Assignment 5: chArm-v2 System Emulator

C S 429, Spring 2024
Unique ID: 50855, 50860, 50865, 50870, 50875, 50880, 50884
Lead TAs: Kiran Chandrasekhar, Kavya Rathod, Prithvi Jamadagni

Assigned: Friday, 29 March 2024 21:00 CT
Checkpoint 1 submission: Thursday, 11 April 2024 23:59 CT
Checkpoint 2 submission: Thursday, 18 April 2024 23:59 CT
Checkpoint 3 submission: Thursday, 25 April 2024 23:59 CT
Last possible hand-in: Saturday, 27 April 2024 23:59 CT

1 Introduction

In this lab, you will be implementing several simulators:

- `ae`, a set of individual hardware elements emulating hardware, including a register file for writing/reading register values and an ALU (Arithmetic Logical Unit) to perform computations;
- `psim`, a standalone simulator for the PIPE implementation of the chArm-v2 instruction set architecture (ISA), assuming an ideal memory system;
- `csim`, a standalone trace-driven cache simulator for a simple two-level memory hierarchy; and
- `pcsim`, an integrated "PIPE-with-CACHE" simulator, by enhancing `psim` to handle variable delays in the memory stage, and connecting it to `csim`, resulting in a simulator for the PIPE implementation of the chArm-v2 ISA with a simple two-level memory hierarchy.

The common executable that you will create will be named `se`. You will be working on the `ae` and `psim` aspects of it in the first two weeks, upgrading the `psim` in the third week, and on the `csim` and `pcsim` aspects in the fourth week.

Outcomes you will gain from this lab include the following:

- You will understand how the SEQ and PIPE- implementations of chArm-v2 work. You will understand the utilities of each stage and how they are connected to each other.
- You will understand how the PIPE implementation of chArm-v2 works. You will understand how stalling, squashing, and forwarding help resolve different hazard conditions.
- You will understand the impact that cache memories can have on the performance of programs.
- You will understand the additional changes that need to be made to the PIPE implementation to accommodate a (semi-)realistic memory hierarchy.

2 Logistics

This assignment lasts for four weeks, with three separate submissions. You are required to perform three submissions:

- AE and PIPE-, due by Thursday, 11 April 2024 23:59 CT.
- PIPE, due by Thursday, 18 April 2024 23:59 CT.
- CSIM and PCSIM, due by Thursday, 25 April 2024 23:59 CT.

Start early enough to get the assignment done before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped internet connections, corrupted files, traffic delays, minor health problems, *force majeure*, etc.

This is an individual or partner project. If you choose to work in pairs, the team may use as many slip days as the partner with the fewest available slip days. That is, if you have two slip days and your partner has three, the team gets two slip days to use for the entire assignment. Both partners will be charged for any slip days used. *Note that using slip days for a checkpoint does not adjust any future due dates.* Choose your partner well; you will not be allowed to split up during the project. Partners on a team do not have to be from the same section of the class.

All hand-ins are electronic. You may do your coding on any machine you choose, *but it is your responsibility to test this assignment for correct build/execution on an UTCS 64-bit x86-64 Linux machine before your final hand-in.* You may not share your work on lab assignments with other students outside your team, but feel free to ask instructors for help (e.g., during office hours or discussion sections). Unless it is an implementation-specific question (i.e., private to instructors), please post it on Ed Discussion publicly so that students with similar questions can benefit as well.

Before you begin, please take the time to review the course policy on academic integrity at: `https://www.cs.utexas.edu/academics/conduct`. Don't copy code from anywhere; do it yourself. This discipline is very important for this class and next classes.

Any updates for this lab will be posted on Canvas. Any clarifications or corrections for this lab will be posted on Ed Discussion.

3 Download and Setup

Go to the GitHub Classroom assignment page found at: `https://classroom.github.com/a/XrpvB4pG`. You will be able to clone your assignment to a lab machine as usual. Also note that since you can work in pairs, GitHub Classroom may prompt you to enter who you are working with to set up your repository with them. You will only need one repository for submitting, but if both you and your partner make one, just decide whose to submit later on.

4 Assignment Details

4.1 Repository Structure

Now that you have your private repository of the code base, confirm that you have the following subdirectories within it.

- **include**: This subdirectory contains all the header files needed for this project. It contains three subdirectories: `pipe`, which contains the header files needed for the pipelined processor implementation; `cache`, which contains the header files needed for the cache simulator; and `base`, which contains the header files needed for the underlying simulator.

- **src**: This subdirectory contains all the source code files needed for this project. It contains four subdirectories: `pipe`, which contains the source code files needed for the pipelined processor implementation; `cache`, which contains the source code files needed for the cache simulator; `base`, which contains the source code files needed for the underlying simulator; and `testbench`, which contains the source code of the executables you can use to compare your simulator to the provided reference.

  To complete the hardware elements in AE, you will modify the file `src/base/hw_elts.c`.

  To complete the pipelined processor emulator `psim` in PIPE, you will modify the following files: `instr_Fetch.c`, `instr_Decode.c`, `instr_Execute.c`, `instr_Memory.c`, `instr_Writeback.c`, `forward.c`, and `hazard_control.c`. All of these files are located in the `src/pipe` directory.

  To complete the standalone cache simulator `csim` for the CSIM deliverable, you will modify only the file `src/cache/cache.c`.

  To complete the final `pcsim`, you will further modify only these files.

- **testcases**: This subdirectory contains test cases for testing your emulator. The `basics`, `alu`, `mem`, `branch`, `exceptions`, and `applications` subdirectories nested inside contain assembly `.s` files, disassembled `.od` files, and ELF binaries that are used to test your simulator. Some of these directories are further subdivided into `simple`, `hazard`, and `hard`, which narrow the focus of the tests. Finally, the `cache` subdirectory contains memory trace files similar to those that you used in MM Lab, which are used to test your cache simulator.

4.2 Simulator Quirks

There are a few things in the simulator that you would not find in normal hardware, so we would like to note them here.

First, it is important to note that `hlt` is a privileged instruction, and `gcc` won't typically compile files with that instruction. To shut down the emulator, we instead check for a `ret` instruction with a return address of 0, which we turn into an emulated HLT instruction that stops the simulator after it reaches the writeback stage. This functionality is implemented for you in `src/pipe/instr_Fetch.c`, and you only need to pass the generated `STAT_HLT` through the pipeline for it to work.

Second, we need ways to view the state of the machine during a program's execution for our testing purposes, so we have special "trap" addresses defined for this purpose. In `src/base/mem.c`, you can find `IO_CHAR_ADDR` and `CHECKPOINT_ADDR` defined. A load from `CHECKPOINT_ADDR` will log the machine state to a provided "checkpoint file" (more on this later), and a store to `IO_CHAR_ADDR` will print the stored register's contents to the terminal.

5 Programming Tasks

This lab is a sequence of two programming parts. In Checkpoint 1, you will implement `ae`, an architecture element emulator (ALU, register file, etc.) and `psim`, a PIPE- simulator. In Checkpoint 2, you will implement `PIPE`, In Checkpoint 3, you will implement `pcsim`, a "PIPE-with-CACHE" simulator, by implementing a cache simulator `csim`, enhancing `psim` to handle variable delays in the memory stage, and connecting them to each other.

The assignment carries 16 points: 8 points for the first submission and four for each of the other two. Your submission will be auto-graded based on its ability to correctly execute the test cases for the corresponding week.

This *Wiki page* is work in progress to help you understand the code base that we have given you, and what you need to it. It will be updated on an ongoing basis through the assignment.

5.1 Part A: Implementing **`psim`**, A Simulator for the PIPE Implementation

The goal for Part A is to implement a PIPE simulator as described in class. You are going to complete the code in files `instr_Fetch.c`, `instr_Decode.c`, `instr_Execute.c`, `instr_Memory.c`, `instr_Writeback.c`, `forward.c`, and `hazard_control.c`, all of which are located in the directory `src/pipe`. You will need to complete all functions in the files that are marked as "STUDENT TO-DO". The hardware elements you need are implemented for you in `src/base/hw_elts.c`.

5.1.1 Architecture Emulator

Start by implementing the hardware elements in the `hw_elts.c` file. These emulate modules found in the hardware that will be called upon in your pipeline. Once you do these correctly, you will be able to pass all the tests in the `bin/test-ae` testbench.

You are required to implement three functions that emulate three hardware modules for your AE simulator:

- `cond_holds()`: This module determines if a given condition code is met based on the NCZV flag state. This should not be directly invoked from your pipeline emulator.
- `alu`: This module computes bitwise and mathematical operations like addition or shifts, and sets the NCZV flags accordingly. This module will also invoke the `cond_holds()` module to set a condition value.
- `regfile`: This module will allow you to simultaneously read and write to registers. Take care to handle writes to and reads from the SP and XZR registers properly, which are not general-purpose.

AE will need to be complete for the PIPE- implementation to function properly. If in doubt, read the ARM manual linked on the syllabus page.

5.1.2 Testing AE

Make your testbench `test-ae` in the usual manner: `make clean; make`. There is no standalone `ae` executable. The standard set of `make` targets that we have been using all semester long are also used in this assignment.

Run your `test-ae` executable using the command line

```
bin/test-ae -v <verbosity-level>
```

The verbosity level can be 0, 1, or 2 (default: 0) and will control how much diagnostic output you will see. Verbosity level 2 will stop your the testbench and print the current and expected state after a failure; this will enable you to determine which inputs cause your hardware modules to fail and allow you to intuit why.

Debug your program in the normal manner using `gdb` directly on the testbench.

5.1.3 Basic Pipelined Implementation

Next, create the simpler PIPE- implementation we have discussed in class. If you do this correctly, you will be able to pass all the tests in the `testcases/basics` and `testcases/*/simple` directories.

You are required to implement five functions that emulate five stages for your PIPE- simulator:[1]

- `fetch_instr()`: Fetch stage (including PC update actions).
- `decode_instr()`: Decode stage.
- `execute_instr()`: Execute stage.
- `memory_instr()`: Memory stage.
- `wback_instr()`: Write-back stage.

There are a number of other functions in this file that correspond to the various logic blocks discussed in class. You must implement these functions as well, and then "wire them up" appropriately within the stage logic.

The fields of the pipeline registers are defined in `include/pipe/instr_pipeline.h` in several `struct` types `*_instr_impl_t` and `pipe_reg_t`. The "clocking" of these registers is handled for you in `src/base/proc.c`. When implementing the combinational logic for a pipeline stage, you will be passed in the appropriate structs as your input and output arguments, which are named `in` and `out`. There are also a few global variables that represent the "backwards" wires sent from one stage to a prior stage, which you will need to update as well.

5.1.4 Hazard Control

For the second week, you are also required to implement stalling, squashing, and forwarding to deal with data hazards and control hazards, as discussed in class. For forwarding, add your implementation in `src/pipe/forward.c` and then call the resulting function(s) from `src/pipe/instr_Decode.c`. For stalling, implement the four functions in `src/pipe/hazard_control.c`, which is called for you in `src/base/proc.c`. You are responsible for setting up the correct `stall` and `bubble` signals for each pipeline stage at each cycle. The task of taking the appropriate actions on the pipeline registers based on these signals is handled for you.

[1]In hardware, all the stages would execute concurrently for multiple instructions in a single cycle of a PIPE implementation. Since C code is executed sequentially, however, the stages run in reverse order (decode stage after execute and memory stages, and memory stage before execute) in order to propagate forwarding values properly among these multiple executions in flight through the pipeline. *This ordering is already handled* ***correctly*** *in the starter code; you do not need to worry about it.*

5.1.5 Testing

Make your `se` executable in the usual manner: `make clean; make`. The standard set of `make` targets that we have been using all semester long are also used in this assignment.

Run your `se` executable using the command line

```
  bin/se -i <testfile-path> -v <verbosity-level> -l <cycle-limit>
        -c <checkpoint-file>
```

where the i flag is mandatory, but the rest are optional. The verbosity level can be 0, 1, or 2 (default: 0) and will control how much diagnostic output you will see. The cycle limit can be an integer greater than 0, and will set the limit on the number of cycles your simulator can run for. Its default value is 500, and most tests do not require more than this, save for some of the ones in `hard` directories. The checkpoint file is a file that will be overwritten with a "checkpoint" of the machine state when the program ends or attempts to load from `CHECKPOINT_ADDR`.

Debug your program in the normal manner using `gdb`. There is also a provided `se-ref` executable for each checkpoint that you can use (with the same arguments as `se`) to check the correct output of a test case. Or you can run an exhaustive test with `test-se`.

5.1.6 Visualization

A pipeline visualization tool is provided for PIPE- and PIPE. This tool is **NOT** a replacement for proper debugging via `gdb`; however, this may help you find in which instruction a testcase goes wrong. Also, this tool is brand-new this semester, and is therefore likely to be buggier than the other parts of the assignment.

To run the visualization, you will need to download the `visualization` folder and its contents to your **local machine**. You only need to do this step the first time.

Next, use the command line:

```
./launch_vis_run.sh bin/se -i <testfile-path> -v <specify value larger than 0>
```

on your **remote UTCS machine** each time you want to visualize a testcase. The verbosity level must be greater than 0. This will start a node server on the remote machine; make sure to stop the server with CTRL-C after you are done looking at the visualization.

If using the terminal in Visual Studio Code, you may skip this next step. If not, then run the command line:

```
  ssh -N -f -L localhost:8080:localhost:8080 <USERNAME@REMOTE_MACHINE>
```

on your **local machine**. This will forward the remote port to your local port, allowing you to access the results of your remote pipeline code. VSCode will usually handle this automatically.

Finally, open the `index.html` file in any browser on your local machine. We have tested on Firefox and Chromium-based browsers (Safari, Chrome, etc.).

5.1.7 Submission

Submit your checkpoint version using Gradescope, by providing a pointer to the private GitHub repository where you have done your work. Remember to include your partner in your submission, if you have one. *Clearing week 1 corresponds to your correctly running the test programs* in the `testcases/basics` and `testcases/*/simple` subdirectories, as well as the `bin/test-ae` testbench. *Clearing week*

2 corresponds to your correctly running test programs in the `testcases/*/hazard` subdirectories. Please note that correctly running test programs means we will match the checkpoint files generated by a reference simulator and your implementation. Printing the same value to the terminal as the reference does not guarantee that your implementation is correct, though it should at least be a decent hint. The utility `test-se` is provided to specify which tests correspond to which week, as well as which tests are passing.

5.1.8 Evaluation

Part A of the assignment counts for twelve points: two for the basic AE implementation (week 1), six for the basic PIPE- implementation (weeks 1 and 2), and four for the full PIPE implementation including handling hazards and forwarding values (week 3). Partial credit is given where applicable, so you will earn points for each individual test passed.

5.1.9 Expected work schedule

You are given two weeks for PIPE-, and are expected to use those two weeks as needed. PIPE- has historically been a very difficult and long assignment, with many possibilities for bugs. It is expected that the AE aspect should be brief and finished within the beginning of the first week, with the rest being alotted to PIPE-. Once you have completed implementing PIPE-, updating it to PIPE should involve very localized changes.

5.2 Part B: Implementing `pcsim`, A Simulator for the PIPE Implementation with A Simple Memory Hierarchy

In this part, you will first write a standalone cache controller simulator `csim` and test it against a number of memory traces. Correctness will be determined by matching the cache events generated by your simulator against a reference. You will then augment `psim` and connect it to `csim` to produce `pcsim`.

5.2.1 Implementation and Testing

- Start your work in the `cache` directory.
- Implement the `get_line()` and `select_line()` helper functions in the file `cache.c`. Implement the `check_hit()` and `handle_miss()` routines in the file `cache.c`. This will give you a skeletal cache simulator that implements the control actions (the three-state finite-state machine cache controller discussed in class) of a write-back cache with LRU replacement and write-allocate policies, for arbitrary numbers of sets, associativity values, and block sizes.
- You can assume that each cache read/write only accesses one single cache line.
- Test your code by running `make` and running `test-csim`. Your implementation is correct when the test score printed out is 40/40.
- Implement the functions `get_word_cache()` and `set_word_cache()` in the file `cache/cache.c`. After completing this task, you will have a fully functional cache simulator that implements both the control and the data portions of the cache.

- Integrate the memory hierarchy simulator into the PIPE simulator by making the appropriate changes in the files that you updated for Part A. This should involve no more than updating the data memory routines to the corresponding cache routines and handling stalls resulting from cache misses.

- Test the correctness of the combined simulator using `test-se`. Several ELF binaries will be run with a few different cache configurations. You may test these yourself by adding the flags `-A <associativity> -B <line-size> -C <capacity> -d <delay-cycles>` to any test from before.

5.2.2 Submission

Submit your final version using Gradescope, by providing a pointer to the private GitHub repository where you have done your work. As before, remember to include your partner in your submission, if you have one.

5.2.3 Evaluation

Part B of the assignment counts for four points: three for implementing the cache and achieving 40/40 on the cache test (week 3), and one for integrating the cache with the PIPE implementation and passing the tests in the `testcases/*/hard` subdirectories (week 4). Partial credit is given where applicable, so you will earn points for each individual test passed. We will use `test-se` to test your integrated simulator.

6 Extra Credit (Optional)

Two options for extra credit are offered on this lab, with up to 2 points assigned to each. Each is to be submitted along the final week of SE Lab.

6.1 Application: Matrix Multiplication

2 points
To explore the effects of the cache in a real application, we consider several implementations of matrix-matrix multiplication in the tests `applications/hard/gemm_ijk`, `applications/hard/gemm_ikj`, and `applications/hard/gemm_block`. These tests take two 64 $\times$ 64 matrices and multiply them, storing the result in another 64 $\times$ 64 matrix.

Note that these tests take several million cycles to run to completion, so you need to give them several seconds to finish running. For this reason, do NOT run your simulator with the verbose -v flag enabled, or else you will create several Gigabyte-sized files and waste large amounts of space on the lab machines.

Using these tests and your `se` executable, answer the following questions. You may use the `se-ref` executable if you are unable to pass all checkpoint 3 tests with your implementation.

- To start, run the tests without the cache enabled. You should set the cycle limit to 8,000,000 with the `-l` flag in order to bypass the default limit of 500. You can filter the output to a checkpoint file to view the number of cycles the simulator runs for, so an example command would be `bin/se -i testcases/applications/hard/gemm_ijk -c checkpoint.out -l 8000000`. Note the number of cycles it takes to run each test to completion. Which of the three tests runs in the fewest number of cycles? Which of the three runs in the greatest number of cycles? Provide some intuition on why this behavior occurs.

- Next, run the tests with cache parameters `[A,B,C,d] = [4,32,512,100]`. Modern processors typically suffer a loss on the order of 100 cycles on a cache miss when the requested data is in DRAM, so this configuration more closely mimics a real scenario. You will now need to increase the cycle limit to 40,000,000 in order to allow the simulator to run to completion, so the example command from before becomes `bin/se -i testcases/applications/hard/gemm_ijk -c checkpoint.out -l 40000000 -A 4 -B 32 -C 512 -d 100`. Note the number of cycles it takes to run each test to completion. Now which test runs in the fewest/greatest number of cycles? Has this changed from before? Use the memory access patterns of each test to explain why the results did or did not change.
- Next, run the tests with different, but reasonable, values for cache parameters `[A,B]`. Keep parameters `[C,d] = [512,100]`. Run each test once more, and answer the same questions as before.
- Next, attempt to find the most inefficient combination of loop ordering (gemm testcase) and cache layout (with parameters `[C,d] = [512,100]`). Leverage your understanding of memory stride access patterns and cache structures to find this.
- Finally, create a writeup of your answers and observations from each part. Include graphs of cycle count data for each run and the cache parameters used.
- There will be a separate week 4 assignment on Gradescope for you to submit answers to these questions.

6.2 chArm-v3

You will be upgrading the system emulator to chArm-v3, a subset which includes various additional branching and conditional instructions. Many of these instructions introduce new hazards or invoke more registers than those in chArm-v2.

(1 point) Implement all of the following four instructions in your `psim`/`pcsim`.

- CSEL
- CSINV
- CSINC
- CSNEG

(1 point) Implement all of the following four instructions in your `psim`/`pcsim`.

- CBZ
- CBNZ
- BR
- BLR

Implementing chArm-v3 will require substantial changes to your hardware elements, pipeline, and hazard control. Additionally, you will likely need to modify some function signatures. You may implement chArm-v3 in weeks 1 and 2, but chArm-v3 extra credit will be graded alongside checkpoint 3. *Clearing charm-v3 corresponds to your correctly running the test programs* in the `testcases/charmv3` subdirectory. Partial credit is given.

6.2.1 Testing

To test the AE changes for chArm-v3, you will use `bin/test-ae -e`. To test chArm-v3 as a whole, you will use `bin/test-se -w3 -e`. You may also test chArm-v3 locally with checkpoint 2, but grading will solely consider successful runs under checkpoint 3.

6.3 Help

Very little additional help will be given on extra credit options, either through the Wiki, Office Hours, or Ed Discussion. When in doubt, read the manual, available on the course syllabus or the course slides, available on Canvas. This should provide all the necessary information on the optional instructions and writeup, respectively.