Computer Architecture
Lab 5: Cache

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

revision: 11-19-2024

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:** 12/9 (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 eight 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 (**you need to copy your riscvlong folder from lab2**)
- `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 fetching 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.

We require you to use the RAM module in vc to implement the cache, rather than directly implementing it using registers. The tag and data must be stored in separate caches.

3.1 Baseline Design

The baseline design for this lab involves implementing a 2KB, directly mapped, write-allocate cache with 64-byte blocks, where tag and data accesses occur in parallel. You should determine the appropriate tag array size based on the given specifications, but you are otherwise free to make design choices. The cache must support consecutive read hits without delays, achieving a sustained throughput of one read hit per cycle. Additionally, it should write back all dirty blocks when the `flush` signal is asserted and indicate completion by asserting the `flush_done` signal.

3.2 Alternative Design

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 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 Memory System

To complete this lab, you will primarily modify `CacheBase.v` and `CacheAlt.v`. While the changes focus on these files, expect complex and extensive additions. The system setup is largely the same as in the previous lab, with the main difference being the integration of a cache module in the system testbench. Your task is to create a transparent cache that handles requests directly rather than forwarding them all to main memory, as the bypass module does. Note that main memory access latency will vary randomly, averaging around 50 cycles.

- For a cache hit, respond within the same cycle.
- For a cache miss (load or store) with a clean block replacement, request the data from main memory, load it into the cache, and then provide it to the processor.
- For a cache miss with a dirty block replacement, perform "spill-before-fill": write back each word of the replaced line, then proceed with loading the new data as above.

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, once you have implemented the `flush` and `flush_done` signals, 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

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. Summarize your findings in your report.

You may need to delay the timeout check when running the random delay benchmark with the `riscvbc` processor. Specifically, you can modify the maximum cycle count in `riscvbc-randdelay-sim.v`.

6 Extensions

6.1 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 or the next cache level, check the victim cache first. This can improve performance by alleviating conflicts in direct-mapped and set-associative caches.

6.2 Early Restart and Critical Word First

Typically, caches receive data in cache line order, from word 0 to word n-1, and wait until all words arrive before restarting the processor. To reduce processor stall time, modify your design to allow the processor to restart as soon as the required data arrives, prioritizing critical words. Ensure only one cache access per cycle—if the fill logic is writing to the cache, the processor should not read from it, and vice versa.

6.3 Instruction Prefetch Buffer

To anticipate instruction cache misses, consider implementing an instruction prefetch buffer. For example, on a miss, you could fetch two consecutive cache lines and restart the processor once the first line is available. Although the serialized memory system may impact the effectiveness, this technique may still improve performance depending on the workload.

7 Submission

7.1 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 simulation results and cycle counts comparing different cache configuration
- **Discussion**: comparison and analysis of benchmark results, discussion of tradeoffs of using dual fetch and dual issue

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.

7.2 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

7.3 Submission Instructions

- Please, keep your code in the folder lab5. If you do not put code in a tarball created from this folder, we will unable to grade it.
- Tarball and lab report should be submitted separately via e3.

8 Grading Rubric

- **Report (30%)**
  - Introduction/Abstract (maximum 1 paragraph)
  - Design
  - Testing Methodology
  - Evaluation
  - Discussion

  - Figures
- **Code (70%)**
  - Baseline Design (40%)
  - Alternative Design (30%)
- **Extensions (Bonus, 10% each, up to 30% total)**

9 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.

10 Acknowledgments

This lab is adapted from ECE 4750 at Cornell University.