Lab 5: Data Cache

Assigned: Wed., 2/25; Due: **Mon., 3/23**

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

1. Introduction

In this lab, you will extend your pipelined MIPS machine to implement a data cache. For the previous labs, we have been assuming that accessing (i.e., reading from and writing to) the memory can be completed within a *single clock cycle* for simplicity. In this lab, you will be using a memory that takes *several cycles* to complete serving memory accesses, resulting in pipeline stalls at the memory stage. The goal of this lab is to add a data cache to reduce the number of stalls due to memory accesses.

2. Additions to the MIPS Machine

2.1. Microarchitectural Specifications

Data Cache. Your goal is to implement a **data** cache without an instruction cache, assuming that reading instructions from memory still takes only a single cycle. The data cache is accessed whenever a load or store instruction is in the memory stage. The specifications of the data cache that you will be implementing are listed below.

Organization. The data cache is **direct mapped** and has a **parameterized** capacity with a **32-bit block size**. The cache is empty to begin with, and it has the default capacity of 8KB.

Hit and Miss Timing. When a load instruction hits in the data cache in the memory stage, the data is retrieved within the *same cycle*. When a store instruction hits in the data cache, the data is written at the end of the cycle. On the other hand, when either a load or a store misses in the data cache, the block must be read from main memory and installed into the appropriate cache set. The top module that links the MIPS processor and memory in **testbench.v** has been modified to reflect the multi-cycle access latency of memory. The memory has been modified so that it takes 4 cycles to read or write the data.

Table 1 shows the timing of a load instruction that misses in the cache. The core accesses the cache in cycle 1 with `cache_addr`, which results in a cache miss. Within the same cycle, the cache issues a memory request with the load address of 0x04000000 through `mem_addr`, which is an address port from the `mips_core` module to the `mips_mem` module. In cycle 5, the data is returned and installed into the cache. After the data has returned to the core, the load instruction can fetch it in the next cycle from the cache through `cache_data_out`.[1]

Cycles	1	2	3	4	5	6
`cache_addr`	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000
`mem_addr`	0x04000000	x	x	x	x	x
`cache_hit`	0	0	0	0	0	1
`mem_data_out`	x	x	x	x	0xdeadbeef	x
`cache_data_out`	x	x	x	x	x	0xdeadbeef

Table 1. Timing of a load that misses in the cache.

[1]The ports, `cache_addr` and `cache_data_out`, are not provided in the starter code. They are used in the handout for demonstration purposes.

Writing Policy. The data cache uses the **write-back** policy. No data is returned from the cache on a store hit. On a store miss, the cache first reads the block from memory into the cache, and after the block is in the cache, it then performs the store operation into the block.

Handling Dirty Block Evictions. When a dirty block is being evicted due to a cache conflict miss, it needs to be written back to memory. In this lab, we will simply write back the dirty data first before we allocate a new block in the set. Table 2 shows the timing of a load instruction (with address 0x04000000) conflicting with a dirty block in the cache, thus evicting it. Assuming the dirty block's address is 0x04008000, the cache will begin writing back the dirty data to the memory starting in the first cycle of the cache-miss operation. Note that `we` is asserted along with the `mem_data_in` in the first cycle. Your design will need to hold the `mem_addr`, which is the address of the dirty block being written back to memory, until the *fifth* cycle due to a four-cycle delay on the data propagation back to the memory. After the data is written back to memory at the fifth cycle, a memory-read operation will begin in order to fetch the new block into the cache. The sixth cycle essentially corresponds to the first cycle in Table 1.

Cycles	1	2	3	4	5	6
`cache_addr`	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000
`mem_addr`	0x04008000	0x04008000	0x04008000	0x04008000	0x04008000	**0x04000000**
`cache_hit`	0	0	0	0	0	0
`we`	**'b1111**	0	0	0	0	0
`mem_data_in`	0xcafecafe	x	x	x	x	x
`cache_data_out`	x	x	x	x	x	x

Table 2. Timing of a load that evicts a dirty block from the cache.

Interface. Replace your 447src directory with the one provided in the tarball. The interface between the memory and the core remains the same. Note that we do not provide a cache interface for you in this lab, you have the freedom of designing your own interface between the cache and the core.

Tests. To test the correctness of your core after adding the data cache, you will be using `memtest0 and memtest1`. Note that we will not include control flow instructions in our test cases to test your code. You will need to write your own test cases to verify various behaviors of your data cache.

3. Submission

3.1. Lab Section Checkoff

So that the TAs can check you off, please come to any of the lab sections *before* Sat., 3/28. Please come *early* during the lab section. During the Lab Section, the TAs may ask you:

- to answer questions about your implementations,
- to simulate your implementations using various test inputs (some of which you may have not seen before),

3.2. Source Code

Make sure that your source code is readable and documented. Please set your default cache size to 4KB. Please submit the lab by executing the following commands.

```
$ cp -r src /afs/ece/class/ece447/handin/lab5/andrewID/
$ cp -r inputs /afs/ece/class/ece447/handin/lab5/andrewID/
```

3.3. README

In addition, please submit two `README.txt` files. To submit these files, execute the following command.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab4/andrewID/README.txt
```

The `README.txt` file must contain the following three pieces of information.

1. A high-level description of your design.
2. The number of cache hits and misses. You will need to add counters to your module to record these.
3. The percentage speedup of using your data cache over your machine without the cache for the `memtest0 and memtest1` input program.

It may also contain information about any additional aspect of your lab.

3.4. Late Days

We will write-lock the handin directories at midnight on the due date. For late submissions, please send an email to `447-instructors@ece.cmu.edu` with tarballs of what you would have submitted to the handin directory.

Remember, you have only 5 late lab days for the entire semester (applies only to lab submissions, not to homeworks, not to anything else). If we receive the tarball within 24 hours after the deadline, we will deduct 1 late lab day. If we receive the tarball within 24 to 48 hours, we will deduct 2 late lab days. During this time, you may send updated versions of the tarballs (but try not to send too many). However, once a tarball is received, it will immediately invalidate a previous tarball you may have sent. You may not take it back. We will take your very last tarball to calculate how many late lab days to deduct. If we don't hear from you at all, we won't deduct any late lab days, but you will receive a grade of 0 for the lab.

4. Extra Credit: Cache Exploration

We will offer up to 50% additional credit *for this lab* (equivalent to 2.5% additional credit for the course) for exploring two different design aspects of the cache.

1. **Critical path:** We will hold a performance competition. Among all implementations that are correct, the "top"[2] students that have the lowest critical path will receive up to 20% additional credit *for this lab* (equivalent to 1% additional credit for the course).
2. **Four-way set-associate cache and replacement policy:** You will first need to design and implement a four-way set-associate cache. On top of your cache, you will explore various cache replacement and/or insertion policies. The *cache replacement policy* specifies which cache block in a set is replaced when a new block is inserted into the cache. The *cache insertion policy* specifies where in the list of blocks the new block is placed. You can, for example, implement a replacement policy that evicts (replaces) the least-recently-used block, and an insertion policy that places new blocks at the most-recently-used position. However, other replacement and insertion policies have been studied, and some have been shown to achieve significantly better performance (fewer cache misses) for certain access patterns [1, 2]. You should experiment with a variety of test programs and optimize the cache replacement/insertion policy.
   Among all implementations that are correct, the "top" students that have the fastest execution time for an undisclosed set of test inputs will receive up to 30% additional credit *for this lab* (equivalent to 1.5% additional credit for the course).

[2]The instructor reserves all rights for the precise definition of the word "top".

Please write a report (`report.pdf`) that briefly summarizes 1) how you optimize the critical path and 2) your findings on cache replacement/insertion policies. Your report does not need to be more than one page. Please also submit the version of your simulator (`src/`) that implements the best performing cache replacement/insertion policies.

All of the guidelines for Lab 5 specified in this handout also apply to the extra credit, except for the following differences.

- Submission path: `/afs/ece/class/ece447/handin/lab5/andrewID/extra` using `cp -r src /afs/ece/class/ece447/handin/lab5/andrewID/extra`
- `Tarball (for late submissions): lab5_extra_andrewID.tar.gz`

References

[1] M. K. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[2] V. Seshadri et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.