Lab 6: Cache Lab

EE312 Computer Architecture

Professor: John Kim

TA: Jiho Kim (EMAIL: jihokim@kaist.ac.kr)

Due data : 2019.12.12 , 13:00 (before class)

1. Overview

Lab 6 is intended to give you hands-on experience in designing a cache microarchitecture. Based on the concepts and skills you have acquired through the lab assignments, you are now ready to implement the cache. Through this lab assignment, you will be gaining an in-depth understanding on the fundamentals of designing how the cache interacts with the processor microarchitecture.

2. Backgrounds

Cache

During this lab, you will implement a cache to mitigate the performance gap between the CPU and memory. In real world applications, a memory operation is much more expensive than an arithmetic operation. In general, memory accesses can take hundreds of cycles so we discussed the notion of memory hierarchy and where cache comes into play to resolve this performance gap. Cache has a much smaller capacity compared to the main memory but exhibits much shorter access latency. While the cache cannot save all the values that a program requires (as the programmer-visible memory address space is much larger than the cache capacity), it is possible for a target program to satisfy a significant fraction of its instruction/memory accesses from the cache thanks to the temporal and spatial locality existent in memory accesses.

Cache Structure

Cache has two data storage banks, the “tag” bank and the “data” bank. The “data” bank stores the actual data value itself it has fetched from the main memory (in *cache-lin*e, also called *cache-block* granularity). A single cache-line is assumed to contain four sequential words. The “tag” bank stores the necessary subset of the effective memory address information to utilize as an identifier to verify whether the cache line actually contains the value we are looking for.

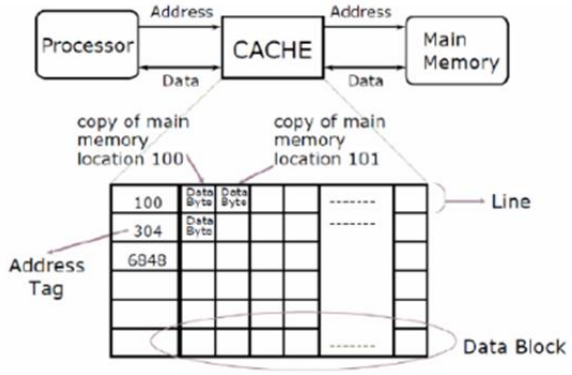

Figure 1. The Cache Structure

Cache Associativity

Because the directed-map cache can suffer significantly from conflict misses, we discussed set-associative and fully-associative caches as a more sophisticated (but also higher overhead in terms of implementation and access latency) microarchitecture design point that helps reduce address conflicts for a given set, improving overall cache efficiency. The figure below shows a 2-way set-associative cache. **In this lab, your goal is to design a simple "direct-mapped" cache.**

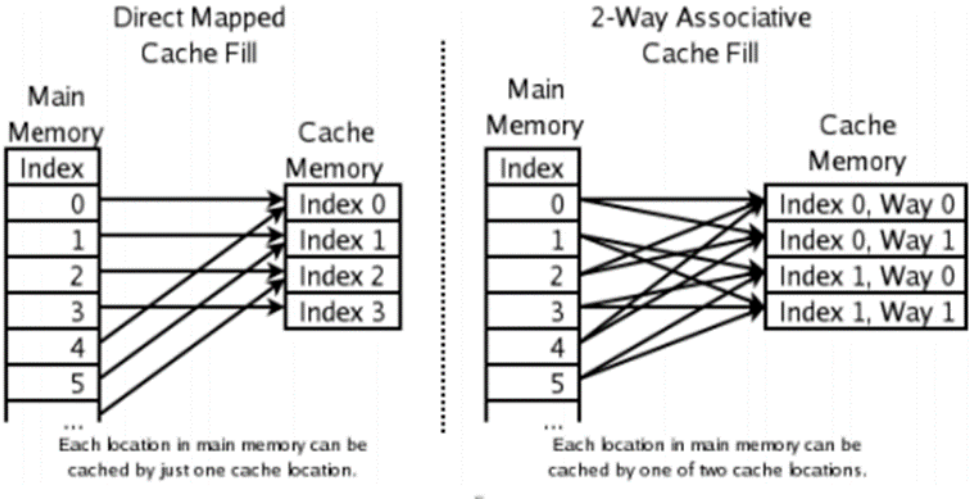

Figure 2. Cache Associativity

Cache Replacement and Write Policy

The cache replacement policy chooses which cache line to evict whenever necessary. The "random", "LRU", "FIFO" are some of the policies we have discussed in class. Each policy has its pros and cons. We also discussed two write-hit policies, "write-through" and "write-back", which dictates how write-hits should be handled upon a write hit operation. Two write-miss policies have also been discussed, "write-no-allocate" and "write-allocate", which is the cache policy that decides whether the cache-line that just missed (for a write operation) should be brought back into that cache or not. Because the cache microarchitecture you'll be designing is a direct-mapped cache, **there is "no" particular cache replacement policy you should be implementing**. In terms of cache write-hit / write-miss policies, you are required to implement the **'write-through'** and **'write-allocate'** policies (no exceptions).

3. Files

As mentioned multiple times during the lecture, you are required to implement the cache microarchitecture on top your pipelined CPU implementation from Lab 5. If you were not able to finish your pipelined implementation, it is okay to utilize the multi-cycle implementation from Lab 4 for this lab (you are not allowed to use the single-cycle CPU design though). If you were not able to implement either one of these (multi-cycle or pipelined CPU), implement a functionally correct Verilog code that contains the cache microarchitecture, without any one of the processor related codes, and send it to the TAs through email as a single Verilog file (cache.v). After verifying the sanity of the codes, the TAs will provide a sample pipelined CPU implementation that you can work with to test your cache implementation.

(Note) If you successfully finished Lab 4/5 (which we will check based on your prior submissions at KLMS), you are not allowed ask for a sample pipelined CPU implementation. Such option is only available for those who haven't been able to finish prior labs successfully but are motivated enough to work on the cache design as a stand-alone lab.

4. Cache Lab

In *Lab 6*, you are required to implement data cache.

Your implementation **MUST** comply with the following rules:

1. RISC-V ISA (RV 32I)
2. **Instruction cache**
    - **A. No instruction cache**
    - **B. Instructions can be fetched within one cycle**
3. **Data cache**
    - **A. Single-level directed-mapped cache which is part of the processor**
    - **B. Capacity:**
        - **i. (Excluding the tag bank)**
        - **ii. 128B cache capacity that can house 32 words (128 bytes / 4 bytes per word)**
        - **iii. 4 words per cache-line**
        - **iv. No replacement policy needed (because this is a direct-mapped cache)**
        - **v. Write-through**
        - **vi. Write-allocate**
        - **vii. Blocking cache**
    - **C. Latency: 1 cycle for cache hit**

D. Memory access granularity : 4 words

4. Memory model

A. Instruction memory can be accessed within one cycle

B. Data memory can be accessed within 6 cycles

i. You fetch 4 words into cache in 6 cycles

5. You need to implement only below instructions

A. JAL

B. JALR

C. BEQ, BNE, BLT, BGE, BLTU, BGEU

D. LW

E. SW

F. ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI

G. ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND

Terminal condition

Since we don't implement instructions which are used to transfer control to the operating system, we set a flag instruction to quit the program. If you get the instruction sequences "0x00c00093 //(addi x1, x0, 0xc) and 0x00008067 //(jalr x0, x1, 0)", you should halt the program. HALT output wire should be set to 1.

Simulation

To test your CPU, you need to implement two additional output, which are *NUM_INST* and *OUTPUT_PORT*.

1. NUM_INST: the number of executed instructions
2. OUTPUT_PORT: the output result,
    a. If the instruction has a destination register (*rd*), then the value that is supposed to be written in the destination register should also be written to *OUTPUT_PORT*.
    b. If the instruction is a branch instruction, 1 is written to *OUTPUT_PORT* if taken; otherwise, 0 is written.
    c. If the instruction is a store instruction, the target address of the store instruction is written to *OUTPUT_PORT*.

5. Grading

The TAs will grade your lab assignments with three *testbench* files in the *testbench folder* that are

already given to you. Your score for this lab is going to be proportional to how many test cases you pass. The implementation takes up 95% of the total score, and the report takes up the remaining 5%. You implement directed-mapped data cache and you can choose any replacement and write policy. **However, if your memory model and cache do not meet the requirement, you will get zero points.** You need to **explicitly** describe your replacement policy, write policy, hit ratio and impact of your cache.

7. Lab Report Guidance

You are required to submit a lab report for every lab assignment. You can write your report either in Korean or English. We don't want you to waste your time writing a lab report. Please keep the report **short**. **Three to four pages** are enough for the report unless you have more to show. You don't need to have too much concern about the report.

Your lab report **MUST** include the following sections:

1. Introduction
    a. *Introduction* includes what you think you are required to accomplish from the lab assignment and a brief description of your design and implementation.
2. Design (Try to assign most of your lab report pages explaining your design)
    a. *Design* includes a high-level description of your design of the Verilog modules (e.g., the relationship between the modules).
    b. Figures are very helpful for the TAs to understand your Verilog code.
    c. The TAs recommend you to include figures because drawing the figures helps you how to *design* your modules.
3. Implementation
    a. *Implementation* includes a detail description of your implementation of what you design.
    b. Just writing the overall structure and meaningful information is enough; you do not need to explain minor issues that you solve in detail.
    c. **Do not copy and paste your source code.**
4. Evaluation
    a. *Evaluation* includes how you evaluate your design and implementation and the simulation results.
    b. *Evaluation* must include how many tests you pass in vending_machine_TB.v
5. Discussion
    a. *Discussion* includes any problems that you experience when you follow through the lab assignment or any feedbacks for the TAs.
    b. Your feedbacks are very helpful for the TAs to further improve *EE312* course*!*

6. Conclusion

    a. *Conclusion* includes any concluding remarks of your work or what you accomplish through the lab assignment.

7. Requirements

You **MUST** comply with the following rules:

- You should implement the lab assignment in **Verilog**.
- You should name your lab report as **Lab6_YourName_StudentID.pdf**.
- You should compress the lab report, and source codes, then name the compressed zip file as **Lab6_YourName_StudentID.zip**, and submit the zip file on the KLMS.