

CS 350c, Spring 2016, Laboratory 3

Increasing Performance with Pipelining

Assigned: Thursday, April 21, 2016

Due: Thursday, May 5, 2016, by 10 am

LATE Submissions Will Not Be Accepted!

1 Introduction

In this lab, you will learn about pipelining the implementation of the SM. This lab will require that you write C-language functions to control various parts of our pipeline-level model of the SM *implementation* data-path. This laboratory will help make clear some of the interactions that can occur in a pipelined processor implementation.

Our SM implementation is just a C-language model of what a highly-simplified pipelined implementation might contain. There are hardware descriptions languages (HDLs), such as VHDL and Verilog, that provide specific language primitives for representing registers, register files, and other common hardware structures. Such HDLs help make what part of the model is a specification hardware that should be manufactured and what part is just supporting infrastructure so that hardware part of the model can be simulated. Our pipeline-level model is just that, a crude, *clock-cycle-level* model of a possible implementation of the SM.

HDLs have a parallel evaluation semantics; this is done to model physical hardware circuits that operate continuously. In our case, each of the pipeline sections (fetch, decode, execute, memory, and write-back) operate in parallel. Our SM pipeline-level implementation (model) must provide control that allows SM instructions to be broken into pieces so that our SM pipeline-level implementation enjoys good performance. Such pipeline-level implementation require control mechanisms so that such a pipeline always functions correctly. Note, our SM pipeline-level implementation is crude – no allowance is made about the resources (area, power, delay, technology) required to implement the functionality desired.

This laboratory is designed to help you more thoroughly understand how a microprocessor is internally configured; in turn, this will help you understand how to be a more effective programmer.

2 Logistics

You are expected to work on this lab alone. However, you may communicate with others concerning your understanding of C code and the various tools, e.g., the compiler, assembler, linker, loader, and other systems issues. And, you may discuss the specification of the class microprocessor, the pipelined-level version of the SM, and any tools that you use – including anything provided to you for our class. The results that you submit in response to laboratory must be created and provided by you alone.

Any clarifications and revisions to the laboratory will be posted on the top-level course web page.

3 Handout Instructions

You will find the file `sm-pipeline.tar` referenced on the homework page of the class website. You will need to download this file so you can use its contents. Of course, there may be changes and/or errors, so please be sure to download the latest version – and periodically check the top-level class web page for any updates or corrections.

This “tar” file will contain a copy of our C-language-based SM microprocessor specification. You will note that this specification loads an initial memory image from a `<filename>` specified on the command line when running the SM emulator.

The format of the ISA-level version of SM and the pipelined version of the SM emulators are both the same. Each line of an input file must contain two numbers: a natural-number memory address (0..65535) and an integer value (-32768..65535) – this *expanded* integer input range allows one to provide input as either natural numbers or integers. In the handout, both the ISA-level version of SM and the pipelined-level version of SM are included in the same file. The SM ISA-level and the pipelined-level microprocessor specifications are included with this laboratory are designed to accept three arguments: `<n>` `<p>` `<filename>`. `<n>` is the number of instructions you want the SM ISA-level emulator to execute, `<p>` is the number of *cycles* you want the SM pipeline-level emulator to execute, and `<filename>` contains lines with address-value pairs.

We will use test datasets to determine whether your solution works correctly. Our test datasets will be in the same format as the SM emulator input file.

Your job in this laboratory is to fill-in several control subroutines that will make the SM pipeline-level emulator work correctly. See the associated file included with this laboratory for further instructions. For improving the control sequencing mechanism of the SM pipeline-level emulator in order to get more performance, you might need to implement some more control subroutine(s) and embed them in the SM pipeline-level emulator. In this case, the pipelined version of SM might be modified.

4 Evaluation

We will run both the ISA-level and the pipeline-level emulators on some datasets of our own creation on your version of the SM pipeline-level emulator control routines, and we hope to receive a report that the program counter, registers, and memory are identical to the SM ISA-level emulator after executing your

version of the SM pipeline-level emulator.

Addresses in the input file need not contiguous nor ordered. Any non-initialized address will be (initialized to) 0 because every memory location (in both the ISA-level and pipeline-level emulators) is initialized to 0 prior to reading your input program.

The maximum score for this laboratory is 100 points. The value of the individual components is as follows.

- To get the SM pipeline-level emulator to work correctly is critical. Do not think of optimizations; create the simplest control strategy possible, and just get the SM pipeline-level emulator to work correctly. Value: 70 points.
- To get a stream of non-interfering sequence of ALU instructions to execute correctly, one per *clock cycle* on the pipeline-level evaluator. Value: 10 points.
- To get a mixture of non-interfering sequence of ALU and memory operations to execute correctly, one per *clock cycle* on the pipeline-level evaluator. Value: 10 points.
- To do the above, with jumps. Value: 10 points.
- To do handle every mixture of instructions without error. Bonus value: 20 points.

5 Running the SM Emulator

To use the SM emulator, you will need to compile “simple-micro-0.7.c” that is provided with the laboratory; this code has been compiled successfully on the departmental Linux machines as well as an Apple Macintosh running OSX 10.11.3. Below is a command sufficient to compile this program so you may run it; you only need to type:

```
gcc -Wall -O2 -o sm simple-micro-0.7.c
```

The resulting compiled program can be run by just typing (all on one line):

```
./sm <num_of_ISA_level_instructions>  
    <num_of_pipeline_level_clock_cycles>  
    <filename_with_memory_image>
```

where `<num_of_ISA_level_instructions>` is a natural number indicating how many ISA-level instructions to simulate, `<num_of_pipeline_level_clock_cycles>` is a natural number of the number of *clock cycles* to simulate, and `<filename_with_memory_image>` contains an initial memory image for both simulators.

After reading the input file, the SM ISA-level emulator will begin execution at memory address 0; that is, the PC will be set to 0. So, your program needs to work starting at address 0. The first thing your program should do is to set the stack pointer to 0xFFFF; that is, put the stack pointer at the top of the memory. The SM ISA-level emulator will be run `<num_of_ISA_level_instructions>` steps.

After the ISA-level emulator has finished its execution, the SM pipeline-level emulator will be run. The SM ISA-pipeline emulator will begin execution at memory address 0 (just like the ISA-level simulator) ; that is, the PC will be set to 0, and execution will begin. To get the SM ISA-pipeline model to *empty* the pipeline of partially executed instructions, you may need to cause the SM ISA-pipeline emulator to (effectively) execute enough NOP instructions to cause the pipeline to empty.

Once both emulations are complete, we will report if any of the state between the two machines is different. If so, then possibly there is a bug in the control logic for the SM ISA-pipeline emulator.

Hints

Start simply. First, get the pipeline to function correctly – even if your control implementation only allows a single instruction to propagate down the pipeline to completion. Once a simple, but inefficient version is working, then you may start to improve the control sequencing mechanism to get more performance from your implementation.

Hand In Instructions

Please follow the instructions below for turning in your work. Note, this laboratory may not be submitted late! This laboratory must be submitted by the due date as this laboratory is due one day before the last day of classes!

- Make sure you have included your identifying information in your submission. In this laboratory, you will provide several C-language subroutines that control the various pipeline stages.
- To submit your lab, you should provide a Unix **tar** file that contains your implementation for the lab. The **tar** file must contain your modified version of the pipelined control file, where you have implemented the control algorithms in the pipe-level SM implementation. The **tar** file can also contain your modified version of the SM file.
- To handin your pipeline laboratory report, please include your report in the **tar** file (mentioned just above) as a PDF file. Name the file containing your report as <YourUTID>.pdf. The Canvas system identifies you in this manner.

Submit your laboratory report by using CANVAS.