UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

DISEÑO Y VERIFICACIÓN EN SYSTEMVERILOG DE UN MÓDULO CONTROLADOR DE CACHÉ PARA RISC-V

Álvaro Fernández Bravo

Tutor: Dr. Jorge Daniel Martínez Pérez

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2018-19

Valencia, 10 de septiembre de 2019

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada REG.NO. ES-CV- 000030

Resumen

Este proyecto se ha llevado a cabo para entender y analizar el funcionamiento de una memoria caché, y cómo esta puede mejorar la velocidad de acceso del procesador a memoria. En concreto, nos hemos centrado en una memoria caché para un procesador RISC-V que trabaja con datos e instrucciones de 32 bits.

Los diseños se han desarrollado con el objetivo de explotar la localidad espacial y temporal, mejorando así el rendimiento. Para ello se han implementado dos modelos de caché: una de mapeado directo y otra asociativa. Además, también se ha desarrollado una memoria secundaria para poder verificar el funcionamiento de ambas cachés.

Finalmente, se han diseñado varios bancos de pruebas: uno para comprobar el correcto funcionamiento de las cachés y analizar el comportamiento de todas las señales de control, y otro para analizar el rendimiento de ambas cachés emulando los accesos a memoria de un programa real ejecutado en un procesador RISC-V.

Resum

Aquest projecte s'ha dut a terme per a entendre i analitzar el funcionament d'una memòria cau, i com aquesta pot millorar la velocitat d'accés del processador a la memòria. Concretament ens hem centrat en una memòria cau per a un processador RISC-V que treballa amb dades e instruccions de 32 bits.

Els dissenys s'han desenvolupat amb l'objectiu d'explotar la localitat espacial i temporal, millorant així el rendiment. Per a dur-ho a terme s'han implementat dos models de memòria cau: un de mapejat directe i un altre associatiu. A mes, també s'ha desenvolupat una memòria secundària per a poder verificar el funcionament d'ambdues caus.

Finalment s'han dissenyat diversos bancs de proves: un per a comprovar el correcte funcionament de les caus i analitzar el comportament de totes les senyals de control, i altre per a analitzar el rediment d'ambdues memòries cau emulant els accessos a memòria d'un programa real executat a un processador RISC-V.

Abstract

This project has been carried out to understand and analyze how a cache works, and how it can improve the processor's access speed to memory. Specifically, we have focused on a cache for a RISC-V processor that works with 32-bit data and instructions.

The designs have been developed with the aim of exploiting the spatial and temporal locality, thus improving performance. For this purpose, two cache models have been implemented: a direct mapping model and a set associative one. In addition, a secondary memory has also been developed to be able to verify the correct functioning of both caches.

Finally, different test benches have been designed: one to check the correct functioning of the caches and analyze the behavior of all the control signals, and another to analyze the performance of both caches emulating the memory accesses of a real program executed in a RISC-V processor.

Índice

Capítulo 1. Introducción

Este proyecto surge a partir del trabajo realizado en la asignatura de Integración de Sistemas Digitales en el cuarto año del grado, en la que se llevó a cabo el diseño y la implementación de un procesador RISC-V de una sola etapa. RISC-V es un conjunto de instrucciones reducido de hardware libre.

El objetivo principal no es otro que el del desarrollo de una memoria caché que pudiera ser utilizada por un procesador como el que hemos comentado e implementada en un dispositivo programable. Esto se ha realizado en paralelo a la implementación de un procesador RISC-V por parte de Izan Segarra Górriz, lo que nos ha permitido llevar a cabo una simulación con los accesos a memoria realizados por este procesador.

1.1 Objetivos

Estos son los objetivos de este proyecto:

- Llevar a cabo una búsqueda de información para comprender cómo funciona una memoria caché.
- Implementar una caché de mapeado directo con un bloque por entrada.
- Diseñar una memoria secundaria.
- Desarrollar una caché de mapeado directo con cuatro bloques por entrada y una caché asociativa *2-way*.
- Comprobar el correcto funcionamiento de las cachés con una serie de lecturas y escrituras conocidas.
- Diseñar un banco de pruebas a partir de los accesos a memoria de un programa real y medir el rendimiento de ambas cachés.

1.2 Metodología

Las herramientas principales que se han usado para llevar a cabo este proyecto han sido *Quartus II* y *Questa Sim*. La primera es una herramienta de software que nos ha permitido realizar el análisis y síntesis de los diseños desarrollados. Gracias a la herramienta *TimeQuest Timing Analyzer* integrada en *Quartus II* hemos obtenido la frecuencia máxima de trabajo de los diseños. Mediante el software *Questa Sim* hemos ejecutado los diferentes bancos de prueba pudiendo comprobar tanto el correcto funcionamiento de los diseños como el rendimiento de estos.

Por otro lado, se ha hecho uso de *Visual Studio Code* como editor de código, así como del sistema de control de versiones *Git*, permitiéndonos llevar un registro de todos los cambios efectuados en el diseño. Todas estas herramientas han sido usadas en el sistema operativo *Windows 10*.

1.3 Plan de trabajo

A continuación, podemos ver cuál ha sido el tiempo aproximado dedicado a cada uno de los objetivos iniciales del proyecto:

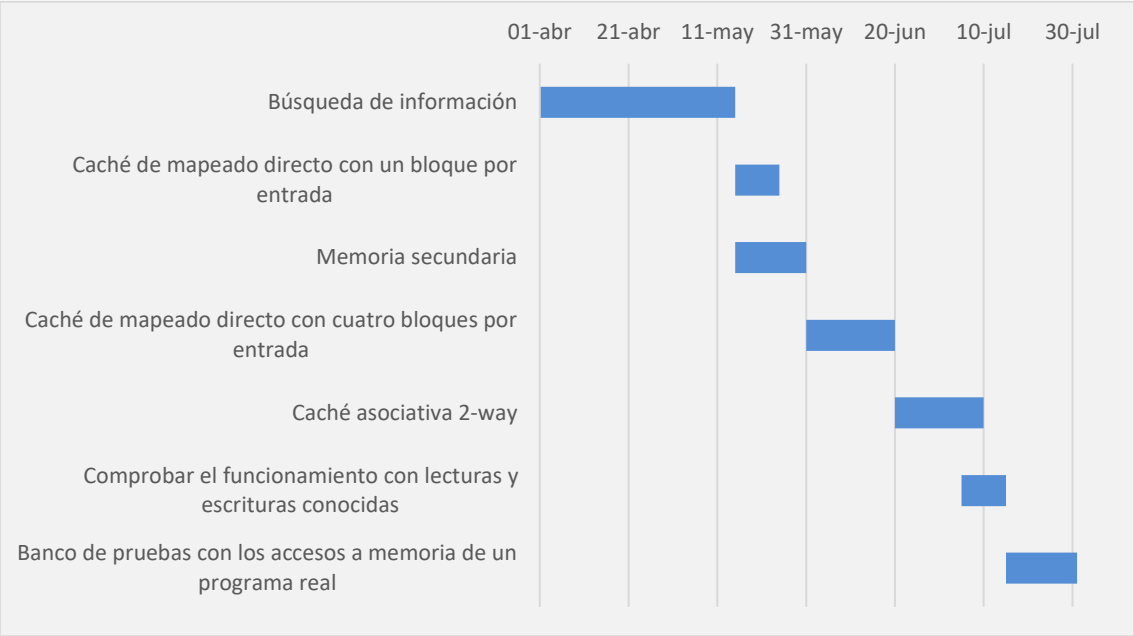

Figura 1. Planificación temporal del proyecto

Capítulo 2. Estudio teórico

2.1 Introducción

Desde la introducción de los primeros ordenadores, el deseo de los usuarios ha sido siempre tener una gran cantidad de memoria relativamente rápida. Los temas tratados en este capítulo ayudan a crear la ilusión de una gran memoria (más barata y lenta) a la que podemos acceder tan rápido como a una memoria más pequeña (más costosa y rápida).

El principio fundamental en el que se basa esta ilusión es que un programa no necesita acceder a todos los datos o instrucciones de una sola vez con la misma probabilidad. De otra forma, sería imposible hacer que la mayoría de los accesos a memoria fueran rápidos y seguir teniendo una gran memoria.

El principio de localidad mantiene que los programas acceden a una relativamente pequeña parte de la memoria en cualquier instante de tiempo. Hay dos tipos de localidad:

- Localidad temporal: si una posición o bloque de la memoria es direccionado, volverá a ser direccionada pronto.
- Localidad espacial: si una posición o bloque de memoria es direccionado, las posiciones adyacentes serán direccionadas pronto.

Extrapolando estos principios de localidad al comportamiento de cualquier programa, surgen dos claros ejemplos. Muchos programas contienen bucles, por lo que una instrucción o dato será requerido repetidamente, mostrando así una alta localidad temporal. Si se accede a las instrucciones de forma secuencial, los programas muestran una alta localidad espacial.

En definitiva, se podría definir la jerarquía de memoria como una estructura que hace uso de diferentes niveles de memorias. A medida que se incrementa la distancia al procesador, el tamaño de la memoria y el tiempo de acceso de esta también se incrementan.

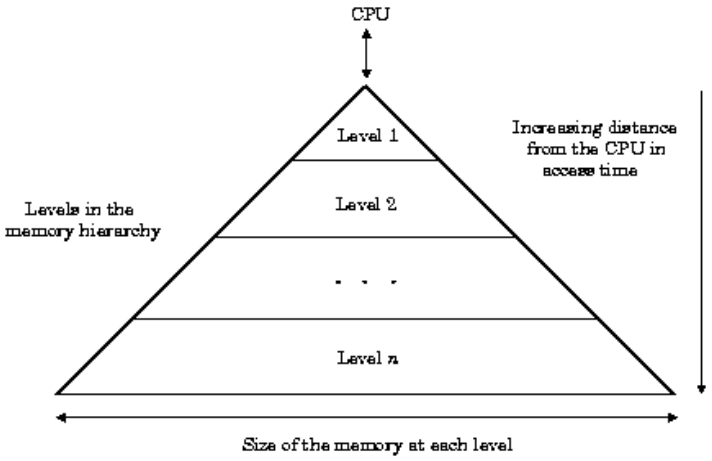

Figura 2. Ejemplo de la estructura de una jerarquía de memoria

Tal y como se muestra en la Figura 2, una jerarquía de memoria puede tener varios niveles, pero los datos son transferidos solo entre niveles adyacentes. La mínima cantidad de información que puede estar o no presente en varios niveles se denomina bloque o línea.

Si el dato requerido por el procesador es encontrado en el nivel superior de la jerarquía se denomina *hit*. Si es dato no se localiza en el nivel superior se denomina *miss*. El *hit ratio* es la fracción de memoria a la que se accede y es encontrada en el nivel superior, el cual suele usarse para medir el rendimiento de la jerarquía de memoria. El *miss ratio* es la porción de memoria a la que se accede y no es encontrada en el nivel superior.

El tiempo para disponer de un determinado bloque de memoria dependiendo de si el acceso ha sido un *hit* o un *miss* es importante. El tiempo de *hit* es el tiempo requerido para acceder al nivel superior de la jerarquía. La penalización por *miss* es el tiempo requerido para reemplazar un bloque del nivel superior por el correspondiente bloque del nivel inferior, más el tiempo necesario para llevar ese bloque al procesador.

2.2 Tipos de memoria

Hay principalmente cuatro tecnologías usadas en las jerarquías de memoria. La memoria principal es generalmente implementada mediante una DRAM (memoria dinámica de acceso aleatorio), mientras que los niveles superiores en la jerarquía y más cercanos al procesador (cachés) usan memorias SRAM (memoria estática de acceso aleatorio). Las memorias DRAM son más baratas que las SRAM, aunque también son más lentas. La diferencia de precio es debida a que las DRAM usan menos área por bit de memoria, por lo que tienen una mayor capacidad para la misma cantidad de silicio. Otro tipo son las memorias flash. Este tipo de memoria no volátil la podemos encontrar normalmente como memoria secundaria en los dispositivos móviles. Por último, encontramos los discos magnéticos o los de estado sólido. Estos son usados para implementar el último (y más lento) nivel de la jerarquía.

2.3 Aspectos básicos de las memorias caché

Cache fue el nombre empleado para representar el nivel de la jerarquía de memoria entre el procesador y la memoria principal en el primer ordenador comercializado en tener este nivel extra. Las cachés aparecieron en la década de 1960 en ordenadores destinados a la investigación y hoy en día las podemos encontrar en cualquier ordenador, desde servidores hasta sistemas embebidos.

A continuación, se verá una caché simple en la que los bloques de memoria consisten en una sola palabra. La Figura 3 muestra un ejemplo sencillo de los datos contenidos en la caché antes y después de que un dato requerido no se encuentre en ella. Antes de la petición, la caché contiene una serie de datos X_1, X_2, …, X_{n-1}, y el procesador solicita el dato X_n que no se encuentra en la caché. Esta petición da lugar a un *miss*, y el dato X_n es traído desde el nivel inferior de memoria hasta la caché.

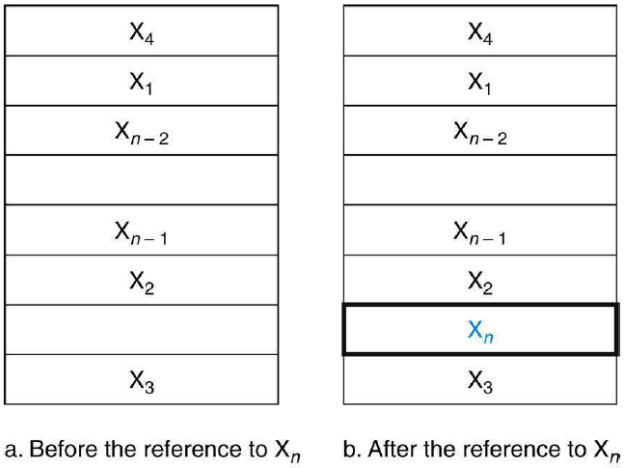

Figura 3. La caché antes y después de la petición del dato X_n que no estaba inicialmente

Después de este ejemplo, se plantea una importante cuestión: ¿cómo podemos saber si un determinado dato se encuentra en la caché? Si cada dato solo puede estar en una determinada posición en la caché, entonces es fácil averiguar si está en la caché. La forma más sencilla de asignar una posición en la caché para cada dato que haya en memoria es asignarla basándose en la dirección del dato en memoria. Esta estructura de caché se denomina de mapeado directo (*direct mapped*), ya que cada posición de memoria es mapeada directamente a una única posición en la caché.

Si el número de entradas de la caché es potencia de 2, el número de bits necesarios para su direccionamiento puede ser calculado con el logaritmo en base 2. Así, una caché de 8 bloques usa los 3 bits menos significativos ($8=2^3$) de la dirección de la memoria. La Figura 4 muestra cómo las direcciones de memoria de la 1 (00001) a la 29 (11101) corresponden a las posiciones de la 1 (001) a la 5 (101) en una caché de mapeado directo de 8 bloques.

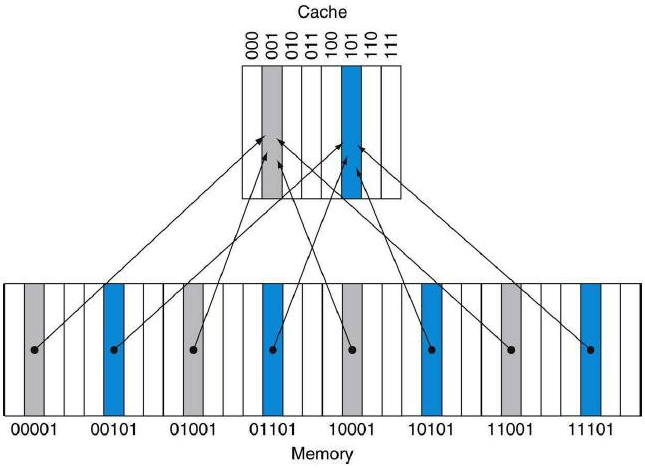

Figura 4. Mapeado entre la memoria y una caché de mapeado directo de 8 bloques

Como cada posición de la caché puede contener diferentes posiciones de memoria, es importante determinar cómo saber si el dato de la caché corresponde a la posición de memoria solicitada. Para ello añadimos el campo *tag* a la caché. Los *tags* contienen la información de la dirección requerida para comprobar si un dato de la caché corresponde al dato solicitado por el procesador. El campo *tag* se corresponde con la parte superior de la dirección, que son los bits que no han sido utilizados en el índice de la caché. Por ejemplo, en la Figura 4 el campo *tag* contendría los dos bits más significativos de la dirección.

También es necesario conocer si un bloque de la caché contiene información valida, ya que cuando se inicia el procesador la caché no contendrá datos válidos y, por tanto, el campo *tag* no tendrá ningún sentido. El método más común es incluir un bit que indique si una determinada entrada contiene información válida.

A continuación, nos centraremos en cómo llevar a cabo las lecturas. Por lo general, una lectura es más sencilla que una escritura, ya que no supone ningún cambio en los datos de la caché.

2.3.1 Accesos a caché

Ahora ya sabemos cómo buscar los datos en la caché para cada posible dirección: los bits más bajos de la dirección pueden ser usados para encontrar la única entrada de la caché posible para esa dirección. La Figura 5 muestra cómo una dirección es dividida en:

- Campo *tag*: es usado para comparar con el valor del campo *tag* de la caché.
- Índice de la caché: es usado para seleccionar un bloque de la caché.

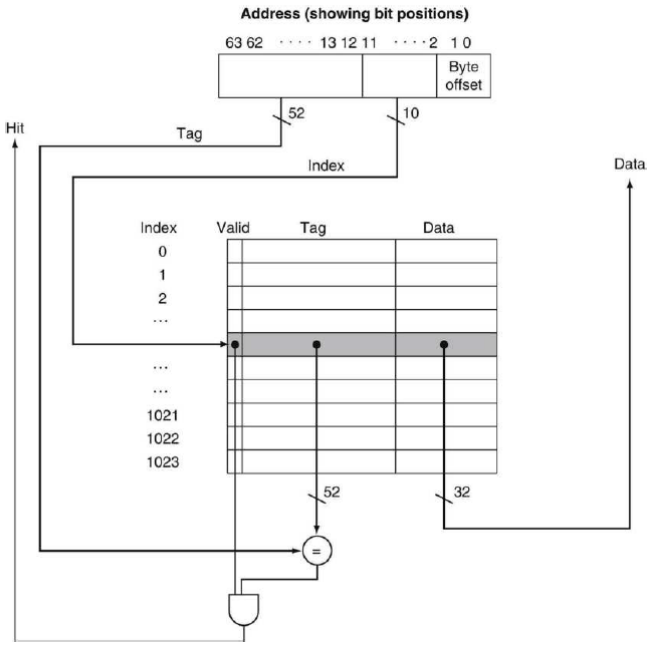

Figura 5. Caché de mapeado directo de 1024 entradas

El índice y el *tag* de un bloque de la caché solo especifican la dirección de memoria de la palabra contenida en ese bloque. Como el índice es usado para direccionar un bloque de la caché y un campo de n-bit tiene 2^n posibles valores, el número total de entradas en una caché de mapeado directo debe ser potencia de 2. Si las palabras no están alineadas en memoria, los dos bits menos significativos de la dirección especifican el byte que contiene la palabra requerida. Sin embargo, si las palabras están alineadas en memoria, los dos bits menos significativos de la dirección pueden ser ignorados. En este proyecto asumiremos que las palabras están alineadas en memoria.

El número total de bits necesarios para una caché depende tanto del tamaño de la caché como del de la dirección. Para determinarlo hay que tener en cuenta dos aspectos:

- El tamaño de la caché es 2^n bloques, así que son necesarios n bits para el índice.
- El tamaño de cada bloque de la caché es de 2^m palabras, por lo que son necesarios m bits para identificar la palabra contenida en el bloque, y dos bits son usados para la parte de *byte offset* de la dirección.

Para una dirección de 64 bits, el tamaño del campo *tag* es:

$$64 - (n + m + 2) \tag{1}$$

El número total de bits en una caché de mapeado directo es:

$$2^n \times (tamaño\ bloque + tamaño\ tag + bit\ dato\ válido) \tag{2}$$

Si el tamaño del bloque es 2^m palabras, y necesitamos 1 bit para señalar si el dato es válido, el número de bits en una caché con palabras de 32 bits como esta es:

$$2^n \times (32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m) \tag{3}$$

En la nomenclatura de la caché solo cuentan los bits dedicados a los datos, por lo que esta es una caché de 4 KiB.

2.3.2 Respuesta frente a un miss

Antes de comenzar a estudiar una caché de un sistema real, vamos a ver cómo controlar las peticiones que acaban en *miss*. La unidad de control debe detectar el *miss* y proceder copiando el dato desde la memoria del nivel inferior a la caché. Si la petición acaba en *hit*, se le pasa el dato al procesador y este continúa funcionando como si no hubiera pasado nada.

Un *hit* no supone ningún cambio en el procesador, sin embargo, lidiar con un *miss* requiere algo más de trabajo. Este trabajo requiere la colaboración de la unidad de control del procesador y de una unidad de control separada que inicie el acceso a memoria y copie en la caché el dato solicitado. El procesado de un *miss* genera una interrupción que requiere congelar todo el contenido de los registros del procesador mientras se espera el resultado del acceso a memoria. Algunos procesadores más sofisticados pueden continuar con la ejecución de instrucciones mientras esperan la respuesta de la caché, aunque en este proyecto asumiremos que el procesador debe esperar al resultado de la caché para continuar con la ejecución.

Vamos a estudiar más detalladamente la respuesta frente a un *miss*. Estos serían los pasos a seguir frente a un *miss* de una instrucción:

1. Enviar el valor original del PC a la memoria.
2. Llevar a cabo una lectura en memoria y esperar a la respuesta de esta.
3. Escribir en la caché, poniendo el dato de leído de la memoria en la entrada correspondiente, escribiendo los bits más significativos de la dirección en el campo *tag* y cambiado el bit de dato válido a 1.
4. Reiniciar la ejecución de la instrucción desde el principio.

En un *miss* de un dato, simplemente habría que parar el procesador hasta que la memoria respondiera con el dato correspondiente.

2.3.3 Respuesta frente a una escritura

Supongamos que en la ejecución de una instrucción de almacenamiento en memoria escribimos el dato solo en la caché, entonces en la memoria habrá un valor distinto al que hay en la caché. En este caso, decimos que hay una inconsistencia entre la memoria y la caché. La forma más sencilla de llevar a cabo una escritura es escribir el dato tanto en la caché como en la memoria. Este esquema se denomina *write-through*.

El único aspecto a tener en cuenta es qué ocurre en un *miss* de una escritura, es decir, que la posición de memoria en la que vamos a escribir no se encuentra en la caché. Después de que el bloque haya sido copiado desde la memoria a la caché, debemos sobrescribir la palabra tanto en la memoria como en la caché. Aunque este diseño simplifica la escritura, no tiene un buen rendimiento, ya que en cada escritura debemos acceder a la memoria para escribir el dato. Estas escrituras bajarían la frecuencia del procesador considerablemente.

Una posibilidad para mejorar este esquema es usar un *buffer* de escritura, almacenando en él el dato mientras esperamos a que sea escrito en memoria. Después de escribir el dato en la caché y en el *buffer*, el procesador puede continuar con la ejecución. Cuando la escritura en memoria es completada, la entrada se puede eliminar del *buffer*. Si el *buffer* estuviera lleno cuando haya una instrucción de almacenamiento en memoria, el procesador debe parar hasta que hubiera de nuevo una posición libre en el *buffer*. Si la frecuencia con la que la memoria puede completar una escritura es menor que la frecuencia con la que el procesador hace una petición de escritura dará igual el tamaño del *buffer*, ya que las escrituras son generadas más rápido que lo que la memoria tarda en completarlas. Aunque la frecuencia de las peticiones de escritura sea menor que la frecuencia con la que la memoria completa la escritura, existe riesgo de que el procesador tenga que parar la ejecución. Esto ocurre cuando las escrituras se producen en ráfagas. Para reducir la aparición de interrupciones, los procesadores normalmente aumentan la capacidad del *buffer* de escritura más allá de una sola entrada.

Otra alternativa es un esquema llamado *write-back*. En este caso, cuando hay una petición de escritura, el nuevo valor es solo escrito en el bloque correspondiente de la caché. El bloque modificado es escrito en memoria cuando sea reemplazado. Este esquema mejora el rendimiento, sobre todo cuando el procesador genera peticiones de escritura a la misma o más frecuencia con la que la memoria puede completar una escritura.

2.4 Rendimiento de la caché

En esta sección, se verá cómo calcular y analizar el rendimiento de la caché. Después veremos otras técnicas para mejorar ese rendimiento. Una de ellas trata de reducir el *miss ratio* reduciendo la probabilidad de que dos bloques distintos de memoria se encuentren en la misma posición en la caché. La otra técnica reduce la penalización por *miss* añadiendo otro nivel a la jerarquía (caché multinivel).

El tiempo que la CPU tarda en ejecutar un determinado programa puede ser dividido en los ciclos de reloj en los que está ejecutando alguna instrucción y los ciclos de reloj en los que espera la respuesta de la memoria. Normalmente, se asume que el retardo de los accesos a caché que son *hits* son parte del tiempo en el que la CPU está ejecutando el programa.

$$Tiempo\ CPU = (ciclos\ de\ reloj\ de\ ejcución + ciclos\ de\ espera\ por\ acceso\ a\ memoria) \times periodo \quad (4)$$

Los ciclos de espera por acceso a memoria son debidos principalmente a los *misses*, por lo que para simplificar asumiremos que no hay ciclos de espera debidos a *hits*.

Los ciclos de espera por acceso a memoria se dividen en los ciclos de espera provenientes de lecturas y de escrituras.

$$Ciclos\ de\ espera\ por\ acceso\ a\ memoria = ciclos\ de\ espera\ por\ lectura + ciclos\ de\ espera\ por\ escritura \quad (5)$$

Los ciclos de espera por lectura pueden ser definidos en función del número de accesos por lectura de un programa, la penalización por *miss* en ciclos de reloj por una lectura y el *miss ratio* de lectura.

$$Ciclos\ de\ espera\ por\ lectura = \frac{Lecturas}{Programa} \times miss\ ratio\ de\ lectura \times penalización\ por\ miss\ de\ lectura \quad (6)$$

Pasamos ahora a analizar las escrituras. Para un esquema *write-through*, tenemos dos motivos de parada: *misses* por escritura, que requiere llevar el bloque de la memoria secundaria a la caché antes de continuar con la escritura, y la espera debida al *buffer* de escritura, que ocurre cuando el *buffer* está lleno y se solicita una escritura.

$$Ciclos\ de\ espera\ por\ escritura = \left(\frac{Escrituras}{Programa} \times miss\ ratio\ de\ escritura \times penalización\ por\ miss\ de\ escritura\right) + espera\ por\ buffer\ de\ escritura \quad (7)$$

Como ya se ha visto en el apartado 2.3.3, la espera por el *buffer* de escritura depende de la proximidad de las escrituras y no solo de su frecuencia, por lo que es imposible reducir este tiempo de parada a una simple ecuación. En sistemas que cuentan con un *buffer* de escritura de mayor capacidad y una memoria capaz de procesar las escrituras más rápido que la frecuencia con la que el procesador las genera, la espera debida al *buffer* de escritura será pequeña y la podremos despreciar, por lo que podemos reducir los tiempos de espera por lectura y escritura a una sola ecuación:

$$Ciclos\ de\ espera\ por\ acceso\ a\ memoria = \frac{Accesos\ a\ memoria}{Programa} \times miss\ ratio \times penalización\ por\ miss \quad (8)$$

El esquema *write-back* también presenta riesgo de parada dada su necesidad de escribir un bloque en memoria cuando este es reemplazado. Aunque en este caso, habría que tener en cuenta que no todos los *misses* por escritura requieren escribir en la memoria secundaria.

2.4.1 Mejorar el rendimiento con otro tipo de emplazamiento de bloques

Hasta ahora, solo se ha visto el esquema de mapeado directo, es decir, cada bloque de memoria se puede situar en un solo lugar en la caché.

En el otro extremo tenemos un esquema donde cada bloque de memoria puede situarse en cualquier posición de la caché. Este esquema se denomina totalmente asociativo (*fully associative*), ya que cada bloque de memoria puede estar asociado a cualquier entrada de la caché. Para localizar un bloque en este esquema debemos buscar en todas las entradas de la caché, lo cual se hace en paralelo con comparadores asociados a cada entrada de la caché. Estos comparadores incrementan considerablemente el coste de la caché, por lo que solo es factible aplicar este esquema en cachés con un número reducido de bloques.

Un punto medio entre la caché de mapeado directo y la totalmente asociativa es una caché asociativa por conjuntos (*set associative*). En este esquema hay un número determinado de entradas de la caché en las que un bloque de memoria puede ser emplazado. Una caché asociativa con n posibles posiciones para un bloque se denomina *n-way*. Una caché asociativa *n-way* consiste en un número determinado de entradas, cada una de las cuales contiene n bloques. Cada bloque de memoria se corresponde con una única entrada en la caché dada el índice de la caché, y puede ser emplazado en cualquiera de los bloques de esa entrada. Como cada bloque puede ser emplazado en cualquiera de las posiciones de la entrada correspondiente de la caché, todos los campos *tag* de todas las posiciones de la entrada deben ser comprobados.

En la Figura 6 se muestra una caché de ocho bloques configurada según estos tres esquemas.

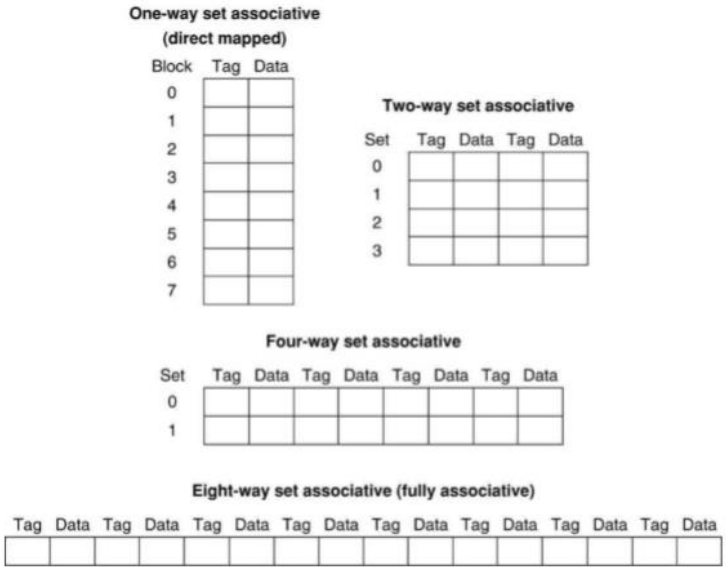

Figura 6. Caché de ocho bloques configurada como mapeado directo, asociativa *2-way*, asociativa *4-way*, y totalmente asociativa

A continuación, vamos a ver cómo evoluciona el *miss ratio* con diferentes esquemas de caché.

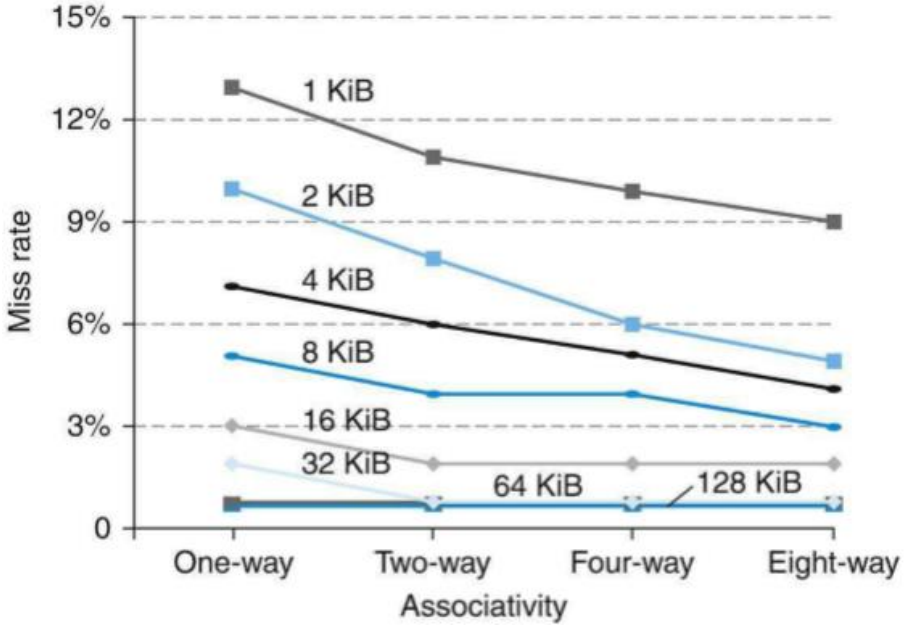

Figura 7. Evolución del *miss ratio* ante distintos grados de asociatividad

Como podemos ver en la Figura 7, mientras la mejora de ir de una caché de mapeado directo a una asociativa de grado dos es importante, la mejora al aumentar el grado de asociatividad más allá de dos no es tan grande. La menor mejora se da al pasar de una caché asociativa *4-way* a *8-way*, que es muy similar al *miss ratio* de una caché totalmente asociativa.

Estos datos han sido obtenidos de un procesador con una organización similar a la del *Intrinsity FastMATH* en las pruebas de *SPEC CPU2000*. SPEC fue creado como un programa para fomentar y reconocer los logros obtenidos por la comunidad académica e industrial en proveer y desarrollar código y datos que serán utilizados en las pruebas de *SPEC CPU*. En la siguiente imagen podemos ver la estructura de la caché del procesador *Intrinsity FastMATH*.

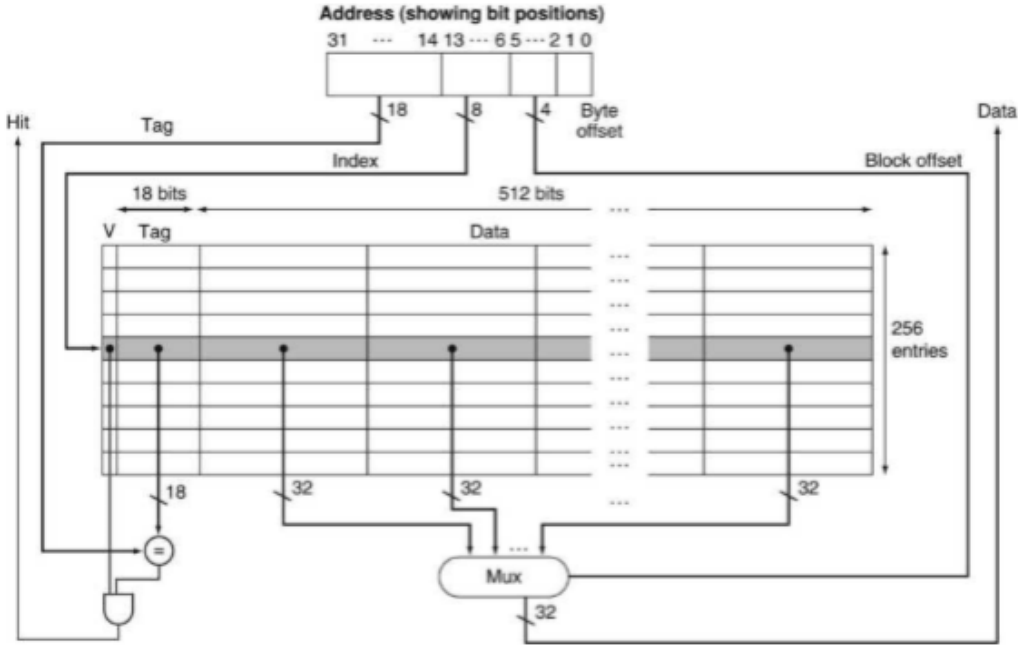

Figura 8. La caché de 16 KiB del procesador *Intrinsity FastMATH*

El *Intrinsity FastMATH* es un microprocesador embebido que usa la arquitectura MIPS y una organización de caché como la que vemos en la Figura 8. En la práctica, para eliminar la necesidad de un multiplexor extra, los campos *tag* y *valid* están en una memoria diferente a los datos.

2.4.2 Localizar un bloque en la caché

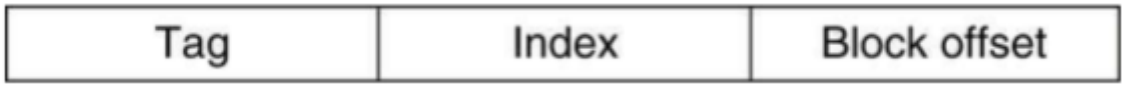

Figura 9. Las tres partes de una dirección en una caché asociativa o de mapeado directo

En este apartado se analizará cómo localizar un bloque en una caché asociativa. Al igual que en la caché de mapeado directo, cada bloque en una caché asociativa incluye un *tag* que indica la dirección del bloque. El campo *tag* es comprobado para ver si coincide con el bloque requerido por el procesador. El valor del índice se usa para seleccionar la entrada de la caché, y los *tags* de todos los bloques de esa entrada son comprobados. Para que esta búsqueda no suponga una gran pérdida de velocidad, todos los *tags* son seleccionados y comprobados en paralelo, tanto en una caché asociativa como en una totalmente asociativa. También se usa un campo de *offset* para seleccionar el dato requerido dentro del bloque, ya que cada bloque puede contener varias palabras.

Si mantenemos el tamaño total de la caché, aumentar el grado de asociatividad incrementaría el número de bloques por entrada. Cada aumento de la asociatividad en una potencia de 2 doblaría el número de bloques por entrada y reduciría a la mitad el número de entradas. Además, reduciría el tamaño del índice en un bit y aumentaría el tamaño del campo *tag* en un bit también.

En una caché totalmente asociativa solo hay una entrada y los campos *tag* de todos los bloques son comprobados en paralelo. Por lo tanto, no hay índice y toda la dirección, excepto el campo *offset*, constituye el campo *tag*.

En una caché de mapeado directo, solo es necesario un comparador, ya que en cada entrada hay un solo bloque al que accedemos mediante el índice.

En la Figura 10 podemos ver como ejemplo una caché asociativa *4-way* que requiere cuatro comparadores y un multiplexor 4 a 1. Los comparadores determinan si alguno de los bloques de la entrada seleccionada coincide con el *tag* de la dirección requerida por el procesador. Las salidas de los comparadores son usadas como bit de selección en el multiplexor, eligiendo así el bloque que contiene el dato requerido.

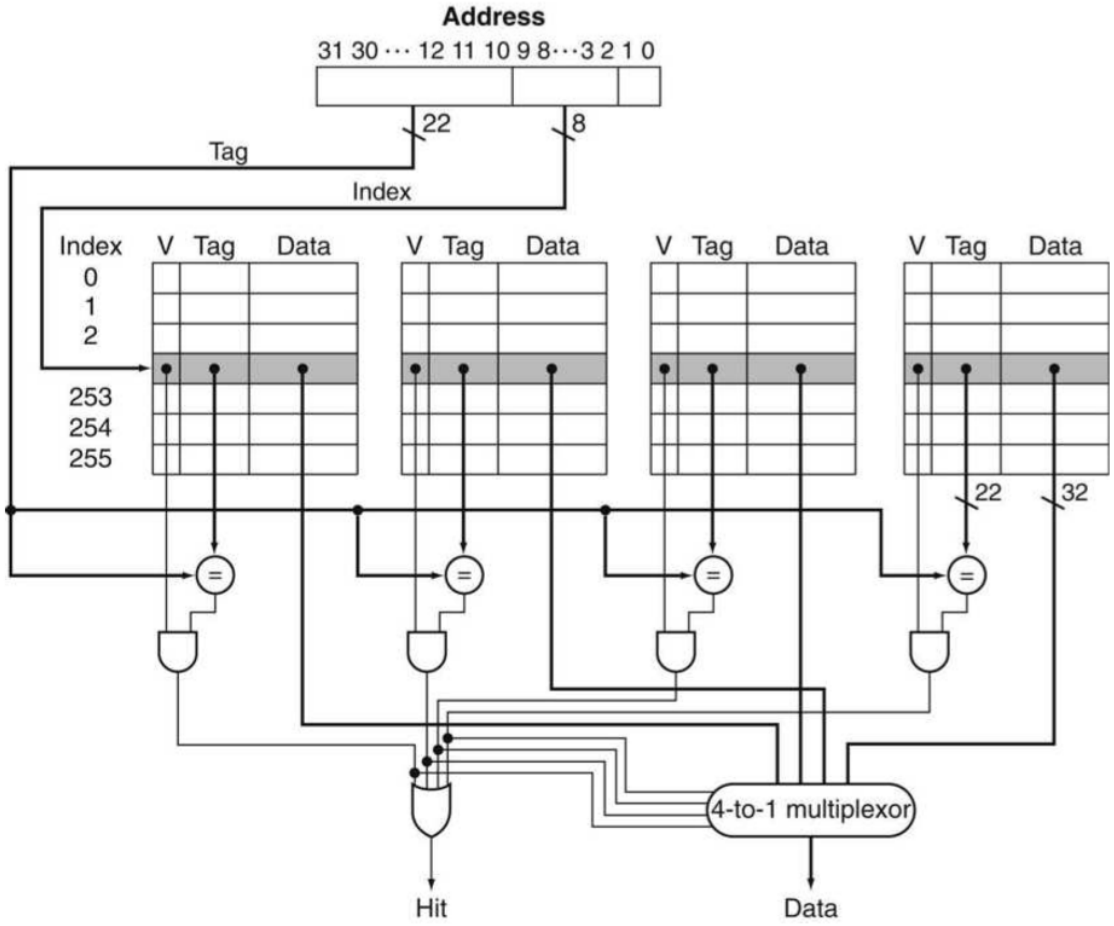

Figura 10. Caché asociativa *4-way*

2.4.3 Elegir qué bloque reemplazar

Cuando se produce un *miss* en una caché de mapeado directo, el bloque requerido solo puede ir en una posición, por lo que se reemplaza el bloque que hay en esa posición. En una caché asociativa o totalmente asociativa, tenemos la posibilidad de elegir en qué posición colocar el bloque requerido y, por tanto, debemos elegir que bloque reemplazar. En una caché totalmente asociativa podríamos reemplazar cualquier bloque, en una caché asociativa debemos elegir entre los bloques de la entrada correspondiente.

El esquema más usado es elegir reemplazar el bloque que más tiempo lleva sin usarse (LRU – Least Recently Used). Este esquema es implementado teniendo un registro de cuando ha sido usado cada uno de los bloques de una determinada entrada. En una caché asociativa *2-way*, para llevar a cabo este registro basta con tener un único bit en cada entrada y usarlo para indicar cuando uno de los bloques es referenciado.

Capítulo 3. Diseño

Para este proyecto se ha decidido diseñar dos esquemas de caché, una de mapeado directo y otra asociativa de grado 2. Con esto se pretende explotar los dos tipos de localidad que se han visto en la sección 2.1, lo cual explicaremos mejor a lo largo de este apartado.

Por otro lado, cabe mencionar que ambas cachés admiten una nueva petición por parte del procesador cada ciclo de reloj. Esto se consigue con un registro por el que pasan estas peticiones y permite mantenerlas hasta que se termine con la operación actual. Mientras la petición por parte del procesador no suponga un *miss* este podrá seguir enviando una nueva petición cada ciclo, de lo contrario desde el controlador de la caché se activará una señal de *stopped* indicando al procesador que debe parar. Cuando el *miss* sea resuelto por la caché, esta señal se desactivará y el procesador podrá seguir ejecutando el programa. Profundizaremos más en este tema en el Capítulo 4, que aborda la verificación funcional del diseño.

3.1 Interfaz de control

Las características y señales de control de estas cachés son:

- Palabras de 32 bits.
- Esquema *write-back* para escrituras.
- Tamaño del bloque de cuatro palabras en el caso de la caché de mapeado directo (128 bits). Bloques de una sola palabra en el caso de la caché asociativa y un grado de asociatividad dos.
- Tamaño de la caché de mapeado directo de 16 KiB y 8 KiB en el caso de la asociativa.
- Bus de direcciones de 20 bits.
- Cada bloque incluye un bit *valid* (indicando si contiene información válida) y un bit *dirty* (que nos indica si el bloque que va a ser reemplazado en una escritura hay que escribirlo antes en memoria). En el caso de la caché asociativa, también tenemos un bit *LRU* con el que podemos saber cuál es el bloque de cada entrada que se ha usado menos recientemente.
- Índice de la caché de 10 bits (1024 entradas).
- En el caso de la caché de mapeado directo, tenemos un campo de *offset* que indica el bloque que se está direccionando desde el procesador.
- Tamaño del campo *tag* de 6 bits en la caché de mapeado directo y de 8 bits en la asociativa.

Las señales entre el procesador y la caché son:

- 1 bit para identificar si es una operación de lectura o escritura.
- 1 bit indicando que hay una operación válida desde el procesador.
- Bus de datos de 32 bits.
- 1 bit para señalar que la caché ha completado la lectura.
- 1 bit para indicar al procesador que ha habido un *miss* y debe parar de solicitar lecturas o escrituras.

Estas son las señales en la interfaz entre la caché y la memoria:

- 1 bit para identificar si es una lectura o escritura.
- 1 bit que indica que hay una operación válida desde la caché.
- Bus de datos de 128 bits en el caso de la caché de mapeado directo y 32 bits en la asociativa.
- 1 bit para indicar que la operación ha sido completada por la memoria.

Con el objetivo de facilitar toda la conexión entre los distintos módulos se ha hecho uso de las estructuras de *SystemVerilog*. Esta herramienta permite declarar una serie de señales dentro de

una estructura y así pasarlas entre diferentes módulos como un conjunto. En la Figura 11 podemos ver un ejemplo de estas estructuras, en este caso es la estructura que define la interfaz de señales del procesador a la caché.

```
//CPU request (CPU -> Cache controller)
typedef struct{
    bit [19:0] addr;
    bit [31:0] data;
    bit rw;                 // 0=read / 1=write
    bit valid;              // valid read/write request from the processor
}cpu_to_cache_type;
```

Figura 11. Estructura que define las señales entre el procesador y la caché

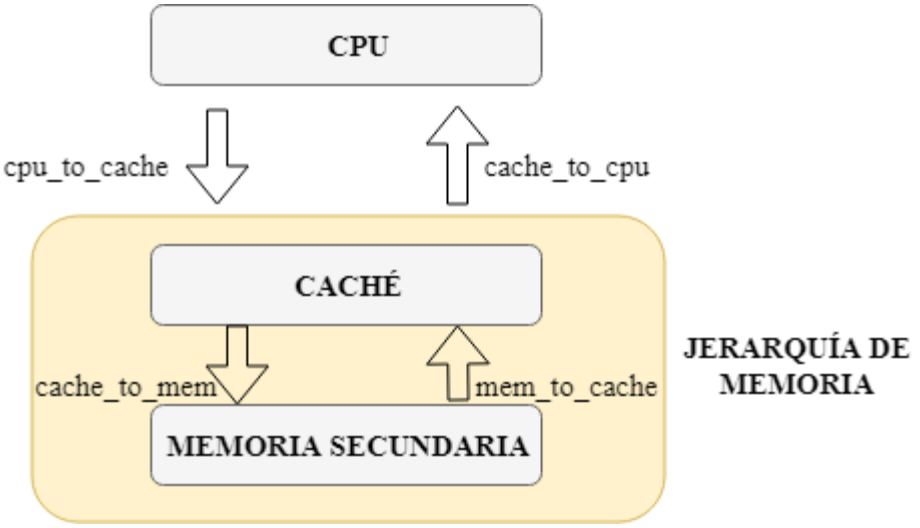

Figura 12. Esquema de la interconexión de los diferentes módulos

3.2 Usar una máquina de estados para controlar la caché

El control de una caché es algo complejo, ya que cada operación consta de una serie de pasos. El control debe especificar qué señales activar en cada uno de esos pasos, así como el siguiente paso en la secuencia. El método más común para este control es usar una máquina de estados finita (Finite State Machine – FSM). Una FSM consiste en un conjunto de estados y unas condiciones sobre cómo cambiar de estado. A continuación, vamos a detallar los tres estados utilizados para controlar las cachés implementadas:

- *Compare Tag*: Cuando llega una lectura o escritura válida desde el procesador, este estado comprueba si esa petición supone un *hit* o un *miss*. Mediante el índice seleccionamos la entrada de la caché cuyos *tags* hay que comparar. Si el dato contenido en el bloque de la caché que hemos seleccionado mediante es válido, y el *tag* del bloque coincide con el campo *tag* de la dirección requerida por el procesador, entonces es un *hit*. Cuando esto sucede el bit *ready* de la caché es puesto a 1. En caso de ser una escritura, también se marcaría el bit *valid* y el *dirty*. Mientras no haya un *miss*, la máquina de estados seguirá en este estado. En caso de que haya un *miss*, primero se actualizará el *tag* del bloque correspondiente y después pasará al estado *Write-back*, si el bloque tenía el bit *dirty* a 1, o al estado *Allocate*, si el bit *dirty* estaba a 0.

- *Write-back*: Este estado escribe el bloque que va a ser reemplazado en memoria utilizando como dirección la concatenación del campo *tag* del bloque a reemplazar y el índice de la caché. Permanecemos en este estado hasta recibir el bit *ready* desde la memoria. Cuando la escritura en memoria haya sido completada, pasaremos al estado *Allocate*.
- *Allocate*: En este estado el nuevo bloque es traído desde la memoria, por lo que permanecemos en aquí hasta que se active el bit *ready* de memoria. Cuando la lectura de memoria es completada, nos movemos al estado *Compare Tag*.

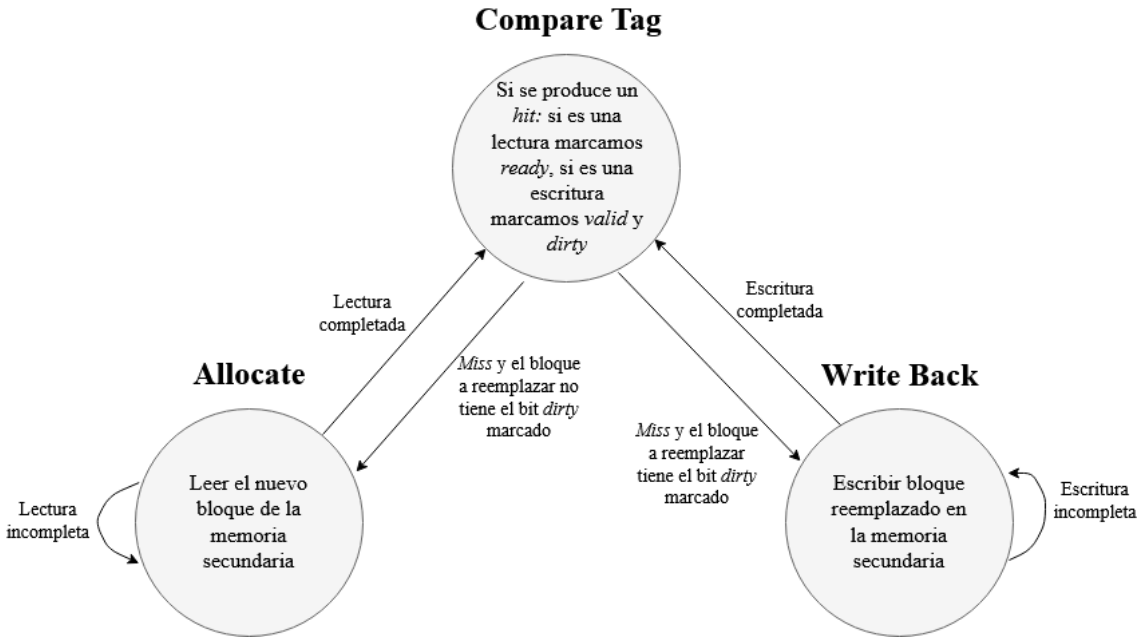

Figura 13. Máquina de estados para controlar la caché

3.3 Caché de mapeado directo

3.3.1 Caché de mapeado directo con un bloque por entrada

Inicialmente, nuestro diseño de la caché de mapeado directo consistía en una caché con un solo bloque por entrada, similar al explicado en la Figura 5. Esta caché consta de dos memorias independientes con 1024 entradas cada una: una que contiene los campos de información de cada una de las entradas (bit *valid*, bit *dirty* y campo *tag*), y otra que contiene los datos. Con dos memorias separadas evitamos un multiplexor extra para seleccionar el dato que queramos. Ambas están diseñadas para escritura síncrona y lectura asíncrona. Tienen un bus de direcciones de 10 bits, la memoria que contiene la información de cada entrada tiene un bus de datos de datos de 10 bits (*valid* + *dirty* + *tag*) y la memoria que contiene los datos de 32 bits.

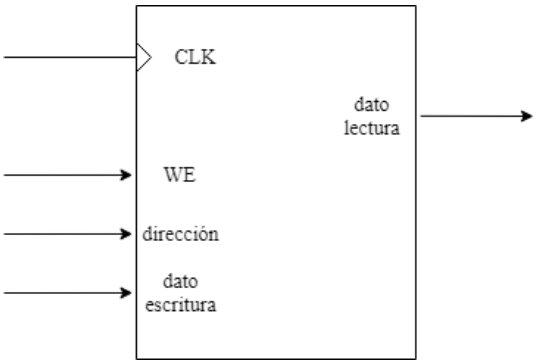

Figura 14. Esquema de las memorias que componen la caché

Para determinar si la petición del procesador resulta en un *hit* o un *miss* basta con utilizar la memoria que contiene los campos de información, ya que solo necesitamos el bit *valid* y el campo *tag*:

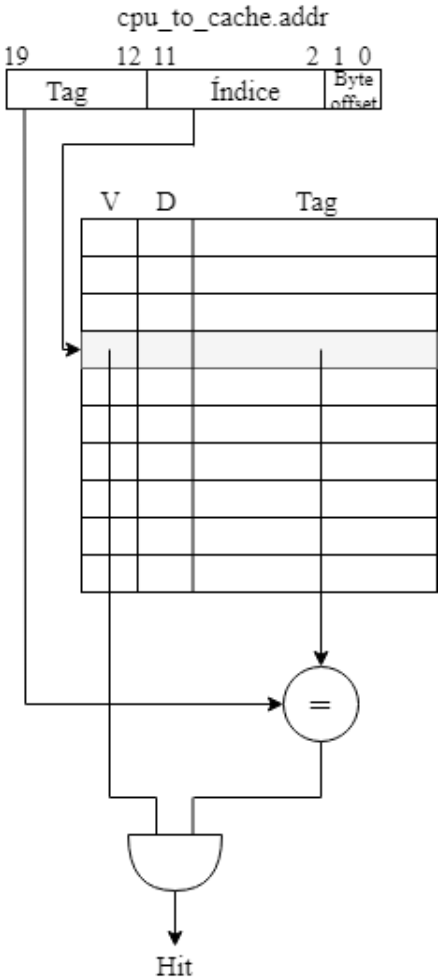

Figura 15. Cómo se determina un *hit* en la caché de mapeado directo

Si determinamos que se trata de un *hit*, leemos o escribimos en la entrada seleccionada por el índice. Si por el contrario se trata de un *miss*, hay que comprobar si el bit *dirty* está marcado. Si no lo está, copiamos desde la memoria secundaria el bloque y reemplazamos el bloque contenido en la entrada correspondiente de la caché. Si el bit *dirty* está marcado, antes de reemplazar el bloque en la entrada de la caché, habrá que escribirlo en la memoria secundaria.

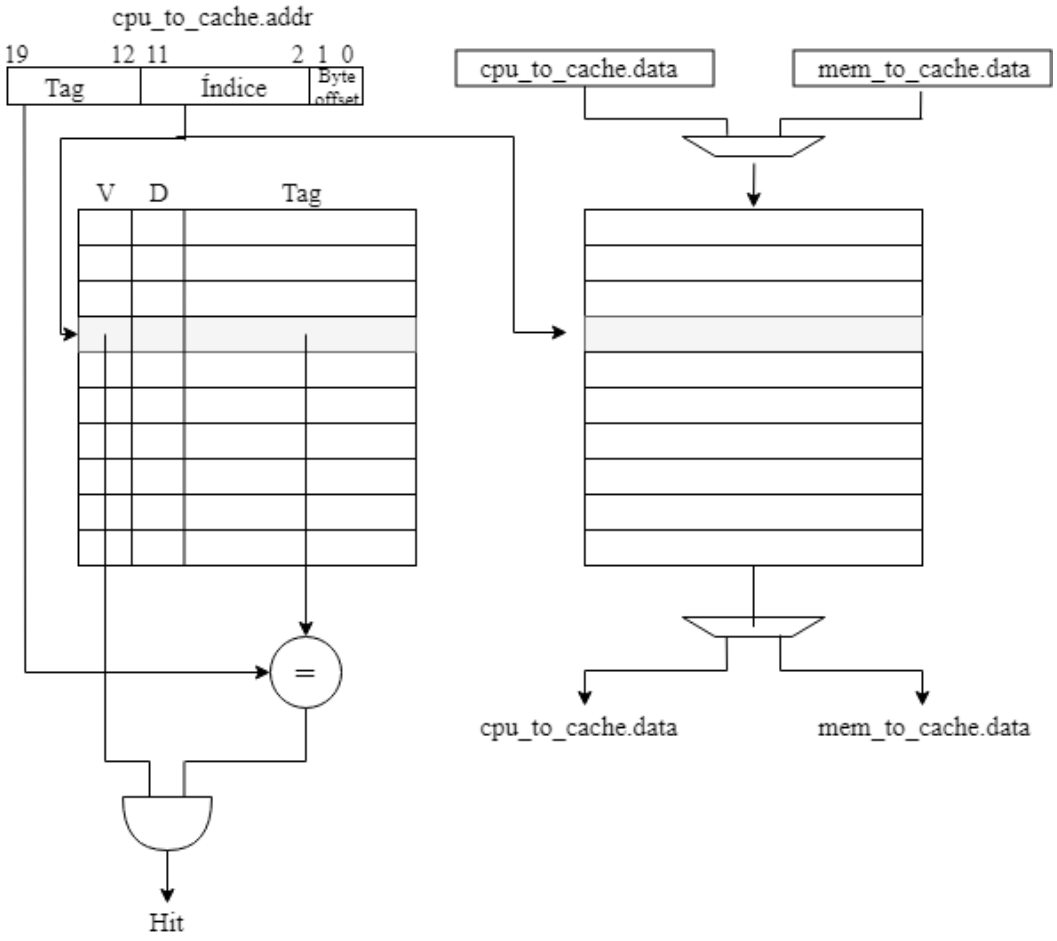

Figura 16.Caché de mapeado directo de 4 KiB con un bloque por entrada

El problema de esta organización es que no supone una gran mejora con respecto a la memoria secundaria más allá de la tecnología de fabricación. Por lo que, en el siguiente apartado vamos a abordar el diseño de una caché de mapeado directo con la que explotaremos la localidad espacial.

3.3.2 Caché de mapeado directo con cuatro bloques por entrada

Este esquema de caché se ha diseñado con el objetivo de explotar la localidad espacial, es decir, cuando se accede a la memoria secundaria para copiar una palabra a la caché, no solo se copia la palabra contenida en la dirección referenciada por el procesador, sino también las adyacentes. De esta forma, si las posiciones adyacentes son direccionadas pronto no será necesario volver a acceder a la memoria secundaria. Para ello, cada entrada de la caché contiene cuatro palabras. La organización de esta caché está basada en la caché del procesador *Intrinsity FastMATH*.

Esta caché tiene la misma organización que la anterior en cuanto a que está compuesta también por dos memorias independientes. En este caso, la memoria que contiene los campos de información tiene un bus de datos de 8 bits, ya que el campo *tag* se reduce dada la necesidad de

destinar dos bits de la dirección a seleccionar el bloque correspondiente de la memoria de datos. La memoria de datos tiene un bus de 128 bits para albergar los cuatro bloques.

El mecanismo para determinar si ha habido un *hit* o un *miss* es el mismo que en el caso anterior. Por otro lado, para evitar tener una señal que habilite la escritura en cada uno de los bloques de la memoria de datos por separado, a la hora de escribir copiamos toda la entrada y sustituimos solo el bloque que nos interesa. Más adelante veremos cómo lo hacemos.

Si hemos determinado que se trata de un *hit*, hay que pasarle al procesador la palabra requerida (o escribir en ella) dentro de la entrada seleccionada por el índice. Si se trata de una lectura, almacenamos en la variable *data read* los cuatro bloques de la entrada seleccionada y, mediante un multiplexor cuya entrada de selección es la señal *block offset*, podemos seleccionar la palabra referenciada por el procesador. Si es una petición de escritura, copiamos en la variable *data write* el contenido de *data read* y direccionamos el dato enviado por el procesador a cuatro multiplexores, cuya entrada de selección será de nuevo la señal *block offset*. Como hemos copiado previamente todo el contenido de la entrada en *data write*, modificaremos solo el bloque que contiene la palabra enviada por el procesador.

Por el contrario, si se produce un *miss* hay que hacer uso de la memoria secundaria. Como ya hemos visto en el apartado anterior, el procedimiento difiere en función de si el campo *dirty* está marcado o no. Si no está marcado, basta con copiar desde la memoria la entrada seleccionada, sin necesidad de escribir la entrada que va a ser reemplazada. En este caso, se copiaría la entrada completa desde memoria en el campo *data write* y se escribiría en la entrada correspondiente. Si el campo *dirty* estuviera marcado, antes de copiar la entrada desde memoria hay escribir en memoria la entrada que va a ser reemplazada. Para ello, copiamos la entrada correspondiente en *data read* y la escribimos en memoria. En la Figura 16 podemos ver el esquema completo de esta caché.

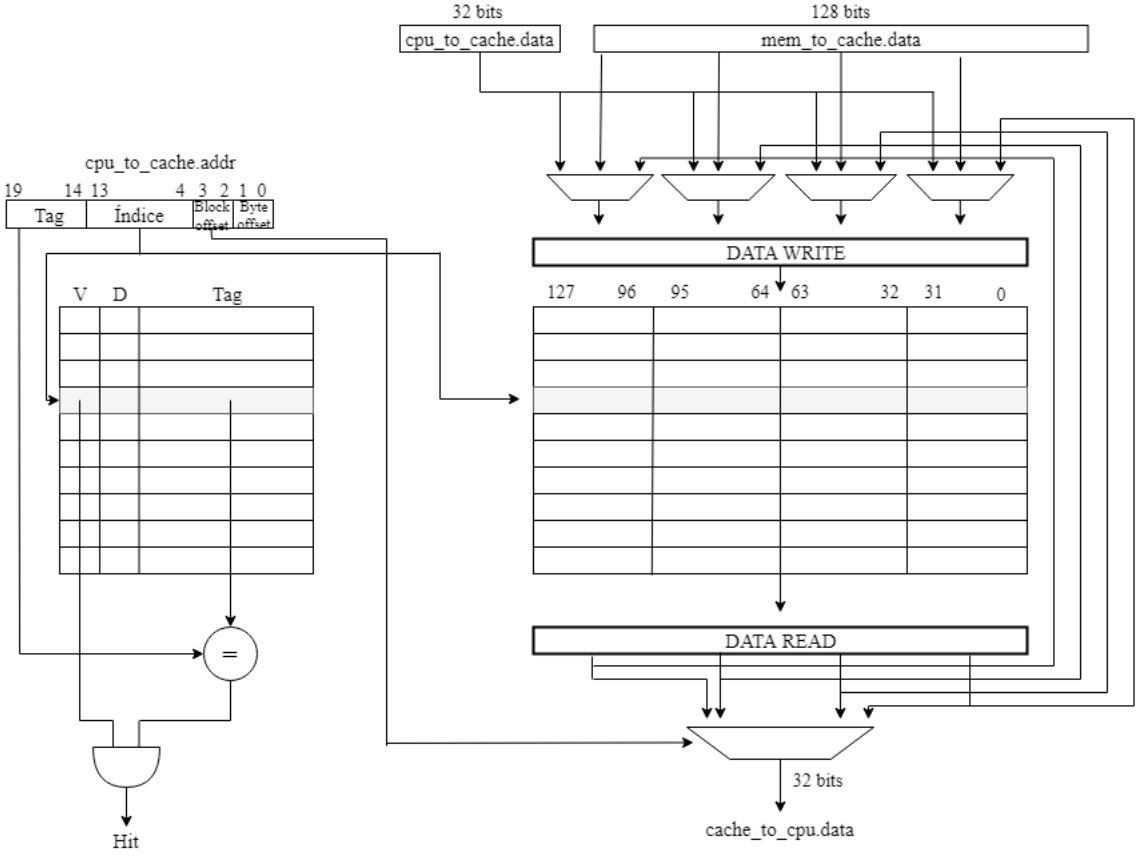

Figura 17. Caché de mapeado directo de 16 KiB con cuatro bloques por entrada

Una vez compilado el diseño, podemos comprobar la frecuencia máxima de trabajo que en este caso es de 122.32 MHz.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	122.32 MHz	122.32 MHz	clk	

Figura 18. Frecuencia máxima de la caché de mapeado directo

3.4 Caché asociativa

En este caso hemos intentado explotar la localidad temporal. Como cada bloque puede ocupar varias posiciones, cuando ocurre un *miss* no necesariamente hay que reemplazar el bloque de la entrada correspondiente, por lo que si es direccionado más adelante seguirá ahí.

La caché asociativa se ha diseñado con cuatro memorias: dos para datos (al ser asociativa de grado dos tiene dos bloques por entrada) y otras dos para las señales de control de ambos bloques de cada entrada. Esta última memoria, además de almacenar los campos *valid*, *dirty* y *tag*, también guarda el bit *LRU* que indica cual es el bloque de cada entrada que lleva más tiempo en desuso. Estas memorias siguen el mismo esquema que las memorias de la caché de mapeado directo,

aunque en este caso las memorias que contienen los campos de información tienen un bus de datos de 11 bits (*LRU + valid + dirty + tag*) y las que contienen los datos de 32 bits.

En este esquema, para comprobar si hay un *hit* o un *miss*, hay que comprobar los dos campos *tag* y *valid* de la entrada direccionada por el índice, haciendo uso de las dos memorias que contienen los campos de información:

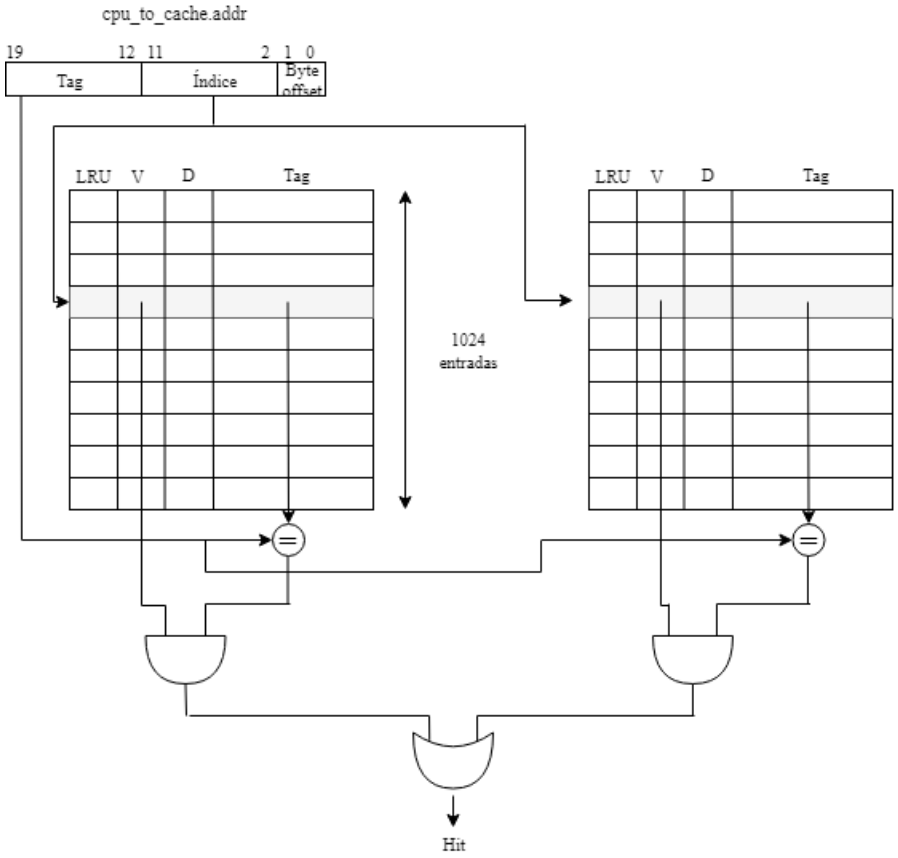

Figura 19. Cómo se determina un *hit* en la caché asociativa

Una vez determinado que se ha producido un *hit*, debemos pasarle al procesador el bloque adecuado. Como cada memoria de control está asociada a una de datos, podemos saber cuál es el dato adecuado. En este caso, el proceso es más sencillo que en la caché de mapeado directo, ya que cada bloque sólo contiene una palabra. Si se trata de una escritura, escribiremos en el que tenga el bit *LRU* a 1, ya que será el que más tiempo lleve en desuso y, por tanto, hay menos posibilidades de que se vuelva a direccionar pronto.

Igual que en la caché de mapeado directo, si se produce un *miss* habrá que acceder a la memoria secundaria. Si el bit *dirty* no estuviera marcado, copiamos el bloque correspondiente de memoria en el bloque de la caché que tenga el bit *LRU* a 1. Si el bit *dirty* estuviera marcado, antes de sustituir el bloque de la caché debemos escribirlo en memoria.

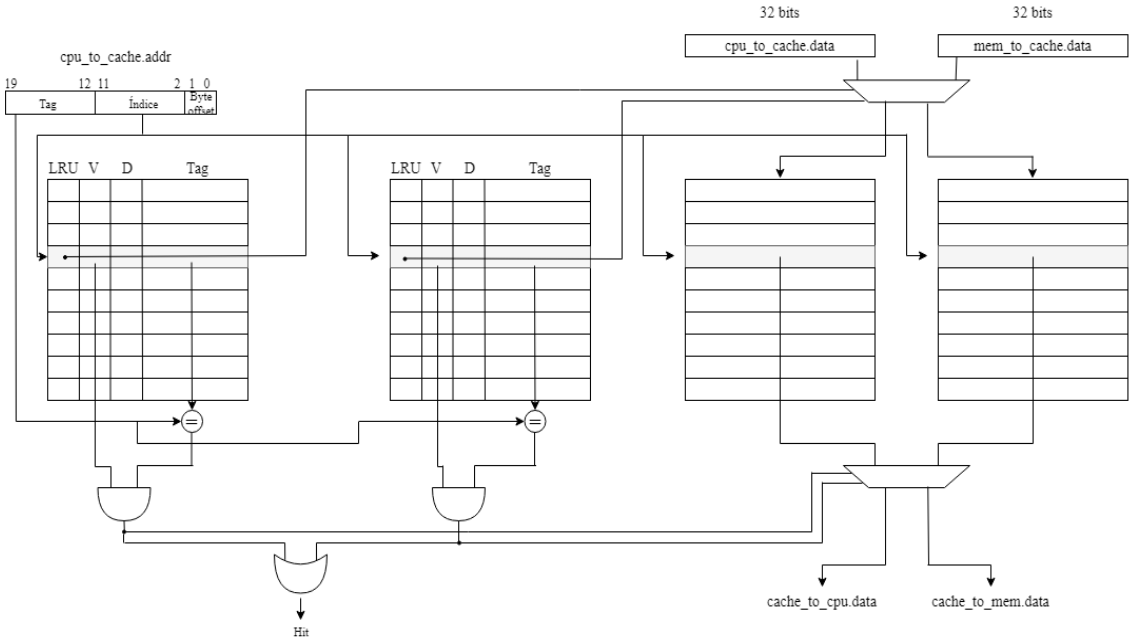

Figura 20. Caché asociativa de 8 KiB

En este caso, la frecuencia máxima de operación es algo inferior a la de la caché de mapeado directo, concretamente es 103.51 MHz.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	103.51 MHz	103.51 MHz	clk	

Figura 21. Frecuencia máxima de trabajo de la caché asociativa

3.5 Memoria secundaria

Para verificar el funcionamiento de ambas cachés es necesario una memoria secundaria. En un comienzo se pretendió usar como memoria secundaria la SRAM de la FPGA DE2-115, ya que es la que tiene una interfaz más sencilla. Esta es una RAM estática de 16M bits, organizados como 1024K palabras de 16 bits. En la siguiente figura podemos ver su diagrama de bloques.

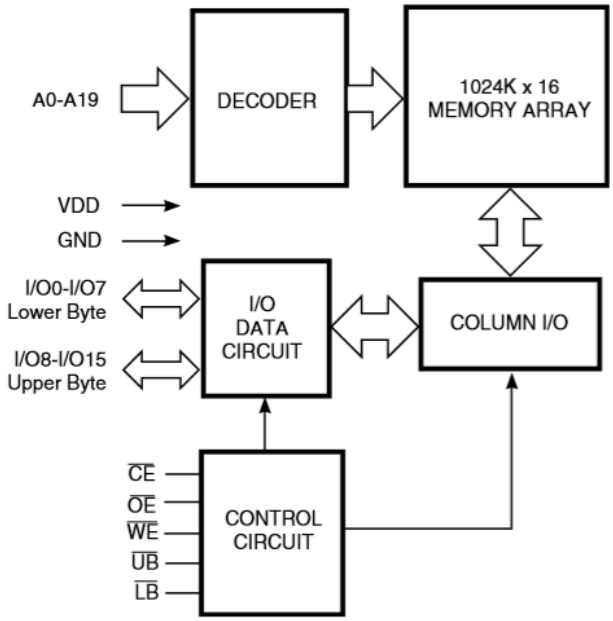

Figura 22. Diagrama de bloques de la SRAM equipada en la FPGA DE2-115

Esta memoria tiene un tiempo de acceso para lectura de 20 ns y de 9 ns para escritura y un control de datos para el bit más y menos significativo. Como podemos comprobar en la imagen anterior, tiene un circuito de control con diferentes señales, por lo que fue necesario diseñar un controlador para su funcionamiento. Suponiendo que la caché tiene una frecuencia de trabajo de 100 MHz y para simplificar el controlador de la SRAM se asimió el mismo tiempo de acceso para lectura y escritura (20 ns), por lo que mantenemos las señales de control dos ciclos de reloj. Además, se diseñó un modelo de la memoria en *Verilog* para verificarla. Este controlador se basa en una máquina de estados, tal y como podemos ver en la Figura 23.

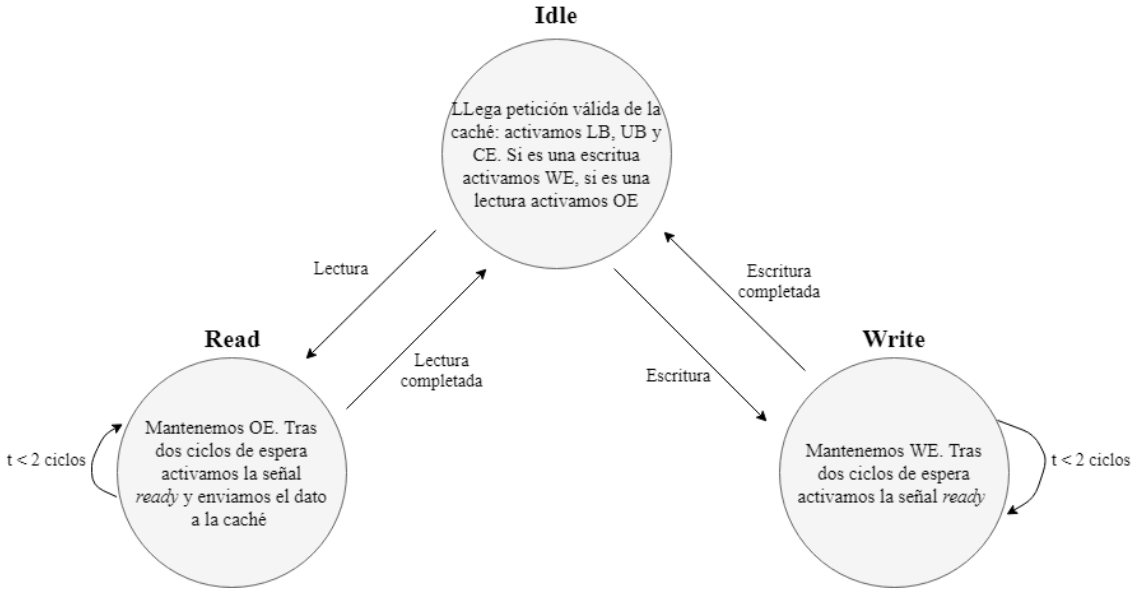

Figura 23. Máquina de estados del controlador de la SRAM

Finalmente, se barajó la posibilidad de utilizar un programa simple diseñado para el procesador RISC-V y emular sus accesos a memoria para verificar el funcionamiento de la caché y nos pareció la mejor opción. Por ello, se descartó la SRAM de la placa, puesto que la caché se ha diseñado para un procesador RISC-V que trabaja con palabras de 32 bits y escritura tipo byte. Además, una SRAM no sería la memoria más recomendada para emplear como memoria secundaria, ya que como se ha visto en el apartado 2.2 es precisamente ese tipo de memoria la que se suele emplear para la caché.

Se optó por implementar una RAM síncrona de esas características. De nevo, hubo que implementar un controlador para manejar las señales de control entre la memoria secundaria y la caché, implementando una máquina de estados muy similar a la anterior.

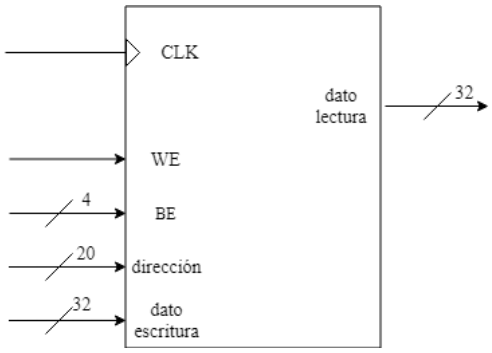

Figura 24. Esquema de la RAM empleada como memoria secundaria

Capítulo 4. Verificación

Para este capítulo se ha implementado una ROM asíncrona donde se cargarán los diferentes accesos a memoria. Estos accesos estarán en hexadecimal en un archivo .txt, cada uno de los cuales incluye un bit para identificar si es lectura o escritura, un campo para la dirección de memoria y otro para el dato a escribir en caso de escritura.

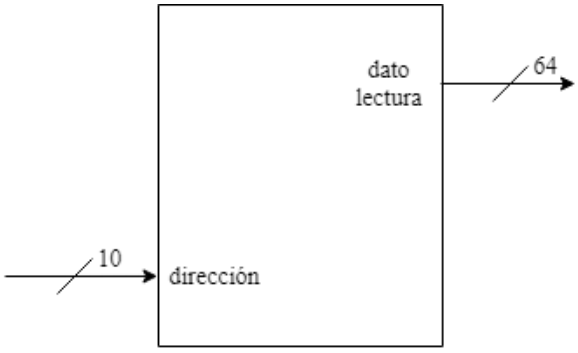

Figura 25. Esquema de la ROM empleada

Para inicializar un archivo en hexadecimal en memoria se utiliza la función *$readmemh*, si estuviera en binario utilizamos la función *$readmemb*. En la Figura 26 podemos ver una de las líneas del archivo .txt que cargamos en la ROM, que en este caso sería una escritura:

lectura/ escritura	dirección	dato
1	00000004	00000004

Figura 26. Formato del archivo .txt cargado en la ROM

Ahora la cuestión es cómo podemos comprobar que se realizan correctamente las lecturas y escrituras. La única forma de comprobar una escritura sería directamente visualizar en contenido de la memoria en *Questa Sim*. Así que se ha optado por comprobar las lecturas, ya que si el dato leído es correcto supone que se hizo bien la escritura previamente. Para comprobar esto se ha optado por una RAM asíncrona sin ningún tipo de retardo similar a la RAM empleada como memoria secundaria que funcionará en paralelo con la caché, de tal forma que si hay una escritura o lectura también se producirá en la RAM.

Supongamos ahora que hay una petición de lectura por parte del procesador que acaba en *miss*. El procesador, tal y como veremos en este apartado, aún puede enviar otra petición en el siguiente ciclo de reloj antes de recibir la señal de *stopped*. Si esa petición también es una escritura, habremos leído dos datos de la RAM cuando aún no hemos resuelto la primera petición desde la caché, por lo que no podemos comparar ambas lecturas. Para resolver este problema se ha optado por implementar una cola en la que iremos metiendo las lecturas de la RAM y comprobándolas cuando la caché las tenga disponibles. Si ambas lecturas no coinciden, *Questa Sim* lanzará un mensaje de error. Por lo tanto, mientras no se reciba la señal de *stopped* por parte de la caché, cada ciclo de reloj el puntero que apunta a la dirección de la ROM irá aumentando, generando así una nueva petición.

```
while (1) @(posedge clk) begin

    if (!cpu_to_cache.rw && !cache_to_cpu.stopped) begin    //read request from CPU
        queue.push_front(check_data_r);     //push data from the RAM into the queue
    end

    if (cache_to_cpu.ready) begin           //read completed
       queue_data = queue.pop_back();

        assert (queue_data == cache_to_cpu.data) else
            $error("Lectura erronea. Dato correcto: %h | Dato leido: %h", queue_data, cache_to_cpu.data);

    end
end
```

Figura 27. Fragmento de código empleado para comprobar si la lectura es correcta

Así, en caso de que la lectura sea errónea, aparecerá un mensaje en *Questa Sim* como el siguiente:

```
# ** Error: Lectura erronea. Dato correcto: 00000008 | Dato leido: 0000004
```

Figura 28. Mensaje de error en caso de lectura errónea

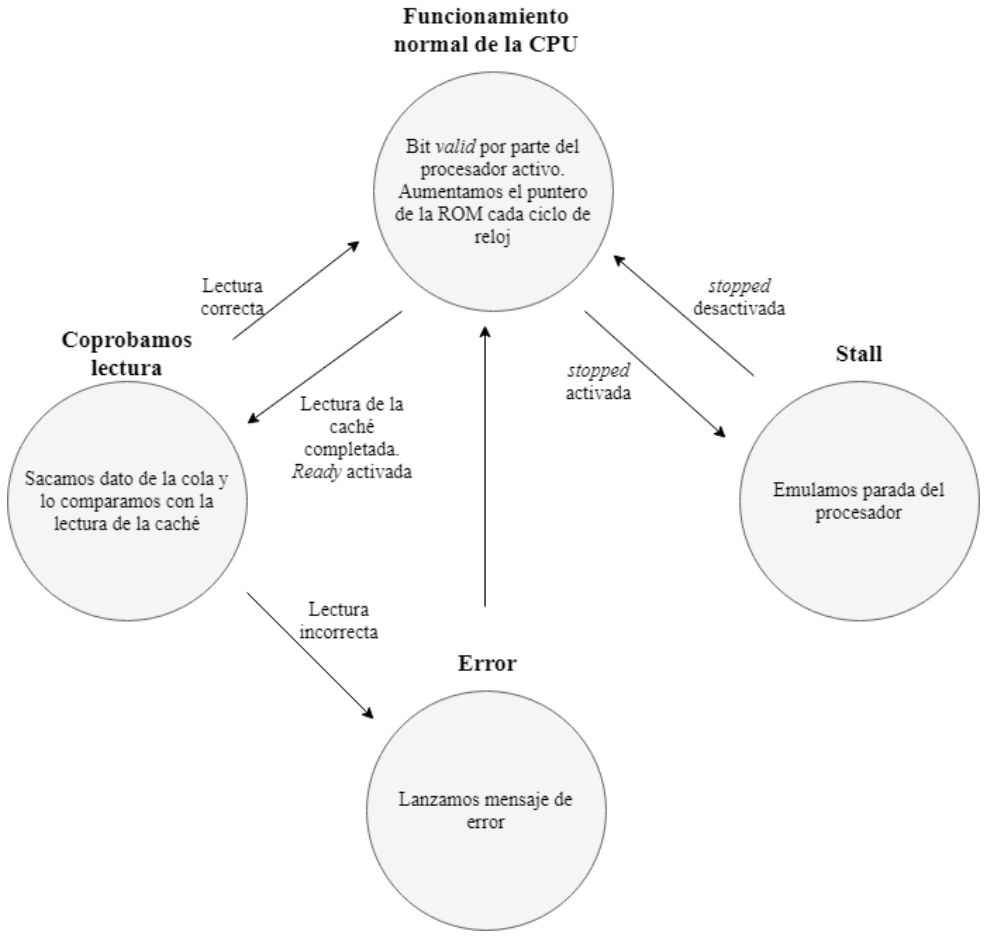

Figura 29. Flujo de trabajo del *Test Bench*

4.1 Verificación RTL

En este apartado vamos a analizar el funcionamiento de la caché en algunos casos que nos han parecido conflictivos (*corner cases*) para comprobar que se comporta correctamente en estas circunstancias. Hay que tener presente que este tipo de simulación no tiene en cuanta los retardos de las puertas lógicas ni los retardos de propagación.

Esta simulación se ha realizado principalmente para identificar los posibles problemas en el diseño y poder comprobar el comportamiento de la caché para cada una de las peticiones, ya que más adelante haremos una verificación con una gran cantidad de accesos a memoria de un programa real y si hubiera algún error sería más difícil identificar la causa. Para simplificar los accesos y que nos sea más sencillo identificarlos, se ha optado por escribir en cada posición de memoria esa misma dirección, es decir, en la posición 0000 escribimos 0000 o en la posición 0004 escribimos 0004. En la Figura 30 podemos ver los accesos simulados en este apartado.

```
10000000000000000
10000000400000004
10000000800000008
10000000c0000000c
000000000xxxxxxxx
000000004xxxxxxxx
000000008xxxxxxxx
00000000cxxxxxxxx
10008000c0008000c
1000c000c000c000c
00000000cxxxxxxxx
00008000cxxxxxxxx
0000c000cxxxxxxxx
```

Figura 30. Conjunto de accesos simulados en la verificación RTL

4.1.1 Verificación RTL de la caché de mapeado directo

A continuación, comprobaremos el correcto funcionamiento de la caché de mapeado directo ante una serie de casos y analizaremos el comportamiento de las diferentes señales de control. En el caso de esta caché, tenemos un bus de datos entre la memoria secundaria y la caché de 128 bits. La memoria secundaria que hemos diseñado tiene un bus de datos de 32 bits, por lo que para la verificación se han usado cuatro RAMs en paralelo y, mediante el controlador de estas, repartimos los 128 bits.

Como podemos comprobar en Figura 31, la primera petición por parte del procesador es una escritura en la posición 0000. El *reset* es activo a nivel bajo, por lo que al ponerlo a 1 (cursor azul) en el siguiente flanco de subida del reloj la señal de *stopped* se activa (cursor rojo), ya que la caché comprueba que no tienen el dato de esa posición y envía una petición de lectura a memoria.

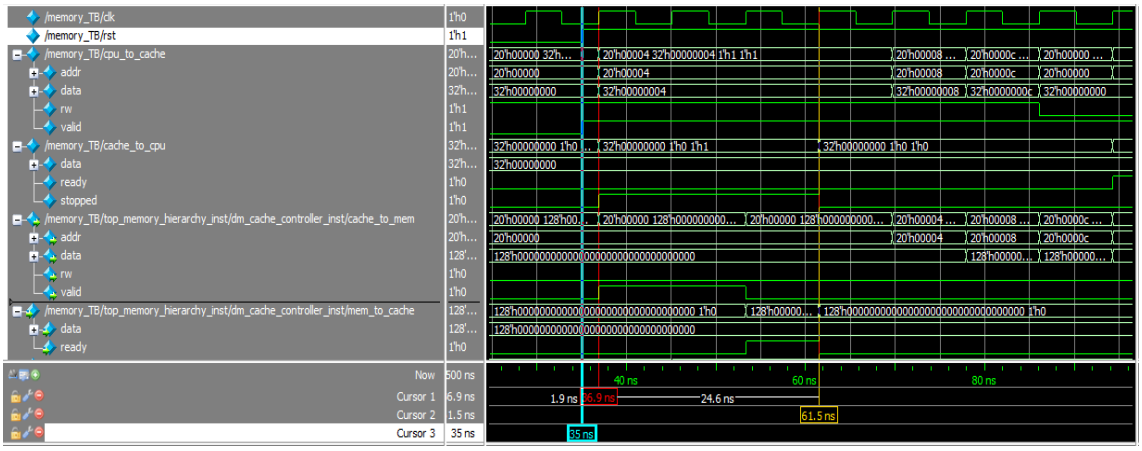

Figura 31. Verificación RTL de la caché de mapeado directo: primeras peticiones de escritura

Podemos ver cómo, al coincidir la activación de la señal *cache_to_cpu.stopped* con el flanco de reloj, el puntero de la ROM aún aumenta una posición, emulando así una nueva petición por parte del procesador. En el instante en que se activa la señal de *stopped*, se genera una petición a la memoria secundaria (se activa la señal *cache_to_mem.valid*) solicitando una lectura de las posiciones 0000, 0004, 0008 y 000C, ya que como ya hemos visto la caché de mapeado directo accede también a las posiciones adyacentes. Cuando se desactiva la señal de *stopped* (cursor amarillo), el procesador vuelve a enviar nuevas peticiones, en este caso escrituras en las posiciones 0008 y 000C, que como ya han sido copiadas desde la memoria secundaria no dan lugar a ningún *miss*. Vemos como la señal de *stopped* ha permanecido activa durante 24.6 ns, que es la menor penalización por *miss* en esta caché ya que en este caso solo ha habido una lectura de la memoria secundaria sin necesidad de escribir en ella.

A continuación, leeremos los datos que acabamos de escribir. En la Figura 32 vemos cómo, tras enviar la petición de lectura (cursor rojo), en el siguiente ciclo de reloj se activa la señal *cache_to_cpu.ready* indicándole al procesador que ya tiene el dato disponible. El tiempo de *hit* ha sido de 8.2 ns (mismo tiempo para lectura y escritura. Vemos también como el procesador puede seguir enviando una nueva petición cada ciclo hasta que se produce el siguiente *miss* (cursor azul).

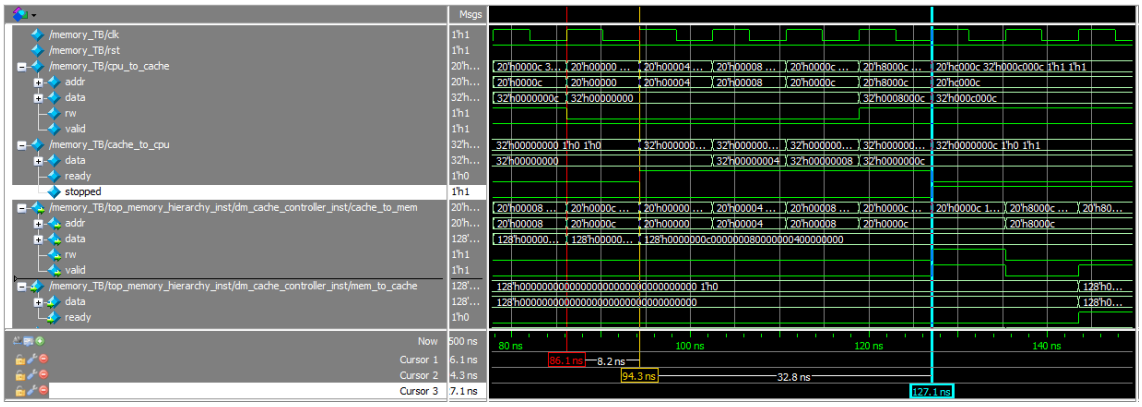

Figura 32. Verificación RTL de la caché de mapeado directo: primeras peticiones de lectura

Una vez comprobada la lectura, nos queda comprobar la escritura en memoria secundaria (*write-back*). Para ello, generamos una petición de escritura en una entrada de la caché que ya hayamos escrito pero con un *tag* distinto, provocando así un *miss* y obligando a la caché a escribir el bloque en memoria antes de ser reemplazado. En la Figura 33 vemos cómo generamos dos escrituras seguidas a la misma entrada de la caché, pero con *tags* distintos (cursor azul). De esta forma, estamos obligando a la caché a reemplazar la entrada en la que ya habíamos escrito antes para después volverla a reemplazar otra vez. Vemos cómo en este caso la penalización por *miss* ha sido de 41 ns, ya que hemos hecho una lectura y escritura de la memoria secundaria.

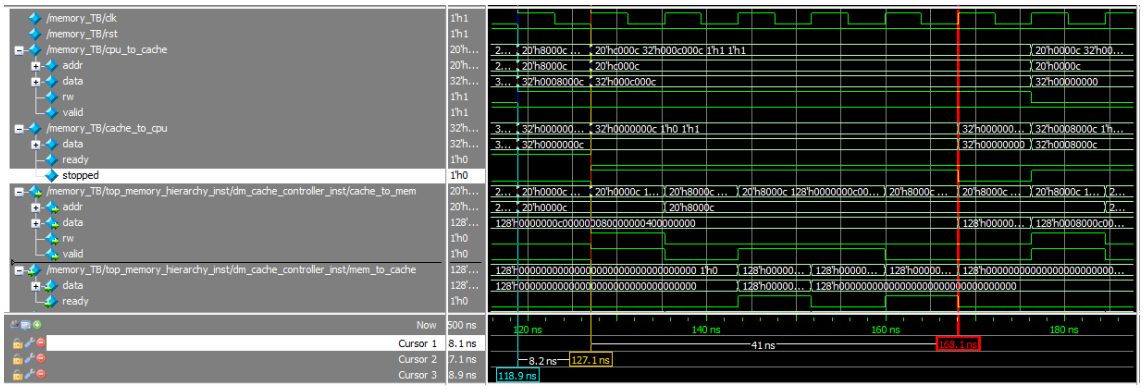

Figura 33. Verificación RTL de la caché de mapeado directo: provocando una escritura en memoria

Por último, solo nos queda leer las primeras posiciones que hemos escrito para comprobar que la escritura en la memoria secundaria se ha hecho correctamente. De nuevo, habrá una escritura en memoria del último bloque que hemos escrito antes de copiar desde memoria la entrada solicitada en la lectura. Esto provocará una gran penalización por *miss*, ya que cuando se genera la petición de lectura de la posición 000C, la caché aún está resolviendo la petición anterior. Como vemos en la Figura 34, desde que se envía la petición de lectura (cursor amarillo) hasta que se activa la señal *cache_to_cpu.ready* (cursor rojo), transcurren 90.2 ns.

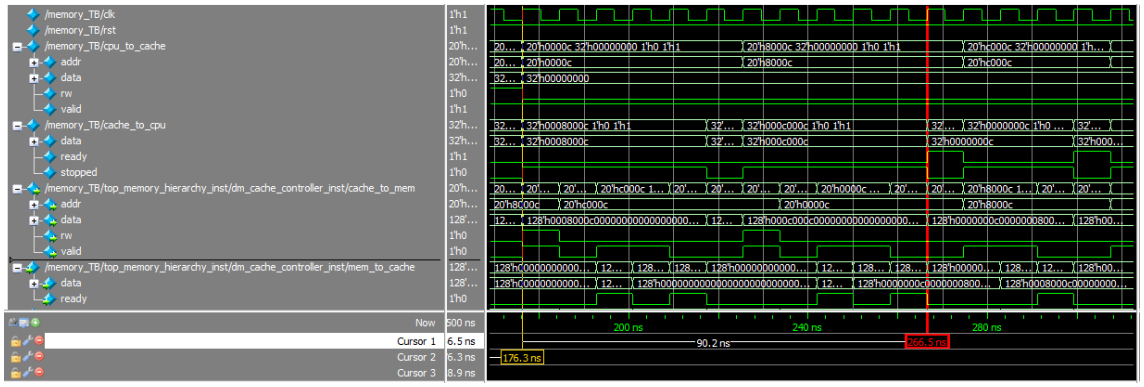

Figura 34. Verificación RTL de la caché de mapeado directo: comprobando *write-back*

4.1.2 Verificación RTL de la caché asociativa

En este caso, realizaremos los mismos accesos en la caché asociativa y veremos cómo se comporta. El bus de datos entre la memoria secundaria y la caché es de 32 bits, por lo que solo es necesario usar una RAM como memoria secundaria.

De nuevo, la primera petición por parte del procesador es una escritura en la posición 0000. Al igual que en la caché de mapeado directo, tras desactivar la señal de *reset* (cursor amarillo) se activa la señal *stopped* y se realiza un acceso a la memoria secundaria (cursor rojo).

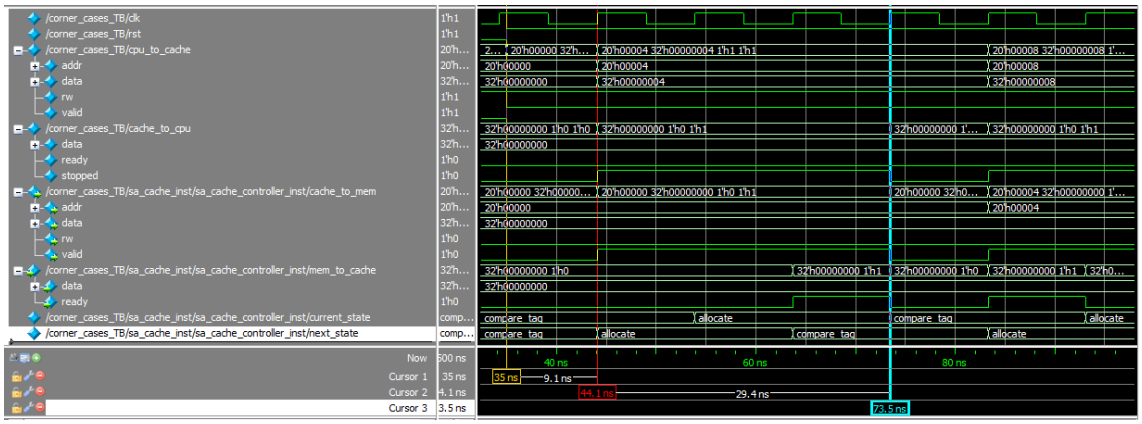

Figura 35. Verificación RTL de la caché asociativa: primeras peticiones de escritura

La principal diferencia que vemos con respecto a la caché de mapeado directo es que los accesos a la memoria secundaria son individuales, es decir, solo leemos o escribimos una sola palabra. En esta simulación, esto perjudica a esta caché ya que la mayoría de los accesos son a posiciones adyacentes, presentando así una alta localidad espacial. En la Figura 35, podemos ver cómo al desactivarse la señal de *stopped* (cursor azul) nos indica que se ha resuelto la primera petición. En el siguiente ciclo de reloj el procesador envía una nueva petición y la señal de *stopped* se vuelve a activar porque la petición anterior (escritura en la posición 0004) ha acabado en *miss*. Vemos como la señal de *stopped* ha permanecido activa 29.4 ns, que es la menor penalización por *miss* que presenta esta caché. Las cuatro primeras peticiones de lectura se han resuelto en 191.1 ns, mientras que en la caché de mapeado directo se resolvieron en apenas 86.1 ns.

Como podemos ver en la Figura 36, la primera petición de lectura se produce a los 161.7 ns (cursor azul), aunque no se resuelve hasta 39.2 ns después (cursor rojo). Aunque esta petición no suponga un *miss*, ha tardado en resolverse puesto que cuando se ha enviado, la caché aún estaba resolviendo la petición anterior (escritura en la dirección 000C). El resto de peticiones de lectura ya las resuelve de forma consecutiva.

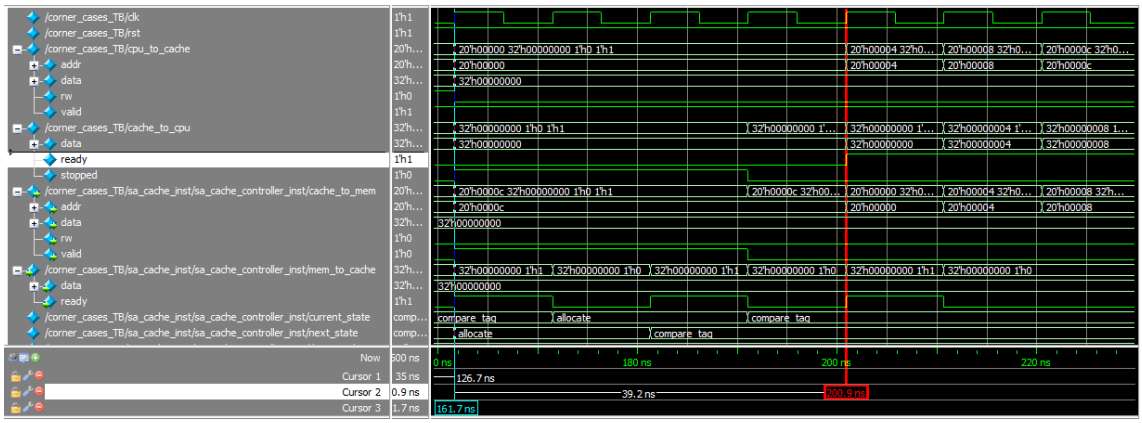

Figura 36. Verificación RTL de la caché asociativa: primeras peticiones de lectura

Ahora vamos a comprobar cómo, ante una petición de escritura en una posición ya ocupada, esta caché hace uso del segundo bloque que tienen libre para esa entrada y no se produce ningún *miss*. Vamos a escribir en las posiciones 8000C y C000C. En la Figura 37 vemos cómo la escritura en la posición 8000C (cursor rojo) no supondrá una escritura en memoria ya que en esa entrada queda un bloque libre. La escritura en la dirección C000C (cursor azul) sí que supondrá una escritura y se deberá reemplazar el bloque que lleve más tiempo en desuso, en este caso el de la dirección 000C, ya que el de la posición 8000C lo acabamos de escribir. La escritura en la posición 8000C se ha resuelto en 29.4 ns, mientras que en la dirección C000C ha tardado 49 ns.

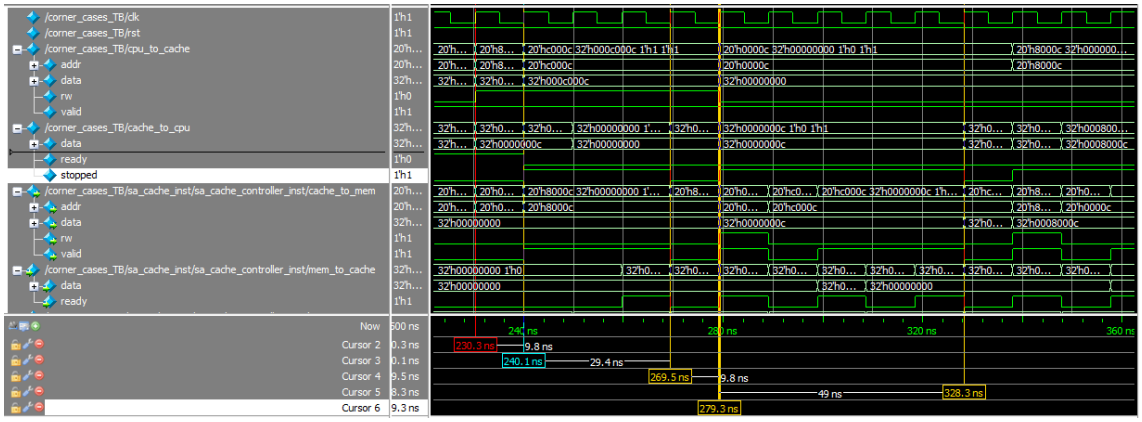

Figura 37. Verificación RTL de la caché asociativa: provocando una escritura en memoria

Por último, comprobaremos que la escritura en la memoria secundaria se ha realizado correctamente. En la Figura 38 vemos cómo enviamos una petición de lectura de la dirección 000C (cursor rojo), que es la que ha sido reemplazada previamente. Para copiar el contenido de esa dirección a memoria, hay que reemplazar uno de los bloques de esa entrada, ya que ambos están ocupados. En este caso se reemplazará el bloque correspondiente a la dirección 8000C, ya que es el que lleva más tiempo en desuso. Esa misma dirección la volveremos a leer (cursor azul) para comprobar de nuevo que la escritura y lectura de la memoria secundaria se realiza correctamente. La lectura de la dirección 000C se ha resuelto en 107.8 ns ya que, además de suponer un *miss*, la caché aún estaba procesando la petición anterior.

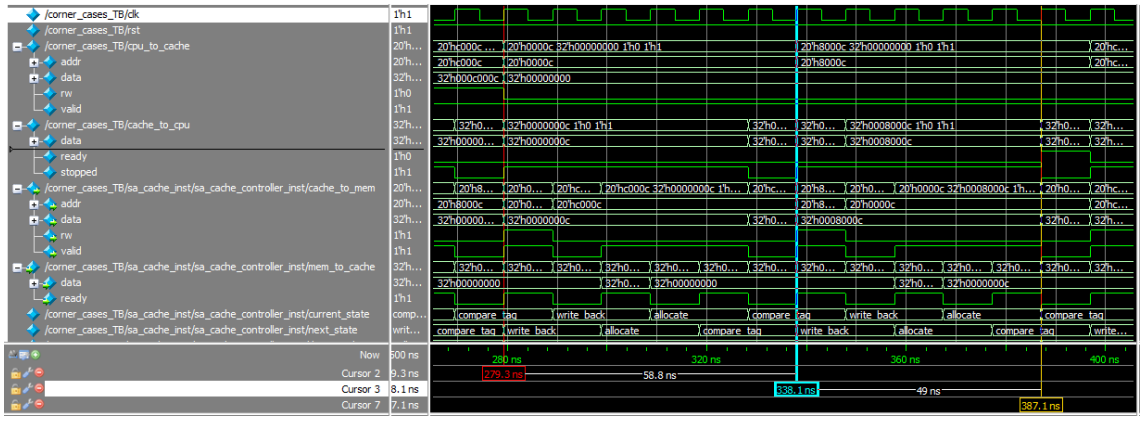

Figura 38. Verificación RTL de la caché asociativa: comprobando *write-back*

4.2 Benchmark

En este apartado mediremos el rendimiento de las dos cachés diseñadas y, para ello, emularemos los accesos a memoria de un programa diseñado para un procesador RISC-V. El programa utilizado es el *Bubble Sort*, que consiste en un sencillo algoritmo de ordenamiento. Funciona revisando cada pareja de elementos de una lista que va a ser reordenada de menor a mayor, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces la lista hasta que no se necesiten más intercambios.

Este programa ha sido escrito en C y, posteriormente, pasado a código ensamblador ejecutable por un procesador RISC-V. Esto ha sido posible mediante una *toolchain* denominada GCC (*GUN Compiler Collection*), que es una colección de herramientas utilizada en los sistemas UNIX. El repositorio que contiene esta herramienta se encuentra en el siguiente enlace: https://github.com/riscv/riscv-gnu-toolchain

De nuevo, se han cargado todos los accesos a memoria en una ROM para ser leídos. Se ha modificado el *Test Bench* para contabilizar el número de escrituras y lecturas que se producen, los *misses* producidos y los ciclos de espera debidos a ellos. Al finalizar la simulación, se lanzará un mensaje desde *Questa Sim* aportando toda esta información.

4.2.1 Benchmark de la caché de mapeado directo

A continuación, vamos a analizar el rendimiento de la caché de mapeado directo. Estos son los datos obtenidos tras la simulación:

```
# Total accesos:         559
# Total lecturas:         427 | Total escrituras:         132
# Misses lectura:           2 | Misses escritura:           9
# Ciclos espera lectura:            6 | Ciclos espera escritura:          27
```

Figura 39. Datos obtenidos tras la simulación del *benchmark* en la caché de mapeado directo

Además, sabemos que el tiempo total de ejecución ha sido de 4891.3 ns. La penalización por *miss* de escritura o lectura es de una media de 3 ciclos de reloj. También podemos calcular el *miss* y el *hit ratio*:

$$miss\ ratio\ (\%) = \frac{total\ misses}{total\ accesos} \times 100 = \frac{11}{559} \times 100 = 1.968\ \% \tag{9}$$

$$hit\ ratio\ (\%) = \frac{total\ hits}{total\ accesos} \times 100 = \frac{559-11}{559} \times 100 = 98.032\ \% \tag{10}$$

$$miss\ ratio\ lectura\ (\%) = \frac{misses\ por\ lectura}{total\ lecturas} \times 100 = \frac{2}{427} \times 100 = 0.468\ \% \tag{11}$$

$$miss\ ratio\ escritura\ (\%) = \frac{misses\ por\ escritura}{total\ escrituras} \times 100 = \frac{9}{132} \times 100 = 6.818\ \% \tag{12}$$

4.2.2 Benchmark de la caché asociativa

Ahora vamos a analizar el rendimiento de la caché asociativa para poder comparar ambas cachés. Estos son los datos que hemos obtenido de esta simulación:

```
# Total accesos:          559
# Total lecturas:         427 | Total escrituras:         132
# Misses lectura:           7 | Misses escritura:          20
# Ciclos espera lectura:          21 | Ciclos espera escritura:          60
```

Figura 40. Datos obtenidos tras la simulación del *benchmark* en la caché asociativa

En este caso, el tiempo total de ejecución ha sido bastante superior, de 6435 ns. Al igual que en la caché de mapeado directo, la penalización por *miss* es de una media de 3 ciclos, aunque como la frecuencia de trabajo es menor supone una mayor penalización temporal. Vamos a calcular el *miss* y el *hit ratio*:

$$miss\ ratio\ (\%) = \frac{total\ misses}{total\ accesos} \times 100 = \frac{27}{559} \times 100 = 4.83\ \% \quad (13)$$

$$hit\ ratio\ (\%) = \frac{total\ hits}{total\ accesos} \times 100 = \frac{559-27}{559} \times 100 = 95.17\ \% \quad (14)$$

$$miss\ ratio\ lectura\ (\%) = \frac{misses\ por\ lectura}{total\ lecturas} \times 100 = \frac{7}{427} \times 100 = 1.64\ \% \quad (15)$$

$$miss\ ratio\ escritura\ (\%) = \frac{misses\ por\ escritura}{total\ escrituras} \times 100 = \frac{20}{132} \times 100 = 15.152\ \% \quad (16)$$

Capítulo 5. Conclusiones y líneas futuras

En el apartado anterior hemos podido comprobar cómo, para la ejecución del *Bubble Sort*, el rendimiento de la caché asociativa es menor que el de la caché de mapeado directo. Esto es debido a que, a pesar de ser un algoritmo con una gran cantidad de bucles y saltos, se accede continuamente a las posiciones de memoria adyacentes, ya que lee una cadena de números y los va ordenando en posiciones contiguas.

Caché	Frecuencia de operación	Tiempo de ejecución	Miss ratio	Penalización por miss	Ciclos espera por escritura	Ciclos espera por lectura
Mapeado directo	122 MHz	4891.3 ns	1.968 %	3 ciclos	27	6
Asociativa	100 MHz	6435 ns	4.83 %	3 ciclos	60	21

Tabla 1. Comparativa del rendimiento de las cachés

Imaginemos que ejecutamos este mismo programa sin utilizar una memoria caché, solamente haciendo uso de la RAM. Si suponemos un tiempo de acceso de 20 ns, aunque sigue siendo rápido para una gran memoria ya que es similar al tiempo de acceso de la SRAM vista en el apartado 3.5, el tiempo total de ejecución hubiera sido de 11180 ns. Vemos cómo con la implementación de cualquiera de las dos cachés este tiempo de ejecución se reduce considerablemente.

Dentro de todo el diseño, cabe destacar la importancia del registro entre el procesador y la caché. Un procesador con una estructura segmentada es capaz de ejecutar una nueva instrucción cada ciclo de reloj y, por ello, es posible que envíe varias peticiones a memoria de forma consecutiva. Ambas cachés son capaces de solventar estas peticiones y aceptar una nueva cada ciclo de reloj (siempre que no se produzca un *miss*), sin afectar así al funcionamiento del procesador.

Ha habido determinadas implementaciones que no se han podido abarcar, pero cabe destacarlas como posibles mejoras de este proyecto:

- Implementación de un *buffer* de escritura, procurando que tenga la suficiente capacidad como para reducir al máximo las paradas por escritura.
- Aumentar el grado de asociatividad de la caché asociativa. Esto habría que estudiarlo detenidamente, ya que como hemos visto en el apartado 2.4.1, para una caché de un tamaño similar a la que hemos implementado un grado de asociatividad mayor de dos no supone prácticamente ninguna mejora.
- Combinar la caché de mapeado directo con cuatro bloques por entrada y la caché asociativa. Aunque, de nuevo habría que estudiarlo detenidamente ya que sería necesaria una memoria de más capacidad, por lo que supondría un mayor coste.

Capítulo 6. Bibliografía

[1] Accellera Organization, Inc. (2004, 13 mayo). SystemVerilog 3.1a Language Reference Manual [PDF]. Recuperado de http://www.ece.uah.edu/~gaede/cpe526/SystemVerilog_3.1a.pdf

[2] DE2-115 Board I/O Pin Assignments [PDF]. (s.f.). Recuperado de http://www.ece.iastate.edu/~alexs/classes/2017_Fall_281/labs/Lab03/DE2-115_PIN_ASSIGNMENTS.pdf

[3] Fletcher, C. W., & UC Berkeley. (2008, 5 septiembre). Verilog: always @ Blocks [PDF]. Recuperado de https://class.ece.uw.edu/371/peckol/doc/Always@.pdf

[4] Integrated Silicon Solution, Inc. (2014, junio). 1M x 16 HIGH-SPEED ASYNCHRONOUS CMOS STATIC RAM WITH 3.3V SUPPLY [PDF]. Recuperado de http://www.issi.com/WW/pdf/61WV102416ALL.pdf

[5] Patterson, D. A., & Hennessy, J. L. (2012). Computer Architecture: A Quantitative Approach (5ª ed.). Ámsterdam, Países Bajos: Elsevier Inc.

[6] Patterson, D. A., & Hennessy, J. L. (2018). Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Ámsterdam, Países Bajos: Elsevier Inc.

[7] Standard Performance Evaluation Corporation. (2005, 1 noviembre). SPEC CPU2000 V1.3. Recuperado de https://www.spec.org/cpu2000/