Universidad de Costa Rica | Facultad de Ingeniería | Escuela de Ingeniería Eléctrica

Bitácora de Laboratorio 6

Roberto Sánchez Cárdenas - B77059
Gabriel Jiménez Amador - B73895

San José, 31 de agosto de 2021
Laboratorio de Circuitos Digitales

Resumen

Se implementó una memoria cache y se conectó directamente al modelo de CPU PicoRV32, de modo que se observaron los beneficios de rendimiento que este tipo de memorias genera. Además se simuló la localidad espacial por medio de retardos en la lectura y escritura de datos en memoria principal.

Índice

1. Introducción

El objetivo principal de esta práctica de laboratorio es la implementación de los conceptos teóricos respectivos a memorias caché aprendidos en los distintos cursos de arquitectura de

computadores e implementar dichas memorias en el lenguaje de descripción de hardware Verilog. Para ello se propone la implementación de una caché de mapeo directo y con escritura writeback.

2. Correcciones realizadas

2.1. Correcciones al diseño original del módulo de memoria

Aunque el módulo funcional de memoria emplea como planteado originalmente un contador para retrasar las lecturas y escrituras a memoria, originalmente se tenía dos bloques `always` con diferentes señales de reloj separados la transacción de lectura y escritura. Esto produjo errores con la comunicación con el core PicoRV32 y se optó por seguir un controlador de memoria similar al que se ha utilizado en `system.v` con la diferencia clave de que la señal `mem_ready` no se levanta hasta luego de cierta cantidad de ciclos de reloj designados para cada transacción. De esta forma el core efectivamente no reconocerá que la transacción se llevó a cabo luego de unos cuantos ciclos de reloj, puesto que depende de `mem_ready`.

Además, se agregaron salidas especiales para interceptar ciertas direcciones y redirigirlas a `out_byte`, esta intercepcción no es una lectura (o escritura) normal de memoria por lo que confunde a PicoRV32 si la tratamos como tal. La forma en que se logró un módulo de memoria funcional con salida `out_byte` fue creando este puerto especial en el módulo.

Este módulo de memoria corregido se encuentra en el repositorio del laboratorio aquí.

2.2. Correcciones al diseño original del caché

El diseño original de la memoria caché de mapeo directo con política write-back (según el Anteproyecto 6) tenía los siguientes 5 estados:

- `IDLE`
- `TAG CHECK`
- `EVICTION`
- `REFILL`
- `DATA R/W`

Se tuvieron que agregar más estados y modificar funciones de otros para poder lograr el funcionamiento esperado con el core PicoRV32 junto al módulo de memoria con retrasos sintetizables. Los estados quedaron:

- Estados que funcionalmente sirven como planteados originalmente:

  - `IDLE`
  - `DATA R/W`

- Estados auxiliares a `DATA R/W`, dividen el funcionamiento planteado en este estado en varios para mejorar la modularidad del código:
  - `WRITE`: Escribe en caché directamente si es un hit, caso contrario pasa a `MEM_WRITE`.
  - `READ`: Lee de caché directamente si es un hit, caso contrario pasa a `MEM_READ`.
  - `MEM WRITE`: Guarda el bloque accesado en caché (si hay espacio en el bloque), actualizando la dirección a escribir pasando a `UPDATE_ADDR`. Si no hay espacio, actualiza tags y pasa al estado `RETURN`.
  - `MEM_READ`: Guarda el bloque accesado en caché (si hay espacio en el bloque), actualizando la dirección a escribir pasando a `UPDATE_ADDR`. Si no hay espacio, actualiza tags y pasa al estado `RETURN`.
  - `UPDATE_ADDR`: Actualiza la dirección que recibirá el módulo de memoria y decide si se debe realizar un `writeback` o un `memory read/write`.
  - `WRITEBACK`: Actualiza flags del módulo de memoria para poder accesar a ella según el bloque de caché a realizar writeback, luego pasa al estado `MEM_WRITEBACK`.
  - `MEM_WRITEBACK`: Realiza el writeback en memoria, toma en cuenta los delays de memoria pasando al estado `WRITEBACK_DELAY`.
- Estados creados por formato especial de comunicación con el core PicoRV32:
  - `RETURN`: Devuelve el dato accesado al core PicoRV32. Esto permite comunicación con el core.
- Estados creados por consideración del módulo de memoria con retardos:
  - `WRITEBACK_DELAY`: El caché se mantiene en este estado hasta que el módulo de memoria verifique la transacción de lectura/escritura.
  - `OFF_LIMIT`: El caché se mantiene en este estado si la dirección se encuentra fuera de los límites establecidos de la memoria.

3. Resultados y análisis

Esta sección presenta los resultados de las implementaciones realizadas. Todos los resultados mostrados son de creación propia y fueron obtenidos utilizando el simulador de Xilinx: Vivado.

3.1. Ejercicio 1

Este primer ejercicio consiste en implementar una lista enlazada directamente en la memoria del pico_rv32. Para ello se guarda siempre en una posición n la dirección del siguiente elemento y en la n+1 se guarda el dato a buscar. Como se tiene esta estructura, busca la dirección en n para leer el siguiente elemento.

En las últimos 48kB se guardan números secuenciales. Una vez se termina de guardar, se lee toda la memoria para obtener los últimos 5 datos impares.

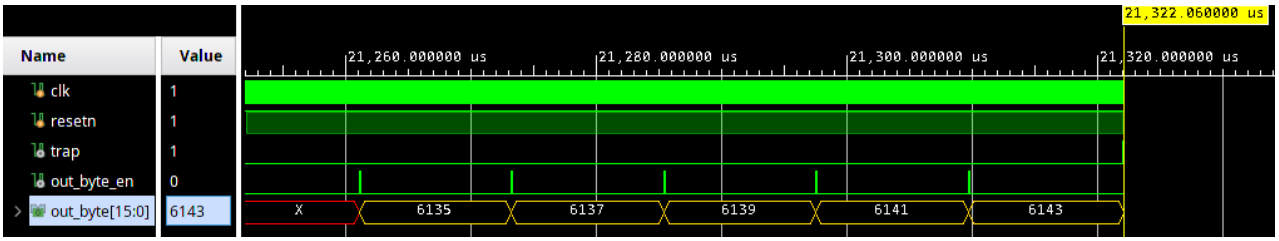

Figura 1: Resultado de la simulación de la creación y recorrido de la lista enlazada con `FAST_MEMORY = 1`

- Slices: 316

Sólo para motivos de comparación más adelante se simuló la creación y el recorrido de la lista enlazada con `FAST_MEMORY = 0` y se obtuvo los resultados de la Figura 2. Se observa una diferencia de 14 ms agregados que tarda el circuito en obtener el primer número impar, comparado al de `FAST_MEMORY = 1`.

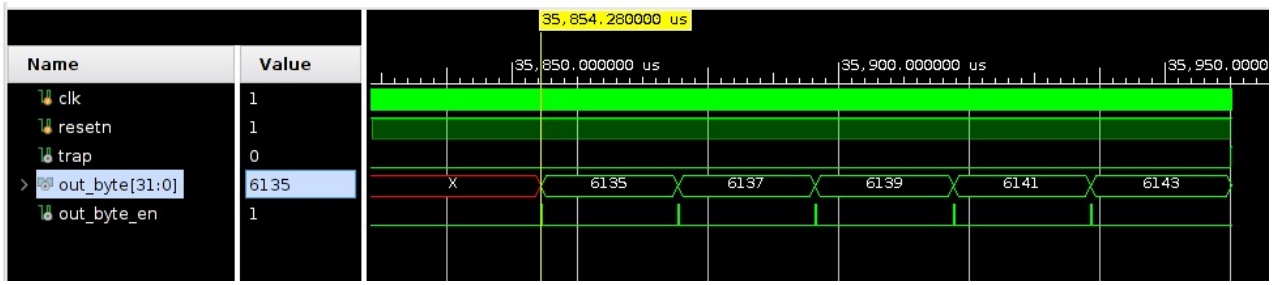

Figura 2: Resultado de la simulación de la creación y recorrido de la lista enlazada con `FAST_MEMORY = 0`

3.2. Ejercicio 2

En la Figura 3 se observa el comportamiento esperado del retardo al realizar una transacción de lectura de memoria. Se realizó un *instruction fetch* al iniciar el programa (firmware) de la primera instrucción en memoria, `0x00004137`. Se observa que la diferencia de tiempo en que se

solicita la transacción y finaliza, siguiendo las señales `mem_valid` y `mem_ready` respectivamente, es de 100 ns, es decir 10 ciclos de reloj[1].

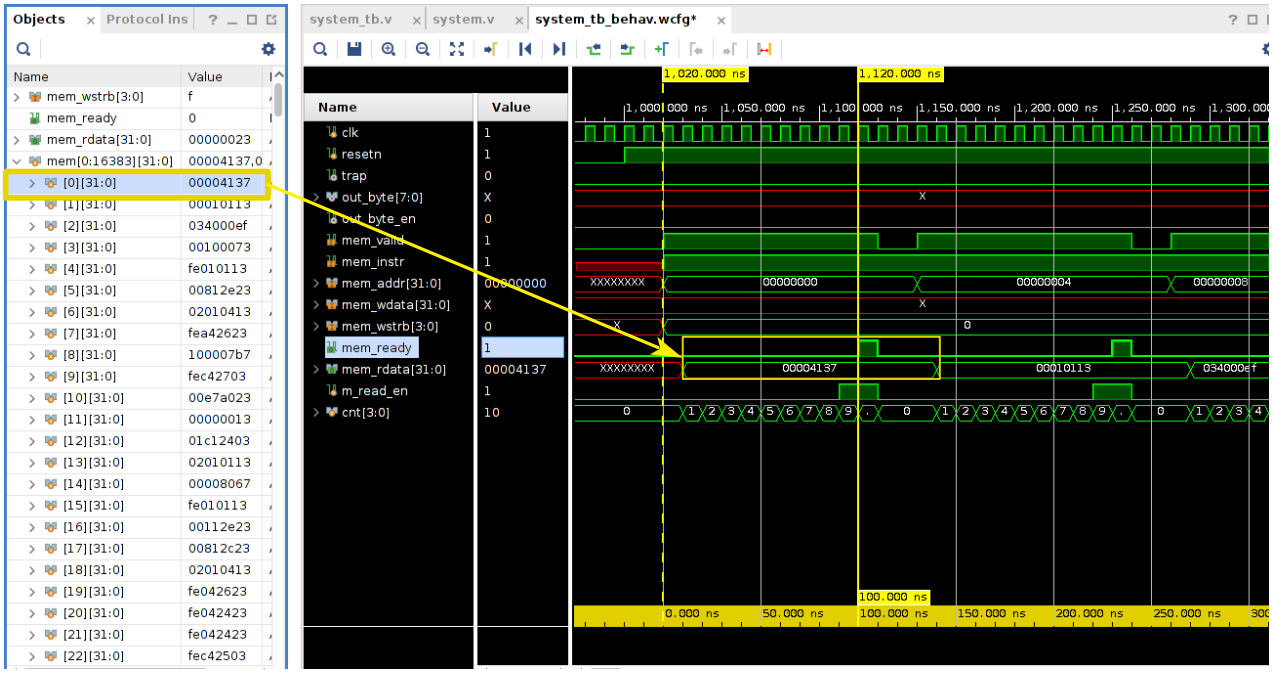

Figura 3: Simulación de una lectura de memoria, se observa que el delay tomado desde que inicia la transacción (levantando `mem_valid`) y finaliza (levantando `mem_ready`) es de 100 ns

En la Figura 4 se observa el comportamiento esperado del retardo al realizar una transacción de escritura a memoria. En esta transacción de ejemplo, el valor de 12 en `mem_wdata` se escribirá en la dirección designada de `0x00003FFC` en la memoria. Se observa que la diferencia de tiempo en que se solicita la transacción y finaliza, siguiendo las señales `mem_valid` y `mem_ready` respectivamente, es de 150 ns, es decir 15 ciclos de reloj.

1 Un ciclo de reloj tarda 10 ns

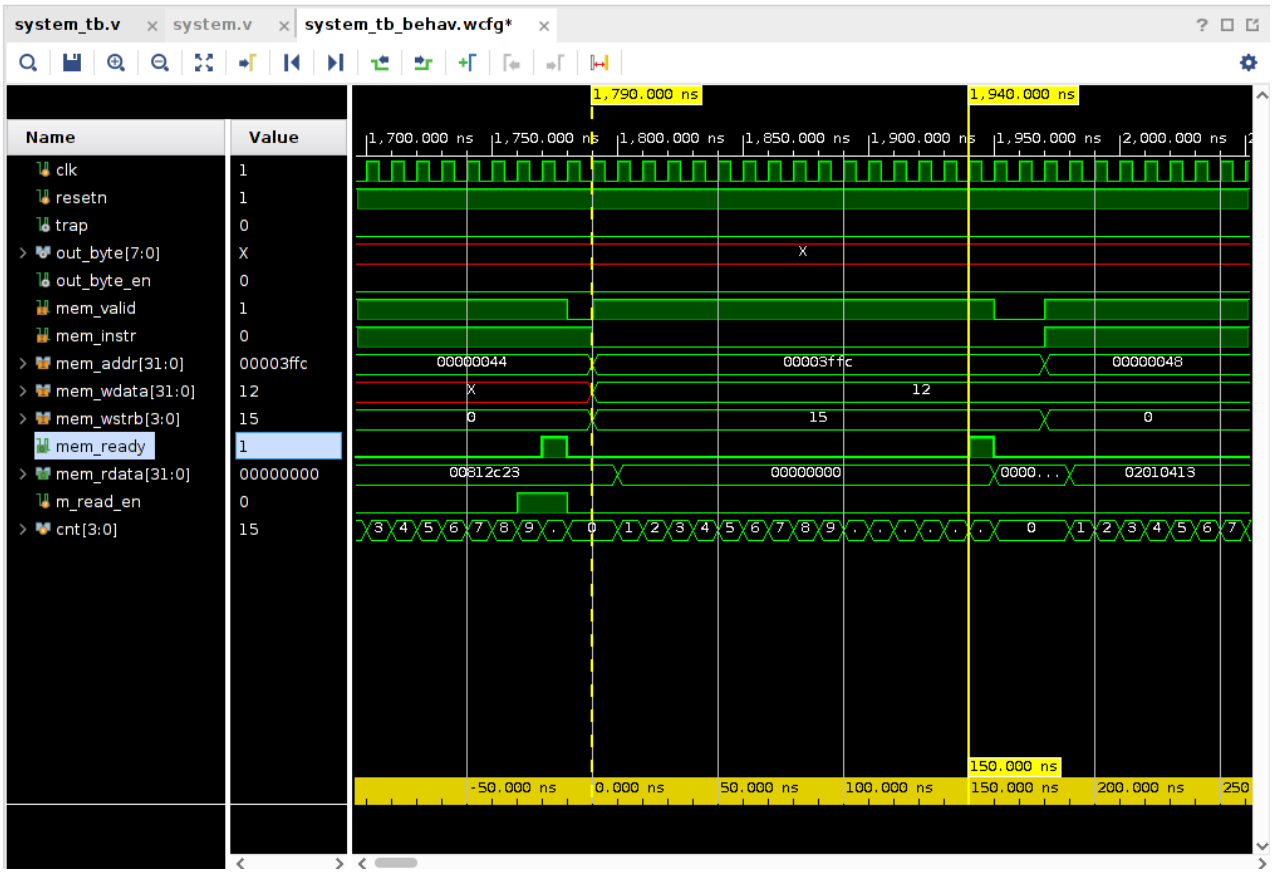

Figura 4: Simulación de una escritura en memoria, se observa que el delay tomado desde que inicia la transacción (levantando `mem_valid`) y finaliza (levantando `mem_ready`) es de 150 ns

Al realizar la creación y recorrido de la lista enlazada del punto anterior con el módulo de memoria con delays, se obtienen los números impares de la Figura 5. Son los mismos números pero ahora tarda cerca de 0.1 s para obtener el primer dato.

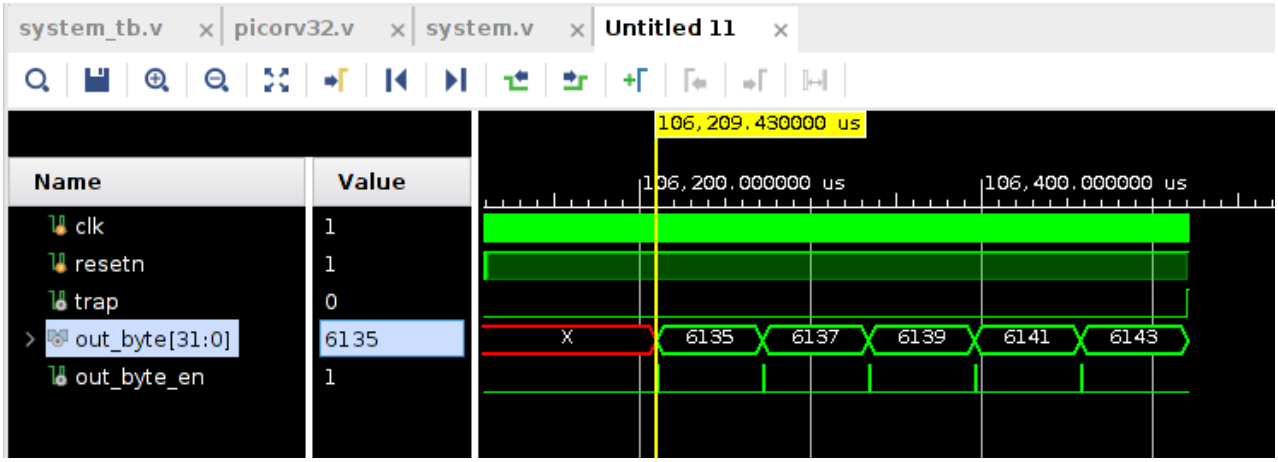

Figura 5: Resultado de la simulación de la creación y recorrido de la lista enlazada con el módulo de memoria.

Para encontrar el punto en donde el circuito inicia a construir la lista enlazada se eligió el punto cuando inicia la transacción de escritura a la dirección `0x4000` y se eligió el punto cuando finaliza la escritura a la última dirección `0xFFFC` como punto de finalización (Figuras 6 y 7). Se obtuvo una diferencia de:

$$\begin{aligned} T_{\text{creacion}} &= 50505680 - 11590 \\ &= 50\,497\,090\,\text{ns} \\ \text{Ciclos de reloj}_{\text{creacion}} &= 5\,049\,709 \end{aligned}$$

Para encontrar el punto en donde el circuito inicia a construir la lista enlazada se eligió el punto cuando inicia la transacción de lectura de la dirección `0x4000` y se eligió el punto cuando finaliza la lectura de la última dirección `0xFFFC` como punto de finalización (Figuras 8 y 9). Se obtuvo una diferencia de:

$$\begin{aligned} T_{\text{recorrido}} &= 106200500 - 50518760 \\ &= 55\,681\,740\,\text{ns} \\ \text{Ciclos de reloj}_{\text{recorrido}} &= 5\,568\,174 \end{aligned}$$

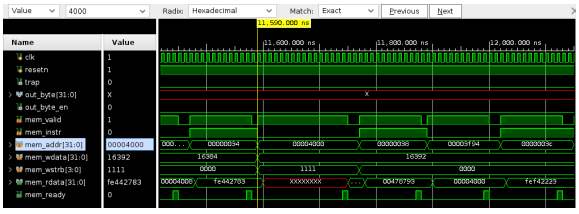

Figura 6: Punto donde inicia la creación de la lista enlazada

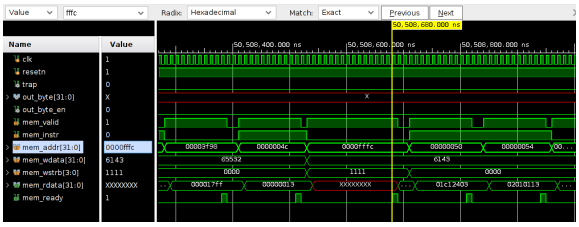

Figura 7: Punto donde finaliza la creación de la lista enlazada

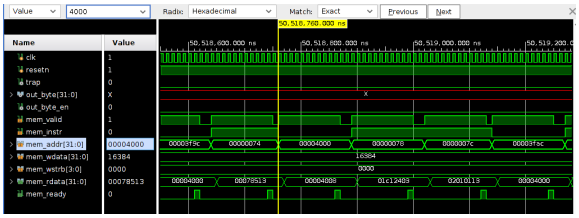

Figura 8: Punto donde inicia el recorrido de la lista enlazada

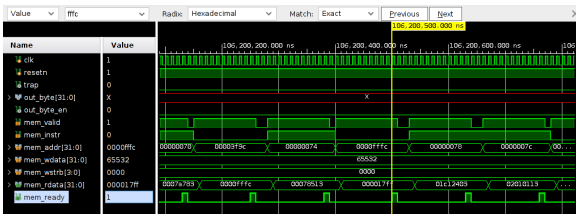

Figura 9: Punto donde finaliza el recorrido de la lista enlazada

Con ambos bucles, el de escritura y lectura se tiene que el circuito tarda 10 617 883 ciclos de reloj. Además, es importante denotar que aunque parezca que la transacción de lectura tiene una duración mayor al de escritura, contradiciendo lo que sugeriría los delays puestos en

el módulo de memoria, se tiene que tomar en cuenta que el proceso de **creación** solo realiza **una transacción de escritura** a memoria para cada dirección, mientras que el proceso de **recorrido** realiza **dos transacciones de lectura** de memoria por cada dirección, una para conseguir el puntero y otra para conseguir el dato.

3.3. Ejercicio 3

En este ejercicio se logró diseñar una cache funcional con retrasos provenientes de la memoria. La implementación en Verilog es parametrizada, por lo que tanto los retardos como el tamaño de la cache y memoria principal puedes ser modificados. Para comprobar el funcionamiento de la misma se utilizó el firmware creado en el ejercicio 1. Al correr la simulación mostrada en la figura 10 se puede notar que al simulador le toma 45.316us completar la corrida.

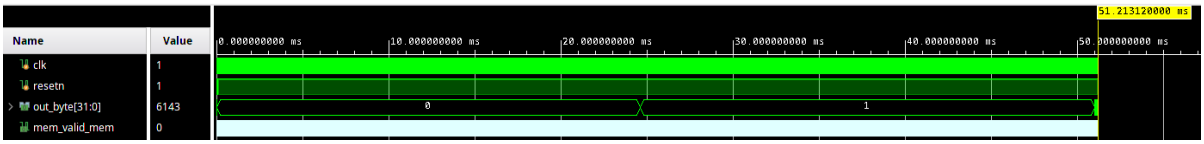

Figura 10: Cache capaz de simular el firmware

Además, en la figura 11 se puede ver como se logran leer con éxito los valores impares que se escribieron en la memoria principal.

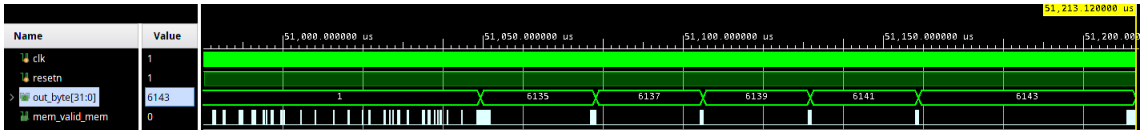

Figura 11: Resultado de números impares

En la figura 11 se puede notar que sí se logra obtener los últimos datos impares de la memoria y ponerlos en `out_byte`, al igual que se hizo cuando el CPU estaba conectado directo al CPU. En un caso idea sin tiempos de retraso por lo que toma ir hasta memoria principal, se tiene que el firmware se ejecuta en 36,950us. Para efectos ilustrativos, se setearon retardos de lectura y escritura directos a la memoria principal, mientras que la caché no tiene retardos.

Se añadieron 10 y 15 retardos en lectura y escritura respectivamente, en un caso sin caché, presentado en el ejercicio 2 se obtuvo que la simulación toma 106,500us. Cuando se añade una caché que tiene un funcionamiento más veloz, el tiempo se reduce a 51,213us, lo cual es menos de la mitad del tiempo.

La idea de una cache es reducir los tiempos que toma ir hasta memoria principal, pues una memoria RAM está espacialmente distante de un CPU, por lo que los resultados obtenidos concuerdan con la teoría.

Además se calcularon los ciclos que toma la formación de la lista y la lectura de la misma, para ello se definieron contadores para cada etapa. Se obtuvo que la formación toma **2462280** y la lectura toma **2641593**. Un motivo por el cual la formación toma menos ciclos es porque a nivel de firmware, se utiliza una única llamada a función para escribir dirección y dato, mientras que para lectura se hacen dos llamados a la misma función.

Finalmente, se tiene una tasa de fallos de un **2.16 %**, por lo que la tasa de aciertos es del **97.83 %**.

Site Type	Used	Fixed	Available	Util%
Slice	1839	0	15850	11.60
SLICEL	1300	0		
SLICEM	539	0		
LUT as Logic	2682	0	63400	4.23
using O5 output only	0			
using O6 output only	2387			
using O5 and O6	295			
LUT as Memory	176	0	19000	0.93
LUT as Distributed RAM	176	0		
using O5 output only	0			
using O6 output only	128			
using O5 and O6	48			
LUT as Shift Register	0	0		
Slice Registers	3940	0	126800	3.11
Register driven from within the Slice	913			
Register driven from outside the Slice	3027			
LUT in front of the register is unused	2348			
LUT in front of the register is used	679			
Unique Control Sets	157		15850	0.99

Figura 12: Número de Slices del reporte de implementación

3.4. Ejercicio 4

En este ejercicio se modificaron los parámetros del caché: el tamaño del bloque (palabras) y el tamaño del caché (KB) y se simuló, sintetizó e implementó para cada uno de los casos. A manera de ejemplo se tiene los números impares en la Figura 13 para una caché con un un bloque de 2 palabras y un tamaño de 2 KB y el número de slices que consume en la Figura 14.

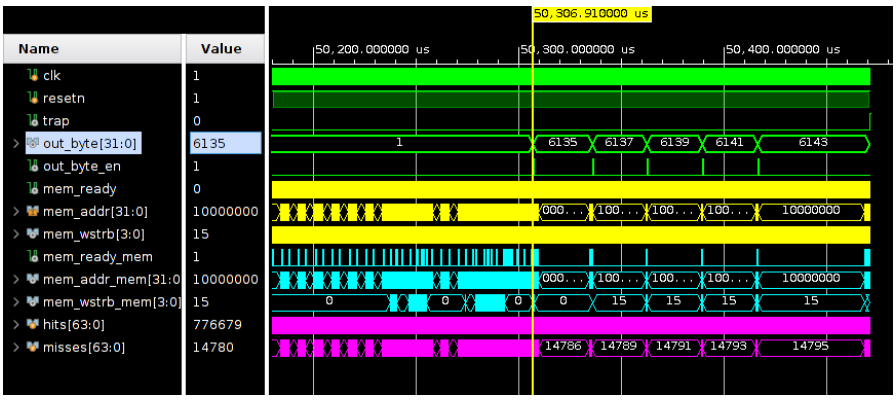

Figura 13: Resultado de números impares para la caché con un bloque de 2 palabras y un tamaño de 2 KB

Site Type	Used	Fixed	Available	Util%
Slice	2781	0	15850	17.55
SLICEL	2029	0		
SLICEM	752	0		
LUT as Logic	4642	0	63400	7.32
using O5 output only	1			
using O6 output only	4089			
using O5 and O6	552			
LUT as Memory	304	0	19000	1.60
LUT as Distributed RAM	304	0		
using O5 output only	0			
using O6 output only	256			
using O5 and O6	48			
LUT as Shift Register	0	0		
Slice Registers	7131	0	126800	5.62
Register driven from within the Slice	1436			
Register driven from outside the Slice	5695			
LUT in front of the register is unused	4713			
LUT in front of the register is used	982			
Unique Control Sets	290		15850	1.83

Figura 14: Número de slices del reporte de implementación para la caché con un bloque de 2 palabras y un tamaño de 2 KB

Con la simulación, se tomaron los puntos de inicio de creación de la lista, finalización de creación de la lista, inicio de recorrido de la lista y finalización de recorrido de la lista, como en la sección 2. Con la información en estos puntos se obtiene el hit rate, miss rate y ciclos de reloj para la formación y recorrido de la lista, así como los resultados totales. Esta información se tiene en resumen en la Tabla 1.

Tabla 1: Resultados de tipos de cachés con diferentes parámetros. Incluye resultados cuando forma la lista, cuando la recorre y resultados totales.

	Parámetros caché		**Formando**			**Recorriendo**			**Totales**			
	Tam. Bloque (palabras)	Tam. Caché (KB)	Hit rate (%)	Miss rate (%)	Ciclos de reloj	Hit rate (%)	Miss rate (%)	Ciclos de reloj	Hit rate (%)	Miss rate (%)	Ciclos de reloj	**Slices consumidos**
Caché 1	2	2	98.12	1.88	2458006	98.14	1.86	2585696	98.13	1.87	5030691	2781
Caché 2	4	2	98.84	1.16	2461676	98.70	1.30	2667386	98.77	1.23	5117538	2093
Caché 3	2	4	98.26	1.74	2455146	98.33	1.67	2567940	98.29	1.71	4993563	4515
Caché 4	4	4	99.01	0.99	2431184	98.98	1.02	2605302	98.99	1.01	5010882	2761

Analizando la Tabla 1 se puede notar en primera instancia que los tiempos de creación y recorrido de la lista enlazada se reducen significativamente con respecto a un sistema sin memoria caché, promedian un 50 %. Es decir, todo el sistema es 2 veces más rápido (en promedio) que si no tuviera memoria caché.

Con respecto al hit/miss rate, siguiendo a la Tabla 1, se tiene un miss rate menor conforme aumenta o el tamaño del bloque o el tamaño de la caché. Cuando los dos aumentan de tamaño (caché 4) se tiene el miss rate mínimo de 1.01 % (total). Se observa que si se tuviera que elegir entre aumentar el tamaño de bloque o el tamaño del caché para mejorar el miss rate, es mejor aumentar el tamaño del bloque (caché 2, miss rate de 1.23 % comparado a la caché 3 con miss rate de 1.71 %).

En tanto a los ciclos de reloj, parece que tienden a estar constantes en 5 M, aunque mejores que el caso sin caché, este tipo de caché no parece tener mejor rendimiento con estos leves cambios de tamaño de bloque y caché, siguiendo una desviación entre valores de σ = 47669,8.

Analizando los slices consumidos por cada tipo de caché, se tiene un impacto grande en la cantidad consumida al duplicar el tamaño del caché de 2781 a 4515 sin modificar el tamaño del bloque. Esto esta relacionado a la cantidad de recursos necesarios para reservar espacio en la FPGA para una caché de mayor tamaño. Mientras que aumentar el tamaño de los bloques en la caché disminuye la cantidad de slices consumidos puesto que habrán mayor cantidad de bytes que representarán un bloque. La cantidad de sets que conforman la memoria caché son proporcionales al tamaño de la caché y son inversamente proporcionales al tamaño del bloque.

4. Repositorio

El código necesario para emular los resultados de este laboratorio se pueden encontrar en Github utilizando el siguiente enlace:
https://github.com/ucr-ie-0424/laboratorio-6-jimenez-sanchez

El ID del último commit es:
c1d488022bcbd7714c9d8f09621e777309cbfbaf

5. Conclusiones y recomendaciones

5.1. Conclusiones

- La implementación de una cache como intermediario de entre una unidad de procesamiento y una memoria principal es capaz de reducir los tiempos de ejecución significativamente, pues se aprovechan los principios de localidad temporal y espacial.
- La manera en que se escribe un firmware puede beneficiar o perjudicar el rendimiento. Pequeños cambios como realizar varias operaciones sencillas en una misma llamada a función son capaces de reducir el número de ciclos que toma al CPU la ejecución. En este caso el efecto es más evidente pues no se implementan predictores de salto.
- Resulta relativamente sencillo construir un módulo de memoria con retrasos que se pueda comunicar con el core PicoRV32, sólo se debe atrasar la señal que indica que se realizó una transacción correcta: `mem_ready`, todas las demás señales se atrasarán con ella.
- Con respecto a la variación de parámetros de la caché, incrementando el tamaño de los bloques junto al tamaño de la caché mejora el hit rate, mientras que incrementar el tamaño de la caché sin modificar el tamaño de los bloques consume cantidad de slices excesivos para el rendimiento que aporta.

5.2. Recomendaciones

- El core PicoRV32 tiene un formato especial como comunicarse con memoria, se debe consultar la fuente para analizar con detalle como funciona y poder realizar transacciones correctas de/a memoria.