VHDL Programming Assignment #5

Architecture of Computers (CSI3102-01), Spring 2020

Welcome to the fifth (and the last) programming assignment of the Architecture of Computers (CSI3102-01) course at Yonsei University! In the previous assignments, we mainly focused on how to implement the processor; we started with simple digital circuits, and then merged the digital circuits into single-cycle and five-stage pipelined processors.

In this programming assignment, we will implement **write-back caches**. Caches play an important role in modern processors as they greatly reduce the memory access latency by exploiting spatial and temporal locality. Especially, write-back caches minimizes the amount of main memory accesses by writing the contents of dirty cache lines back to main memory only when the dirty cache lines get evicted. In addition, this assignment is intended to demonstrate how we can exploit VHDL's `generics` to reuse the same piece of VHDL code. With `generics`, you can instantiate different write-back caches having different sizes, associativity, block sizes, address sizes, etc, without writing multiple pieces of VHDL code.

The deadline for this assignment is **11:59pm on June 21st, 2020 (Sun)**. No extension will be given.

1. Direct-Mapped Caches (50 Points)

Extend `Cache.vhd` to support instantiation of direct-mapped caches.

Direct-mapped caches allow a memory block to be mapped to only one cache block. The target cache block for an access to a memory address is calculated by splitting the memory address into three parts: tag, index, and offset. The index field specifies which of the cache blocks the memory address maps to, and the offset field indicates which of the byte(s) of the target cache block the memory address refers to.

Figure 5.10 in the textbook shows the datapath for a 4-KiB direct-mapped cache which supports 64-bit memory addresses and whose block size is one word (i.e., 32 bits):

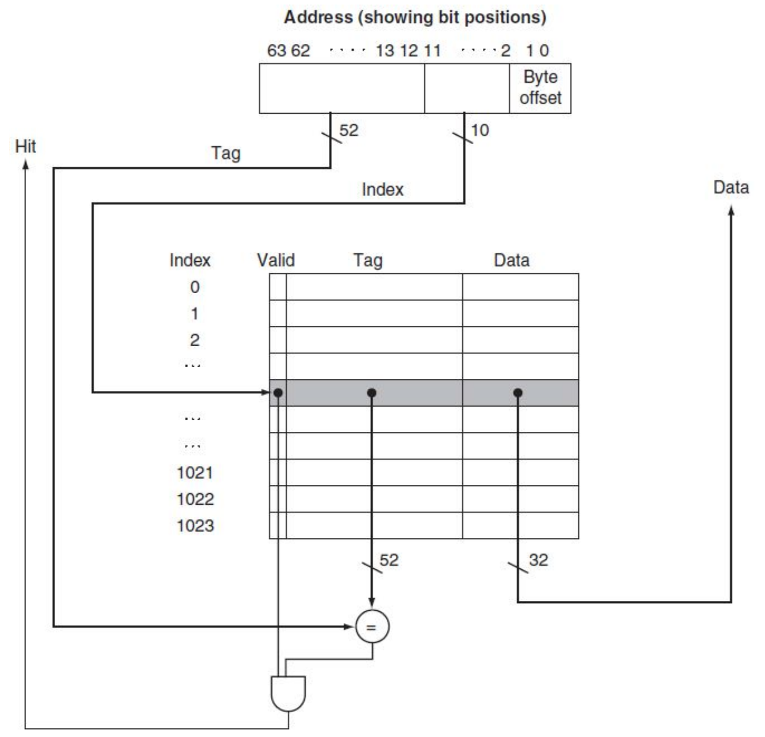


Figure 5.39 in the textbook shows the control for controlling direct-mapped write-back caches:

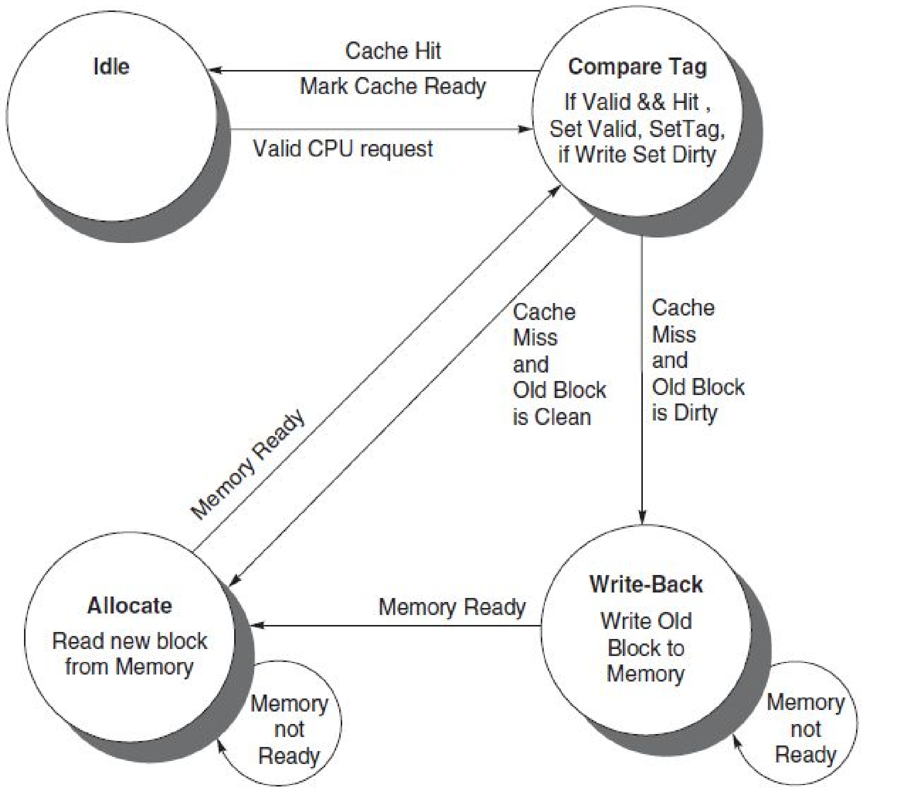

Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

FIGURE 5.39 Four states of the simple controller.

Assume that transitioning from one state to another takes 1 clock cycle.

Now, let's have a look at the skeleton VHDL code. `Cache.vhd` defines an entity named `Cache`, and the entity has several `generics` and `ports`.

`generics`:

- `addrSize`: the size (the # of bits) of the (byte) memory address
- `accessGranularityInBytes`: the cache access granularity in bytes
- `totalSizeInBytes`: the size of the cache (counts the # of bytes for storing data)
- `numWays`: the associativity of the cache
- `blockSizeInBytes`: the size of each cache block

`ports`:

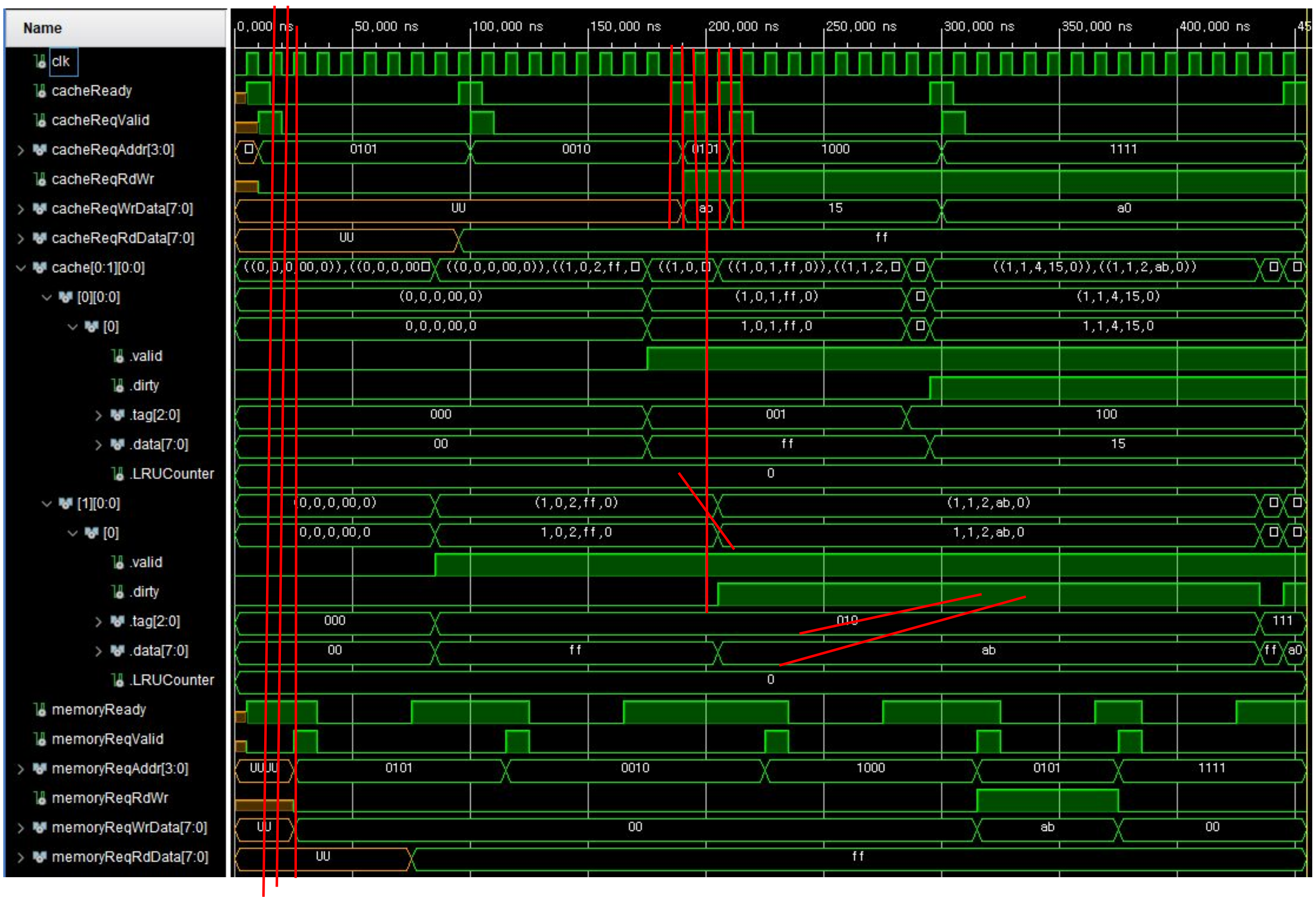
- `clk`: the clock signal
- `ready`: 1 if the cache is ready to accept a new request, 0 otherwise.
- `reqValid`: 1 if there is a valid request issued by the processor
- `reqAddr`: the memory address the processor wishes to access
- `reqRdWr`: 0 if the request is a read request, 1 if the request is a write request
- `reqWrData`: the data to write to the cache when `reqRdWr` = 1
- `reqRdData`: the data returned by the cache when `reqRdWr` = 0

To issue a memory request to the cache, signals reqValid, reqAddr, reqRdWr, and reqWrData should be set. When the cache is ready, it starts serving the memory request by transitioning from the `Idle` state to the `Compare Tag` state. After going through a series of the control states, the cache becomes ready again. When the cache becomes ready again, the cache should return the requested data through the `reqRdData` signal. The data returned through `reqRdData` signal should be valid for only one clock cycle.

Note that your implementation of architecture `Behavioral` for entity `Cache` should adapt to changes in the values of the `generics`. For example, if `addrSize` is set to 4, your cache should handle 4-bit memory addresses. In addition, your implementation of direct-mapped caches should be driven by the `clk` signal; any behaviors by direct-mapped caches should be triggered by the rising edges of the `clk` signal. As an example, only one transition between the four control states can occur within a single clock cycle.

Example #1:

This example demonstrates a 4-byte direct-mapped cache which accepts 4-bit memory addresses. The cache block size is 1 byte, and the processor accesses the cache by issuing 1-byte requests. The initial values of the data memory is set to `0xFF`, and the memory access latency is 4 clock cycles.



See "csi3102-assn5-prob1-ex1.jpg" for a more clear image.

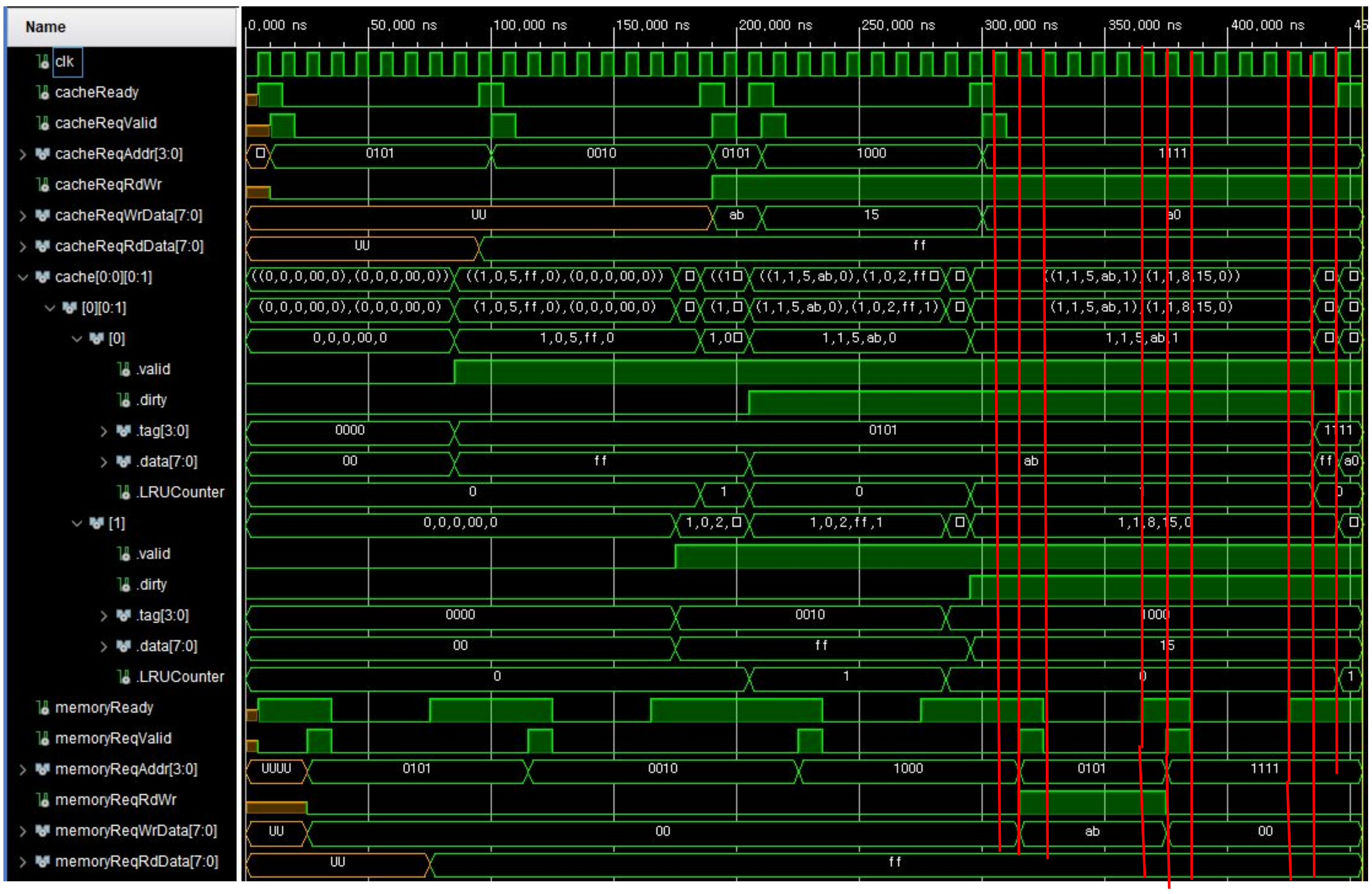
2. Set- & Fully-Associative Caches (50 Points)

Further extend Cache.vhd to support instantiation of set-associative and fully-associative caches.

Unlike direct-mapped caches, we need a cache replacement policy for set- and fully-associative caches as a memory block can be mapped to multiple cache blocks. Among various cache replacement policies, you should implement **Least-Recently Used (LRU) policy**. The LRU policy associates a counter to each cache block. The counter of a cache block tracks how long the cache block has not been accessed. This counter is defined as `LRUCounter`; when a cache block is accessed, the `LRUCounter` for the cache should become 0 to indicate that the cache block is the most-recently used cache block. The `LRUCounters` of the other cache blocks which belong to the same cache set should be incremented by one.

Example #1:

This example demonstrates a 4-byte fully-associative cache which accepts 4-bit memory addresses. The cache block size is 1 byte, and the processor accesses the cache by issuing 1-byte requests. The initial values of the data memory is set to `0xFF`, and the memory access latency is 4 clock cycles.



See “csi3102-assn5-prob2-ex1.jpg” for a more clear image.