Assignment 5: Y86-64 System Emulator

CS 429, Spring 2022
Unique IDs: 51215, 51220, 51225, 51230, 51235, 51240
Lead TA: Zach Leeper

Assigned: Saturday, 2 April 2022 11:00 CT
Week 1 Submission: Friday, 8 April 2022 23:59 CT
Week 2 Submission: Friday, 15 April 2022 23:59 CT
Week 3 Submission: Friday, 22 April 2022 23:59 CT
Due: Friday, 29 April 2022 23:59 CT
Last possible hand-in: Sunday, 1 May 2022 23:59 CT

1 Introduction

In this lab, you will be implementing two simulators:

- `psim`, a standalone simulator for the PIPE implementation of the Y86-64 instruction set architecture (ISA), assuming an ideal memory system; and
- `pcsim`, an integrated "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, producing a simulator for the PIPE implementation of the Y86-64 ISA with a realistic memory hierarchy.

For reference, we are also providing you with the source code for `ssim`, a standalone simulator for the SEQ implementation of the Y86-64 ISA.

Outcomes you will gain from this lab include the following:

- You will understand how the SEQ implementation of Y86-64 works. You will understand the utilities of each stage and how they are connected to each other.
- You will understand how the PIPE implementation of Y86-64 works. You will understand how stalling, squashing, and forwarding help handle 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 realistic memory hierarchy.

2 Logistics

This assignment lasts for four weeks and consists of two interrelated parts. You are required to perform four submissions:

- The Week 1 submission, due by Friday, 8 April 2022 23:59 CT.
- The Week 2 submission, due by Friday, 15 April 2022 23:59 CT.
- The Week 3 submission, due by Friday, 22 April 2022 23:59 CT.
- The Final hand-in, due by Friday, 29 April 2022 23:59 CT.

The purpose of the checkpoints are to make sure that you have started and made some demonstrable progress on the assignment.

You may use up your remaining late (slip) days for the checkpoints and final uploads of this assignment. *Note that using slip days for a checkpoint does not adjust any future due dates.* 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, 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 most available slip days. That is, if you have 2 slip days and your partner has 3, the team gets 3 slip days to use for the entire assignment. 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's an implementation-specific question (i.e., private to instructors), please post it on Piazza 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 Piazza.

3 Download and Setup

1. Go to the GitHub Classroom assignment page found at: `https://classroom.github.com/a/bOzahenf`. You'll 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're working with to set up your repository with them. You'll only need one repository for submitting, but if both you and your partner make one, just decide whose to submit later on.

2. From the project root directory, run `make` to compile all the y86 modules and utility programs (see details below). Alternatively, you can use `bash all.sh` to run tests along with the compilation, but a lot of failure messages will pop up, since the simulators haven't been implemented yet.

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:

- **misc**: Source code files for utilities such as `yas` (the Y86-64 assembler) and `yis` (the Y86-64 instruction set simulator). It also contains the `isa.c` source file that is used by all of the processor simulators.
- **seq**: Source code for the SEQ simulator. This is solely for your reference. See file `README` for instructions on compiling and using the simulator.
- **pipe**: Source code for the PIPE simulator. You will modify `psim.c` to complete this simulator in Part A. See file `README` for instructions on compiling and using the simulator.
- **cache**: Source code for the CACHE simulator. You will modify `cache.c` to complete this simulator in Part B. `csim` simulates a cache controller, and is tested using the `test-csim` executable.
- **pipe-cache**: Source code for the "PIPE-with-CACHE" simulator. You will modify `cache.c` and `pcsim.c` to complete this simulator in Part B. See file `README` for instructions on compiling and using the simulator.
- **y86-code**: Y86-64 assembly code for some of the example programs shown in CS:APP3e Chapter 4. You can automatically test your SEQ and PIPE simulators on these benchmark programs. Additionally, there are a few more example programs in this directory with the naming convention `prog-.ys`, which will be used to test the basic PIPE- implementation. See file `README` for instructions on how to run these tests.
- **ptest**: Scripts that generate systematic regression tests of the different instructions, the different jump possibilities, and the different hazard possibilities. These scripts are very good at finding bugs in your implementation. See file `README` for instructions on how to run these tests.

4.2 Utility Programs

The `misc` directory contains two useful programs:

- **yas**: The Y86-64 assembler. This takes a Y86-64 assembly code module (a text file with extension `.ys`) and generates a Y86-64 object module (a "binary" file [1] with extension `.yo`). The easiest way to invoke the assembler is to use or create assembly code files in the y86-code subdirectory. For example, to assemble the program in file `prog1.ys` in this directory, use the command:

  ```
  linux> make prog1.yo
  ```

  Note: The `yas` binary is pre-compiled and is only guaranteed to work on the UTCS x86-64 Linux machines.
- **yis**: The Y86-64 ISA simulator. This program interprets the instructions in a Y86-64 machine-level program according to the instruction set definition, and is the "gold standard" IaaT definition of instruction semantics that your simulators must replicate accurately. To run the program `prog.yo` from within the subdirectory `y86-code`, simply run:

  ```
  linux> ../misc/yis prog.yo
  ```

[1] The generated file actually contains an ASCII version of the object code, and is therefore not truly a binary file.

`yis` simulates the execution of the program and then prints changes to any registers or memory locations on the terminal, as described in CS:APP3e §4.1. For your convenience, `yis` also prints out any intermediate changes for each step.

4.3 Helper Functions

There are several helper functions provided to help you with your `psim.c` implementation:

- `HPACK()` is a macro defined that takes two 4-bit quantities and combines them into a single byte. `HPACK(0x5, 0xB)` returns `0x5B`.
- `HI4()` is a macro defined that takes a byte and returns the upper 4 bits of it. Similarly, `LO4` is a macro that takes a byte and returns the lower 4 bits of it. `HI4(0x5B)` and `LO4(0x5B)` return `0x5` and `0xB` respectively.
- `get_byte_val()`, `get_word_val()`, and `get_reg_val()` are functions for accessing memory or the register file. They are fully declared in `isa.c`, but they essentially take a byte array (simulating the machine's memory) and an address or register id to look up. Accessing memory also requires providing a pointer to the read memory's destination.
- `cond_holds()` is a method provided to check whether a conditional jump or move should be taken, that takes the current condition codes and the condition to be evaluated.
- `compute_alu()` and `compute_cc()` are methods provided for computing the result of an arithmetic function and its resulting condition codes when an `OPq` instruction is being executed.

5 Implementing Simulators

For the PIPE processor implementation, we have provided a dummy implementation in `psim.c` that you can compile and run before you start filling in the missing details.

5.1 Simulator Command Line Options

You can specify several options from the command line of the simulator:

- `-h`: Prints a summary of all of the command line options.
- `-l` *m*: Sets the instruction limit, executing at most *m* instructions before halting. The default limit is 10,000 instructions.
- `-v` *n*: Sets the verbosity level to *n*, which must be between 0 and 3 (for SEQ) or between 0 and 2 (for PIPE), with a default value of 2.
- `-i`: Runs the simulator in interactive mode.

The final command-line argument specifies the object filename.

Here are some typical invocations of the simulators from within the `seq` subdirectory, for example:

```
linux> ./ssim -h
linux> ./ssim ../y86-code/prog1.yo
linux> ./ssim -i ../y86-code/prog1.yo
```

The first case prints a summary of the command line options for `ssim`. The second case runs `ssim` on object file `prog1.yo` until the simulator halts. The resulting register and memory values are compared with those from the ISA simulator (`yis`). The third case launches `ssim` in interactive mode, reads object file `prog1.yo`, and waits for further commands.

In interactive mode, you can use more commands to step through the program:

- `help` : print help message (or use the first letter `h` for short, similar for below)
- `go` : run program to completion
- `next n` : advance `n` instructions
- `cycle n` : advance `n` cycles
- `memory` : display differences in memory
- `registers` : display differences in registers
- `arch` : display processor state
- `undo n` : step back `n` instructions (or use `u n` for short)
- `back n` : step back `n` cycles
- `pipe X` : display pipeline info for stage `X` (`f`, `d`, `e`, `m`, `w`)
- `quit` : exit the program

The same invocations work for the PIPE simulator `psim` from within the `pipe` subdirectory, and with `pipe-cache/pcsim` as well.

The dummy implementation we provide doesn't update any state and never halts, so you will see the simulator falling into an infinite loop and stopping only after reaching the 10,000-instruction limit.

6 Programming Tasks

This lab is a sequence of two programming parts. In Part A, you will implement `psim`, a PIPE simulator. In Part B, 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 nine points: one point for the checkpoint, and four points each for Parts A and B. There is an additional three-point bonus for the extra-credit section.

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

The goal for Part A is to implement a PIPE simulator as described in CS:APP3e §4.5. You are going to complete the code in `pipe/psim.c`. The only functions you will modify are `sim_step_pipe()` and several helper functions in it. The detailed instructions are provided in the comment blocks marked "TODO".

6.1.1 Basic Pipelined Implementation

You are required to implement five functions in `sim_step_pipe()` that emulate five stages for your PIPE simulator:

- `do_fetch_stage()`: Fetch stage.
- `do_decode_stage()`: Decode stage.
- `do_execute_stage()`: Execute stage.
- `do_memory_stage()`: Memory stage.
- `do_writeback_stage()`: Write-back stage.

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, you need to process the stages 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 in the code skeleton; you do not need to worry about it.*

The pipeline register contents for PIPE are declared as `structs` in `stages.h`. The contents of an input pipeline register into the next stage are pointed by `XX_input` (which you should update), and the contents of an output pipeline register from the last stage are pointed by `XX_output`. Thus, `do_decode_stage()` will read from `D_output` and write to `E_input`. See function `sim_init()` in `psim.c` to get an idea of how it works.

6.1.2 Hazard Control

You are also required to implement stalling, squashing, and forwarding to deal with data hazards and control hazards, as described in CS:APP3e §4.5.5. For stalling, you are provided with a helper function `do_stall_check()`. Implement this function to make function `update_pipes()` handle stalling correctly. For forwarding, add your own implementation in the functions for stage updating mentioned above.

6.1.3 Testing

Use the following command to run your PIPE simulator against the standard YIS simulator on a Y86-64 machine-level program (`prog1.yo`, for example):

```
linux> make clean; make
linux> ./psim ../y86-code/prog1.yo
```

There is also a provided `psim-ref` executable that you can use (with the same arguments as `psim`) to check the correct output of a test case. Or you can run an exhaustive test with `ptest`:

```
linux> make clean; make
linux> cd ../ptest
linux> make SIM=../pipe/psim
```

When the ptest program detects an erroneous simulation, it leaves the corresponding .ys file in the directory so that you can debug on it.

6.1.4 Submission

Submit your checkpoint version using Gradescope, by providing a pointer to the private GitHub repository where you have done your work. *Clearing week 1 corresponds to your correctly running test programs* `y86-code/prog1-.yo` *through* `y86-code/prog6-.yo`. *Clearing week 2 corresponds to your correctly running test programs* `y86-code/prog1.yo` *through* `y86-code/prog9.yo`. We will run your PIPE simulator using *ptest* (a regression suite of 743 tests) to check the correctness of your implementation for week 3, along with what's in part B.

6.1.5 Evaluation

Part A of the assignment counts for seven points - three for the basic PIPE- implementation, absent of hazard checking, and three for the full PIPE implementation, including handling hazards. The final point will be for passing all 743 tests in *ptest*. Partial credit is given where applicable, so you'll earn points for each individual test passed.

6.2 Part B: Implementing `pcsim`, A Simulator for the PIPE Implementation with A Realistic 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`.

6.2.1 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 test-csim` and running `test-csim`. Your implementation is correct when the test score printed out is 40/40.
- Next, copy the contents of your `psim/psim.c` into `pipe-cache/pcsim.c`. This will give you a baseline code to start from. Copy stage by stage, and make sure to take note of the existing code provided in the memory stage of `pcsim.c` to see how the new memory functions are used.
- Implement the four functions `get_byte_cache`, `get_word_cache`, `set_byte_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 file `pcsim.c`. This should involve no more than updating the memory routines to the corresponding cache routines and handling stalls resulting from cache misses. In the fetch stage, you might

need the functions `get_byte_val_I` and `get_word_val_I`. In the memory stage, you might need the functions `get_word_val_D` and `set_word_val_D`.

- Check the correctness of the combined simulator using `make test-pipe-cache` in the `ptest` directory. There are a total of 64 tests run against the simulator: eight programs, each tested against eight different cache configurations.

6.2.2 Submission

Submit your final version using Gradescope, by providing a pointer to the private GitHub repository where you have done your work.

6.2.3 Evaluation

Part B of the assignment counts for five points - two for implementing the cache and achieving 40/40 on the cache test, and three for integrating the cache with the PIPE implementation and passing the 64 tests. Partial credit is given where applicable, so you'll earn points for each individual test passed. We will run your `pcsim` simulator using the command `make test-pipe-cache` in the `ptest` directory against eight programs, using eight different cache configurations for each program.

7 Extra Credit (Optional)

There are three optional extra-credit opportunities, each worth one point *towards your course total*.

7.1 Custom Y86 tests

Create an original test case in the file `y86-code/myprog.ys`. In order to earn the bonus, two conditions need to be satisfied:

- This test case needs to be non-trivial and demonstrate some interesting feature or quirk of the architecture. To clarify what we mean by "non-trivial" a bit, try to write a test case that is at least 15 instructions long, don't use `nop` instructions, and avoid copying parts of the provided programs. If you're looking for suggestions, a program that needs to deal with different kinds of pipeline hazards, or is able to take advantage of the cache would be a good place to start.
- Both your `psim` and `pcsim` simulators must correctly execute this test case.

7.2 Additional Y86 Instructions

Implement all of the following instructions as an extension to the Y86 ISA in your PIPE simulator:

- `leaq D(rB), rA`: Load the effective address into `rA` following the Y86 Base + Displacement addressing mode.
- `vecadd rA, rB`: Add the corresponding bytes in `rA` and `rB` together and save the result to `rB`.
- `shlq | shrq | sarq rA, rB`: The x86-like left shift, right logical shift, and right arithmetic shift instructions.

The `yas` and `yis` programs have been implemented to handle these custom instructions already. You can use them as the reference. Modify your PIPE simulator code in-place, and make sure it still works for all previous tests in `ptest`.

7.3 Improved Cache Implementation

For the cache in Part B, we assumed that each cache read/write only accesses a single cache line. Improve your cache implementation in `cache.c` so that it can handle read/write requests spanning multiple cache lines. Run `make cache-bonus` in the `ptest` directory to check your implementation. Your improved cache must still work correctly for the original tests.