# Computer Architecture
# Lab 5: Cache

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

revision: 11-17-2025

In this lab, you will enhance your pipelined processor model by adding a data cache (D-cache) and an instruction cache (I-cache). You will be provided with a Verilog testbench and a functional-level (FL) cache bypass module, which will serve as a "gold standard" for verifying cache functionality. Use this FL cache bypass module to validate and troubleshoot your own cache implementation, helping you ensure correctness in your design. As part of the lab requirements, you will implement both baseline and alternative cache designs, verify them through an effective testing strategy, and evaluate their performance through comparison.

This lab will provide you with experience in:

- Fundamental principles of memory system design
- Designing complex finite-state machine (FSM) controllers for cache
- Microarchitectural strategies for implementing cache associativity

## 1 Reminder

- **Submission Deadline:** 12/8 (Mon) 11:59 p.m.
- Please refer to Lab 0 for the academic integrity requirements.
- **No sharing or distribution of lab materials is allowed.**

## 2 Setup

### 2.1 Getting Started

Once you have the Lab 5 materials, extract them by entering the following commands:

```
% tar -xf lab5.tar
% cd lab5
% export LAB5_ROOT=$PWD
```

Within the lab root directory, you will find six subdirectories, each serving a specific purpose:

- `build`: Contains the Makefile and compiled code.
- `riscvlong`: Pipelined RISC-V processor integrated with pipelined muldiv unit and bypass logic.
- `riscvbc`: Pipelined RISC-V processor integrated with blocking cache.
- `tests`: Assembly test build system
  - `riscv`: RISC-V assembly tests

  - `scripts`: miscellaneous scripts for build system
- `ubmark`: Benchmarks for evaluation
- `vc`: Contains additional Verilog components

Most directories remain the same as in the previous lab. The new directory, `riscvbc`, includes a functional-level model of the cache, which primarily bypasses all requests, along with the other files necessary for your implementation.

### 2.2 Building the Project

We will begin as usual by compiling the reference processors and running the RISC-V assembly tests:

```
% cd $LAB5_ROOT/tests
% mkdir build && cd build
% ../configure --host=riscv32-unknown-elf
% make && ../convert

% cd $LAB5_ROOT/build
% make
% make check
% make check-asm-riscvlong
% make check-asm-rand-riscvlong
% make check-asm-riscvbc
% make check-asm-rand-riscvbc
```

The methodology is the same as in the previous lab. Refer to the lab 1 handout for details about the test suite.

## 3 Cache Design

Accessing main memory often takes hundreds of cycles, but caches can drastically reduce average memory access latency, especially for predictable address patterns. Caches achieve faster access times by being smaller and closer to the processor. However, because caches can only store a portion of the total memory, effective management of stored data is essential. A cache hit occurs when the requested data is already in the cache, while a cache miss requires refilling data from main memory. Caches leverage spatial and temporal locality to increase hits; spatial locality means that accessing one address makes nearby addresses more likely to be accessed soon, while temporal locality suggests that recently accessed addresses are likely to be reused shortly.

We've provided a functional-level (FL) cache model that simply forwards all cache requests to main memory and passes responses back to the cache. Although basic, this FL model helps develop and verify test cases using the test memory before applying them to your baseline and alternative designs.

**Note:** You are required to implement your cache using the **RAM module** provided in `vc`, rather than building the structure directly with registers. Store the tag and data in separate RAM-based arrays (a tag array and a data array), just as in a realistic hardware design.

### 3.1 Baseline Design (I-cache)

The baseline design for this lab is a 2KB, direct-mapped, write-allocate cache with 64-byte blocks, where the tag and data arrays are accessed in parallel. You should compute the required tag array size from these specifications, but you are otherwise free to make reasonable design choices. The cache must sustain one read hit per cycle, meaning consecutive read hits should complete without any stalls.

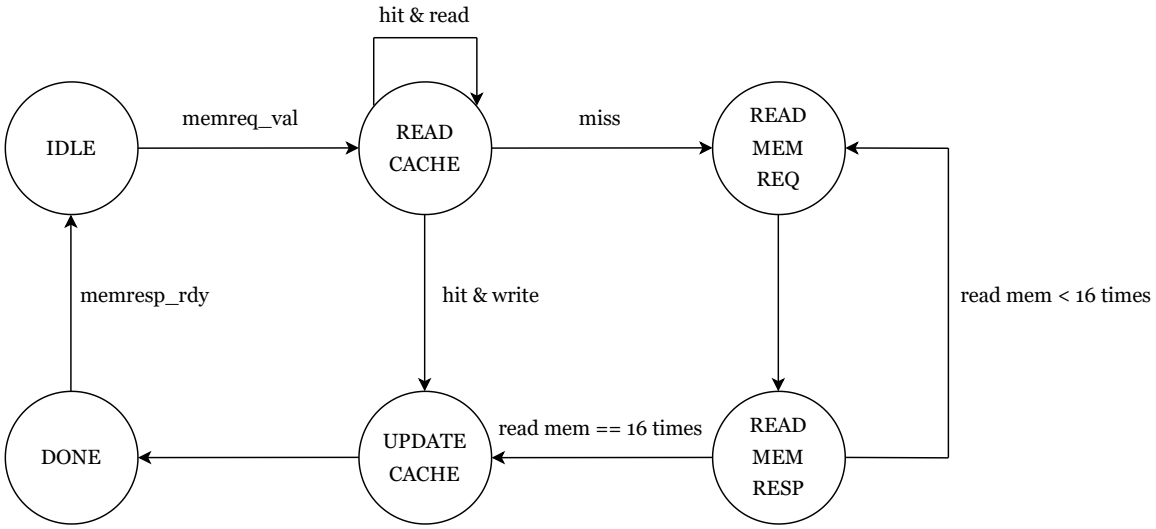


Figure 1: **Baseline FSM Control Unit**

Figure 1 shows the baseline finite state machine provided for this lab. You may modify it if you prefer an improved design. In this FSM, a read hit completes in 1 cycle, while a write hit requires 4 cycles.

The cache controller starts from the **IDLE** state. When a request is received from the processor, it transitions to the `READ_CACHE` state. In this state, the cache data and tag are read, and a tag comparison is performed. To support back-to-back read hits, the controller remains in the `READ_CACHE` state as long as consecutive read hits occur and return read hit data to processor. If a write hit occurs, the controller transitions to `UPDATE_CACHE`. If a cache miss occurs, it will send request to memory. `READ_MEM_REQ` sends a read request to the memory, while `READ_MEM_RESP` receives the data returned from memory. After the entire cache block has been received, the controller transitions to the `UPDATE_CACHE` state. In this stage, the data obtained from either a write hit or a memory refill (due to a miss) is written into the cache, and the cache state is updated accordingly. And the controller enters the **DONE** state, if the original request was a read miss, the data is returned to the processor.

## 3.2 Alternative Design (D-cache)

For the alternative design, implement a 4KB, 2-way set associative, write-allocate cache with 64-byte blocks. The FSM will be similar to the baseline design, but with adjustments to the address mapping and control signals. You'll need separate valid bits and dirty bits for each way and must carefully manage these to determine cache hits or misses by AND-ing each tag match result with the corresponding valid bit. The control unit should employ a least-recently-used (LRU) policy to decide between ways during eviction, with the LRU status tracked by dedicated bits in the control logic.

## 3.3 Victim cache

Enhance your cache by adding a victim cache to temporarily store one or two lines evicted from the main cache. Before sending a miss request to main memory, check the victim cache first. This helps reduce conflict misses and improve performance in direct-mapped or set-associative caches.

You need to implement your own 2-entry fully associative victim cache in `riscvbc-VictimCache.v` and integrate it into your D-cache. Modify your datapath and control logic to correctly handle victim hits, swaps, and evictions for seamless cooperation between the main cache and the victim cache.

### 3.4 Memory System

To complete this lab, you will primarily modify `CacheBase.v`, `CacheAlt.v`, and `VictimCache.v`. Although the edits are concentrated in these files, expect the changes to be substantial. The overall system setup is similar to the previous lab, with the main difference being that the system testbench now includes a cache module. Your goal is to implement a transparent cache that directly serves load and store requests, rather than forwarding every access to main memory as the bypass module did. Keep in mind that main memory latency is variable and averages roughly 50 cycles.

- **A cache read hit must complete in one cycle, allowing consecutive back-to-back read hits without any stalls.**
- When issuing memory requests, the cache does not need to wait for the previous memory response before sending the next request.
- For a cache miss (load or store) with a clean block, request the block from main memory, fill the cache, and then return the data to the processor.
- For a cache miss with a dirty block, perform a spill-before-fill: write back each word of the evicted line, then fetch the new block and proceed as above.

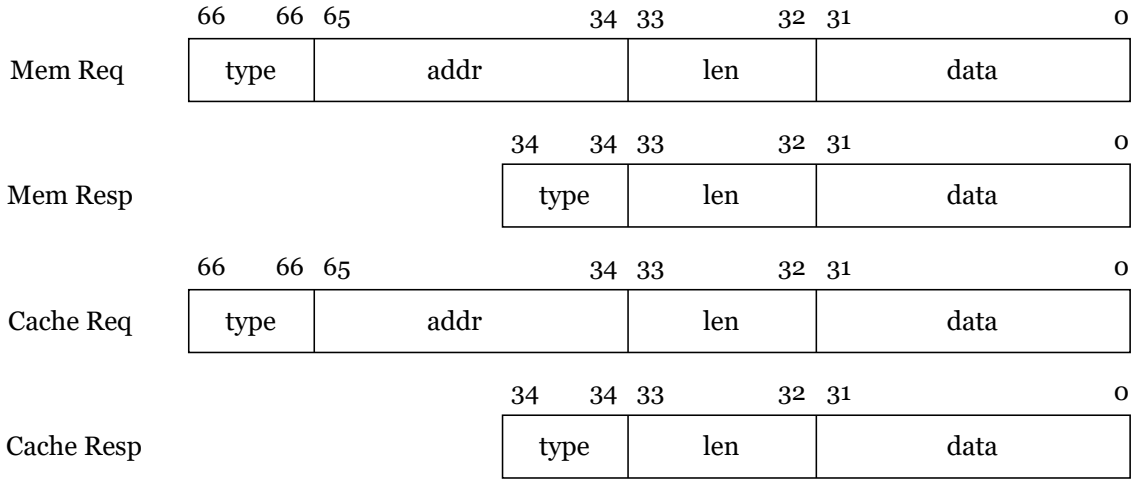


Figure 2: **Memory and Cache Request/Response Message Formats** – "Memory" represents the direction from processor to cache, while "Cache" represents the direction from cache to memory.

### 3.5 Requirement

The following are the requirements for the cache design:

- The cache specifications (such as size and set associativity) must strictly follow the specified values.
- You are required to implement the cache using the RAM module provided in `vc`; the use of registers is strictly prohibited.
- Within a single type of cache, the data cache and tag cache must be separated. The tag cache includes tag and cache state information such as the valid bit, dirty bit, and LRU bits.
- You only need to implement the victim cache for the D-cache; it is not required for the I-cache.

# 4 Testing Methodology

Although the system-level test is similar to the previous lab, a major focus here is verification. You must demonstrate correct cache functionality in your processor through assembly testing. This includes testing all typical scenarios and edge cases, with added random delays in cache-related modules.

For example, you can test them as follows:

```
% make DESIGN=${DESIGN}
```

Available designs include: None (FL-cache bypass), Icache (Base), Dcache (Alt), and All (Base+Alt).

Add directed and random tests to your unit testbench—expanding beyond a single test case. Writing directed tests for caches requires most of the miss path to be functional before you can reliably test the hit path, as the miss path is more complex. This necessitates building much of the cache structure before starting directed testing.

Here are suggested test cases, each best handled as a separate test:

- Read and write hit paths for clean lines
- Read and write hit paths for dirty lines
- Read and write misses with refills (with and without eviction)
- Stress tests that use the entire cache rather than a few lines
- Tests for conflict and capacity misses
- LRU policy verification by filling up a set
- Edge cases and corner cases
- Random delays in source, sink, and memory
- Simple address patterns with single or mixed request types
- Randomized address patterns, request types, and data
- Unit stride and stride patterns, with random data and mixed locality
- Stress tests repeatedly access the same set to induce conflict misses.

# 5 Evaluation

In this lab, you will evaluate the performance of the `riscvbc` processor across different cache configurations: None (FL-cache bypass), ICache (Base), DCache (Alt), and All (Base+Alt). For each benchmark, record the cycle count and IPC (instructions per cycle) for each setup. In your report, analyze performance differences, discussing design trade-offs under various levels of spatial and temporal locality. Identify scenarios where each configuration enhances or limits performance and cases with minimal impact. We recommend using at least six patterns, mixing reads, writes, and random access, to effectively highlight these contrasts. Explain the impact of the victim cache on performance and discuss the underlying reasons. Summarize your findings in your report.

When running the random-delay benchmark with the `riscvbc` processor, you may need to increase the timeout threshold. You can do this by adjusting the maximum cycle count in `riscvbc-randdelay-sim.v`. The following instruction sequences are provided for evaluation, but your primary focus should be the random test, as it best reflects real-world memory latency behavior.

```
% cd $LAB5_ROOT/build
% make run-bmark-riscvbc
% make run-bmark-rand-riscvbc
```

Table 1: **Cycle count for each random benchmark**

| benchmark | None | Icache | Dcache | All |
|---|---|---|---|---|
| bin-search-rand | | | | |
| cmplx-mult-rand | | | | |
| masked-filter-rand | | | | |
| vvadd-rand | | | | |

You need to include Table 1 in your report, recording the cycle count for each random benchmark under different cache configurations.

# 6 Submission

## 6.1 Lab Report

In addition to the source code for the lab, you would need to submit a lab report that includes the following sections:

- **Introduction/Abstract (1 paragraph maximum)**: introductory paragraph summarizing the lab
- **Design**: describe your implementation, justifications for design decisions (if any), deviations from the prescribed datapath, and discussion. **Remember that you must provide a balanced discussion between what you implemented and why you chose that implementation.**
- **Testing Methodology**: describe how you tested the modules and your overall testing strategy (any corner cases?)
- **Evaluation**: report your random simulation results and cycle counts comparing different cache configuration
- **Discussion**: comparison and analysis of benchmark results, discussion of tradeoffs of Icache, Dcache and All. Explain the impact of the victim cache on performance.
- **Figure**: provide the Icache and Dcache **FSM diagram** for your control logic, and ensure that the conditions for state transitions are clearly specified.

Avoid scanning hand-written figures, and **certainly, do not capture hand-written figures with a digital camera**. The lab report holds too much importance to jeopardize its readability with illegible figures. Please ensure that each section is clearly numbered. The lab report should not exceed a **maximum of 4 pages**, and there will be penalties imposed for exceeding this limit. Figures do not count against this limit.

## 6.2 Modified Files

For this assignment, the following files should be modified:

- riscvbc-CacheBase.v
- riscvbc-CacheAlt.v
- riscvbc-VictimCache.v
- riscvbc-CacheMsg.v

## 6.3 Deliverables

For submission, please turn in a .tar.gz file of your working directory without changing the original directory structure. All source files should be located in `$LAB5_ROOT/riscvbc` and/or in the `test/` directory. Please be sure to delete any generated waveforms or compiled code by running `make clean` in each of the build directories. Also, delete the build directories of the tests and ubmark directories.

```
% cd $LAB5_ROOT/build
% make clean
% cd $LAB5_ROOT/tests
% rm -rf build
% cd $LAB5_ROOT/ubmark
% rm -rf build
```

You can execute the following commands to make a tarball of your completed lab, assuming that you did not change the name of the lab root directory.

```
% cd $LAB5_ROOT
% cd ..
% tar -cvzf student_id-lab5.tar.gz lab5
```

Below is a list of files we will need to submit to meet the expectations of this lab:

- `riscvbc` source code
- Custom assembly tests
- Lab report

### 6.4 Submission Instructions

- Keep your code in the lab5 folder. If the code is not in this tarball, we cannot grade it.
- Submit the tarball and lab report separately via e3.
  - student_id-lab5-report.pdf
  - student_id-lab5.tar.gz

## 7 Grading Rubric

- **Report (30%)**
  - Introduction/Abstract (maximum 1 paragraph)
  - Design
  - Testing Methodology
  - Evaluation
  - Discussion
  - Figures
- **Code (65%)**
  - Baseline Design (25%)
  - Alternative Design (30%)
  - Victim cache (10%)
- **Performance (5%)**
  - Baseline (1%)
  - Alternative (1%)
  - All (3%)

# 8 Tips

- Develop incrementally—code and test small sections at a time to avoid overwhelming debugging.
- Always draw the hardware design before coding, ensuring clear interaction between control logic and the datapath.
- Modify the control unit as needed—it's a starting point for your own logic and signals.
- Start early and run simulations regularly to catch issues early in the process.
- If things don't fully work, clearly document your progress and debugging efforts in the lab report.

# 9 Acknowledgments

This lab is adapted from ECE 4750 at Cornell University.