Computer Assignment 2 Deadline: Sunday, Dec. 5, 11:55 PM

Fall 2021 (Upload it to Gradescope.)

The ultimate goal of these computer assignments is to create a (simple) C/C++ simulator for a (simple) RISC-V CPU. At each step of this project, we will gradually add more units/capabilities to our processor.

Here are the important rules that we will use in all of these computer assignments:

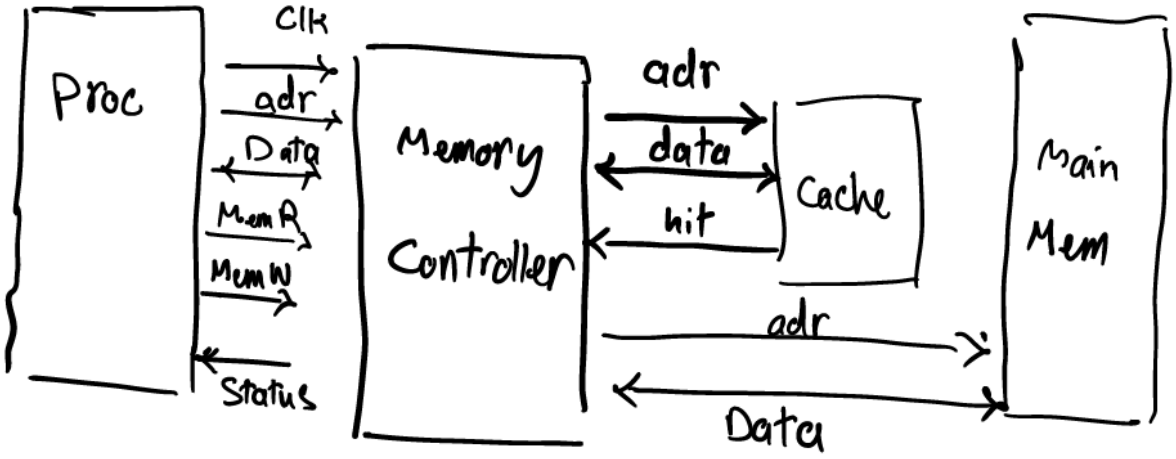
- We will use the 32-bit version of RISC-V ISA and will focus on implementing only 10 instructions that were described in Lecture 5.
- Later on, we might add more instructions and/or capabilities to our processor.
- You should do all your work in the C/C++ programming language. Your code should be written using only the standard libraries. You may or may not decide to use classes and/or structs to write your code (C++ is preferred). Regardless of the design, your code should be modular with well-defined functions and clear comments.
- You are free to use as many helper functions/class/definitions as needed in your project.
- There is no restriction on what data type (e.g., int, string, array, vector, etc.) you want to use for each parameter.
- Unfortunately, experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered, or if something needs to be described better. It is your responsibility to check the website often and read new versions of this project description as they become available.
- Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy. You are allowed to compare your results (only for the debug part) with others or discuss how to design your system. You are also allowed to ask questions and have discussions on Campuswire as long as no code is shared.
- You have to follow the directions specified in this document and turn in the files and reports that are asked.

(Changes to this file will be shown in red.)

Project Description

Overview

In computer assignment 2, we want to design a cache/memory controller and add it to our pipelined processor. To make the design and debugging easier, in this project we will only focus on your cache controller as an independent component. In the next project, we will describe how to connect this to your pipeline design. The image below shows this design in detail. You are responsible for designing the memory controller, cache, and main memory.



The input to your controller is a 32-bit address coming from the CPU (i.e., from the MEM stage). Your controller is responsible for loading the data from the memory (if it is an `LW`) or storing the value at the correct address if it is an `SW` (this is done by checking `MemR` and `MemW` signals). Your memory controller should input/output two things: "data" and "status" (we will describe later what this value should be).

Your memory hierarchy **has one level of cache and the main memory** (i.e., the same array/vector you used in Project 1).

You have to implement three separate cache models (details later) a direct-mapped a fully-associative, and a 4-way set-associative cache, and use (ideal) LRU replacement policy (for FA and SA). Each cache has 1 byte per block and a total of 16 sets (for DM) and 4 sets (for SA). The main memory is 2048 bytes.

(Changes to this file will be shown in red.)

As mentioned earlier, in this project we will focus on how to design the controller and won't connect this to a real processor (i.e., your code in project 1). Instead, you are given a driver that emulates the behavior of the processor. (More details about how this works later.)

Description

There will be three possible scenarios for your memory controller: load, store, and non-memory instructions (e.g., R-type). Your controller essentially is an FSM that depends on the inputs, control signals, and the previous state makes a decision, outputs something, and moves to another state. We will describe each in the following. Note that your controller gets the Status as an input. This input shows what the previous status of your controller was.

(You can add/remove inputs and outputs to any of the functions described below.)

FSM Design:

Same as your processor, the cache controller has a(n) (implicit) "clock" and an (infinite) while loop. During each iteration, your controller checks the (previous) Status and does the following based on this value:

1. If it is 1, it checks control signals (i.e., `MemR` and `MemW`). If either is one, it calls the corresponding function (i.e., `Load` or `Store`).
2. If it is zero, it outputs `data=MM[adr]` and `Status=1`.
3. If it is negative, it increments the `Status` by one.

LW Function Design:

1. It should call "Search" function, which has two inputs: (`adr, data`) where both are passed by reference, and a boolean output: `hit`.
2. In "Search", using the address, your code should first calculate the correct offset and index, and the tag. Then, it should access the cache (i.e., by using the index to read the tag store array).
3. If it is a direct-mapped cache, then "Search" should just use the index to access the cache and check the stored tag with the new tag. If it is a FA cache, then it should search ALL sets. If it is a SA, it should do both.
    a. If it is a match (for one set in DM, or any of sets in FA/SA), then it is a "hit". You should then access the Data Store to read the actual value and update `Data`. "Search" should then call "Update" function (details later), and finally return True.
    b. Otherwise, it is a "miss". In that case, your code should return "false".

(Changes to this file will be shown in red.)

4. Returning from "Search", there are two scenarios: if it is a "hit", then your controller should assign `"status = 1"`. If it is a miss, it should assign `"Status = -3"` (this is to model the read delay in the MM. i.e., in this case, it takes four cycles to read from the memory). It then calls "CacheMiss".
    a. In "CacheMiss", it has to read the data from the main memory (i.e., accessing `MM[adr]`). It then calls "Evict", and then returns.
    b. In "Evict", three things need to be done: 1- Finding a candidate for eviction (finding `lru_position ==0`). 2- Updating the counters (i.e., calling "Update"). 3- Updating the tag and data stores with the new values (read from MM). It then returns.
5. In "Update" function, you should update the LRU counters. Counters will be updated if either there is a "hit", or when a new line is installed from MM.

SW Function Design:

1. Same as LW, it first calls "Search" function (same as LW for steps 1-3). The only difference is that if it is a hit, instead of reading the value from the Data store, the Data Store should be updated with the new value.
2. Returning from "Search",
    a. If it is a "hit", you have to also update the `MM[adr]` (i.e., write-through strategy).
    b. If it is a miss, you again update the `MM[adr]` but do not allocate a new line in your cache (i.e., write-no-allocate strategy).
3. In either case (hit or miss), you assign `Status=1` and `Data=0`.

Cache_Driver Code and Benchmarks:

The entry point of your project is "`memory_driver.cpp`". The program should run like this:

`"./memory_driver <inputfile.txt> type",`

where `type` indicates whether the cache design is direct-mapped (type = 0) or fully-associative (type = 1), or set-associative (type=2). Your program should print the total clock cycles and miss-rate in the terminal (you need a counter to count misses):

`"(CLK, MR)"`

The template for the driver is given. The controller reads an input file ("trace.txt") line-by-line, and loads in a temporary array called trace. Each line is in this format:

`"MemR, MemW, Adr, Data"` → `(e.g., R-type: "0, 0, 100,200", SW: "0, 1, 100, 50")`

(Changes to this file will be shown in red.)

In the `While` loop, if Status=1, it reads a new line from the trace and acts accordingly (see the sample code for details). The `While` loop will be broken once we reach the end of the trace (code provided). Two trace files were given for you to test.

Same as the previous project, it is your choice how you want to structure your code (e.g., whether you want to have separate objects for each class, or you want to instantiate an object within another class, or even not use any class at all and utilize functions and structs, etc.).

Bonus Part:

For the bonus part, you should implement a level 2 cache (an 8-way set associative with 8 sets). The changes that need to be made are the following:

- On a miss, the data should be brought to L1 first. After evicting from L1, you should install the line in L2. If L2 is full, you should evict a line from L2. (i.e., your cache is exclusive).
- For search, you should first check L1. If it is a miss, you should search L2. If it is there, you should assign 'Status = -1', otherwise, it is on your main memory (in other words, L2 acts like your main memory but with 2 cycles delay instead of 4).
- The rest of the functionalities (e.g., updating LRU positions, updating values, etc.) are the same.
- (if more details are needed, they will be added here.)

Questions:

(Answer these questions for the "test.txt" file.)

1. Which cache has a higher miss-rate?
2. Assuming that 40% of instructions are loads and stores and for all other instructions, IPC=1, what would be the overall IPC for the DM, FA, and SA caches?
3. (bonus part) What is the miss rate for the cache with the 2-level cache?

What to submit.

You need to submit the following files on Gradescope.

1. Your well-commented code (all the files). Note that we will use a different trace to test your code's correctness. Your code should be compiled with the following command: "g++ *.cpp -o memory_driver" (in case you are using C, it would be

(Changes to this file will be shown in red.)

"gcc *.c -o memory_driver"). If the code fails to compile, you will lose points. Your code should produce the results in the format described on the previous page. Failing to create the above format will result in losing points.

2. A short report (a PDF file) that answers the above questions.