ECE 4750 Computer Architecture, Fall 2023
Lab 3: Cache

School of Electrical and Computer Engineering
Cornell University

revision: 2023-11-02-01-45

The purpose of this lab is to add a data cache (D-cache), an instruction cache (I-cache) to your pipelined processor model. You will be given a Verilog system integration testbench along with a FL cache bypass module. You are to use the FL cache bypass at your discretion as a "gold standard" to test cache functionality. It may be useful to instrument the FL version of the cache model and use it to compare your cache version against it when debugging. Aside from adding the caches, there is one extra-credit section in this lab. It is completely optional, and is discussed at the end of the handout.

You are required to implement the baseline and alternative designs, verify the designs using an effective testing strategy, and conduct an evaluation comparing the two implementations. **You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with:

- basic memory system design;
- complex finite-state-machine cache controllers;
- microarchitectural techniques for implementing cache associativity;
- abstraction levels including functional- and register-transfer-level modeling;
- design principles including modularity, hierarchy, and encapsulation;
- design patterns including message interfaces, control/datapath split, and FSM control;
- agile design methodologies including incremental development and test-driven development.

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. To begin, download the starter files from the course website and extract them into a directory titled `sim` with the following commands:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750

# download the file to this folder from Canvas

% tar -xvzf lab3.tar.gz
% cd sim
% make setup
```

You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece4750/sim/lab3_cache
% make run-all
```

All of the tests you used in the last lab should pass in the provided functional-level model. For this lab you will be working in the `lab3_cache` subproject which includes the following files:

- `tb_Cache.v` – System Level Testbench for the Processor with cache
- `CacheBypass.v` – A FL simulator for the cache (cache bypass).
- `CacheBase.v` – Base cache design template
- `CacheAlt.v` – Alt cache design template
- `CacheNone.v` – Wrapper file for the two bypass caches (No I or D-cache)
- `CacheI.v` – Wrapper file for base design as I-cache and bypass cache as D-cache
- `CacheD.v` – Wrapper file for bypass cache as I-cache and alt design as D-cache
- `CacheAll.v` – Wrapper file for base design as I-cache and alt design as D-cache
- `asm/` – Symlink to lab2/asm

1. Introduction

Accessing main memory can require hundreds of cycles, but cache memories can significantly reduce the average memory access latency for well-structured address patterns. Caches are faster than main memory because they are smaller and are located close to the processor; but because a cache can only hold a subset of all memory locations at any one time, we must carefully manage what data we keep in the cache. A cache hit occurs when the data we are requesting is already in the cache, while a cache miss occurs when the data we are requesting is not in the cache and thus requires accessing main memory. Caches exploit spatial and temporal locality to increase the number of cache hits. In an address pattern with significant spatial locality, if we access a given address we are likely to access an address close to the first one in the near future. When temporal locality is exhibited, if we access a given address, we are likely to access that same address again in the near future.

We have provided you with a functional-level model of a cache, which essentially just passes all cache requests through to the memory interface, and passes all memory responses through to the cache response interface. While this might not seem useful, the functional-level model will enable us to develop many of our test cases with the test memory before attempting to use these tests with the baseline and alternative designs.

2. Baseline Design

The baseline design for this lab is to implement a 2kB, directly mapped write-allocate cache with 64-byte blocks. The tag and data access should happen in parallel. You are expected to choose the appropriate size of the tag array and follow the specifications outlined in this document but otherwise you are free to make your own design choices. Your cache should be able to handle back-to-back read hits without delays between requests. The cache should write back all its dirty blocks when the `flush` signal is asserted, and assert the signal `flush_done` when it's done.

3. Alternative Design

For the alternative design, add a 4kB, 2-way set associative write-allocate, cache with 64-byte blocks, utilizing LRU replacement policy. Your cache should be able to handle back-to-back read hits without delays between requests. The cache should write back all its dirty blocks when the `flush` signal is asserted, and assert the signal `flush_done` when it's done.

4. Memory System

To implement a correct solution for this lab, you should only have to change the `CacheBase.v` and `CacheAlt.v` files. However, do not be misled: The additions to this file can be difficult and numerous. The system-level setup is mostly the same as the one from the last lab. The main difference is the inclusion of a cache module in the system-level testbench. It is your job to create a transparent cache that will serve the requests without forwarding all of them to main memory, which is what the provided bypass module does. In this lab, the main memory latency will be random, and is approximately 100 cycles.

- on a cache hit, the response should happen in the same cycle
- on a cache miss (load or store) where the block being replaced is not dirty, send the request to the Main Memory bring it into the cache, and lastly return the result to the processor.
- on a cache miss where the block being replaced is dirty, perform spill-before-fill: First send a request for each word from the replaced line. Again, the request should happen within the same cycle once the cache lookup determines there is a miss. After the data is written back, the cache miss can proceed as in the bullet item above.

5. Testing Strategy

While the system-level test can be run in a similar manner as last lab, a key part of this lab is verification: you will be required to show, with assembly testing, that your cache is working as expected when integrated in your processor. Furthermore, you will have to demonstrate full line and toggle coverage whenever possible, along with testing all common and edge cases –which includes random delay testing of the all the cache related modules. In additional, **you will write individual unit testbenches for all the modules** inside the `lab3_cache` folder, which you can run as follows:

```
% cd ${HOME}/ece4750/sim/lab3_cache
% make utb_XXXX.v.sim [DESIGN=${DESIGN}$] [RUN_ARG=--trace] [COVERAGE=${COVERAGE}]
```

As an example for assembly testing, after you make sure the `flush` and `flush_done` signals are correctly implemented, you can do the following:

```
% make add.hex.diff DESGIN=${DESIGN} [COVERAGE=${COVERAGE}]
```

The design names are: CacheNone (FL-cache bypass), CacheI(Base), CacheD(Alt), CacheAll(Base+Alt)

You will add your directed and random tests in your unit testbench. You will be adding more test cases — do not just make the given test case larger. A key challenge in writing directed tests for cache memories is that most of the miss path must be working before you can test the hit path. The miss path is significantly more complicated than the hit path, so this lends itself more towards a monolithic design process. Most of the cache must be implemented before we can run our first directed test.

Some suggestions for what you might want to test are listed below. Each of these would probably be a separate test case.

- Read hit path for clean lines
- Write hit path for clean lines
- Read hit path for dirty lines
- Write hit path for dirty lines
- Read miss with refill and no eviction
- Write miss with refill and no eviction

- Read miss with refill and eviction
- Write miss with refill and eviction
- Tests which stress entire cache, not just a few cache lines
- Conflict misses
- Capacity misses
- LRU replacement policy by filling up a way
- Tests specifically designed to trigger corner cases
- Testing all or some of the above using random source and sink delays and test memory delays
- Simple address patterns, single request type, with random data
- Simple address patterns, with random request types and data
- Random address patterns, request types, and data
- Unit stride with random data
- Stride with random data
- Unit stride (high spatial locality) mixed with shared (high temporal locality)

6. Evaluation

You are required to conduct a holistic evaluation of the cache performance of the base and alternative designs and the design tradeoffs with different exposures to spatial and temporal locality. We strongly recommend including some patterns that mix reads/writes and random patterns. We recommend a total of six or more patterns for evaluation. Obviously, these patterns need to be carefully chosen to highlight the differences between the baseline and alternative designs. As another idea, it would be interesting for you to analyze other replacement policies and describe your findings in your report.

7. Submission

Do not modify the system level techbench interface, this applies to students doing the lab or the extra credit. The code submission for both milestone and final submission is the exact same process. For this lab you will be submitting a compressed tar file (with the file ending `.tar.gz`) to Canvas with the following files/folders:

- `Makefile` – Makefile to run the Verilog simulator
- `verilator.cpp` – C++ harness for the Verilog simulator
- `group$XX.txt` – XX is your group number. File contains all the netids of all of the group members (one per line).
- `lab1_imul/` – The lab1 sub-project folder with everything you submitted in lab1.
- `lab2_proc/` – The lab2 sub-project folder with everything you submitted in lab2
- `lab3_cache/` – The lab3 sub-project for the files you have created in this lab. (See below)

The `lab3_cache` sub-project folder should have the following files at minimum:

- `tb_Cache.v` – System Level Testbench
- `CacheBypass.v` – Functional Level Cache
- `CacheBase.v` – Base cache Design
- `CacheAlt.v` – Alt cache Design
- `CacheNone.v` – Wrapper file for the two bypass caches (No I or D-cache)
- `CacheI.v` – Wrapper file for base design as I-cache and bypass cache as D-cache

- `CacheD.v` – Wrapper file for bypass cache as I-cache and alt design as D-cache
- `CacheAll.v` – Wrapper file for base design as I-cache and alt design as D-cache
- `default.config` – Config file for Makefile to allow testbench reuse.

In addition to the following mandatory files as listed above, you should submit any additional modules, `.config` files, and testbenches (files with the `tb/ub/utb` prefix) you have created. Make sure you do not modify the provided module declarations, the Makefile, `verilator.cpp`, and `vc` files.

You can create a compressed tar file by the following command:

```
% tar -czvf lab3.tar.gz file1 [file2] [...]
```

For your convenience, we have a make rule to assist you in creating the tar file. However, you are ultimately responsible for the contents (and the lack of) of the tar file you submitted.

```
% source setup-ece4750.sh
% cd ${HOME}/ece4750/sim
% make build-tar
```

8. Extra Credit

8.1. Victim Cache (up to 0.10 points)

Modify your caches to incorporate a victim cache with them. You will keep one or two lines lying around in the victim cache and check the victim cache before issuing the miss to main memory/higher level cache. This may help performance when you have conflict problems with direct-mapped and set-associative caches.

8.2. Early Restart and Critical Word First (up to 0.10 points)

The most intuitive way to implement the cache is to have the memory system deliver the words of the cache line in cache line-aligned order starting from word 0 to word n-1. The processor waits for all words to be delivered before restarting. You can change your design to allow the processor to restart as soon as possible, along with having the memory system prioritize values the processors need to resume faster. This technique should reduce the amount of time the processor is stalling for memory, as it no longer has to wait until the full cache line is loaded in. Be sure that on any given cycle there is only one cache access. For example, if the fill logic is writing the cache, then the processor should not read the cache and vice versa.

8.3. Instruction Prefetch Buffer (up to 0.05 points)

Since the next instruction cache miss is likely to be the cache line following the last miss, you could implement a prefetch buffer, or simply fetch two lines on an instruction cache miss and restart the processor after the first miss is complete. This may not be a good idea in terms of performance for this processor since the memory system is quite serialized. Or it might be, depending on the application.

Acknowledgments

Socrates Wong, Cecilio C. Tamarit, and José Martínez prepared this version of Lab 3 as part of the course ECE 4750 Computer Architecture at Cornell University. Significant parts were adopted from earlier versions by Shunning Jiang, Shuang Chen, Ian Thompson, Moyang Wang, Christopher Torng, Berkin Ilbeyi, Shreesha Srinath, Christopher Batten, and Ji Kim.