# Lab 6: Cache Lab

EE312 Computer Architecture

Professor: Minsoo Rhu

TA (Lab 6): Jinha Chung, Head TA : Youngeun Kwon

EMAIL: kaist.ee312.ta@gmail.com

## 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.

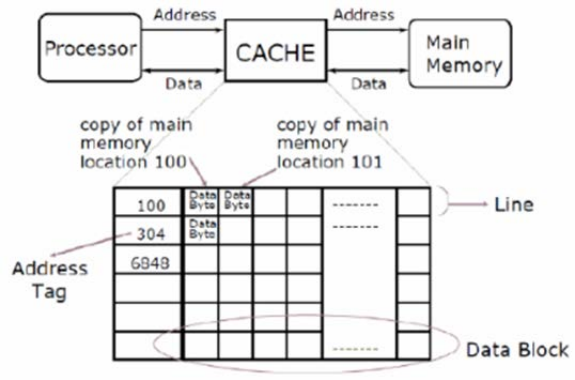


**Figure 1. The Cache Structure**

### Cache Associativity

Because the direct-mapped 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.**

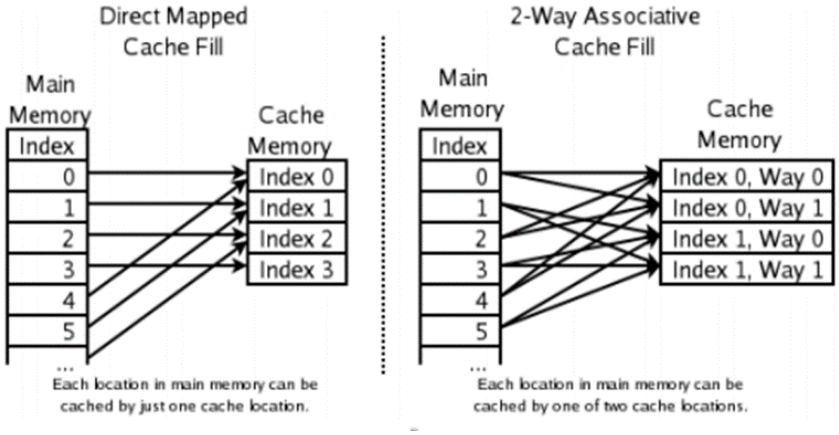


**Figure 2. Cache Associativity**

### Cache Replacement and Write Policy

The cache replacement policy chooses which cache-line to evict whenever necessary. The "random", "LRU", and "FIFO" are some of the policies we have discussed in class. Each policy has its pros and cons. We also discussed two white-hit policies, "write-through" and "write-back", which dictate 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 are the cache policies that decide 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

**You are required to implement the cache microarchitecture on top of 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). However, if you implement on top of your multi-cycle CPU implementation, your maximum score will be 80% of the maximum score you can get when implementing on top of the pipelined CPU**

### 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 (RV32I) + custom instructions
    A. Refer to RISC_V_101.pdf for the detailed descriptions about the custom instructions.
2. **Instruction cache**
    **A. No instruction cache**
    **B. Instructions can be fetched within one cycle**
3. **Data cache**
    **A. Single-level direct-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, 1 cycle for cache update**
    **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 8 cycles**
        **i. You fetch 4 words into cache in 8 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
    H. MULT, MODULO, IS_EVEN

**Elaborating on cache hit/miss latencies**

Here we provide some information so that the above information cannot be interpreted in different ways. In accordance with what has been mentioned above, cache access latency `CACHE_ACCESS_LATENCY` is 1 cycle, cache update latency `CACHE_UPDATE_LATENCY` is also 1 cycle, and memory access latency `MEMORY_ACCESS_LATENCY` is 8 cycles.

**Read accesses to cache**

1. A *read hit* will take 1 cycle to complete.
   Only cache access (read) takes place.

2. A *read miss* will take 10 cycles to complete.
   A cache read access incurs a `CACHE_ACCESS_LATENCY` latency, and read from memory incurs `MEMORY_ACCESS_LATENCY` of latency. Afterwards, the new value brought in from memory must be updated (written) to cache, taking `CACHE_UPDATE_LATENCY`. Thus, `CACHE_ACCESS_LATENCY + MEMORY_ACCESS_LATENCY + CACHE_UPDATE_LATENCY` is required for a read miss to complete.

**Write accesses to cache**

1. A *write hit* will take 9 cycles to complete.
   A cache write access incurs `CACHE_ACCESS_LATENCY` latency, after which a cache update will take place. Here, *instead of* taking another `CACHE_UPDATE_LATENCY`, we assume the access and update will be resolved in the same cycle. After the update has been taken care of, the value will also need to be stored into memory, because you are implementing a write-through cache. Memory write incurs `MEMORY_ACCESS_LATENCY` of latency. All these latencies are added together, because they are performed sequentially, amounting to a total of `CACHE_ACCESS_LATENCY + MEMORY_ACCESS_LATENCY`.

2. A *write miss* will take 18 cycles to complete.
   A cache write access incurs a `CACHE_ACCESS_LATENCY` latency. After the cache miss occurs, it will take `MEMORY_ACCESS_LATENCY` to access memory, Where the *cache-line containing the current word being accessed* is stored. Once the cache-line has been brought into cache, the word we were to write to is updated, causing a `CACHE_UPDATE_LATENCY` latency. But the updated cache-line must be written back to memory (because it is a write-through cache) and

this process takes another `MEMORY_ACCESS_LATENCY` of latency. Adding the latencies, `CACHE_ACCESS_LATENCY + CACHE_UPDATE_LATENCY + 2×MEMORY_ACCESS_LATENCY` is required for a write miss to be resolved.

**Memory (Mem_Model.v) interface**

Up until now, the memory movement granularity was 4B, and thus the in/out ports of the SP_SRAM_DATA module are also 4 bytes (32 bits). However, the data movement granularity between the cache and memory is same as the cache-line, which is 16 bytes (128 bits). This **DOES NOT** mean you may change the memory module interface. Rather, you should make the memory module and cache module communicate with each other inside the top file (RISCV_TOP.v) so that the cache can send to (and receive from) the memory module **one word at a time, but four times sequentially**, so that the total data moved in between the two will result in 16 bytes (4 words). This is possible, because the memory access latency `MEMORY_ACCESS_LATENCY` is longer than 4 cycles.

**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 two instructions** "0x00c00093 //(addi x1, x0, 0xc)" and "0x00008067 //(jalr x0, x1, 0)" **in a row**, 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 (Not all testbenches have the same portion). The implementation takes up 75% of the total score, and the report takes up the remaining **25%** (Refer to Section 6 for detailed evaluation metric). Nonetheless, even if you pass all the test, if your design contains flaws, **we will deduct your points accordingly**. If the total number of cycles is not the same as the reference implementation, you will also get a deduction (e.g., for a testbench without any load or store instruction, the number of cycles

should stay the same with the cycle of implementation that does not contain cache implementation). **If your memory model and cache do not meet the requirement, you will get zero points.** In the report, you need to **explicitly** describe where your cache module is implemented in your code, your write policy (*write-through* and *write-allocate*), hit ratio and impact of your cache (cycle information) in the report.

Also, there are some guidelines that can affect your grades.

- **Do not** use "blocking operator" in the sequential logic part (e.g., always @(posedge clk)). Use non-blocking operator for sequential logic and blocking operator for combinational logic. If you don't abide by this rule, there will be a deduction.
- **Do not** use delay operator (e.g., #50, #100). If you use "delay operator" in your code (except the given testbench code), you will automatically get zero points for that lab assignment.
- **Do not** use "wait()" function for your design. We will not give any points for a design that is using "wait()" function.

### 6. 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 that you 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. **Report the number of cycles with cache and without cache and compare them.**

   c. **The waveform screenshots of ModelSim in each of the cases of different cache accesses, read miss, read hit, write miss, and write hit. Each screenshot takes up 5% of the total score.**

   d. *Evaluation* must include how many tests you pass in the *testbench folder*.

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**.
- Since we do not provide the template code this time, you may implement cache directly on top of your pipelined CPU. But you may only modify the **TODO** parts of the (previously) provided template, while you are free to add additional modules and files, as long as you abide by our stated rules.
- You should name your lab report as **Lab6_YourName1_StudentID1_YourName2_StudentID2.pdf**.
- You should name your honor pledges **HonorPledge_YourName1_StudentID1.pdf** and **HonorPledge_YourName2_StudentID2.pdf**.
- You should compress the honor pledge(s), lab report, simulation results, and source code, then name the compressed zip file as **Lab6_YourName_StudentID.zip**, and submit the zip file on KLMS. *All* names of your submitted files, *including* your zipped file, **should not contain any spaces in them**. If you need to put a space in the name, substitute it with an underscore ("_") instead. You **will be penalized** if you fail to comply with this rule.

## 8. FAQ

- The write-through and write-allocate write policies may not look like a smart choice compared to what you have learned in class ("write-back + write-allocate" and "write-through + write-no-allocate") but the proposed write policies ("write-through + write-allocate") will make the implementation easier, and we make it clear that there will be no exceptions for implementing different write policies.
- We decided not to provide the expected execution cycle of the test benches.