CDA 4150 Project 3

Adding Caches to the Processor Model

Due Tuesday, Dec 1st, 11:59pm

Computer Systems Laboratory, University of Central Florida Orlando, FL 32816.

1. Objective

The purpose of this lab is to add both a data cache (D$) and instruction cache (I$) to your pipelined processor model. You will be given two Verilog models—a working pipelined model and a pipelined model with support for caching. The latter model is not complete and it is your assignment to complete it. The former model can be used at your discretion as a "gold standard". It may be useful to instrument the working version and use it to compare your cache version against when debugging. Neither version has a multiply/divide unit. You may add those in as a part of the extra credit.

After adding the caches, there are 3 extra-credit sections in the lab. These are completely optional, and are discussed at the end of this handout.

2. Lab Setup

- Go to your home directory (type `cd`)
- `mkdir cda4150/lab3`
- `cd !$`
- `cp ~hgao/cda4150/lab3.tgz .`
- Uncompress the file with `tar -xzf lab3.tgz`
- You should now have a `pipe/` and a `pipe+cache/` directory

The `pipe/` directory is the working MIPS pipelined model. It is a solution to lab2 (w/o a multiply/divide unit). Please be sure you check out the solution and understand the material from lab2. The `pipe+cache/` directory is the working directory for lab3 that contains the incomplete cache model.

3. Adding the Caches

The goal of this lab is to add an 8KB direct-mapped, 32B line, I$ and an 8KB direct-mapped, 32B line, writeback D$ to the processor model in the `pipe+cache/` directory. The caches have already been created via a special PLI routine in an initial block in `mips.v`. A later section describes all the new PLI routines in detail.

To implement a correct solution for this lab, you should need only to change the `mem.v` file. However, do not be misled. The additions to this file can be difficult and numerous. In addition, although you do not need to modify the other files in the processor, `mips.v` and `cpu.v` have changed significantly with respect to the `pipe/` model. These changes are intended to simplify the lab, and allow you to focus on implementing the cache control in `mem.v`.

3.1. The Memory System

`mips.v` is heavily changed from the previous labs. The main difference is that the processor/memory interface changes considerably when the processor has on-chip instruction and data caches. In particular, there is no need to have separate address and data busses for instructions and data. Instead, there is just one `Bus` for data and one `Addr` bus for the miss address. The data bus is bi-directional (keyword *inout* in Verilog). The control signals `Read` and `Write` are the same as in previous models. There is one additional interface signal, `Valid`, that is set by the memory system when it is returning valid data on the `Bus`. If you examine the `mips.v` file and read the comments, you will see that these changes to the model have already been made! From the comments, it should also be clear what the processor model needs to do to interact properly with this memory system:

- on an I$ miss, assert `Read` for one cycle, and drive `Addr` with the `PC`. The signals should appear on the bus one cycle after the I$ lookup determines there is a miss. Some amount of time later, the memory system will assert the `Valid` line and drive the `Bus` with the data for the entire I$ line (one word per cycle). The processor should inspect the `Valid` line on the posedge of the CLK.
- on a D$ miss (load or store) where the line being replaced is not dirty, the control is identical to the I$ miss above except that the `Addr` bus is driven from the `MAR` not the `PC`.
- on a D$ miss where the line being replaced is dirty, perform spill-before-fill. First assert `Write` for *each* cycle that you must writeback a word from the replaced line. Again, the bus transactions should begin one cycle after the D$ lookup determines there is a miss. At the time you assert `Write` you must drive `Addr` with the replacement word address, and `Bus` with the replacement data word. After the last write cycle, the D$ miss can proceed as in the 2nd bullet item above.

3.2. The Processor Flow Control

If you look at cpu.v, you will find that all of the flops have been grouped into a single always block. This makes it easier to stall the machine. There are two cases when the machine must stall. The first is on a `decodeStall` from lab2, and the second is on an I$ or a D$ stall. The pipeline in `cpu.v` handles this latter case by stalling whenever the signal `pipeInhibit` is asserted. This signal is an output of `mem.v`, but is not yet implemented. You must set this signal appropriately in `mem.v` for your design to work properly.

There is another place in `cpu.v` where some gating is needed. The PC-logic in the working pipelined model flops the `PC` on the negedge of the clock unless `decodeStall` is asserted. When we add caches, we must add a condition to that statement as well. This condition is already added in `cpu.v`, and the `PC` is not flopped if `decodeStall` or a new signal called `pcInhibit` is asserted. This signal is an output of `mem.v`, but is not yet implemented. You must set this signal appropriately in `mem.v` for your design to work properly.

3.3. The Cache PLI Routines

The actual cache state, tag, and data storage arrays have already been implemented in C for you. Your task is to write the control logic in `mem.v` that handles the cache lookup, stalling the processor, and interfacing to the memory system. The Verilog needs read and write access to the state, tag, and data arrays of each of the caches. The following PLI routines are defined for your use:

- `$icache_tag_read(index, set)`
- `$icache_state_read(index, set)`
- `$icache_data_read(index, set, word offset)`
- `$icache_tag_write(index, set, tag)`
- `$icache_state_write(index, set, state)`
- `$icache_data_write(index, set, word offset, data)`
- `$dcache_tag_read(index, set)`
- `$dcache_state_read(index, set)`
- `$dcache_data_read(index, set, word offset)`
- `$dcache_tag_write(index, set, tag)`

- `$dcache_state_write(index, set, state)`
- `$dcache_data_write(index, set, word offset, data)`

These PLI routines **must** be used in `always` blocks. They cannot be used in `assign` statements. The `set` numbers start at 0, so for direct-mapped caches, that parameter will always be 0. Similarly, the `word offset` starts at 0 and go to linesize (in words) - 1. The `state` consists of 2-bits for this lab. Bit 0 is the Valid bit for the cache line, and bit 1 is the Dirty bit. Recall that only the D$ needs to track a Dirty bit.

3.4. The MEM Pipeline Stage

The `mem.v` file is where you need to implement the cache control described in the memory interface section above using the PLI routines described in the previous section. You also need to drive the two processor flow-control signals mentioned above. The given interface to `mem.v` is correct. You simply need to implement the proper body. Here are some important notes:

- the logic for both the I$ and D$ accesses should go in this file
- the cache statistics are kept in `mem.v`. A template for the stats is given. Be sure to increment the given stats at the proper time.
- assume that the cache can be read or written in half a clock cycle. This means you do not have to worry about pipelining writes to the writeback cache. In fact, the cache write code is almost completely given to you. Writes happen on the negedge of the clock just to be sure the address and hit determination are stable before writing. *This is the only logic in the file that should be negative edge triggered.*
- `cache.h` contains useful defines for the given cache configuration. You should use these defines in many many places.
- an I$ miss and a D$ miss can occur at the same time! Because of the way the PC is flopped on the negedge, I$ misses will actually occur "sooner" than a coincident D$ miss. I found it convenient to service the I$ miss first in that case (remember that since we only have one data and address bus to the main memory system, we now have to implement these misses serially).
- remember that when doing a cache fill from the memory system and writing the cache tags and data appropriately, you should not also be reading the cache tags and data in your lookup code. A cycle after the last word of the line is filled from the memory system on a miss, you should allow the access to be retried, and it should now be a hit.

You can test your cache code with the `test/hello` and `test/host` programs given. Feel free to write your own test cases as well.

4. Extra Credit 1: Changing the Cache Configuration (5%)

Change the data cache size to a 16KB direct-mapped cache with a 64B line size, and instruction cache to an 8KB direct-mapped cache with a 64B line size. You do not need to turn in the code for this model. In your `README` file, just list the parts of your design that you had to change to change the configuration, and list your cache stats for both the original and the new configuration when running the supplied `test/host` program.

5. Extra Credit 2: Load/Store word Left/Right (10%)

Implement the lwl, lwr, swl, and swr instructions in your pipelined model (yes, they are real MIPS instructions!). Your implementation should allow back-to-back lwl/lwr and swl/swr instructions without a stall, which will involve adding a new bypass path. To test your implementation, come up with a C program that generates these instructions (or, an assembly language program). In your `README` file, indicate the name of your test program and include your test program in your lab submission.

6. Extra Credit 3: Multiply/Divide (20%)

Since the multiply and divide instructions only write the `hi` and `lo` registers, you do not need to include the multiplier or divider in any bypassing logic. Your pipelined processor should ship multiplies and divides to the multiply/divide unit and keep operating the pipeline normally even though the multiply or divide may still be in progress. The only thing you need to do is ensure that any `mfhi` and `mflo` instruction stalls in the decode stage when the multiplier or divider is busy, and that you detect structural hazards on the multiplier and divider. For full credit on each part you must implement both the signed and unsigned versions of the operation (e.g. mult and multu, div and divu).

7. Submitting Your Lab

The submission procedure is the same as the previous labs. If your lab does not work entirely, please be sure your `README` file explains what works and what does not. If you did any or all of the Extra Credit, remember that your `README` file should include the discussions mentioned in the Extra Credit sections above.