Universidad de Costa Rica

Bitácora

Experimento 4

Prof. José Daniel Hernández

Luis Soto Camacho, B57082
Yeison Rodríguez Pacheco, B56074

Grupo 2

18 de noviembre de 2019

Índice

Índice de figuras

1. Objetivos

1.1. Objetivo general

- Aplicar los conceptos de arquitectura de computadoras para el diseño de jerarquías de memoria.

1.2. Objetivos específicos

- Analizar interfaces de memoria de un CPU e implementar un cache sencillo
  Recolectar y analizar datos de rendimiento de un cache.

2. Correcciones realizadas al anteproyecto y a la solución propuesta en el mismo

No se realizaron correcciones al anteproyecto, ya que se contempló todo lo necesario para la realización de este experimento.

3. Resultados, justificación y preguntas de la guía del laboratorio

3.1. Ejercicio 1

En esta sección se llenó la memoria con una lista enlazada con el formato presentado en el anteproyecto de este laboratorio, a continuación se presenta el código que realiza esta tarea:

```
#include <stdint.h>

static void putuint(uint32_t i, uint32_t direccion) {
        *((volatile uint32_t *)direccion) = i;
        }


void main() {
        uint32_t direccion_1 = 0x10000000;


        uint32_t puntero = 0x00004000;
        uint32_t dato = 0;
        while(puntero <= 0x00010000){
                putuint((puntero + 8),puntero);
                puntero += 4;
                putuint(dato,puntero);
                puntero += 4;
                dato += 1;
        }

        uint32_t contador = 0;
        puntero -= 4;
        while (contador < 6)
        {
                if(*((volatile uint32_t *)puntero) %2 == 1){
                        putuint(*((volatile uint32_t *)puntero),direccion_1);
                        contador += 1;
                }
                puntero -= 8;
        }
```

```
        while(1);
}
```

Lo que se realiza en el primer while del código es llenar la memoria a partir de la dirección 0x4000 con un dato (numero que empieza en 0) y el puntero al siguiente elemento. La memoria se llena hasta la dirección 0x10000.

En el segundo while lo que se hace es encontrar los últimos 5 datos impares y los escribe en la dirección 0x10000000, esto para que lo intercepte el archivo system.v y de esta forma pueda ser desplegado en los display de 7 segmentos.

En la figura 1 podemos ver en la parte izquierda la memoria, se ve como almacena el dato y el puntero al siguiente elemento en cada caso. Para enviar los últimos 5 elementos impares se utiliza la salida out_byte, podemos ver que esta salida está cambiando debido a que los display de 7 segmentos funcionan realizando multiplexación de cada uno. Al ejecutar el programa es posible observar que los 5 números son desplegados cada uno por 4 segundos aproximadamente.

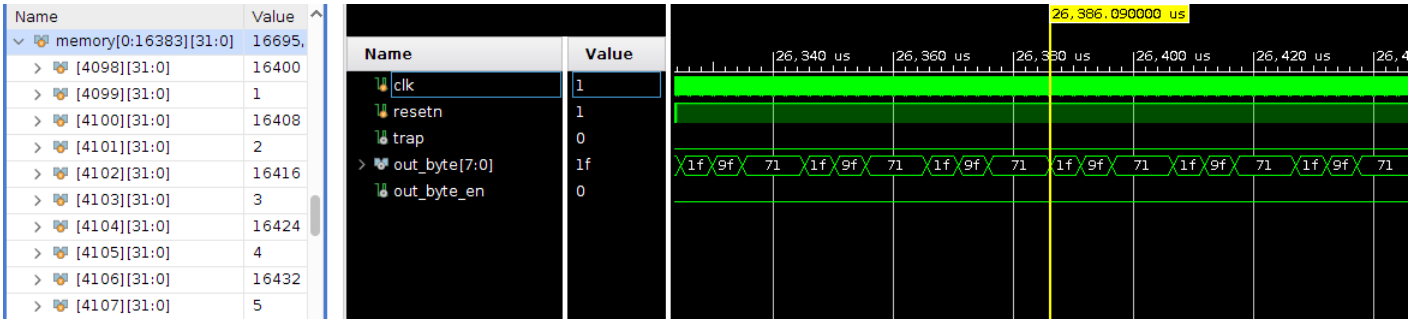

Figura 1: Simulación ejercicio 1.

3.2. Ejercicio 2

Para la implementación de la cache, se definieron las siguientes entradas y salidas:
Entradas

- clk: Señal de reloj.
- resetn: Señal de reset.
- address: Dirección enviada desde CPU.
- valid: Bandera de dato válido.
- instr: Bandera de instrucción.
- wdata: Datos a ser escritos.
- wstrb: Bandera de escritura de bits.
- ready_mp: Bandera de dato listo desde memoria principal.
- rdata_mp: Dato leido desde memoria principal.

Salidas

- clk: Señal de reloj.
- resetn: Señal de reset.
- address: Dirección enviada desde CPU.
- gotHit, gotMiss: Banderas que indican el resultado del último acceso a memoria.
- ready: Bandera de dato listo de salida.
- rdata: Dato enviado a CPU durante lectura.
- valid_mp: Bandera de dato válido a memoria principal.
- instr_mp: Bandera de instrucción a memoria principal.
- wdata_mp: Datos a ser escritos en memoria principal.
- wstrb_mp: Bandera de escritura de bits a memoria principal.

Así mismo, se crearon las siguientes estructuras a nivel interno de la cache:

- cacheData1, cacheData2: Matrix de registros donde se guarda los datos de cada way.
- cacheLRU: Array donde se guarda el valor del way menos usado recientemente.
- tag1, tag2: Array con los tags de los sets guardados en la cache.
- cacheDirty1, cacheDirty2: Array de bits de dirty de los sets de la cache.
- offset, index, tag: Registros donde se guarda los datos de la actual dirección.
- mod: Residuo del cociente entre la dirección y el tamaño de bloque.
- waitData: Bandera que indica si se esperan datos de la memoria principal.
- i: Contador para la transferencia de bloques a o desde memoria principal.
- address_, wdata_, ready_, valid_, wstrb_: Registros en los que se guarda los datos de entrada si se activa la bandera de valid.

La estructura de la caché implementada consiste en una máquina de estados con seis estados: IDLE, CHECK_HIT, WRITE_MEM LOAD_MISS1, LOAD_MISS2 y READ_WRITE, los cuales se explicarán detalladamente a continuación. Por otra parte, para la implementación de conteo de HIT y MISS se usó un módulo externo de detector de flancos, de forma que cada vez que la señal de hit y miss cambiara, esta sumaba uno a un contador de salida.

La cache también cuenta con dos parametros por los que se les ingresa las dimensiones a las estructuras de la cache, y estos parámetros son CACHE_SIZE (en KB) y BLOCK_SIZE (en bytes).

3.2.1. Estado IDLE.

El estado de IDLE se encarga de esperar a que se reciban datos validos desde el CPU, en cuyo caso, se salta al estado CHECK_HIT.

3.2.2. Estado CHECK_HIT.

Este estado extrae el index, offset y tag de la dirección actual enviada por el CPU, posteriormente compara el tag obtenido con el tag guardado en el registro index del array Tag1, en caso de tenerse match, se activa la señal de gotHit y se apaga la de gotMiss. Posteriormente, se verifica si el dato se debe escribir o leer, en caso de leerse, se envía el dato correspondiente al index y offset del way1 a rdata, y se activa ready; de necesitarse escribir el dato, se escribe en la casilla correspondiente al index y offset del set 1, y se activa el bit dirty del registro index en cacheDirty1, y por último se salta al estado IDLE. Si no se tiene un match, se hace la comparación pero con el tag correspondiente del way 2, en el array tag2, en caso de tenerse un match se repite el proceso pero esta vez para el way 2.

Si no se consigue un match con ninguno de los dos ways, se actualizan las banderas de gotHit y gotMiss, se calcula el valor de mod, y se salta al estado WRITE_MEM.

```
CHECK_HIT: begin

offset = address_[OFFSET_BITS-1:0];
index = address_[INDEX_SIZE+OFFSET_BITS-1:OFFSET_BITS];
tag = address_[31:INDEX_SIZE+OFFSET_BITS];


if (tag == cacheData1[index][offset]) begin

        cacheLRU[index] = 'b0;
        gotHit = 'b1;
        gotMiss = 'b0;
        if (wstrb_ == 'b0000) begin
                rdata = cacheData1[index][offset];
                ready = 'b1;
        end     else begin
                cacheData1[index][offset] = wdata_;
                cacheDirty1[index] = 'b1;
        end
               NXT_ST = IDLE;
end else begin
if (tag == cacheData2[index][offset]) begin

        cacheLRU[index] = 'b1;
        gotHit = 'b1;
        gotMiss = 'b0;
        if (wstrb_ == 'b0000) begin
                rdata = cacheData2[index][offset];
                ready = 'b1;
        end     else begin
                cacheData2[index][offset] = wdata_;
                cacheDirty2[index] = 'b1;
        end
        NXT_ST = IDLE;
```

```
end else begin
        gotMiss = 'b1;
        gotHit = 'b0;
        mod = address/4 % (BLOCK_SIZE/4);
        NXT_ST = WRITE_MEM;
end
end
end
```

3.2.3. Estado WRITE_MEM.

Al no tenerse un match, es necesario ingresar un dato desde memoria a la cache, sin embargo, al ser una cache write back, se debe verificar si el dato fue escrito antes de sacarlo de la cache, por lo que se verifica si el bit de dirty del bloque a reemplazar se encuentra sucio, en cuyo caso se procede a enviar los datos a memoria.

Para esto se envían los datos uno por uno por los puertos de salida dirigidos a memoria, de forma que se activa el bit de valid a mp, se ingresa indica con el wstrb que se va a escribir y se envian los datos a la dirección del {index, tag, 0} hasta la dirección {index, tag, block_size-1}, de la misma forma se indexan y envían los datos, con cada dato enviado, se suma 1 al contador i, si este contador aumenta el valor del block size, se devuelve a cero.

Si el último dato es enviado, se limpia el bit de dirty del set y se salta al estado LOAD_MISS 1 en caso de haberse reemplazado un bloque del way 1, y LOAD_MISS en caso de haber sido del way 2.

```
WRITE_MEM: begin

if(cacheDirty2[index] && !cacheLRU[index]) begin

        if(!ready_mp) begin
                address_mp = {tag2[index],index,i};
                valid_mp = 'b1;
                wdata_mp = cacheData2[index][i];
                instr_mp = instr;
                wstrb_mp = 'b1111;

                i = i + 1;

                if(i == BLOCK_SIZE/4-1) begin
                        cacheDirty2[index] = 0;
                        NXT_ST = LOAD_MISS2;
                end

                if(i>=BLOCK_SIZE/4)      begin
                        i = 'b0;
                end
        end
end
```

```
if(cacheDirty1[index] && cacheLRU[index]) begin

        if(!ready_mp) begin
                address_mp = {tag1[index],index,i};
                valid_mp = 'b1;
```

3.2.4. Estado LOAD_MISS1

Este estado se encarga de cargar los bloques solicitados desde la memoria principal, comienza por verificar que el bit de dirty sea 0 y que sea el way 1 el que se deba reemplazar por medio de LRU, despues de esto, y de forma similar al anterior estado, se solicita uno por uno los datos desde la memoria principal, para lo que se envía un valid activo y un wstrb con ceros, indicando la lectura, despues se itera desde la dirección {address-mod} hasta {address-mod+BLOCK_SIZE-1}.

Si la señal de ready de la mp esta en cero, se activa waitData, indicando espera de dato.

Si se tene la señal de ready activa y waitData activo, significa que se tiene el dato solicitado, el cual se guarda en el espacio correspondiente de la cache, y se desactiva waitData.

Una vez que todos los datos han sido recibidos, se actualiza la bandera de LRU, se actualiza el tag del set que se acaba de traer de memoria, se reinicia el contador i, y se salta a READ_WRITE.

```
LOAD_MISS1: begin

if(!cacheDirty1[index] && cacheLRU[index]) begin

        if(!ready_mp && !waitData) begin
                address_mp = address-mod+i;
                valid_mp = 'b1;
                instr_mp = instr;
                wstrb_mp = 'b0000;

                i = i + 1;

                if(i == BLOCK_SIZE/4-1) begin
                        cacheLRU[index] = 0;
                        NXT_ST = READ_WRITE;
                end

                if(i>=BLOCK_SIZE/4)       begin
                        i = 'b0;
                end
        end

        if(!ready_mp) begin
                waitData = 'b1;
        end

        if (ready_mp && waitData) begin
```

```
                cacheData1[index][(BLOCK_SIZE/4)-mod-1+i] = rdata_mp;
                waitData = 'b0;
```

3.2.5. Estado LOAD_MISS2

Este estado se encarga de hacer lo mismo que el estado LOAD_MISS1 pero en este caso para traer datos desde memoria al way 2 de la cache.

3.2.6. Estado READ_WRITE

Por último, se tiene el estado de READ_WRITE, el cual se encarga de enviar los datos al CPU o escribirlos, una vez que se trajeron los datos necesarios de la memoria, para esto se comienza verificando cual de los dos ways debe ser leído (en cual de los dos e acaba de cargar el dato), se comienza por actualizar el LRU, despues de esto se verifica si se debe leer o escribir, en cuyo caso para una lectura se envía el dato a CPU y se activa la señal de ready; si es una escritura, se escribe el dato correspondiente y se activa el bit de dirty del set. Por último, se salta a IDLE y se termina con la lógica de la cache.

```
READ_WRITE: begin

if (tag == cacheData1[index][offset]) begin

        cacheLRU[index] = 'b0;

        if (wstrb == 'b0000) begin
                rdata = cacheData1[index][offset];
                ready = 'b1;
        end else begin
                cacheData1[index][offset] = wdata;
                cacheDirty1[index] = 'b1;
                end
                NXT_ST = IDLE;
end else begin
if (tag == cacheData2[index][offset]) begin
        cacheLRU[index] = 'b1;
        if (wstrb == 'b0000) begin
                rdata = cacheData2[index][offset];
                ready = 'b1;
        end     else begin
                cacheData2[index][offset] = wdata;
                cacheDirty2[index] = 'b1;
        end
        NXT_ST = IDLE;
        end
end
```

3.2.7. Implementación de la cache.

Al simular la cache implementada en esta práctica, se tiene que el procesador picorv32 instanciado activa la señal de trap, por lo que entra en un estado donde no interactua con la cache, de forma qu no envía señales de valid, o las direciones a acceder o datos a guardar, por lo que no se pudo probar la misma. Se intento usar un sistema de recepción de datos simmilar al de system.v de forma que no presentara problemas, sin embargo, esto no fue suficiente para hacer que esta no entrara en este estado, por lo que no fue posible efectuar la simulación de la caché.

A continuación se presenta los resultados de la síntesis de esta:

3.3. Ejercicio 3

Para esta parte, se debe instanciar la cache, definiendo los parámetros de CACHE_SIZE y BLOCK_SIZE primero como 2 y 8, despues como 2 y 16, posteriormente como 4 y 8 y por último como 4 y 16, respectivamente, de esta forma se cambian las dimensiones de las estructuras internas de la cache, las cuales son parametrizables, posteriormente, se obtienen los resultados de los contadores de HIT y MISS, se suman en un mismo registro que tenga el total de los accesos a memoria, y se dividen los resultados de HIT y MISS entre el total de accesos, obteniendo el hitRate y el missRate.

Al no ser posible la simulación por el comportamiento de la bandera de trap en el procesador, no es posible hacer la prueba del rate de hit y rate de miss, por lo que se muestra el resultado de la síntesis de cada caché parametrizada con estas variables.:

Tamaño del bloque	Tamaño de la cache	Miss rate	Cantidad de slices
2 palabras	2048	-	36150
4 palabras	2048	-	36200
2 palabras	4096	-	-
4 palabras	4096	-	36680

Tabla 1: Comparación entre caches de diferente tamaño de bloque y total

4. Repositorio

Enlace a repositorio de GitLab: https://gitlab.com/YeisonRP/Lab_digitales_2_B56074_B57082

5. Conclusiones y recomendaciones

A continuación se presentan las principales conclusiones encontradas en este experimento:

- Es posible crear una lista enlazada desde el firmware con el fin de llenar la memoria física que será cargada en la FPGA, además de imprimir números de la lista en los display de 7 segmentos.
- Se pudo observar que la implementación de la caché era sintetizable y parametrizable, aunque la simulación no se pudo ejecutar por el trap que generaba el procesador.
- Vemos que al aumentar el tamaño de la caché o de los bloques de la misma, se aumenta la cantidad de slices presentes en la síntesis.

Seguidamente se mencionarán algunas recomendaciones para facilitar la recreación de este experimento.

- Es muy recomendable el implmentar este tipo de dispositivos como una máquina de estados, ya que de esta forma la lógica se vuelve más intuitiva y organizada en comparación a hacerlo en código lineal.
- El comportamiento de un dispositivo como una caché conlleva una complejidad considerable, por lo que es recomendable dividir la lógica de esta en diferentes etapas, y de esta forma tener una mejor organización en el manejo de banderas o guardado y actualización de variables.
- Estudiar con mucho detalle la temporización en cada una de las señales de entrada/salida del picorv32, debido a que si no se conocen estos datos la comunicación entre la caché y el procesador se vuelve una tarea muy complicada.
- Diseñar un testbench en el anteproyecto que envíe datos a la caché con el fin de no utilizar el picorv32 para comprobar el funcionamiento de la implementación.