Proyecto de fin de carrera

Simulación de caches usando una plataforma FPGA

Autor: Eduardo Begué Granged

Director: Javier Resano Ezcaray

Departamento de informática e ingeniería de sistemas

gaZ

Grupo de arquitecturas de Zaragoza

Centro politécnico superior

Ingeniería informática

Junio 2010

ÍNDICE

ÍNDICE DE FIGURAS Y TABLAS

SIMULACIÓN DE CACHES USANDO UNA PLATAFORMA FPGA.

1. RESUMEN

La utilización de dispositivos lógicos programables (FPGAs) en el diseño de hardware va cobrando cada día más importancia. Una de sus aplicaciones consiste en desarrollar plataformas de simulación que produzcan resultados rápidos y con la máxima precisión. Este proyecto fin de carrera ha surgido como respuesta a esta tendencia desarrollando sobre una FPGA un simulador de memorias cache totalmente configurable.

El simulador de memorias cache es un diseño hardware que permite emular el comportamiento de una cache con unas características concretas (se permiten más de 380 tipos diferentes de memoria cache) para obtener información fiable acerca de qué configuración de cache es la más adecuada para una determinada situación.

El hardware desarrollado se ha diseñado utilizando el lenguaje de descripción de hardware VHDL y se ha implementado sobre la FPGA Xilinx Virtex II. El proyecto se ha realizado utilizando variables “genéricas” durante el diseño del hardware. Estas variables se utilizan para describir la cache que se quiere simular definiendo el tamaño, emplazamiento, política de reemplazo, escrituras… Con esta aproximación se consigue que un único módulo VHDL sea capaz de implementar el simulador de cualquiera de las memorias caches posibles. Estos simuladores trabajan con una traza de accesos a memoria y replican la gestión de las direcciones de una memoria cache real, generando estadísticas tras analizar cada acceso. La traza la envía un computador a la FPGA a través del puerto serie, y las estadísticas las devuelve la FPGA por ese mismo puerto. Se ha comprobado que el simulador genera los datos correctos comparándolo con un simulador SW equivalente usando las mismas trazas.

Este proyecto también ofrece una visión global acerca de las ventajas e inconvenientes que ofrece la utilización de hardware programable con respecto a la simulación software. Las ventajas principales serían que se puede ejecutar en lugar de simular, que el tiempo de ejecución menor, que es posible observar ciclo a ciclo el comportamiento de los distintos módulos y que se obtiene información adicional sobre el coste de la implementación y el retardo del sistema.. En cuanto a los inconvenientes, el diseño es bastante más complejo, y en especial la depuración del proyecto. También nos encontramos con las limitaciones de área de la plataforma que impiden simular memorias cache muy grandes.

2. INTRODUCCIÓN

2.1 Motivación

Las FPGAs son dispositivos programables que se pueden utilizar para diseñar y evaluar circuitos hardware complejos. La utilización de las FPGA's como herramienta de desarrollo de plataformas de simulación / emulación va cobrando cada vez más importancia en la actualidad. Poco a poco, estos dispositivos se están convirtiendo en una seria alternativa a la utilización de simuladores software. Las principales ventajas son:

- Plataformas económicas.
- No se simula, sino que se ejecuta un diseño hardware real por lo que se puede obtener mayor precisión.
- La velocidad de ejecución en la FPGA puede ser varios órdenes de magnitud mayor que la simulación software.
- La misma FPGA puede configurarse para realizar distintas simulaciones e incluso puede realizar varias en paralelo si su capacidad se lo permite.
- Puede interaccionar con otros dispositivos en tiempo real.
- Proporciona información sobre el coste de implementación y el retardo del circuito.

Son múltiples las aplicaciones de simulación/emulación en las que se están usando FPGA´s :

- **Predicción de saltos**: La creación de herramientas predictivas que detecten según determinados parámetros si va a tomarse o no un salto en un programa son muy útiles para incrementar la velocidad de ejecución del software. En [ref1] se describe un sistema basado en una FPGA que interacciona en tiempo de ejecución con un procesador evaluando distintas estrategias de predicción.

- **"Network on a chip"**: En [ref2] se describe un sistema basado en FPGA que permite probar distintas topologías de interconexión para un multiprocesador y evaluar los resultados de ejecución.

- **RAMP Project**: Es una macro plataforma con muchas FPGAs que permite simular el comportamiento de multiprocesadores, incluyendo sus sistemas operativos, y ejecutar software sobre ellos [ref3].

Viendo estas aplicaciones y las características de las FPGAs en este proyecto nos hemos planteado utilizarlas para seleccionar la memoria cache más adecuada para un determinado sistema.. La capacidad de configuración de la FPGA nos permite simular el comportamiento de distintas memorias cache simplemente cargando un archivo u otro en la FPGA y su rapidez de ejecución nos proporciona un rendimiento superior al de simuladores software equivalentes.

[ref1]: **"Implementation of a hardware branch-predictor evaluation platform based on FPGAs"**, Enrique Sedano, Daniel Chaver, Javier Resano.5th International Conference on Ph.D. Research in Microelectronics & Electronics (PRIME). 2009

[ref2]: **"NoC Emulation: A Tool and Design Flow for MPSoC"**, Nicolas Genko, David Atienza, Giovanni De Micheli, Luca Benini, *IEEE Circuits and Systems Magazine*, IEEE Press, ISSN: 1531-636X, Vol 7, Nr. 4, pp.42-51, diciembre 2007.

[ref3]: http://ramp.eecs.berkeley.edu/

2.2 ¿Cómo funciona una memoria cache?

La memoria principal de un ordenador es demasiado lenta como para que el procesador acceda a los datos de un modo eficiente. Para solucionar este problema se utilizan las memorias cache.

Una memoria cache es mucho más rápida que la memoria principal aunque también tiene una capacidad de almacenamiento significativamente menor. En este tipo de memorias se saca partido a los principios de **localidad temporal y localidad espacial** para guardar aquellos datos que van a ser accedidos por el usuario con mayor probabilidad.

La jerarquía de memorias de un ordenador actual funciona de la siguiente manera: si se realiza un acceso a un dato que está almacenado en la memoria cache el tiempo empleado en ello es mínimo. Sin embargo, si este dato no se encuentra en cache (**miss**), se traerá de memoria principal el bloque en el que dicho dato está guardado siendo este el peor caso. De este modo, es muy importante utilizar una configuración de memoria cache (tamaño, políticas de escritura, políticas de reemplazo, asociatividad…).tal que el número de misses sea mínimo. Es en este punto donde es de utilidad el simulador de memoria cache presentado en este proyecto fin de carrera.

Por último, es importante añadir que en los ordenadores actuales de altas prestaciones hay diversos niveles de memoria cache para optimizar su funcionamiento. Sin embargo, en este simulador sólo se ha contemplado un único nivel de memoria cache que es el caso más habitual que se encuentra en los sistemas empotrados basados en computador.

2.3 ¿Qué es una FPGA?

Una FPGA (“Field programable gate array”) es un dispositivo lógico programable que permite crear diseños hardware. La estructura interna de una FPGA se basa en una serie de bloques lógicos simples que se pueden interconectar de diferentes maneras. De esta manera, según las conexiones que hagamos crearemos un diseño u otro. La configuración de la FPGA se realiza mediante la utilización de lenguajes de descripción de hardware tales como VHDL, ABEL o Verilog.

Una FPGA se comporta como hardware convencional pero tiene la ventaja de ser totalmente configurable, es decir se puede cambiar su funcionalidad para que implemente diseños distintos. Este proceso de configuración es muy rápido (décimas de segundo, o segundos dependiendo

del protocolo utilizado). De este modo, según carguemos un diseño u otro en el dispositivo podremos im

plementar un comparador, un codificador o, como es nuestro caso, un simulador de caches en cuestión de segundos.

En cuanto a la estructura interna de la FPGA, ésta se compone de tres tipos de estructuras diferentes: los bloques de lógica configurables (CLB's), los Bloques de Entrada/Salida (IOB's) y toda la lógica de interconexión de los elementos funcionales de la FPGA.

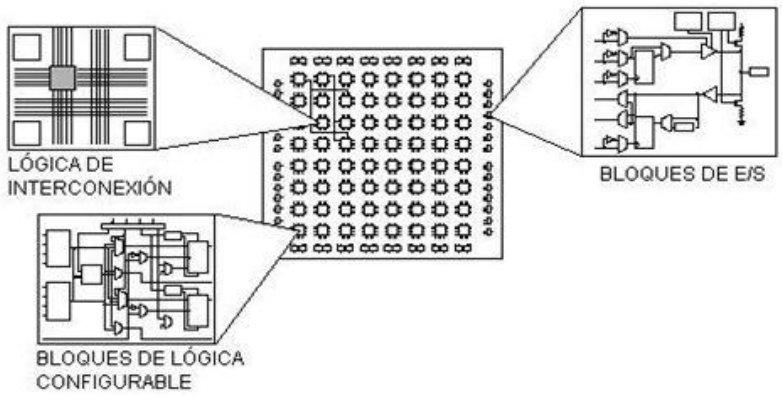

Figura 1. Estructura de una FPGA

Un **CLB** está formado por un conjunto de **celdas** (**LC**'s). Dichos LC's contienen unas tablas llamadas **LUT**'s (Look up tables) mediante las cuales se pueden generar diferentes funciones. Los CLB's tienen además lógica que interconecta los LC's para formar funciones más complejas.

La lógica de interconexión está configurada por los valores guardados en el interior de las celdas de los CLB's. En cuanto a los bloques de E/S, simplemente se encargan de comunicar la FGPA con el exterior.

Para crear un determinado diseño hardware, hay que utilizar lenguajes de descripción de hardware mediante los cuales se crea un fichero que configura la FPGA (ficheros .bit). Mediante estos ficheros se dan valores a las LUT's, se activan las interconexiones necesarias y se utilizan los bloques de E/S adecuados.

Por último hay que resaltar que en las FPGAs hay numerosos chips de memoria que reciben el nombre de **BlockRAMs.** Usando varios de estos bloques se pueden crear memorias RAM de diferentes tamaños donde almacenar datos.

2.4 Objetivos del proyecto

Veamos los objetivos más relevantes que se pretenden alcanzar mediante el desarrollo de este proyecto fin de carrera:

- Aprender a utilizar dispositivos lógicos programables (FPGAs).
- Familiarizarse con el uso de lenguajes de descripción de hardware (VHDL).
- Desarrollar un simulador de caches implementado en hardware que pueda ser de utilidad para determinar qué configuraciones de memoria cache ofrecen mejor ratio misses / tamaño.
- Realizar un diseño que pueda tener utilidad como herramienta docente.
- Comparar los resultados de mi diseño hardware con los obtenidos por un simulador software equivalente (DINERO).
- Sacar conclusiones acerca de las ventajas e inconvenientes del uso de simuladores hardware frente a simuladores software.

3. TECNOLOGÍA UTILIZADA

3.1 Entorno de trabajo: Hardware

El hardware en el que se basa el proyecto es la FPGA modelo Xilinx Virtex II pro. Gracias al programa de colaboración de Xilinx con universidades este modelo se adquiere por trescientos dólares. Esta placa es bastante antigua (2002), pero su precio y su disponibilidad en un gran número de universidades, la hace ideal para este proyecto.

Como se puede ver en el dibujo, la FPGA en sí es una estructura cuadrada de metal bastante pequeña colocada en el centro de una plataforma en la cual la mayor parte de los componentes son puertos de diferentes tipos para comunicarse con el exterior. También hay una ranura para poder añadir una SDRAM de gran tamaño.

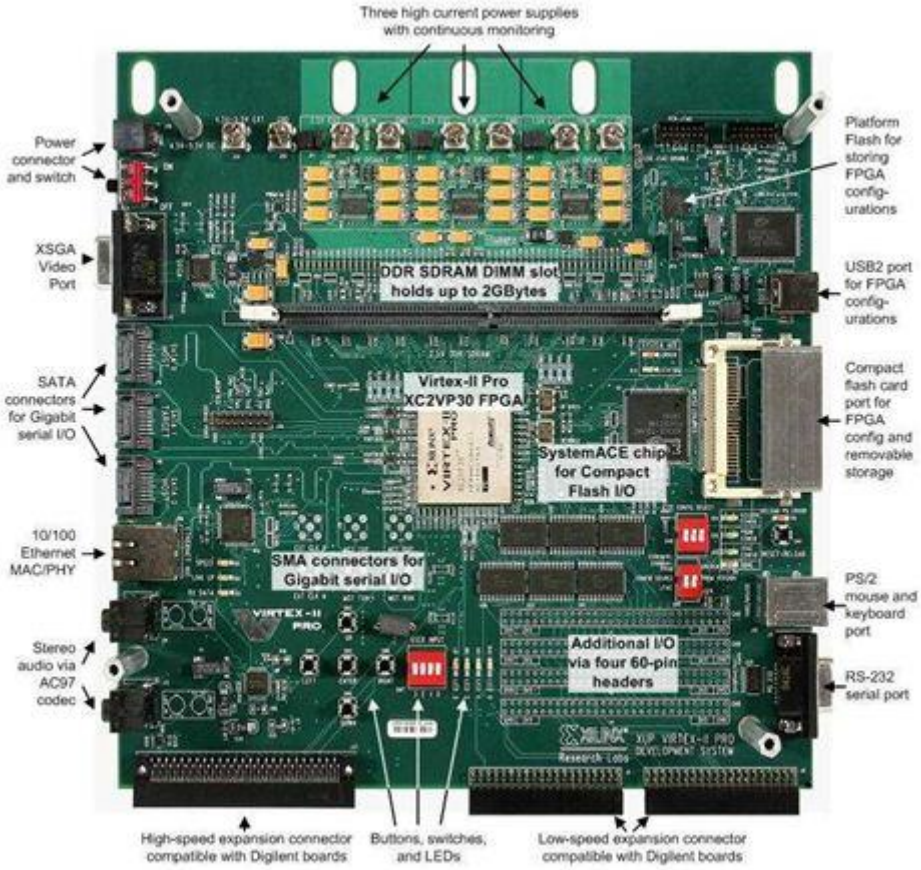

Figura 2. Xilinx Virtex II PRO

En este proyecto se utiliza el puerto USB para configurar la FPGA desde un ordenador, y posteriormente el puerto línea serie (RS 232) para comunicar la FPGA con el ordenador. A través de este puerto el ordenador envía los datos a simular y posteriormente recibe el resultado.

3.2 Entorno de trabajo: Software

Hay cuatro programas diferentes necesarios para realizar este proyecto de fin de carrera. Veámoslo detalladamente:

- **XILINX ISE 10.1**: Es el entorno de trabajo que proporciona la empresa Xilinx. Es un editor que permite diseñar hardware utilizando tanto VHDL como Verilog. Además también nos proporciona una serie de librerías con módulos ya implementados y un compilador que convierte los ficheros escritos en lenguaje de descripción de hardware en ficheros de configuración (.bit) adecuados para la FPGA. Veamos en la siguiente figura cómo es el aspecto de este software:

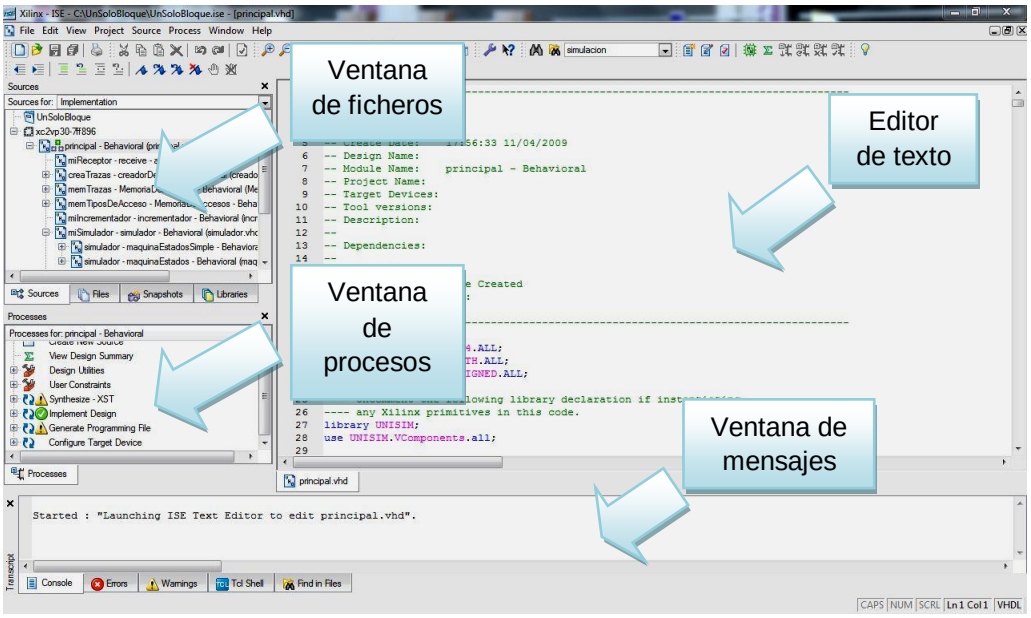

Figura 3. Xilinx ISE 10.1

Hay cuatro partes diferenciadas: el editor de texto, la ventana de procesos, la ventana de ficheros y la ventana de mensajes. En el editor se programa en el lenguaje de descripción hardware (VHDL), en la ventana de ficheros se selecciona el archivo que queremos editar, en la ventana de procesos se pueden controlar las distintas fases de diseño (síntesis, implementación y generación de fichero de configuración). En cuanto a la ventana de mensajes simplemente sirve para comunicarnos Warnings o Errores.

Hay que realizar tres procesos una vez hemos descrito en el editor de texto el diseño hardware. Primero se realiza la **síntesis del diseño** la cual consiste en, a partir del fichero fuente, concretar los diferentes bloques que se deberán crear en la FPGA. En esta fase se hace el "check syntax" que comprueba que el código fuente escrito es sintácticamente correcto, y posteriormente se genera un diseño hardware con componentes genéricos. También se crea el "summary" en el cual podemos ver el porcentaje de recursos utilizados. Posteriormente se pasa a la fase de **implementación** en la cual se da forma al diseño hardware tal y como posteriormente se

creará en la FPGA, para ello se asigna a cada elemento del diseño original un recurso físico de la FPGA, y posteriormente se interconectan todos los elementos entre si. Este proceso puede ser bastante lento para diseños grandes (varios minutos en un procesador reciente). Por último se pasa a la **creación del fichero de configuración** (fichero **.bit**). En esta etapa se traduce el diseño creado en la implementación a una secuencia de unos y ceros que al escribirse en los elementos de configuración de la FPGA crean el circuito diseñado.

Para dar comienzo a los tres pasos del diseño simplemente hay que hacer doble click sobre "**generate programming file**" tal y como se muestra en la siguiente figura:

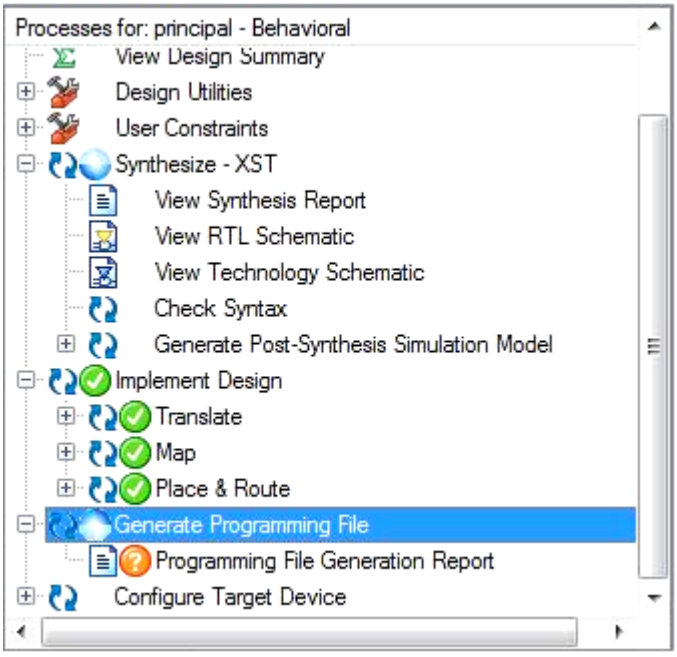

Figura 4. Ventana de procesos

- **MODELSIM 6.2**: Es un potente simulador de circuitos que permite simular ciclo a ciclo los diseños realizados en vhdl. El Modelsim es una herramienta vital a la hora de depurar errores en el diseño hardware. Este programa está asociado al software ISE 10.1 de Xilinx de modo que tras hacer un fichero de test (es decir, un banco de pruebas en el que describimos una secuencia de entradas que queremos comprobar) podemos simularlo en el Modelsim simplemente haciendo doble click sobre "**simulate behavioral model**" tal y como se muestra en la siguiente figura:

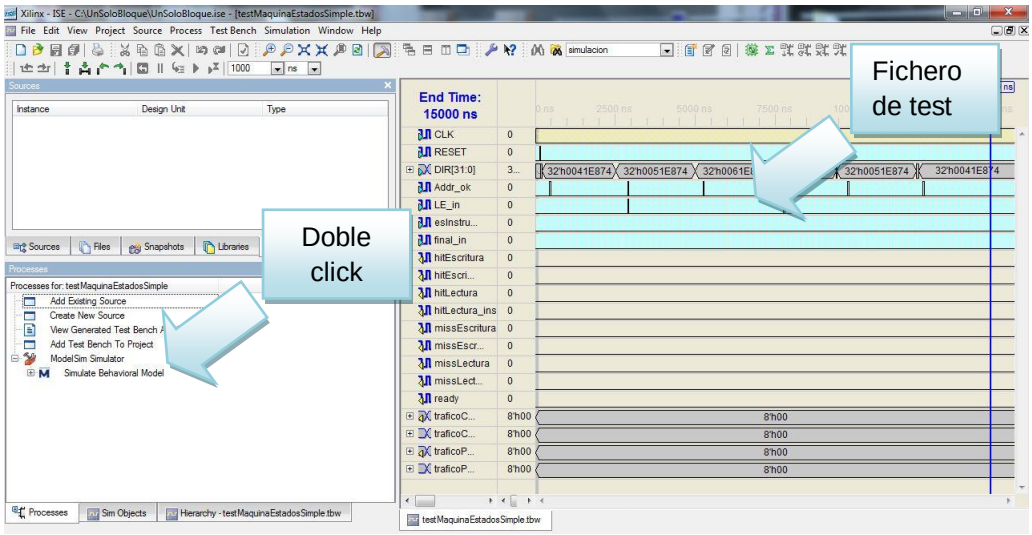

Figura 5. Fichero de test (.tbw)

Al hacer doble click se inicia el modelsim y, una vez seleccionadas las señales que queremos ver, se muestra el valor de estas en un cronograma:

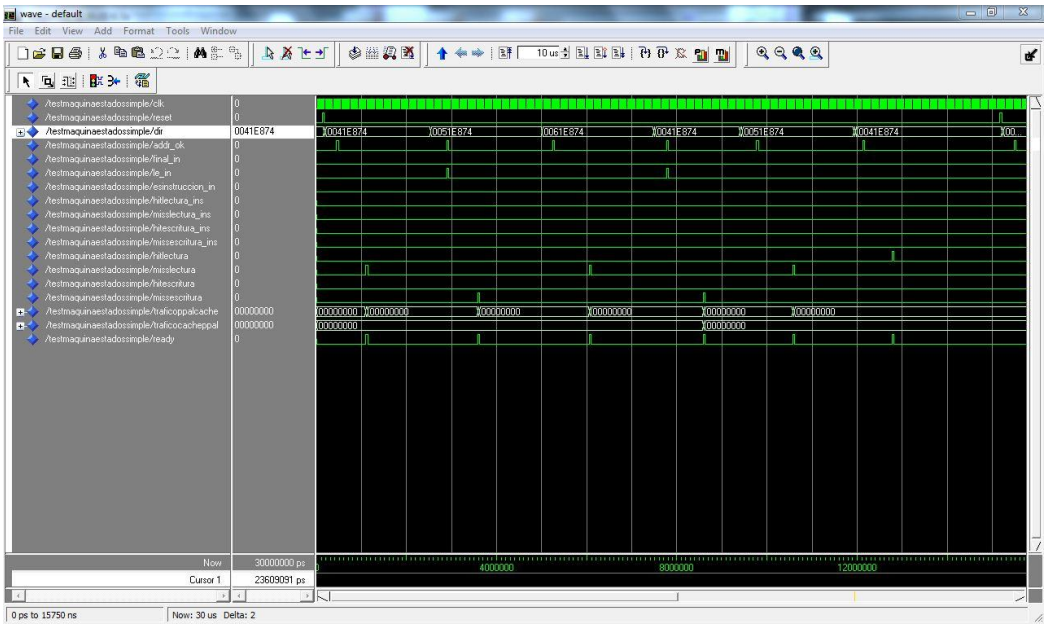

Figura 6. Cronograma del Modelsim

- **IMPACT:** Este programa es muy simple. Lo único que hace es localizar el fichero de configuración (.bit) creado con el ISE de Xilinx y cargarlo en la FPGA. Al finalizar el proceso, el diseño hardware se ha creado correctamente y ya puede utilizarse.

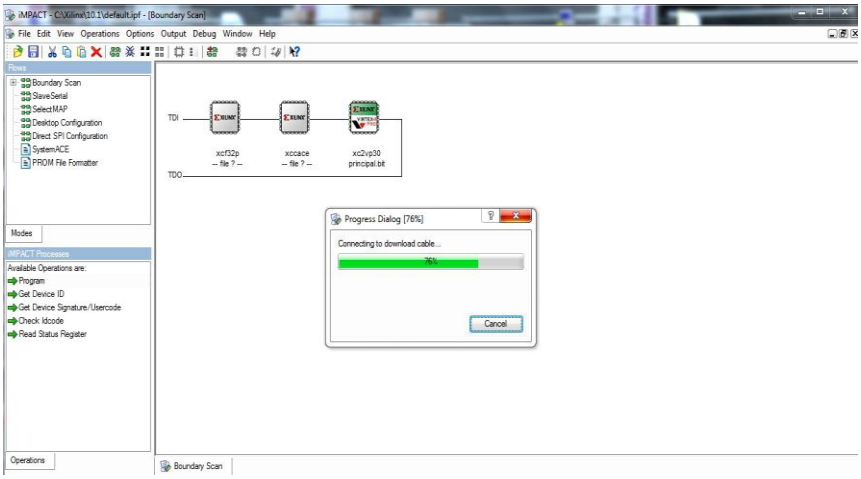

Figura 7. Programa IMPACT

- **REALTERM:** La comunicación entre la FPGA y el ordenador se lleva a cabo mediante el puerto línea serie (**RS 232**). Para utilizar dicha interfaz utilizamos un programa llamado Realterm que se puede descargar de manera gratuita. Usando esta aplicación mandamos las trazas a la FPGA y recibimos los resultados de ésta en formato hexadecimal tal y como se muestra en la figura 8.

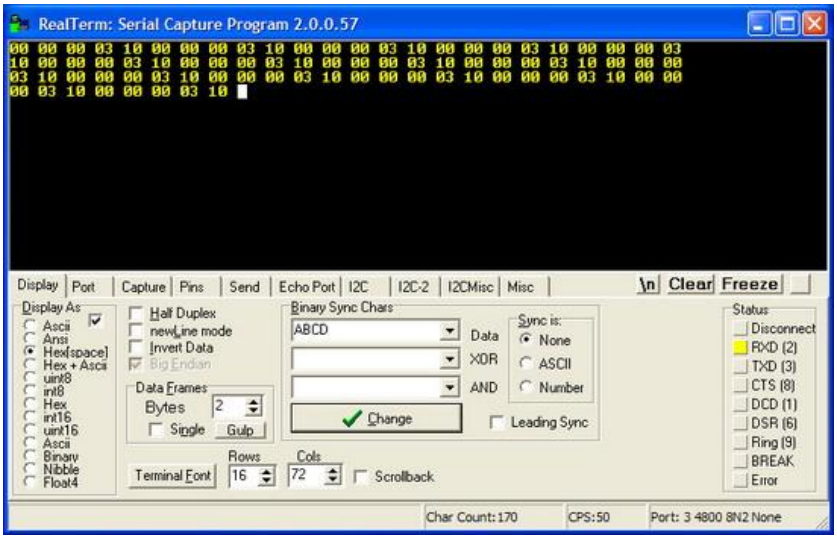

Figura 8. Interfaz de Realterm

- **CONVERSOR:** Se ha tenido que implementar en **lenguaje C** un programa muy sencillo que convierte los caracteres ASCII almacenados en el fichero de trazas en números enteros utilizando la función "**atoi()**". El fichero generado tiene la extensión **.bin** y ya puede ser pasado como entrada al programa Realterm.

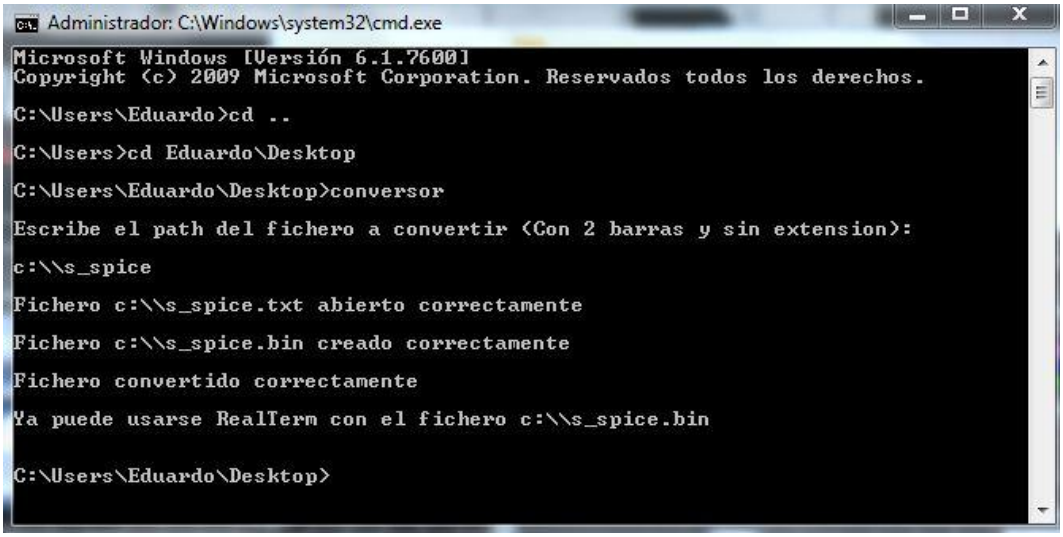

Figura 9. Programa conversor

3.3 Lenguajes de descripción de hardware

3.3.1 ¿Qué es el VHDL?

El lenguaje de descripción de hardware utilizado en este proyecto de fin de carrera es el VHDL. Dicho acrónimo resulta de la combinación de VHSIC (Very High Speed Integrated Circuit) y HDL (Hardware Description Language). El VHDL es, junto con ABEL y Verilog, uno de los lenguajes más utilizados en el campo del diseño hardware.

3.3.2 ¿Cómo se programa en VHDL?

Para usar un lenguaje de descripción de hardware hay que mentalizarse de que es algo muy diferente a la programación software a la que estamos habituados. La sintaxis de VHDL es muy parecida a la usada en lenguajes como ADA o Pascal pero no hay que confundirse. Son conceptos muy diferentes. Cuestiones como el uso de variables o la utilización de sentencias

del tipo for…loop no tienen sentido en el diseño hardware (salvo contadas excepciones).

Un programa hecho en VHDL se divide en cuatro partes diferenciadas:

- Definición de **puertos de entrada / salida** así como de valores **genéricos**: El programa forma un bloque con una serie de puertos que pueden ser de uno o más bits. De esta manera puede utilizarse el bloque implementado como una caja negra en otros programas. También pueden definirse valores genéricos que se suelen usar para concretar el tamaño de los puertos.

- Declaración de **señales**: Como hemos dicho antes, en un HDL no se usan variables. En vez de ello se utilizan señales que no son otra cosa que la representación de un cable de uno o más bits que puede tomar diferentes valores. Si la señal es de un bit se define como un STD_LOGIC o como un STD_LOGIC_VECTOR (n-1 downto 0) si es de n bits.

- Instanciación de **bloques**: Un diseño hardware está formado por un conjunto de bloques lógicos cada uno de los cuales realiza una función. Cada bloque se implementa por separado de forma que en el programa principal se usan como "cajas negras" sin saber qué es lo que pasa en su interior. Así, para crear un determinado sistema lógico podremos utilizar un bloque que sea un AND, otro que sea un CODIFICADOR, etc de forma que no tenemos por qué saber el funcionamiento interno de estos componentes (los podría haber hecho otro programador). Esta característica hace del VHDL un lenguaje muy bien estructurado.

- Definición de la **máquina de estados**: Para interrelacionar las señales de los diferentes bloques instanciados en el programa se usa una máquina de estados. Dicha estructura está formada por dos procesos, uno **síncrono** que se ejecuta con cada ciclo de reloj y que se encarga de cargar el siguiente estado y un proceso **combinacional** mediante el cual se decide cuál va a ser el siguiente estado de la máquina y qué valores van a tomar cada una de las señales. Los procesos en VHDL llevan asociados una "**lista de sensibilidad"** en la cual se indican qué señales van a ser usadas en dicho proceso. Si una señal utilizada en un proceso no se referencia en la lista de sensibilidad, se ignorará el valor de dicha señal y el funcionamiento del hardware será erróneo. De ahí la importancia de este concepto.

La Figura 10 presenta el esquema básico de un programa en VHDL:

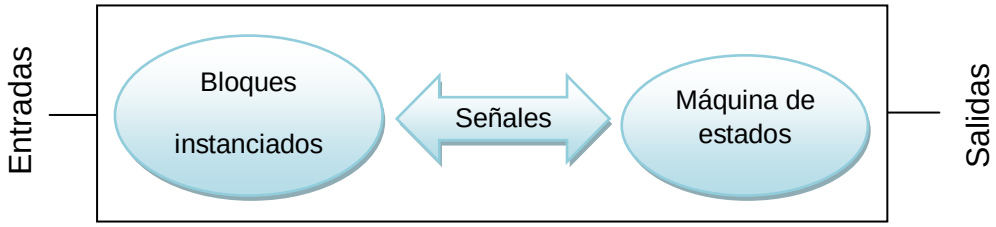

Figura 10. Estructura de programa VHDL

Hay que resaltar otra idea importante a la hora de diseñar con un lenguaje de descripción de hardware. Los procesos definidos en un programa así como los diferentes bloques instanciados **se ejecutan en paralelo**. Al contrario que en un lenguaje de programación software, el escribir algo primero no indica que se vaya a ejecutar primero. El único lugar en el que se utiliza la ejecución secuencial de las distintas sentencias es en el interior de los procesos.

Por último, hay que mencionar a la sentencia "**generate"** debido a su gran utilización en este proyecto.

3.3.3 Sentencia "generate" y uso de "genéricos"

Un bloque diseñado en vhdl no sólo puede tener entradas y salidas. También puede presentar una serie de parámetros configurables denominados **"genéricos"** según los cuales el comportamiento del hardware se puede modificar. Así, por ejemplo, se puede hacer un registro "genérico" que se pueda configurar para guardar datos de 1 bit, 16 bits o 32 bits según le demos uno u otro valor al correspondiente genérico. Veamos cómo se instanciaría un registro genérico para que albergase 32 bits:

registroEjemplo: registro **generic map** (**32**) port map (CLK, E, S);

La utilización de bloques genéricos se complementa con el uso de la sentencia **"generate"**. Mediante esta sentencia podemos crear sistemas lógicos complejos **replicando** un número determinado de veces un mismo bloque dándole diferentes valores a sus genéricos según convenga. Así pues, utilizar "generates" no sólo permite **compactar el código** sino que también da la posibilidad de diseñar **módulos adaptables**, de forma que un mismo módulo vhdl pueda utilizarse para implementar módulos de distinto tamaño, o incluso con distinto comportamiento. La idea es definir una serie de señales genéricas en función de las cuales el comportamiento del módulo variará. Cuando se quiera implementar el módulo se fijarán los valores que se deseen en estas

señales y se obtendrá una de las múltiples configuraciones posibles del módulo original. Veamos un ejemplo de la utilización de la sentencia “generate” y de los bloques genéricos:

```
genComp: if p /= 0 generate

      gen9: for j in (2**p)-1 downto 0 generate


            miComparador: comparadorNbits

                        generic map (d-m+p+2)
                        port map ( entradaComparador(j),
                        TAG_MAS_BITS, entradaOR(j));

      end generate gen9;

end generate genComp;


genCompZero: if p = 0 generate

      miComparadorZero: comparadorNbits

                        generic map (d-m+p+2)
                        port map ( salidaMem, TAG_MAS_BITS, igualesIn);

end generate genCompZero;
```

En este ejemplo, si p es distinto de cero se instancian 2^P comparadores utilizando una sentencia del tipo **for…generate**. En cambio, si p es igual a cero se instancia un único comparador.

En este proyecto se han utilizado variables genéricas en todos los diseños realizados en VHDL para permitir que el mismo código vhdl pueda utilizarse para implementar todos los simuladores de caches desarrollado. Esta aproximación complica el diseño del hardware y la depuración del proyecto, pero es imprescindible para evitar que el aumento exponencial de archivos vhdl.

3.3.4 Dificultades que presenta utilizar VHDL

Veamos las principales dificultades que presenta la utilización de un lenguaje de descripción de hardware como VHDL:

- **Adquirir la mentalidad propia del diseño hardware:** Esto ha sido lo más difícil del proceso de adaptación a los lenguajes de

programación hardware. Hay muchas estructuras típicas del software que no tienen sentido a la hora de crear hardware (sentencias for…loop, while…loop, variables, etc). También hay que acostumbrarse a usar señales y a concebirlas como un “cable físico” que interconecta bloques instanciados. Y por último asumir que todos los bloques en el diseño se están ejecutando en paralelo.

- **Estructura nueva de los programas:** Se calculan los resultados de los procesos y los bloques en paralelo de modo que hay que olvidarse del orden de las sentencias propio del software. La estructura de los programas está compuesta siempre por la instanciación de los bloques y la creación de una máquina de estados que los interconecta.

- **Evitar los “Flip Flop Latches”: Debemos asignar siempre un valor a** todas las señales. Si en un estado determinado dejamos una señal “al aire” por que no la estamos utilizando el entorno de Xilinx reacciona creando un Flip Flop donde guarda el valor de dicha señal, dado que interpreta que si no se le asigna valor, no es porque no se esté usando, sino porque se quiere preservar el valor asignado previamente. Esta situación puede causar resultados inesperados y debe evitarse.

- **Comprensión de conceptos totalmente nuevos:** Por ejemplo, la creación de **bloques genéricos** y el uso de la sentencia **“generate”** al instanciarlos es uno de los pilares básicos del diseño hardware. En un principio no es fácil acostumbrarse a utilizar esta técnica de descripción de hardware.

- **Dificultad en la depuración de errores:** Se dedica una cantidad de tiempo muy grande a la depuración de errores. La complejidad que adquieren las máquinas de estados y el elevado número de señales que se manejan incrementan mucho la posibilidad de cometer errores. La transparencia del diseño hardware es realmente buena gracias al simulador Modelsim pero puede haber errores que se den en la FPGA y que no se vean en el simulador. Éstos son muy costosos de encontrar y solucionar, dado que la FPGA es para nosotros una caja negra, y solo podemos acceder a los puertos de entrada y de salida. Para depurar el funcionamiento en la FPGA se ha tenido que modificar el diseño para que envíe información de depuración a través de los puertos de salida.

4. DESARROLLO DEL PROYECTO

4.1 Visión global del proyecto

En este proyecto fin de carrera se ha creado un **simulador de memorias cache**. Según cómo configuremos dicho simulador podemos recrear el comportamiento de más de **380** tipos diferentes de memoria cache. Es importante recalcar que en este proyecto no se crean memorias cache sino que sólo **se simulan**. Para ello se implementa todo el hardware necesario para la gestión de las direcciones, pero no se incluye el almacenamiento de los datos. En otras palabras, el diseño hardware elaborado **no almacena datos** sino que simplemente guarda constancia de a qué datos se ha querido acceder y de si dichos datos estarían en la memoria (hit) o si bien habría que traerlos de memoria principal (miss).

Como **entrada** se le pasan al simulador una secuencia de accesos a memoria real generada por un determinado programa de prueba en tiempo de ejecución. Dichos accesos vienen dados en forma de direcciones de memoria de 32 bits las cuales, junto con la información acerca del tipo de acceso (lectura, escritura, acceso a dato, acceso a instrucción) forman una **"traza"**. Los conjuntos de trazas se almacenan en ficheros de texto.

Como **salida**, el simulador devuelve una serie de **estadísticas** (número de misses, tráfico de datos, recursos utilizados…) que nos permitirán determinar **qué configuración de cache es la óptima** para una situación concreta. Veamos la estructura básica del proyecto en un sencillo esquema:

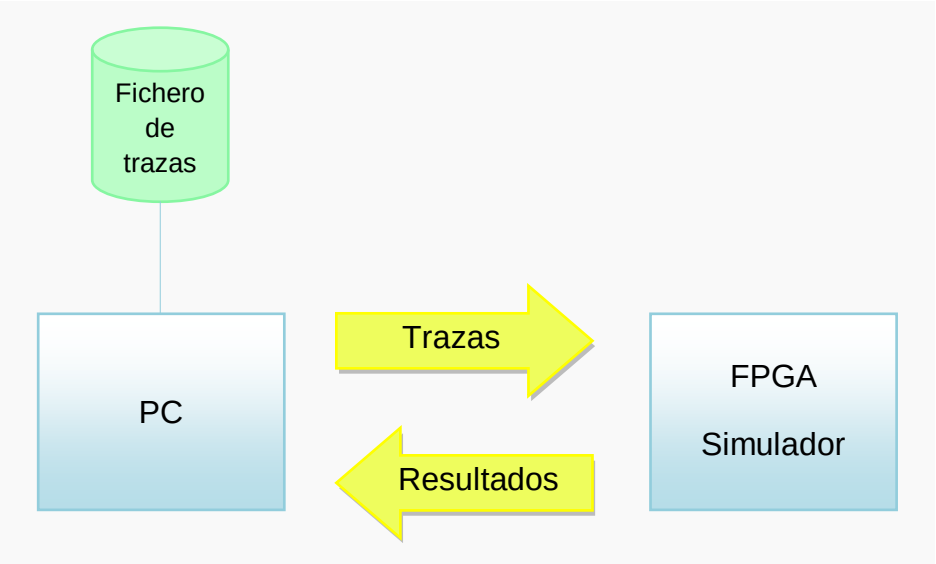

Figura 11. Esquema general

Como podemos ver en la figura 11, las trazas se almacenan en un fichero guardado en el PC. Por otro lado, el simulador configurado con una de las 380 combinaciones de parámetros posibles está cargado en la FPGA listo ya para usarse. A través del puerto línea serie (RS 232) se envían las trazas del ordenador a la FPGA y, mediante el mismo interfaz, el simulador devuelve los resultados (número de misses + otras estadísticas).

4.2 Opciones de simulación

Hay más de **380 configuraciones posibles** del simulador de caches. En el siguiente cuadro se muestran los diferentes **genéricos** que determinan el tipo de cache simulada:

<table>
<tr><th></th><th>Tamaño de memoria</th><th>Tamaño de bloque</th><th>Asociatividad ($S = 2^P$)</th><th>Política de reemplazo</th><th>Política de escritura</th><th>Tipo de memoria</th></tr>
<tr><td rowspan="3">Cache de datos</td><td rowspan="3">m = …</td><td rowspan="3">b = 1,2,3…</td><td rowspan="3">p = 0, 1, 2, 3</td><td rowspan="2">LRU</td><td>CB</td><td rowspan="2">Unificada</td></tr>
<tr><td rowspan="2">WT_con</td></tr>
<tr><td>Aleatorio</td><td>WT_sin</td><td>Disociada</td></tr>
<tr><td rowspan="2">Cache de instrucciones</td><td rowspan="2">m_i = …</td><td rowspan="2">b_i = 1,2,3…</td><td rowspan="2">p_i = 0, 1, 2, 3</td><td>LRU_i</td><td rowspan="2">X</td><td rowspan="2">Disociada</td></tr>
<tr><td>Aleatorio_i</td></tr>
</table>

Tabla 1. Opciones de simulación

Las abreviaturas WT_con, WT_sin y CB significan “Write Trough con asignación en escritura”, “Write Trough sin asignación en escritura” y “Copy Back” respectivamente. La política Write Trough consiste en que cada vez que hay que escribir en un bloque un dato nuevo, no sólo se guarda dicho dato en la cache sino que también se hace en memoria principal. La diferencia entre **WT_con** y **WT_sin** consiste en que en la primera, cuando hay un fallo bien en escritura o bien en lectura, se trae el bloque de memoria principal a cache. (Siguiendo la terminología estudiada en la asignatura de Diseño de Arquitecturas esta opción de simulación aplica una técnida de "write-through + write allocate + fetch on write miss") En cambio, en WT_sin sólo se trae el bloque a cache cuando el fallo es en lectura. Si el fallo es en escritura se escribe el dato nuevo en memoria principal directamente sin copiar ningún bloque en cache (en este caso sería "write-through + write allocate + no fetch on write miss"). En cuanto a la política de escritura **CB**, consiste en que cuando hay un acceso en escritura sólo se escribe el nuevo dato en cache pasando

éste al estado de **"bloque sucio"** (el **dirty bit** guardado junto al TAG se pone a 1). Cuando un bloque sucio se va a reemplazar primero se escribe dicho bloque entero en memoria principal (este caso sería "copy-back + write allocate + fetch on write miss).

En cuanto a la **asociatividad** de la cache (S) es un parámetro que indica el número de bloques que hay en un mismo conjunto. De este modo, si la asociatividad es S = 2 (p = 1) la cache estará dividida en conjuntos formados por dos contenedores. Así, cuando accedemos a un bloque, sacamos de la dirección el campo "número de conjunto". Con dicho campo ya podemos localizar el conjunto donde debería estar el bloque. Una vez estamos en el conjunto ya sólo hace falta comparar uno por uno el TAG de cada bloque del conjunto con el TAG del bloque buscado. Cuando la asociatividad es igual al número de bloques de la cache el bloque buscado puede estar en cualquier lugar y la cache recibe el nombre de **"totalmente asociativa"**. Dicha cache es totalmente **inviable** a la hora de ser implementada debido a la gran cantidad de hardware necesaria para llevarla a cabo. Si la cache es 1-asociativa (p = 0) significa que el bloque buscado sólo puede estar en un lugar. Éste es el caso más sencillo (se simplifica mucho la ruta de datos) y la memoria recibe el nombre de **"cache de mapeo directo"**.

Para comprender mejor las opciones anteriores veamos un ejemplo. Queremos crear un simulador con cache de datos de tamaño 32K (2^{15} Bytes) y cache de instrucciones con tamaño 16 K (2^{14} Bytes), bloques de 16 Bytes (2^4 Bytes), ambas caches de asociatividad 2, política de reemplazo LRU para ambas caches y política de escritura Copy Back. Habría que dar los siguientes valores a la *tabla 1*:

	Tamaño de memoria	Tamaño de Bloque	Asociatividad ($S = 2^P$)	Política de reemplazo	Política de escritura	Tipo de memoria
Cache de datos	m = 15	b = 4	p = 1 (S = 2)	LRU = 1	CB = 1	Unificada = 0
Cache de instrucciones	m_i = 14	b_i = 4	p_i = 1 (S= 2)	LRU_i = 1	X	

Tabla 2. Ejemplo de opciones de simulación

4.3 Diseño del proyecto

4.3.1 Esquema detallado del simulador

Vamos a analizar el esquema del simulador que se muestra en la siguiente página (*figura 12*).

Cada uno de los cuadrados del esquema representan los principales módulos hardware instanciados, el trapecio rosa es el puerto RS 232 y el círculo naranja es la máquina de estados.

Veamos por separado la función de cada bloque del simulador:

- **Receive:** Módulo utilizado para recibir datos del ordenador de Byte en Byte a través del puerto línea serie.
- **CreaTrazas:** Sirve para formar una traza de 32 bits a partir de cuatro Bytes recibidos por el módulo Receive.
- **Memoria de trazas:** Es una memoria donde se guardan las direcciones de 32 bits transmitidas desde el ordenador a la FPGA.
- **Memoria de tipos de acceso:** Es una memoria donde se guardan los tipos de acceso (**0** para lectura de datos, **1** para escritura de datos, **2** para lectura de instrucciones) transmitidos desde el ordenador a la FPGA.
- **Simulador de una traza:** Es el componente principal del esquema. Se encarga de analizar una traza y ver si es un acierto o un fallo sacando una serie de estadísticas como resultado.
- **ContadorHitMiss:** Módulo que lleva la cuenta total de todas las estadísticas suministradas por el simulador de una traza.
- **Transmit:** Módulo utilizado para transmitir al ordenador de Byte en Byte a través del puerto línea serie las estadísticas finales proporcionadas por el bloque ContadorHitMiss.

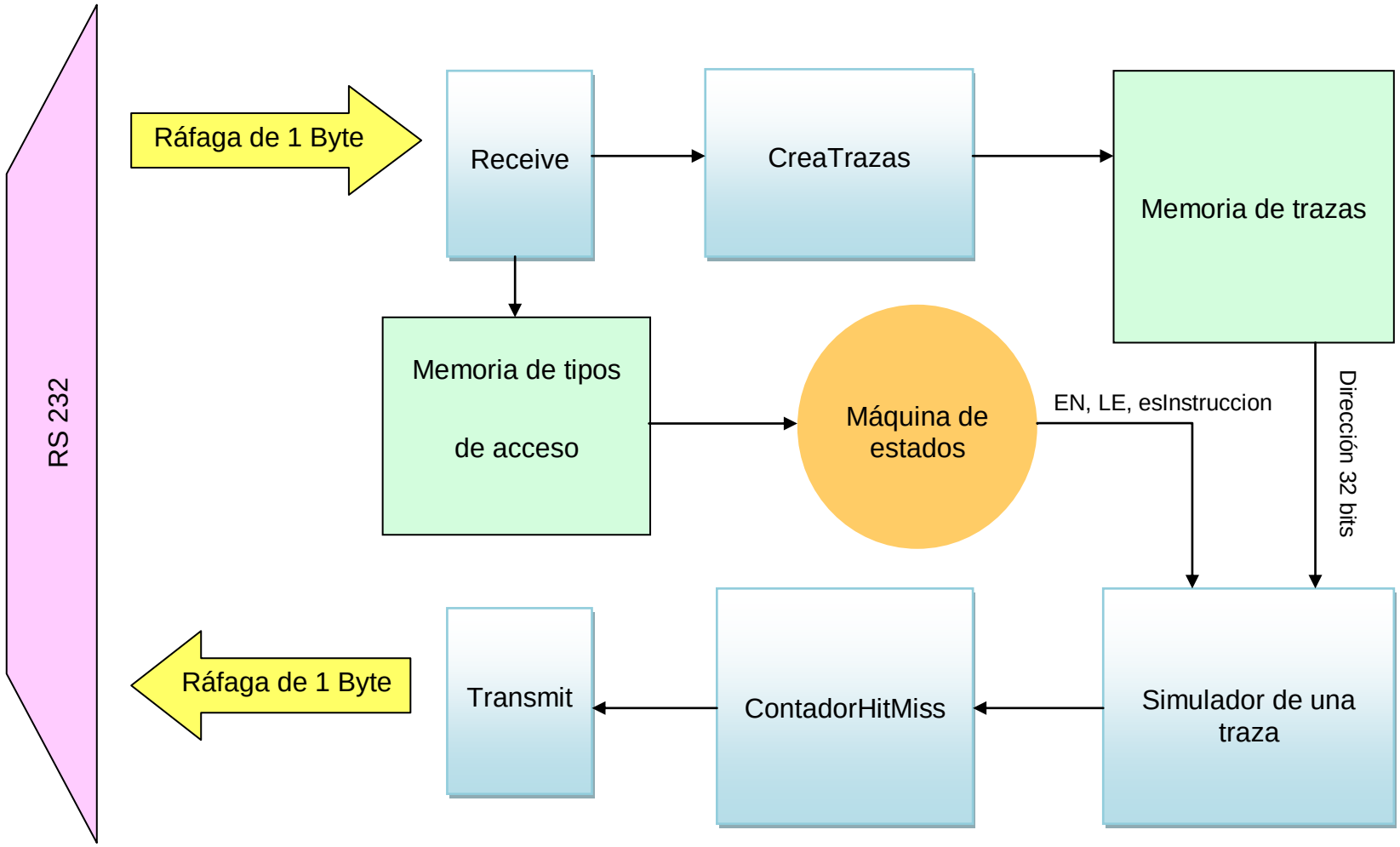

Figura 12. Esquema del simulador

El simulador del esquema anterior funciona siguiendo tres etapas diferenciadas (*figura 12*):

- **Recepción y almacenamiento de trazas**: Usando el módulo **"receive"** se reciben ráfagas de un Byte. El primer Byte corresponde con el tipo de acceso y se guarda en la **memoria de tipos de acceso**. Las cuatro ráfagas siguientes son la dirección de 32 bits. Se une la secuencia de Bytes en un único vector de 32 bits gracias al módulo **"CreaTrazas"** y se guarda la traza completa en la **memoria de trazas**.

- **Simulación de trazas**: Una vez cargadas la memoria de trazas y la memoria de tipos de acceso, se procede a la simulación de las trazas de una en una. Se lee una traza, se lee el tipo de acceso asociado a ella y comienza la simulación. Los resultados obtenidos van a un módulo llamado **"contadorHitMiss"** que lleva la cuenta de las estadísticas.

- **Transmisión de los resultados**: Una vez finalizada la simulación se transmiten por línea serie las salidas de "**contadorHitMiss**" que son los resultados de la simulación (número de misses, etc)

4.3.2 Esquema detallado del simulador de una traza

El bloque encargado de simular una traza es el componente principal del proyecto. Primero vamos a ver las entradas y salidas de este bloque así como los diferentes genéricos:

- Genéricos:
  - **d**: Número de bits de la traza (por defecto 32 bits).
  - **b** y **b_i**: Número de bits del Byte en bloque (de la cache de datos y la cache de instrucciones).
  - **p** y **p_i**: Asociatividad de la cache de datos y la de instrucciones respectivamente.
  - **m** y **m_**i: 2^m y 2^{m_i} son el tamaño de la cache de datos y la cache de instrucciones respectivamente.

- **LRU** y **LRU_i**: Si valen 1 se aplica la política de reemplazo LRU a ambas caches. Si no, se usa la política por defecto (reemplazo de bloque aleatorio).
- **CB, WT_con, WT_sin:** Se aplica aquella política de escritura que tiene valor uno.
- **Unificada:** Si está a 1 indica que sólo hay una única cache. Si vale 0 la cache está dividida en cache de instrucciones y cache de datos.

- Entradas:
  - **CLK**: Señal de reloj.
  - **RESET:** Si vale uno se resetea el simulador.
  - **ENABLE:** Cuando vale uno comienza la simulación.
  - **DIR:** Dirección de d bits.
  - **final:** Si vale uno, ya se han ejecutado todas las trazas y comienza la operación de **"flusheado de la memoria"** mediante la cual se escriben en memoria principal los bloques sucios guardados en cache.
  - **LE:** Indican si el acceso a la cache es escritura o lectura (si vale 1 es escritura, si vale 0 es lectura).
  - **EsInstruccion:** Indica si el acceso es a un dato o a una instrucción.

- Salidas:
  - **Diferentes estadísticas:** número de misses, hits, etc.
  - **Ready:** Vale uno cuando la simulación ha concluido.

Una vez conocidos los parámetros del simulador pasaremos a ver un esquema detallado del mismo (figura 13):

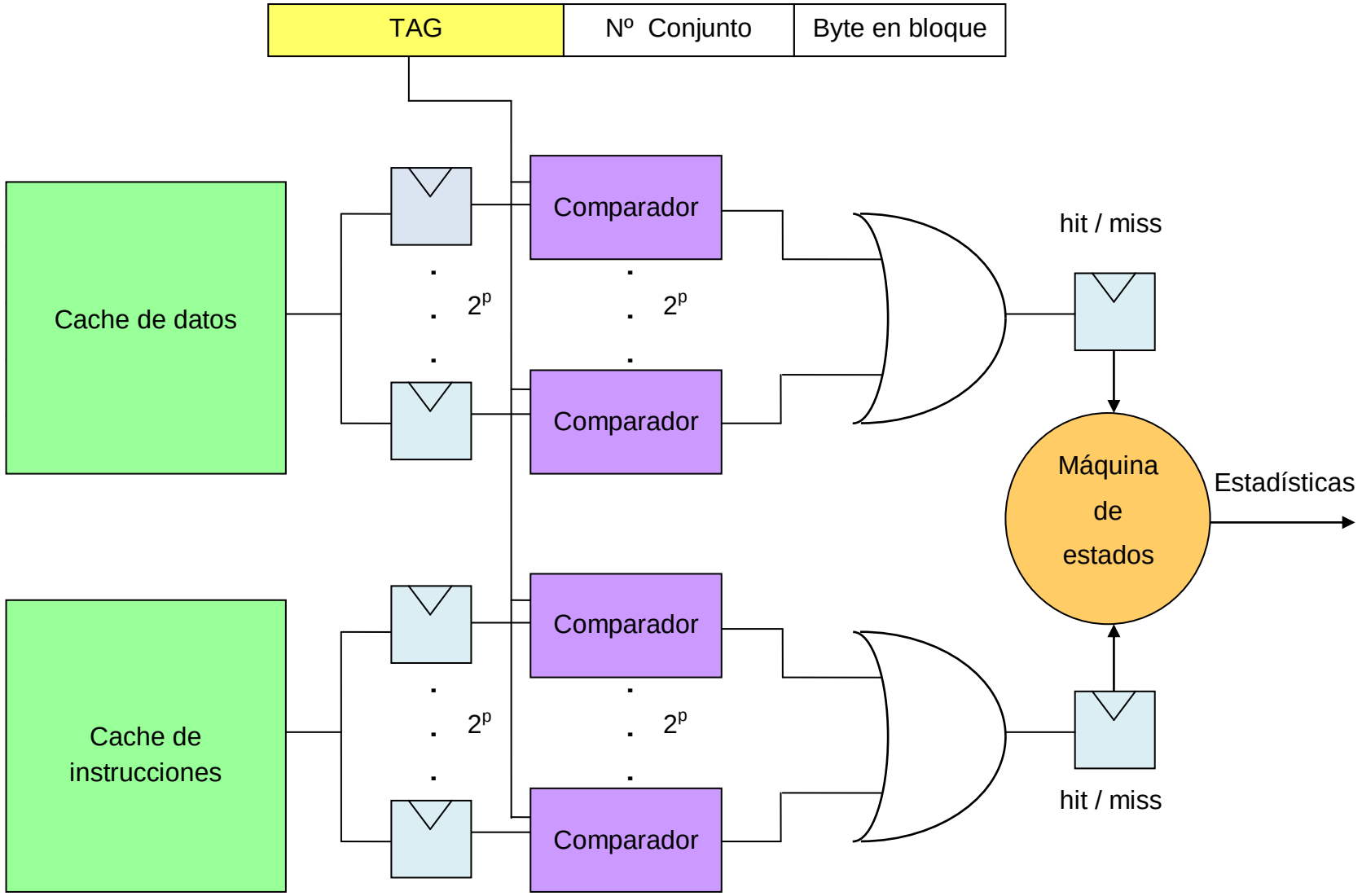

Figura 13. Esquema del simulador de una traza

Una vez hemos visto los diferentes puertos del simulador vamos a analizar el funcionamiento del simulador de una traza (ver *figura 13*).

El primer paso de la simulación consiste en averiguar si el bloque al que queremos acceder se encuentra en la cache (hit) o si bien hay que traerlo de memoria principal (miss). Para ello, se han de leer todos los bloques que forman el conjunto en el que estaría el bloque a acceder. Se ha optado por realizar una **lectura secuencial**. Para ello, se crean tantos registros del tamaño del TAG como el valor de la asociatividad ($S = 2^P$). De este modo, si la asociatividad es dos, se lee de la cache (bien de la de datos o de la de instrucciones) la etiqueta (TAG) del primer bloque y se guarda en el primer registro. Luego se lee la etiqueta del segundo bloque y se guarda en el segundo registro. Ahora ya se tienen en los registros los TAGs de los bloques que forman el conjunto. Ya solo queda compararlos con el TAG del bloque al que queremos acceder para ver si se produce un hit o no. Para ello se instancian tantos comparadores como registros ($S = 2^P$) y se realiza la comparación en paralelo. De este modo, si alguno de estos comparadores devuelve un uno, significa que se ha producido un hit. Es por ello que se añade un OR que tiene por entrada la salida de los comparadores. El resultado obtenido se guarda en un registro que será leído por la máquina de estados actuando en consecuencia (según la política de reemplazo, política de escritura, etc).

4.3.3 Ficheros de trazas

Cuando hablamos de trazas hacemos referencia a las entradas que se les pasa al simulador implementado en la FPGA. El formato de éstas se ha diseñado de tal modo que sean totalmente **compatibles** con el simulador **DINERO** utilizado en la asignatura “Diseño de Arquitecturas”. De este modo, un mismo fichero de trazas podrá ser utilizado por el simulador hardware y el simulador software pudiendo comprobar el correcto funcionamiento de nuestro proyecto de forma rápida y sencilla. En la figura 14 se muestra la estructura de una traza.

Figura 14. Formato de trazas

El tipo de acceso es un entero que puede tomar los valores **0** (lectura de dato), **1** (escritura de dato), **2** (lectura de instrucción). El resto de la traza se corresponde con la dirección de memoria a la que se quiere acceder y se escribe en el fichero mediante ocho cifras hexadecimales (32 bits). En los benchmarks utilizados por DINERO las direcciones de acceso a instrucciones (tipo de acceso 2) sólo tienen seis cifras hexadecimales ya que se obvian los ceros a la izquierda (las direcciones referenciaban zonas bajas de la memoria de instrucciones). Esto se ve claramente en la figura 15 dónde los accesos a instrucciones utilizan direcciones más cortas.

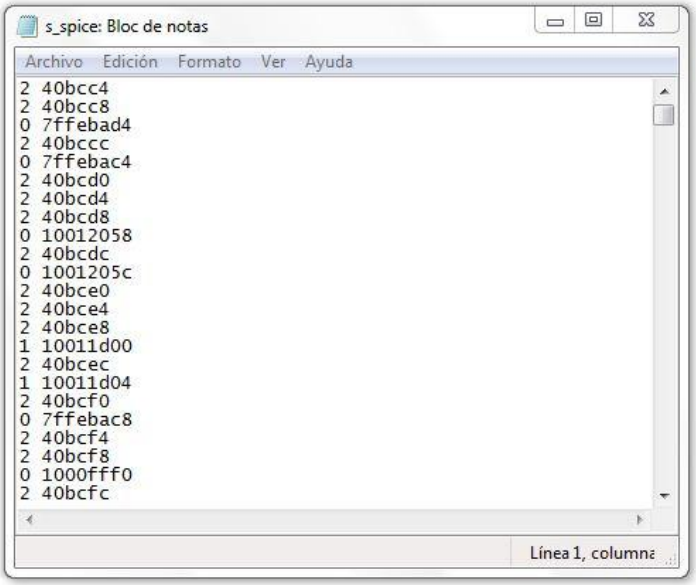

Figura 15. Benchmark s_spice.din

Dado que el simulador requiere direcciones de 32 bits tendremos que añadir los ceros omitidos. De esto se encarga el programa **conversor** explicado detalladamente en la sección 3.2 (Entorno de trabajo: software) el cual además convierte los caracteres ASCII en números en binario. Como ya dijimos antes, el fichero de texto resultante de la conversión adquiere la extensión **.bin** y ya puede ser usado por el programa RealTerm.

4.3.4 Comunicación usando puerto RS 232

La comunicación entre el ordenador y la FPGA se realiza mediante el puerto línea serie (RS 232). El dispositivo programable tiene puerto línea serie pero el portátil utilizado no. Es por ello que se ha tenido que usar un **adaptador USB – RS 232**. Para cargar el diseño hardware en la FPGA se utiliza el programa IMPACT y para enviar las trazas y recibir los resultados se usa el programa RealTerm.

Para que el simulador cargado en la FPGA reciba datos de Byte en Byte se instancia el módulo **"Receive"**. Para transmitir los resultados de Byte en Byte se usa el bloque **"Transmit"**. Por último, para sincronizar la recepción y transmisión de datos usando el puerto línea serie se configura el programa Realterm a 115200 baudios.

4.3.5 Esquema detallado de la memoria cache simulada

Ahora vamos a analizar detalladamente la estructura interna de la cache de instrucciones y la cache de datos que se acaban de mostrar en la *figura 15* (ambas son el mismo bloque). En estas memorias se guardan los **TAG's** (etiquetas que identifican los bloques de datos) junto con un **bit de validez** y un **"dirty bit"** (en el caso de política copy back). El bit de validez sirve para saber que la entrada de la memoria tiene un valor útil y el dirty bit sirve para saber si se ha realizado alguna escritura en el bloque de datos referenciado por el TAG.

Como se aprecia en la *figura 16*, la memoria cache está formada por un conjunto de **blockRAMs**. Como ya explicamos en el apartado 2.3, una blockRAM es una memoria construida usando los chips de memoria RAM que hay en la FPGA. Cada blockRAM utilizada tiene 2^{10} entradas de 16 bits cada una.

Si el TAG a guardar es mayor de 16 bits, instanciaremos tantas blockRAMs como sea necesario. Haremos lo mismo con el número de entradas. Si hacen falta más de 2^{10} entradas en la cache se instanciaran tantas blockRAMs como haga falta. A la hora de leer la memoria se elegirá mediante un **multiplexor** la salida de la blockRAM adecuada.

Como se ve en la figura 16, a partir de los valores de los genéricos **d** (número de bits de la dirección), **m** (2^m = tamaño de la cache), **p** (2^p = asociatividad) y **b** (2^b Bytes por bloque) se descompone la dirección para extraer los campos necesarios (TAG, control del multiplexor y dirección).

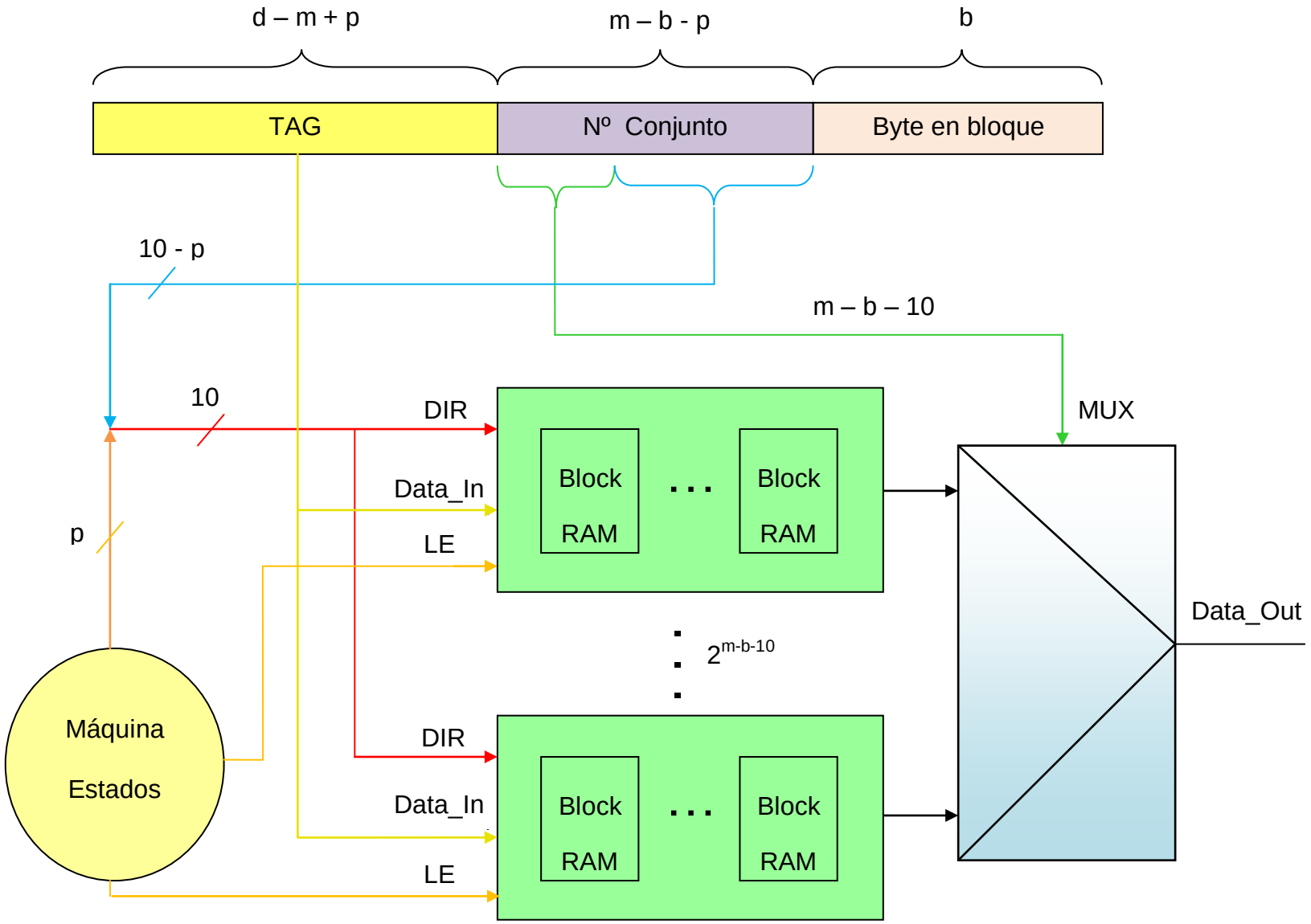

Figura 16. Esquema de memoria cache

4.3.6 Diseño de la política LRU.

Una de las tareas de mayor complejidad realizadas a la hora de desarrollar el simulador de caches ha sido diseñar la política de reemplazo LRU. Veamos cómo se ha llevado a cabo:

La política LRU ("Least Recently Used") consiste en, ante un acceso que ha resultado ser miss, reemplazar el bloque que lleva más tiempo sin utilizarse de entre todos los que pertenecen al conjunto. Para ello hay que tener una **"memoria de edades"** en la que se guarde la edad de cada uno de los bloques que están en la cache. El último bloque en ser accedido tendrá **edad 0** mientras que el que ha sido usado hace más tiempo tendrá edad $\mathbf{2^P - 1}$ siendo 2^P la asociatividad de la cache. Ésta es una forma de aprovechar el **"principio de localidad temporal"** el cual dice que hay muchas probabilidades de que un dato accedido sea utilizado otra vez en un breve periodo de tiempo. De esta manera, si mantenemos en cache los bloques usados hace poco disminuiremos el número de misses y por tanto evitaremos retardos innecesarios.

En la simulación de cada traza usando LRU habrá pues que hacer dos operaciones. Primero encontrar el bloque del conjunto accedido que tiene mayor edad y posteriormente actualizar la edad de los bloques de dicho conjunto (se incrementan en uno las edades de los registros que son menores que la edad del bloque accedido). De este modo, necesitamos leer de la memoria de edades las edades de los bloques que forman el conjunto. Dichas lecturas las haremos **en paralelo** con las lecturas de la cache de datos / instrucciones para ahorrar ciclos. Una vez tenemos los bloques leídos ya podremos averiguar cuál es la posición del bloque de mayor edad. El hardware necesario para crear la memoria de edades consiste en dos módulos de memoria (uno para instrucciones y otro para datos) y 2^P registros a la salida de cada memoria para almacenar las edades de los bloques del conjunto, y una máquina de estados que se encarga de calcular la posición del bloque de más edad (un valor entre 0 y 2^P-1). Así pues el hardware empleado en las dos memorias de edades es análogo al mostrado en la parte izquierda de la *figura 16*. Veamos el funcionamiento de la política LRU con asociatividad 4:

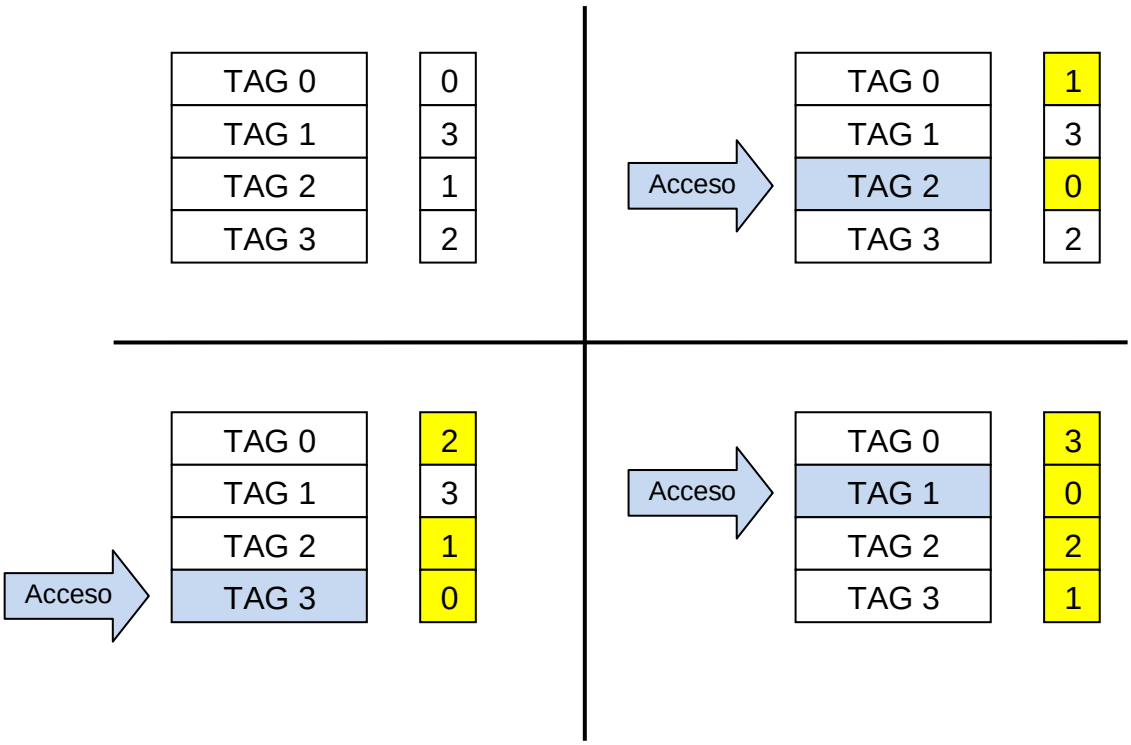

Figura 17. Funcionamiento de LRU

5. RESULTADOS

Como ya se ha comentado anteriormente, con este simulador se puede reproducir el comportamiento de más de 380 tipos de memoria cache diferentes. Ante tantas configuraciones posibles nos vamos a limitar a mostrar algunos resultados ilustrativos, utilizando las trazas de cuatro benchmarks: spice, s_spice, tex, cc1. Debido a las limitaciones de espacio en la FPGA sólo se han simulado X accesos. A continuación se presentan una serie de gráficas y se analizan los resultados en profundidad para obtener las pertinentes conclusiones.

5.1 Misses VS asociatividad

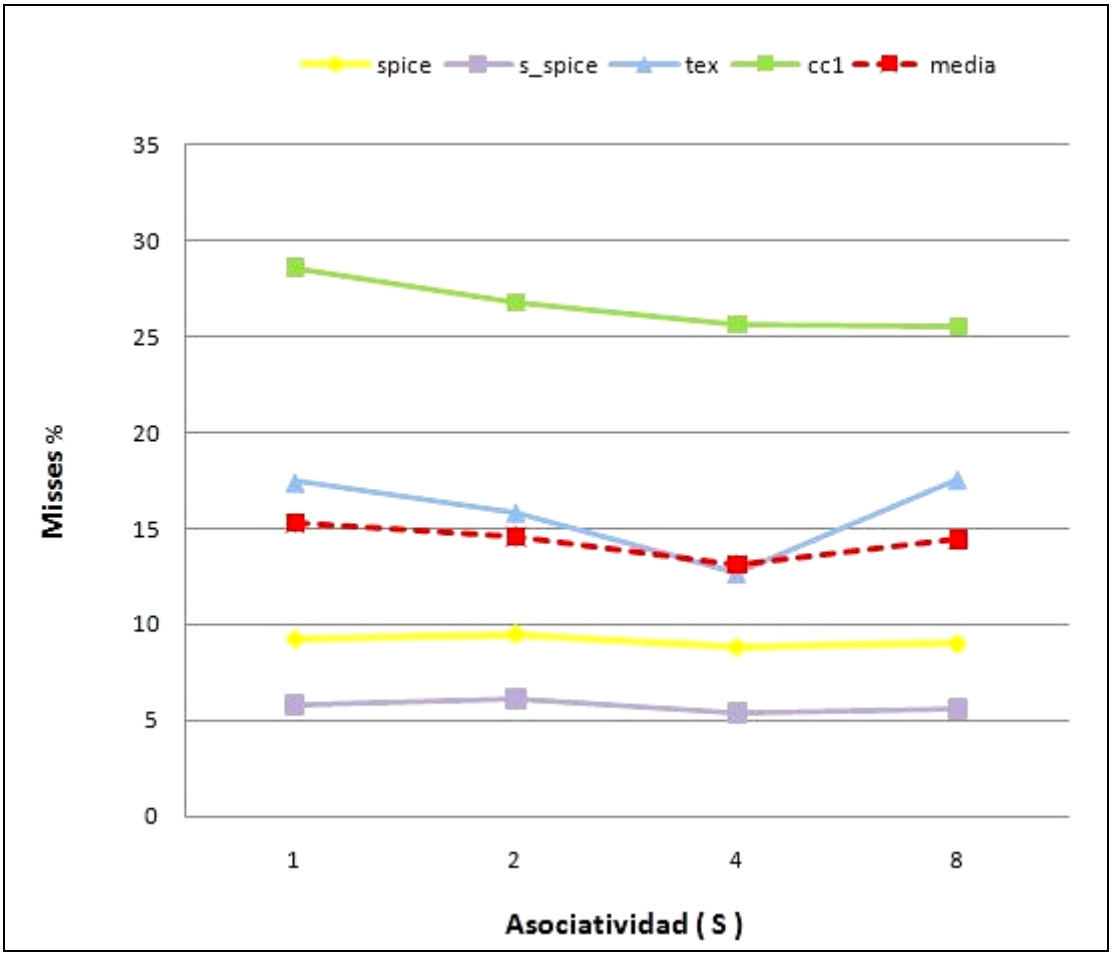

Figura 18. Gráfica misses vs asociatividad

En la gráfica se muestra el porcentaje de misses en función del valor de la asociatividad. El tamaño de las caches de datos y la cache de instrucciones es de 256 bytes, la política de escritura es copy back, la política de reemplazo

es LRU y el tamaño de bloque es de dieciséis Bytes. Los cuatro benchmarks presentan funciones muy parecidas salvo el caso del ejemplo cc1.din que presenta un inesperado repunte de misses con asociatividad ocho. Si nos fijamos en la función media (línea roja discontinua) podemos llegar a la conclusión de que inicialmente al aumentar la **asociatividad disminuye el número de misses**. Sin embargo, hay que tener en cuenta que aumentar la asociatividad también aumenta la complejidad del hardware (y por tanto el coste de la memoria. La asociatividad uno es una mala opción porque da demasiados misses y la asociatividad ocho presenta unos resultados iguales e incluso peores que la asociatividad cuatro. Así pues parece que la **asociatividad óptima sería la cuatro** aunque la asociatividad dos también sería una opción a tener en cuenta.

5.2 Trafico vs asociatividad

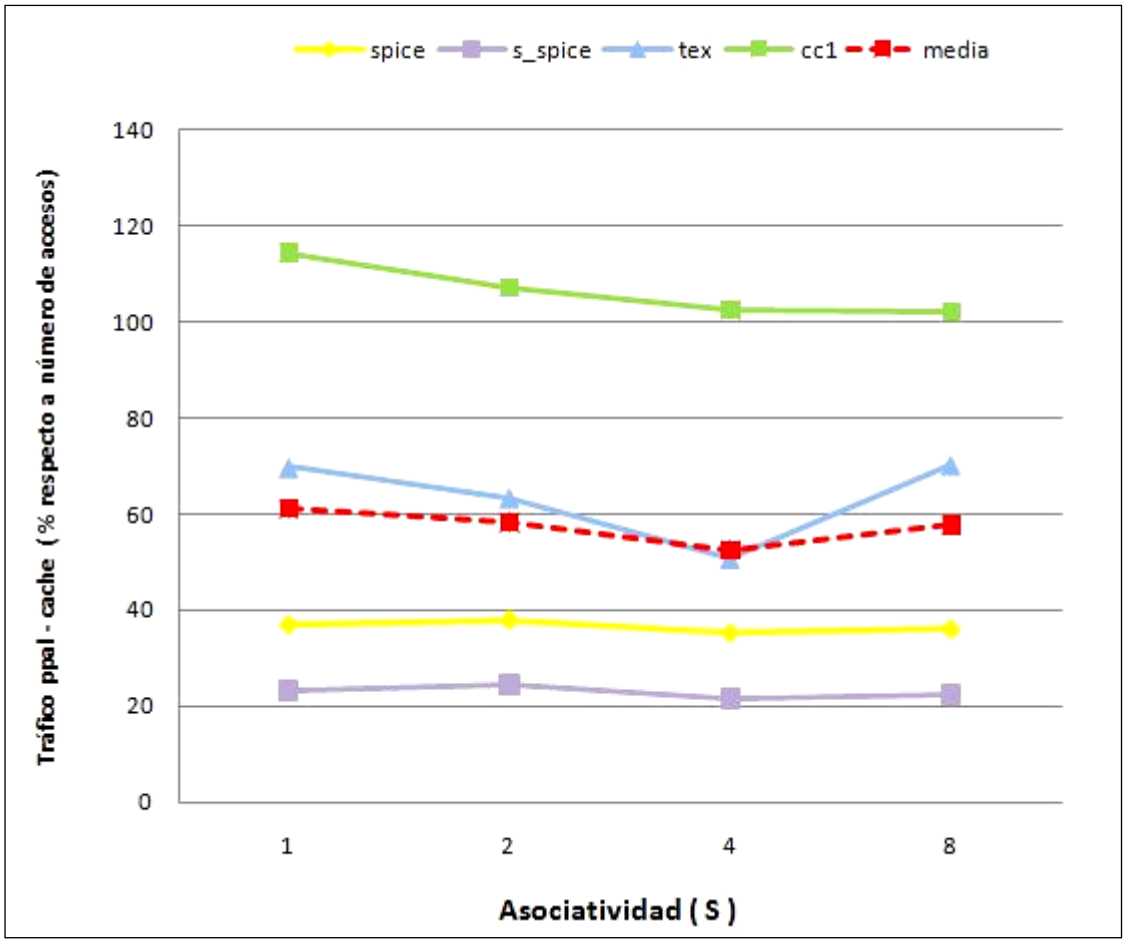

Figura 19. Tráfico Ppal-cache vs asociatividad

En esta gráfica se puede observar el porcentaje de palabras leídas de memoria principal y llevadas a la memoria cache respecto al número total de accesos realizados. El tamaño de las caches de datos y la cache de instrucciones es de 256 bytes, la política de escritura es copy back, la política de reemplazo es LRU y el tamaño de bloque es de dieciséis Bytes. La forma de las funciones es prácticamente igual a las de la figura 18 (misses vs asociatividad). Este hecho es totalmente lógico ya que cuando se produce un miss hay que traer el nuevo bloque de memoria principal. Así, el tráfico de memoria principal a cache es proporcional al número de misses. De este modo, el **valor cuatro de asociatividad sigue siendo el óptimo**.

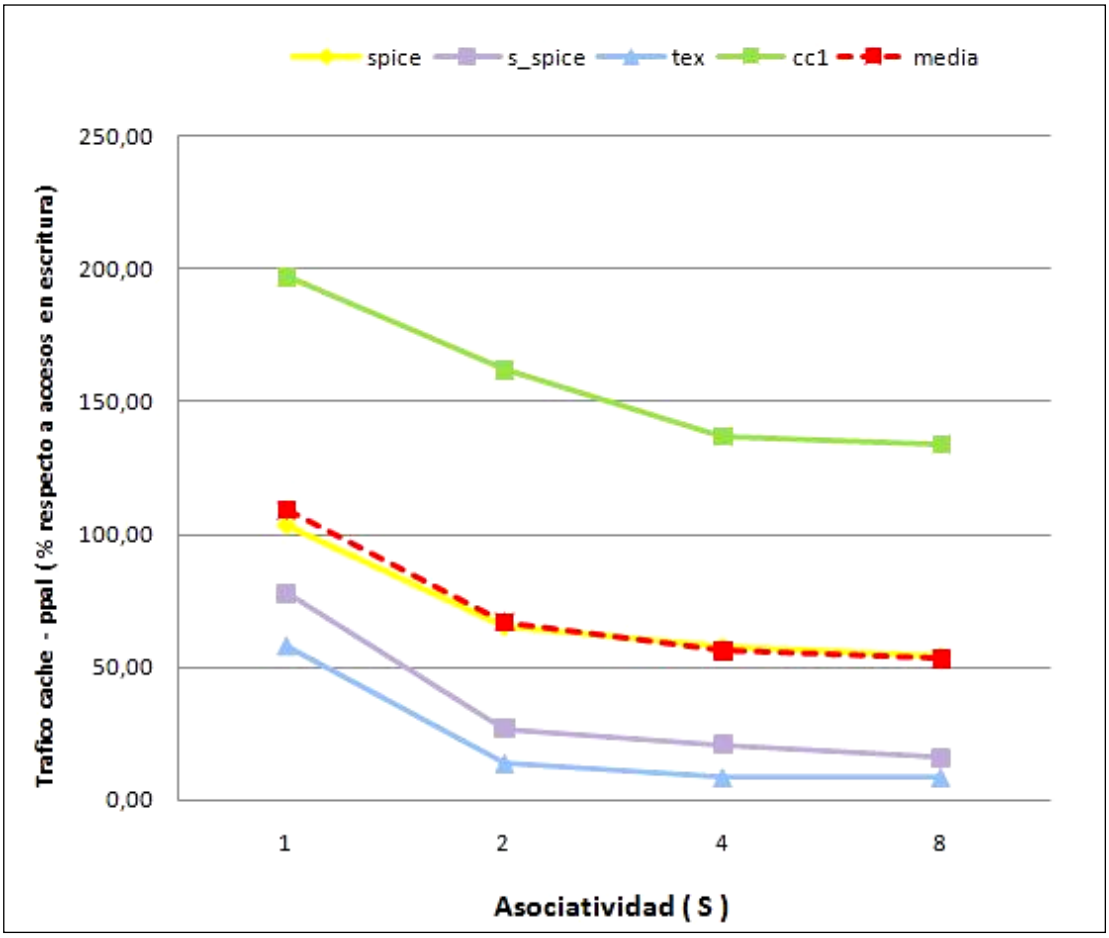

Figura 20. Tráfico cache-Ppal vs asociatividad

Aquí vemos el porcentaje de tráfico de cache a memoria principal respecto al número de acceso en escritura realizados. Los parámetros fijados tienen los mismos valores que en la gráfica anterior. Cuanto mayor es la asociatividad menor es el tráfico de cache a memoria principal. Como vemos no hay mucha diferencia entre el tráfico entre **asociatividad cuatro** y asociatividad ocho siendo más recomendable la primera (menor complejidad hardware).

Así pues, de las dos gráficas anteriores hemos deducido que la opción más recomendable es la de la **asociatividad cuatro**. Es muy importante usar un tipo de cache que genere el menor tráfico posible para **no saturar el bus** entre memoria principal y memoria cache.

5.3 Misses vs tamaño

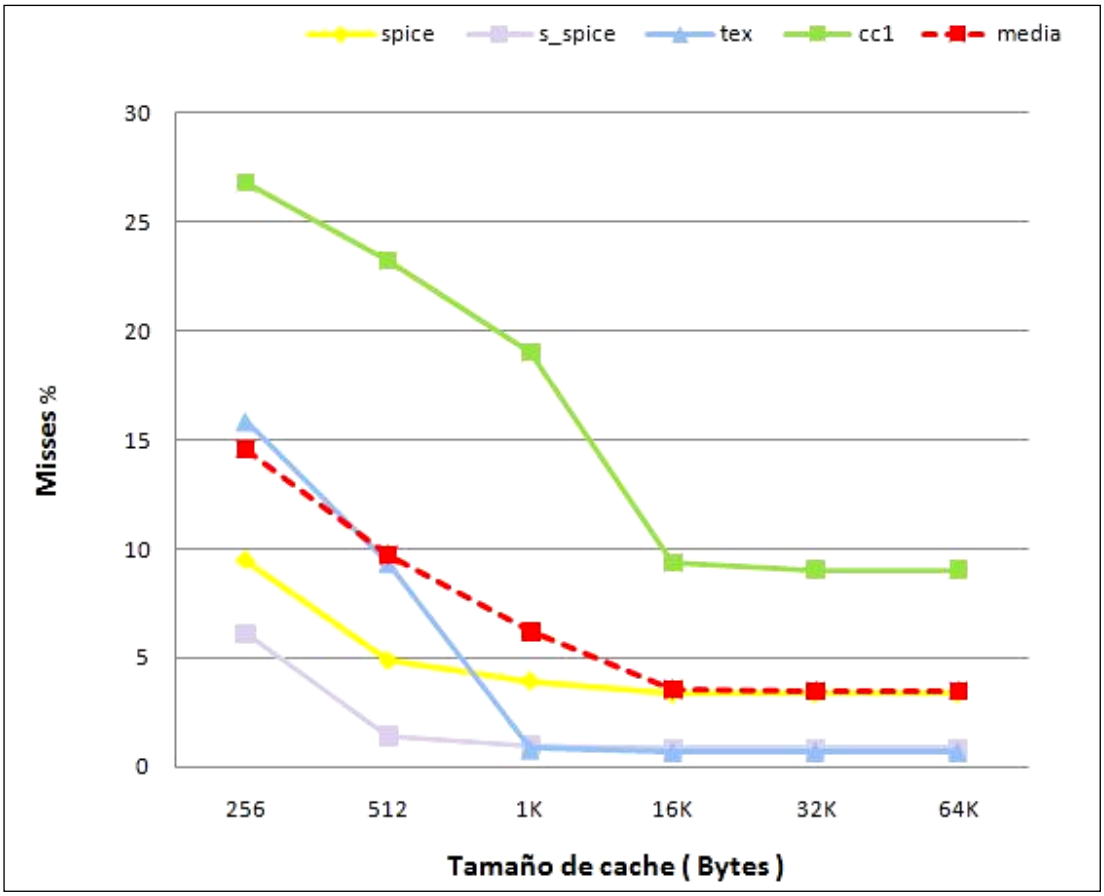

Figura 21. Misses vs tamaño

En esta gráfica se muestra la relación entre el porcentaje de misses y el tamaño de las caches de datos e instrucciones. El resto de parámetros están fijados a asociatividad 2, política de escritura copy back, política de reemplazo LRU y 16 bytes por bloque. Como se puede apreciar en la función media de la figura 21, cuanto mayor es el tamaño de la cache menor es el número de misses. Es un resultado comprensible ya que cuanto mayor es la cache mayor es el número de datos que se pueden guardar en ella y por tanto menor es el número de misses. En la gráfica se aprecia un hecho muy interesante. A partir

del tamaño de 16KBytes el número de misses se estanca y ya no baja más. Así pues, podemos decir que **para estos cuatro benchmarks en concreto el tamaño óptimo para las caches de datos e instrucciones es de 16KB**.

5.4 Retardo vs tamaño

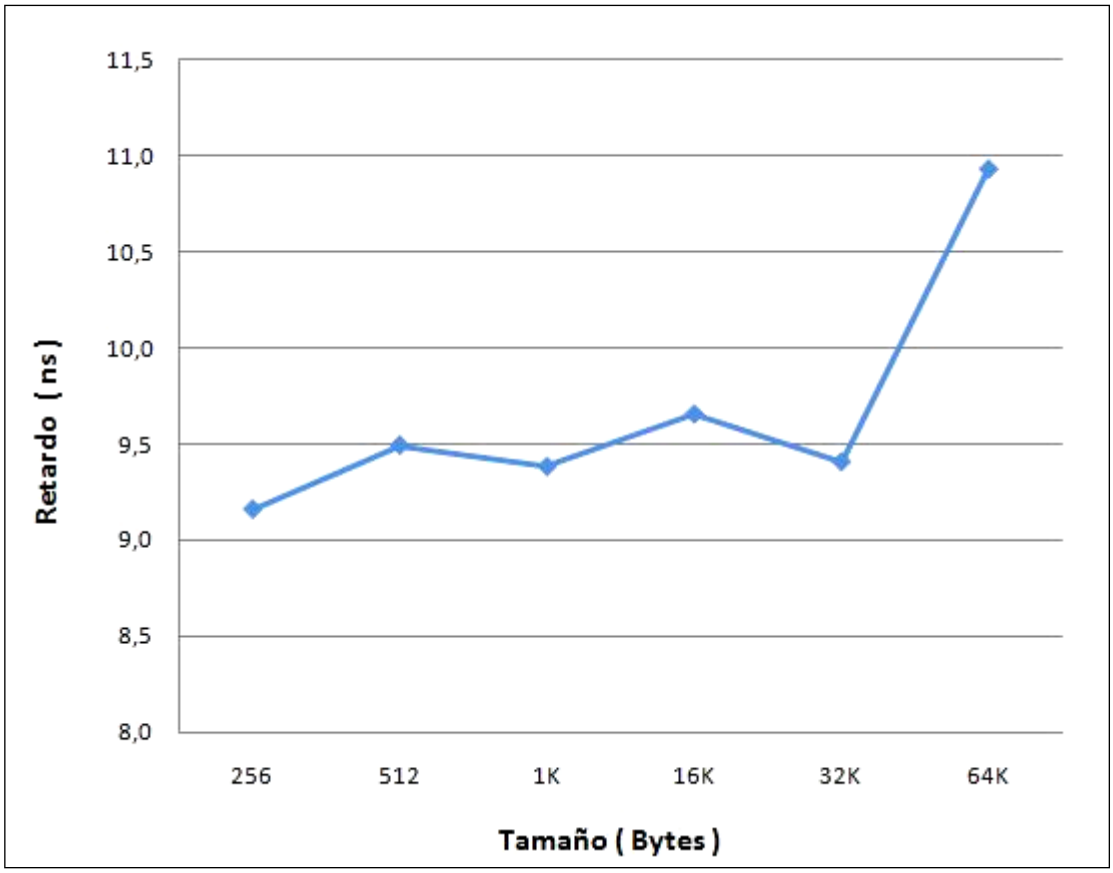

Figura 22. Retardo vs tamaño

En esta gráfica se compara el retardo que presenta el hardware respecto al tamaño de las caches de datos e instrucciones. El resto de parámetros se fijan del mismo modo que en la figura 21. Hasta los 32 KB el retardo oscila entre los 9 y los 9,5 nanosegundos aproximadamente. Es en los 64KB cuando el valor del delay se dispara hasta los once nanosegundos. Teniendo en cuenta los resultados de las anteriores gráficas parece que un tamaño de **16 kB o 32 KB** serían los más recomendables. En todo caso estos en resultados se puede ver que los retardos obtenidos dan una información interesante pero que no se puede tomar al pie de la letra, dado que la tecnología de la FPGA y el trabajo de las herramientas de síntesis hacen que pueda haber algunas oscilaciones extrañas, como el hecho de que en ocasiones al aumentar el tamaño de la memoria, el retardo disminuya ligeramente.

5.5 Retardo vs asociatividad

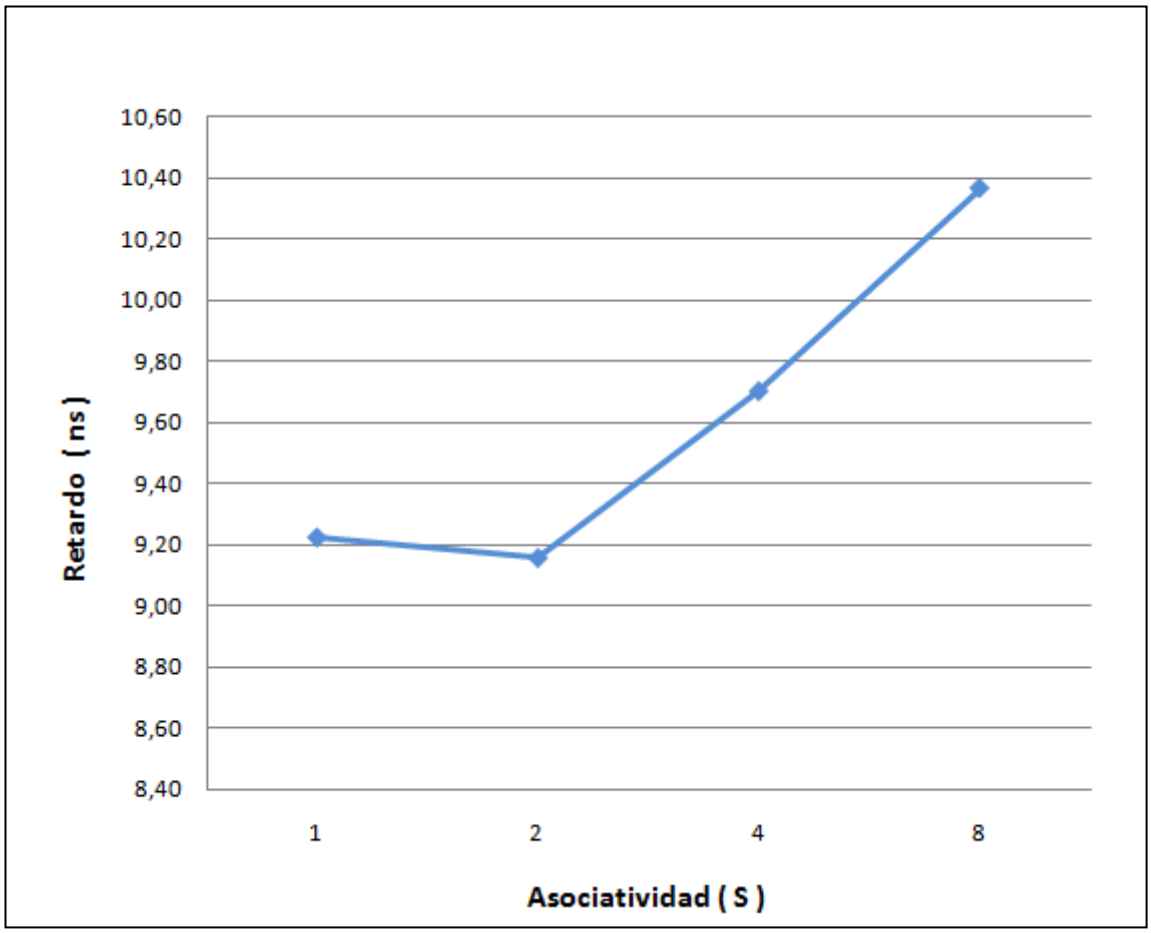

Figura 23. Retardo vs asociatividad

En la figura 22 se han fijado el tamaño de las caches de datos y la cache de instrucciones a 256 bytes, la política de escritura es copy back, la política de reemplazo es LRU y el tamaño de bloque es de 16 Bytes. El diseño hardware para la asociatividad 1 es bastante diferente al diseño para asociatividades mayores que 1. Es por ello que inesperadamente el retardo para asociatividad 1 es mayor que para la asociatividad 2. Además, de nuevo el trabajo de la herramienta de síntesis puede ser distinto en cada diseño produciendo inesperados descensos. Sin embargo, se puede tomar como norma general que **cuanto mayor es la asociatividad mayor es el retardo del hardware diseñado, y esa es exactamente la tendencia que presenta la gráfica.**.

5.6 Slices vs asociatividad

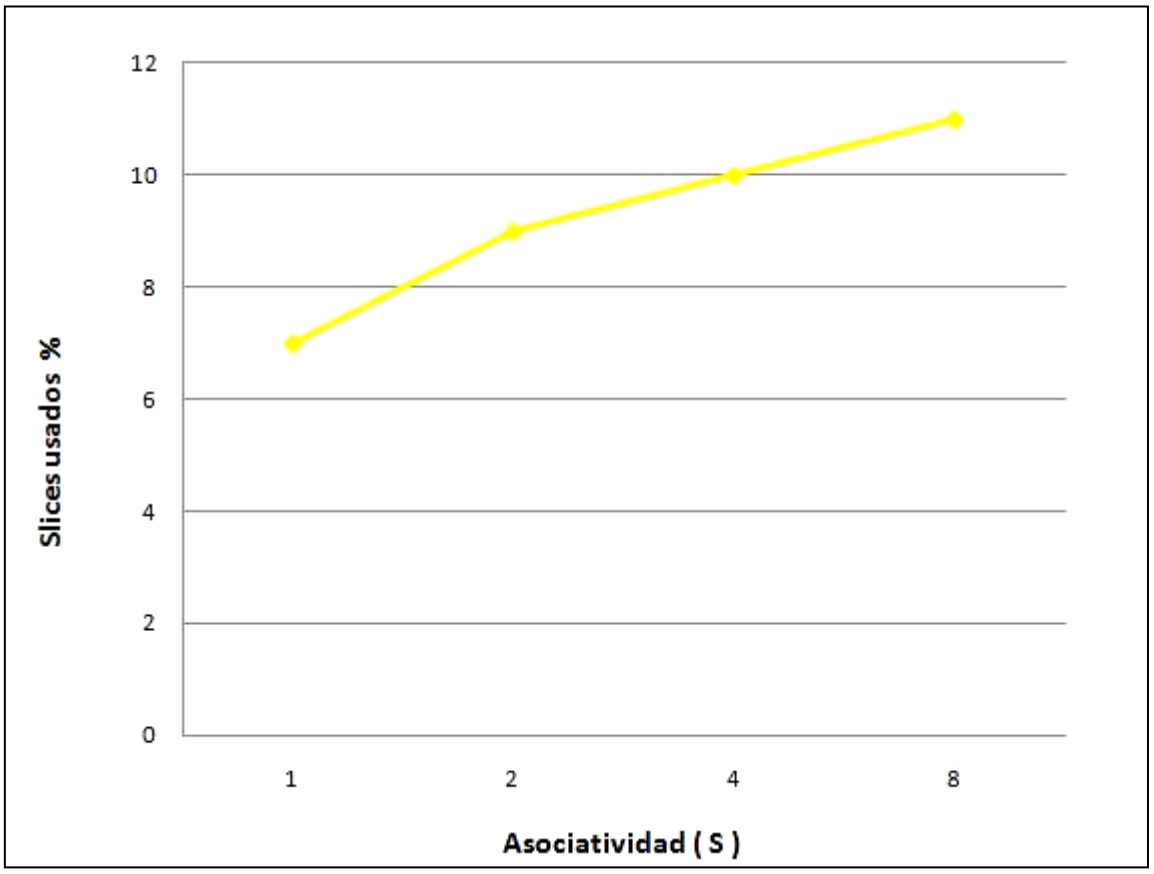

Figura 24. Registros vs asociatividad

Los parámetros fijados para este resultado son los mismos que para la figura 22. Los “slices” son el bloque básico con el que se construye el diseño de la FPGA. Como ya he comentado antes, cuanto mayor es la asociatividad mayor es la complejidad del diseño hardware. Por tanto, **cuanto mayor es la asociatividad mayor es el porcentaje de “slices” utilizados**.

5.7 Block RAMs vs tamaño de cache

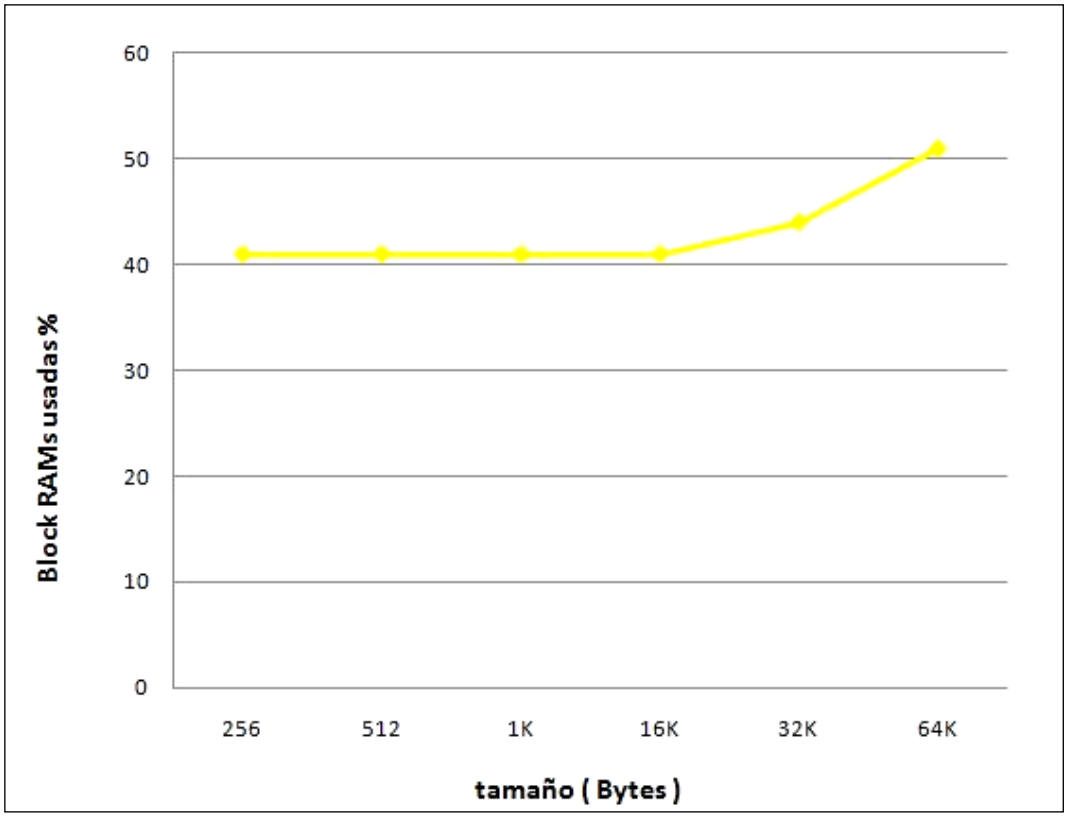

Figura 25. Block RAMs vs tamaño cache

En esta figura la asociatividad está fijada a 2, la política de escritura es Copy back, la política de reemplazo es la LRU y el tamaño de bloque es de 16 Bytes. Hasta 16KB de tamaño de cache caben en un único block RAM es por ello que el porcentaje de block RAMs no varía. Sin embargo, **a partir de 16KB el porcentaje de utilización de block RAMs aumenta a medida que el tamaño de cache se incrementa.**

5.8 Tiempo DINERO vs tiempo diseño hardware

Asociatividad	1	2	4	8
Diseño hardware	0,62	0,41	0,45	0,63
DINERO	55	51	51	51

Tabla 3. Tiempo DINERO vs tiempo diseño hardware (mseg)

En esta tabla se muestra el tiempo que tarda el simulador de caches hardware en calcular los resultados en comparación con el tiempo de ejecución del programa DINERO. Las simulaciones se han realizado con el benchmark s_spice.din y fijando el tamaño de cache de datos e instrucciones a 32 KB, la política de escritura a copy back, la política de reemplazo a LRU y el tamaño de bloque a 16 Bytes. Los resultados obtenidos son muy claros. **El tiempo que tarda DINERO es dos órdenes mayor que el tiempo empleado por el simulador hardware.**

6. CONCLUSIONES Y TRABAJOS FUTUROS

A lo largo de esta memoria hemos podido constatar que se han alcanzado todos los objetivos que se fijaron al comienzo del proyecto fin de carrera (*sección 2.4*).

He conseguido diseñar un simulador de caches que funciona correctamente y que permite determinar cuál de las 380 configuraciones de cache que permite emular es la más adecuada para una determinada secuencia de accesos. Durante el proceso de implementación he aprendido a utilizar el lenguaje de descripción hardware VHDL para crear diseños complejos y he conseguido familiarizarme con el manejo de las FPGAs. El simulador diseñado es totalmente apto para utilizarlo como herramienta docente (por ejemplo en la asignatura “Diseño de arquitecturas”) mostrándole al alumno de forma transparente el funcionamiento interno de una cache. Además he podido comprobar la corrección de mi diseño hardware con la versión software DINERO y las he podido comparar. Las principales ventajas

que ofrece el simulador hardware respecto al software es su rapidez (del orden de dos veces más rápido), su transparencia y el posible uso de éste como herramienta docente al permitir ver las señales internas de la memoria cache, y su evolución ciclos a ciclo utilizando ModelSim. Como inconvenientes se debe destacar la complejidad del diseño hardware, comparado con la programación software, especialmente a la hora de verificar el diseño en la placa. Otra desventaja importantye es la limitación de los recursos disponibles que inpiden simular memorias cache de más de 64Kbytes.

Este proyecto fin de carrera puede servir como base para trabajos futuros más complejos. Veamos algunas posibilidades:

- **Conectar el simulador de caches a un procesador real** implementado en una FPGA en vez de utilizar como entrada los ficheros de trazas. Para ello habría que retocar el diseño hardware realizando la lectura de todos los TAG's de un conjunto en un único ciclo (**en paralelo**) en vez de secuencialmente.

- **Implementar nuevas políticas de reemplazo**: Sería muy interesante añadir otras opciones de reemplazo a parte de la política LRU (por ejemplo implementando **FIFO**).

- **Diseñar más de un nivel de cache**: Gracias a esta mejora el simulador podría ser de ayuda a la hora de buscar la mejor configuración de cache en ordenadores de altas prestaciones que usen varios niveles de memoria cache.

- **Acoplar una memoria RAM grande a la FPGA** para poder simular caches de un tamaño mayor. La placa en la que está la FPGA utilizada permite incluir una memoria DDR de hasta 512 MBytes. Si se incluye un controlador para utilizar esta memoria el problema del espacio desaparecería y se podrían simular FPGAs de gran tamaño.

7. PLANIFICACIÓN

Como podemos ver en las figuras 18 y 19, hay una diferencia de más de **un mes de retraso** entre la planificación inicial y el desarrollo real del proyecto. Este hecho se ha debido principalmente a dos factores. El primero es el coste de tiempo mayor que llevó la realización del diseño del **"simulador básico"**. El segundo y más importante factor fue el aumento del **periodo de pruebas y**

depuración de errores. Hubo un error inesperado en el diseño de la memoria de trazas de modo que cada cierto tiempo se pasaba una traza errónea al simulador. También hubo muchos errores relacionados con la política de reemplazo LRU. Estos errores no aparecían en las simulaciones, sino en la ejecución en la FPGA, por lo que resultó complejo identificarlos.

Se puede ver en los diagramas (parte en gris) que tuve que interrumpir el desarrollo del proyecto durante los meses de diciembre y enero para dedicarme íntegramente a la realización de los dos últimos exámenes de la carrera.

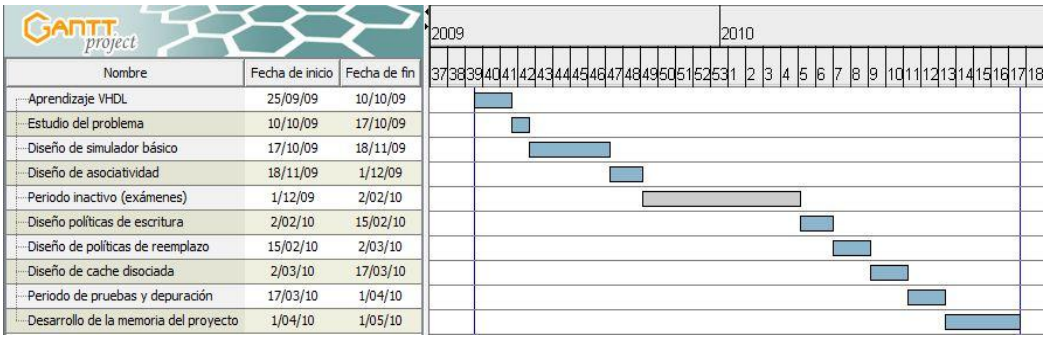

Figura 26. Diagrama de Gantt de la planificación del proyecto

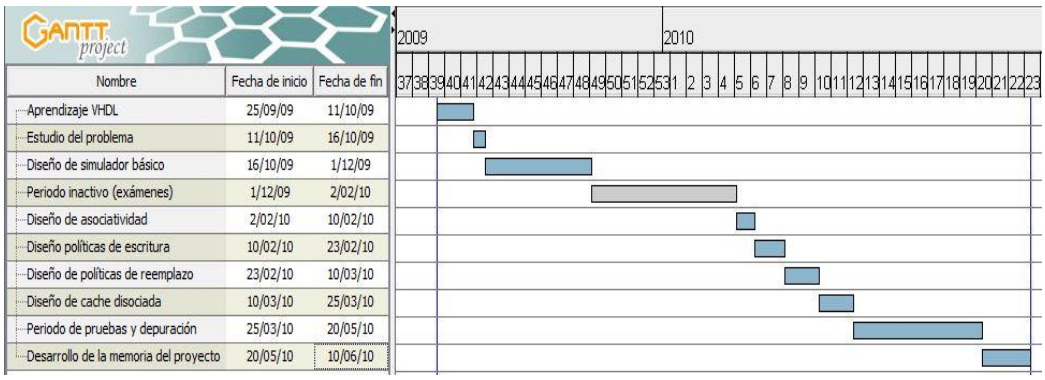

Figura 27. Diagrama de Gantt del desarrollo real del proyecto

He empleado 195 días en este proyecto dedicando una media de cuatro horas por día. De este modo, el tiempo dedicado ha sido de unas 780 horas.