UChicago CMSC-22200 Computer Architecture, Autumn 2016
Lab 4: Simulating Caches

Instructor: Prof. Yanjing Li
TAs: Xin-Chuan (Ryan) Wu, Maria Hyun, Mohammad Reza Jokar, Gushu Li

Assigned: Thur., 11/03, 2016
Due: **11:59 pm, Saturday, 11/12, 2016**

Introduction

In this lab, you will extend the 5-stage pipelined ARM machine that you have implemented in previous labs with level 1 (L1) instruction and data caches. We will fully specify the microarchitecture of the caches (see "Microarchitectural Specification" section). Skeleton source code files for this lab can be downloaded from the course webpage. In the `src/` directory, we provide two files (`shell.c` and `shell.h`) that already implement the shell and another six files (`pipe.h pipe.c bp.h bp.c cache.c cache.h`) that you are allowed to change. You have already implemented the pipelined simulator and branch predictor in `pipe.h pipe.c bp.h bp.c`. You should implement caches (`cache.c cache.h`) and employ the caches in your pipelined simulator (`pipe.h pipe.c`).

1 Microarchitectural Specification

1.1 Instruction Cache

The *instruction cache* is accessed every cycle in the fetch stage (unless the pipeline is stalled).

Organization. It is a **four-way** set-associative cache that is **8 KB** in size with **32 byte** blocks (this implies that the cache has **64** sets). When accessing the cache, the set index is calculated using bits [10:5] of the PC.

Miss Timing. When the fetch stage *misses* in the instruction cache, the block must be retrieved from main memory. An access to main memory takes **50** cycles. On the 50th cycle, the new block is inserted into the cache. In total, an instruction cache miss stalls the pipeline for 50 cycles. In the 51st cycle, the data is returned to the processor, so the Instruction Fetch stage is completed at the 51st cycle.

Replacement. When a new block is retrieved from main memory, it is inserted into the appropriate set within the instruction cache. If any block within the set are empty, the new block is simply inserted into an empty block. However, if none of the blocks in the set are empty, the new block *replaces* the *least-recently-used* block. For both cases, the new block becomes the *most-recently-used* block.

Control-Flow. While the fetch stage is stalled due to a miss in the instruction cache, a control-flow instruction further down the pipeline may *redirect* the PC. As a result, the pending miss may turn out to be unnecessary: it is retrieving the wrong block from main memory. In this case, the pending miss is *canceled*: the block that is eventually returned by main memory is *not* inserted into the cache. (A test input of this case is provided in inputs/cancel_req.s.) Finally, note that a redirection that accesses the *same* block as a pending miss does *not* cancel the pending miss.

1.2 Data Cache

The *data cache* is accessed whenever a load or store instruction is in the memory stage.

Organization. It is an **eight-way** set-associative cache that is **64 KB** in size with **32 byte** blocks (this implies that the cache has **256** sets). When accessing the cache, the set index is calculated using bits [12:5] of the data address that is being loaded/stored. The L1 data cache is a write-back, allocate-on-write cache.

Miss Timing & Replacement. Miss timing and replacement of the data cache are identical to the instruction cache.

Handling Load/Store. Both load and store misses stall the pipeline for 50 cycles. They both retrieve a new block from main memory and insert it into the cache. The Memory stage of the load/store instruction is completed at the 51st cycle.

Dirty Evictions. When a "dirty" block is replaced by a new block from main memory, it must be written back into main memory. For the purpose of this lab, we will assume that such dirty evictions are handled *instantaneously* – i.e., they are written immediately into main memory in the same cycle as when the new block is inserted into the cache.

1.3 Assumptions & Initial States

- Assume that both caches are initially empty.
- Assume that a program that runs on the processor *never* modifies its own code (referred to as self-modifying code), and that a given block *cannot* reside in both caches.
- Assume that if we hit either cache, the data is returned immediately (in the Fetch stage for I cache, and Memory stage for D cache).
- Initialize all data structures related to the caches (e.g., cache blocks, tag/valid/dirty-bit arrays, etc.) to 0.

Testing Your Code

Use the simulator that you developed in lab 3 as a starting point to implement the pipelined simulator with L1 caches. Your simulator in lab3 can be used as a reference to verify that your simulator (with cache) functions correctly.

We are also providing you with a reference simulator and a set of test files, which you can also use to verify the functionality of your cache implementaion. Note that the reference simulator provides hit and miss information for every memory access. Refer to the course webpage for guidelines on using the reference simulator to test your code.

Moreover, like previous labs, you should also write more test cases on your own.

Handin

You will be working in groups of 2, and only one copy of the code needs to be submitted for one group, but `please make sure you write down both names and cnetid at the top of the pipe.c and cache.c file.`

You should electronically hand in your code (all files in the `src/` directory) into lab4 folder of your own SVN repository. Contact the TAs if there is any issue with the handin SVN repository. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in `inputs/` subdirectory.

You should stick with the same partner as Lab3, and we will take the submission from the student whose name comes first in alphabetical order.