ECE 571: INTRO TO SYSTEMVERILOG

Winter 2021

Final Project Report

Cache Controller for a Data Cache

Group #4

Durganila Anandhan (anandhan@pdx.edu)

Erik Fox (erfox@pdx.edu)

Manjari Rajasekharan (manjari@pdx.edu)

Prem Sai Gudreddygari Chandra (premsai@pdx.edu)

Introduction

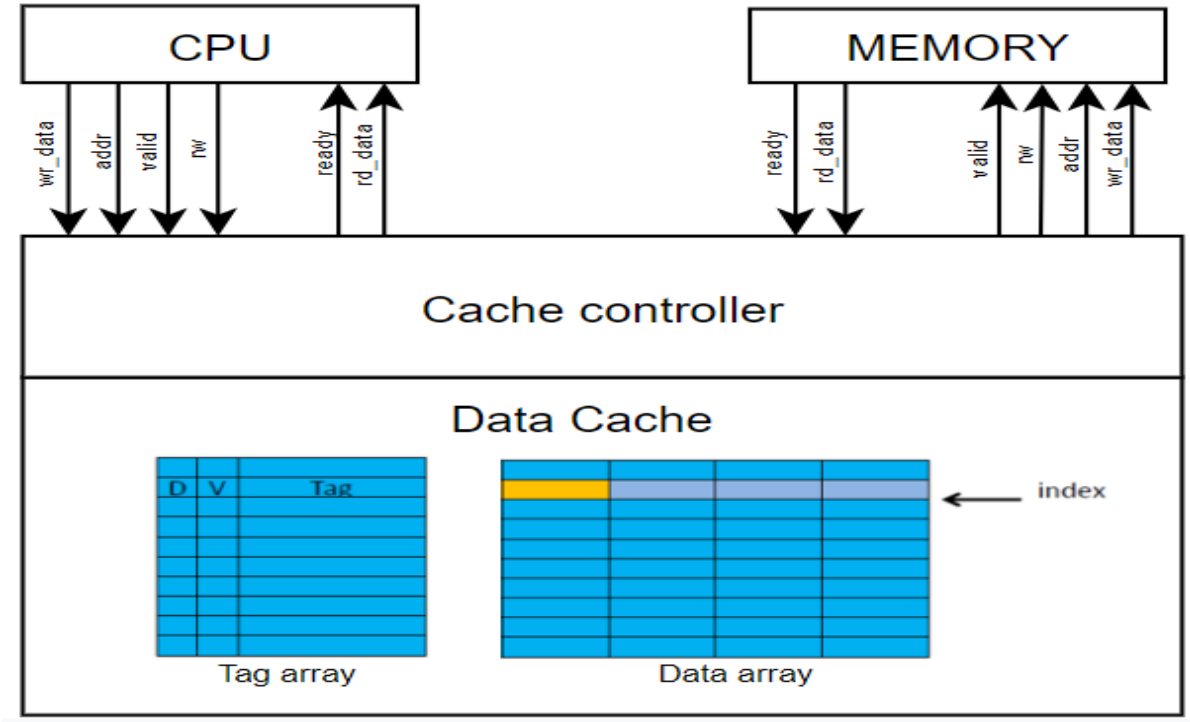
The aim of this project is to upgrade the existing design of a cache controller for a data cache and verify its functionality using Systemverilog.

Design Decisions

We used a design for a direct mapped cache controller described in System Verilog from "D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* ARM® edition, Elsevier/Morgan Kaufmann, 2017, sec. 5.9,5.12."

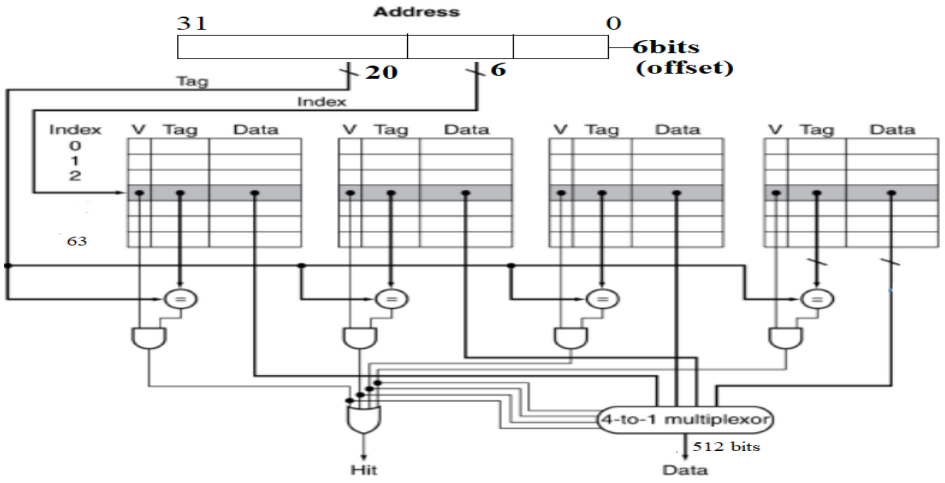
Key characteristics of the Existing Design:

1. Direct-mapped cache
2. Write-back using write allocate
3. Block size is four words (16 bytes or 128 bits).
4. Cache size is 16 KiB, so it holds 1024 blocks.
5. 32-bit addresses
6. The cache includes a valid bit and dirty bit per block

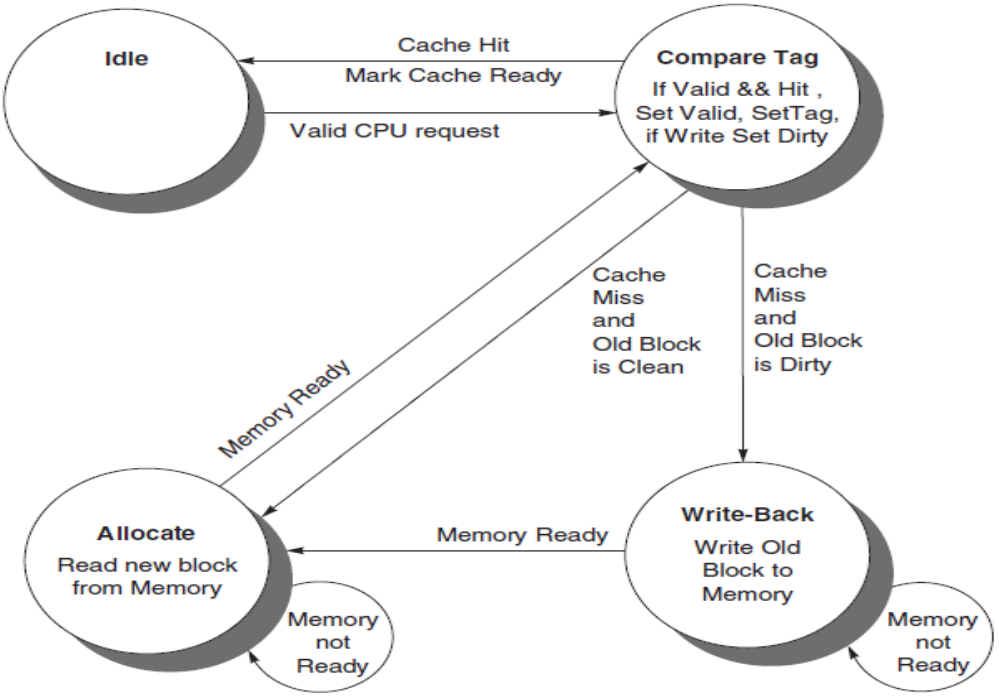


Upgrades to the existing Design:

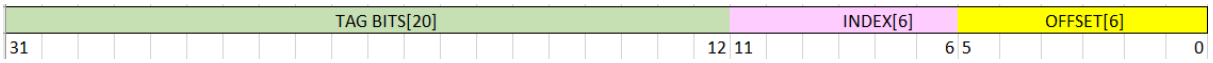
1. 4-way set associative cache
2. Pseudo LRU replacement policy
3. Block size: 64 bytes
4. New cache address mapping
5. Parametrization of block size and address mapping of cache controller



Four States of the Cache Controller:



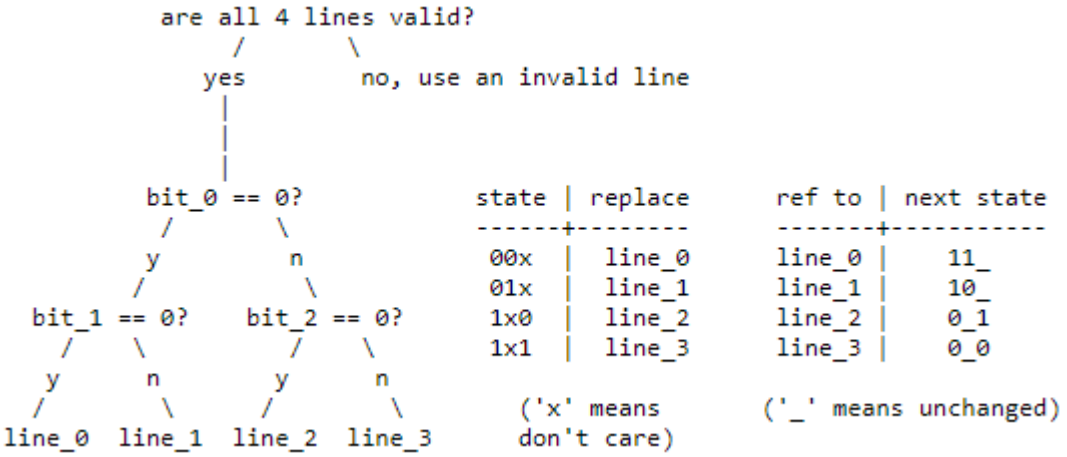
Address Mapping:



Cache size: 16KiB
Block size: 64 bytes => 6 bits required
Number of total blocks: 16KiB/64B = 256
Associativity: 4 => 4 ways in 1 index
Number of sets => 256/4 = 64 => 6 bits required
Address bits = 32 => Tag bits = 32 -6-6 = 20 bits

Replacement Policy

As a part of the upgrade of our cache to four-way set-associative, we had to choose and implement a replacement policy. We chose a pseudo LRU replacement policy. PLRU policies use less memory than LRU, three bits per index versus six for a four-way set associative cache, are easier to implement, and still provide strong performance. Below is a visual description of the PLRU policy we chose. To properly integrate this into our program we had to tailor our implementation of the policy for two different scenarios: a cache hit and a cache miss. For a cache miss, the cache controller needs to be told which way to evict and the PLRU state needs to be updated based on the way evicted, and on a cache hit, the PLRU state needs to be updated to reflect the way being referenced.

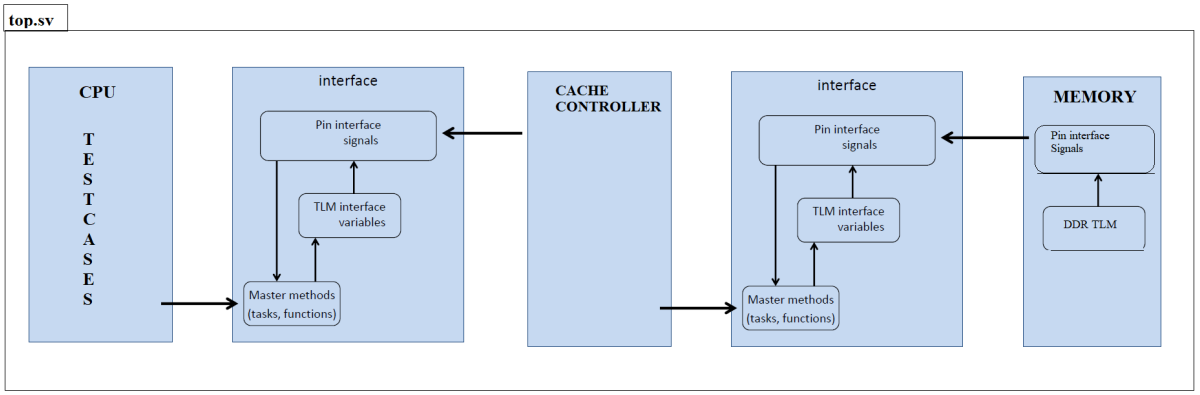


state	replace
00x	line_0
01x	line_1
1x0	line_2
1x1	line_3

ref to	next state
line_0	11_
line_1	10_
line_2	0_1
line_3	0_0

Verification Methodology:

Model Used:



We have designed a Bus Functional Model of CPU and Memory to accurately reflect the I/O level behavior of the cache controller.

CPU - Cache Interface:

For verification purposes, a CPU module was created to send test vectors to our cache. To connect the two modules, we developed an interface with leader (cpu) and follower (cache) modports. In addition to these modports, the CPU - Cache interface contains read and write tasks which are imported as a part of the leader modport. The read task works placing an address on the address bits, de-asserting the rw bit, and pulsing the valid bit. It then waits for the ready signal to be sent from the follower, indicating that valid data is on the data bits. The write task works by placing data and an address on the data and address bits, asserting the rw bit, and pulsing the valid bit. Write then waits for the ready signal, indicating that the cache is ready for the next instruction.

CPU Model

A transaction level model of the CPU is designed to inject stimulus (read-write) to the cache controller. We have defined 6 six types of stimulus both directed and random to verify cache controller functionality. These are:

1. Simple write -read

   Issue continuous write and reads to all the words in the first cache line. Then, modify only a few words and ensure that only intended words are modified.

2. Cache eviction

   Issue five continuous writes to the same index which causes eviction of first access to the index. Then, read is issued to make sure the evicted cache is written back to memory and is possible to read.

3. Random write-read

   Issue writes and reads with random data.

4. Random address and random data.

   Issue writes and reads with random data and address. This may cause access to non-existent memory regions, such access captured and flagged using assertions.

5. Constraint random address and random data.

   Issue writes and reads to only addresses defined in memory with random data.

6. Random cache eviction

   Same as testcase 2 but, tag bits and offset are randomized and index kept the same.

One of the above test cases can be selected during simulation using the plusarg “TESTCASE”.

Cache-Memory Interface:

A memory module was created to mimic the behavior of main memory responding to our cache. To connect the two modules, we developed an interface with master (cache) and slave (memory) modports.

Memory Model:

The memory was modelled using case statements to describe the various states of the memory : Idle, read, write and respond. In “Idle” state, the memory waits for a valid bit on the bus to initiate a transaction. Depending on rw signal, it goes to either “Read” or “Write” State. In Read and Write states, a delay is specified after which the complete 512 bits of data is obtained. Then it moves to the “respond” state to send the ready signal to the cache controller, indicating that valid data is on the data bits.
It also models access delay and burst to memory (DDR4).

Conclusion

We have implemented a simulator for a cache controller for a 4-way set associative, write back, write allocate cache with LRU replacement policy. The functional verification of the design was done and we have successfully demonstrated our knowledge on various systemverilog constructs from ECE 571.d2

REFERENCE

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* ARM® edition, Elsevier/Morgan Kaufmann, 2017, sec. 5.9,5.12.