6.S195: Lab 4
Starting with SMIPS

Andy Wright [*]

October 2, 2013

Due: Wednesday October 9, 2013

1 Introduction

This lab introduces the SMIPS processor and the toolflow associated with it. The lab begins with the introduction of a single cycle implementation of a SMIPS processor. You will be building upon this design to create a two cycle, non-pipelined implementation. Next you will pipeline the two cycle implementation so fetch and execute are happening in parallel. Last you will create a four cycle, non-pipelined implementation so you have a starting point for later labs.

This lab corresponds to chapters 5 and 6 in the course book.

2 Testing Infrastructure

Start by checking out the `lab4.git` repository. This repository contains a one-cycle SMIPS implementation in `1cyc.bsv`. There is also placeholder `mkProc` modules in the files `2cyc.bsv`, `2stage-pipeline.bsv`, and `4cyc.bsv` for later implementation. In addition to these files, there is also a link a script, `smips`, designed to test your processor implementations against a suite of software tests.

Lets start by testing the processor implemented in `1cyc.bsv`. The `smips` script takes a variety of inputs depending on what you are trying to test. The command below cleans your build directory (`-x`), compiles 1cyc.bsv (`-b 1cyc.bsv`), compiles all the tests found in `/mit/6.s196/programs/src` (`-c all`), and runs these tests on the processor (`-r`).

```
    ./smips -x -b 1cyc.bsv -c all -r
```

The output from this script is a combination of the output from the Bluespec compiler from building `1cyc.bsv` and the output from the test programs running

[*]The previous version of this lab was written by Muralidaran Vijayaraghavan and Sang-Woo Jun

on the SMIPS processor. If you look at the output from this script, you will see `PASSED` in a lot of places showing that the processor is working, but there are two places that may not seem right.

2.1 Printing from SMIPS programs

First look at the output from `/mit/6.s195/programs/src/print`. This program tests printing functions designed for the SMIPS processor, and its output can be seen below.

```
chello world
dhello world
        25
```

This *almost* looks like your standard hello world program, except there are a few things that seem off. If you could see the original code for this program, then you would be able to figure out if this behavior was expected or not. Luckily the print example is one of the programs that was compiled from C to SMIPS using a cross compiler called by the `smips` script. The original source code can be found in `/mit/6.s195/programs/src/print/`.

Exercise 1 (2 Points): Judging from `main.c`, is hello world supposed to be repeated twice? What are those characters there for? What is some other functionality tested in this program? Write your answers in discussion.txt.

This answered many questions, but what about the 25? It clearly came from `printInt(25)`, but why is there space before it? To answer this question, we dive into how printing works for this test setup.

The printing functions used in `main.c` are found in the file `/mit/6.s195/programs/lib/print.c`. These functions write characters to one coprocessor registers for `printChar` and `printString`, and it writes to another coprocessor register for `printInt`. The coprocessor sends information about these writes out of the processor using the `cpuToHost` portion of the interface. The test bench receives these writes and uses `$display("%c",c)` for characters and `$display("%d",i)` for integers. The function `$display("%d",i)` adds whitespace to the outputted number corresponding to the largest possible number of digits needed for the type of `i`. This is why there is whitespace before the 25.

2.2 Debugging messages from processor code

Now take a look at the output from the `smips` script again. At the very bottom you will notice an error:

```
Executing unsupported instruction at pc: 000010b0. Exiting
```

This error was not printed by a SMIPS test program, it was generated from within `1cyc.bsv` when an unsupported instruction was decoded. This error was printed to `stderr` by bsim. There is additional debugging information written to `stdout` instead, but this information is piped to a `simOut` file when the simulation is run from `smips`. This `simOut` file can be found in the build directory at `build/1cyc.bsv/hwmultiply/simOut`. This file contains the following line for the unsupported instruction.

```
cycle         31
pc: 000010b0 inst: (00850018) expanded: mult 'h04 'h05
```

This shows that the unsupported instruction is a `mult` instruction (makes sense considering the name of the program is hwmultiply). Since our processor's ALU does not have a multiplier in it, the `mult` instruction cannot be performed. This program will always fail due to unsupported instructions until the final project. Don't worry about it before then.

You should now have enough understanding of the testing structure around the SMIPS processor to make modifications to a processor and test them against the provided test programs.

3 Multi-cycle SMIPS Implementation

The provided code, `1cyc.bsv`, implements a one-cycle non-pipelined SMIPS implementation. This code uses functions, modules, and types defined in the bsv files found in `/mit/6.s196/common-lib/`. In this lab you will make two different multicycle implementations motivated by possible structural hazards. You will also get some practice pipelining a processor with a two-stage SMIPS pipeline.

3.1 Two-cycle SMIPS implementation to support memory sharing

The one-cycle SMIPS processor can only work if there are separate instruction and data memories. If there is only one memory that holds both instructions and data, then there is a structural hazard (assuming the memory cannot be accessed twice in the same cycle). To get around this hazard, you can split the processor into two cycles: instruction fetch and execute.

1. Instruction fetch reads the current instruction from the memory and decodes it.

2. Execute reads from the register file, does ALU operations, does memory operations, and writes back to the register file.

When splitting the processor to a two cycle implementation, you will need a register to keep intermediate data between the two stages, and you will need a state register to keep track of the current state. The intermediate data register will be written to during fetch, and it will be read from during execute. The state register will toggle between fetch and execute. You can use the provided `Stage` typedef as the type for the state register to make things easier.

Exercise 2 (5 Points): Implement a two-cycle SMIPS processor in `2cyc.bsv` using a single memory for instructions and data. The module `mem` has been provided for you to use as your single memory. Test this processor using the following command:

```
    ./smips -x -b 2cyc.bsv -c all -r
```

3.2 Four-cycle SMIPS implementation to support memory latency

The one and two-cycle SMIPS processors assume a memory that has combinational reads; that is, if you set the read address, then the data from the read will be valid during the same clock cycle. Most memories have reads with longer latencies: first you set the address bits, and then the read result is ready on the next clock cycle. If we change the memory in the previous SMIPS processor implementations to a memory with a read latency, then we introduce another structural hazard: results from reads cannot be used in the same cycle as the reads are performed. This structural hazard can be avoided by further splitting the processor into four cycles: instruction fetch, instruction decode, execute, and write back.

1. Instruction fetch sets the address lines on the memory to `PC` to read the current instruction.
2. Instruction decode gets the instruction from memory, decodes it, and reads registers.
3. Execute will perform ALU operations, write data to the memory for store instructions, and set memory address lines for read instructions.
4. Write back will get any read results from the memory and it will write back to the register file (either from the ALU or from the memory).

This processor will require more registers between stages and an expanded state register. You can use the modified `Stage` typedef as the type for the state register.

A one-cycle read latency memory is implemented by `mkDelayedMemory`. This module has an interface, `DelayedMemory`, that decouples memory requests and memory responses. Requests are still made in the same way using `req`, but this method no longer returns the response at the same time. Instead you have to call the `resp` action value method in a later clock cycle to get the memory response from the previous read. More details can be found in the source file `DelayedMemory.bsv` in `/mit/6.s195/common-lib/`.

Exercise 3 (5 Points): Implement a four-cycle SMIPS processor in `4cyc.bsv` as described above. Use the delayed memory module `mem` already included in `4cyc.bsv` for both instruction and data memory. Test this processor using the following command:

```
    ./smips -x -b 4cyc.bsv -c all -r
```

3.3 Two-stage SMIPS pipeline

While the two-cycle and four-cycle implementations allow for processors that handle certain structural hazards, they do not do well in performance. All processors today are pipelined to increase performance, and they often have duplicated hardware to avoid structural hazards such as the memory hazards seen in the two- and four-cycle SMIPS implementations. Pipelining introduces many more data and control hazards for the processor to handle. To avoid data hazards for now, we will only look at a two-stage pipeline.

The two-stage pipeline uses the way the two-cycle implementation splits the work into two stages, and it runs these stages in parallel using separate instruction and data memories. This means as one instruction is being executed, the next instruction is being fetched. For branch instructions, the next instruction is not always known. This is known as a control hazard.

To handle this control hazard, use a PC+4 predictor in the fetch stage and correct the PC when mispredictions occur. Make sure to kill wrong path instructions as shown in lecture.

Exercise 4 (5 Points): Implement a two-cycle pipelined SMIPS processor in `2stage-pipeline.bsv` using separate instruction and data memories (with combinational reads, just like the memories from 1cyc.bsv). Test this processor using the following command:

```
    ./smips -x -b 2stage-pipeline.bsv -c all -r
```

3.4 Instructions per cycle

Processor performance is often measured in instructions per cycle (IPC). This metric is a measure of throughput, or how many instructions are completed per cycle on average. The one-cycle implementation gets 1.0 IPC, but since the clock required for the one-cycle implementation to work is very slow, this is not as fast as it sounds. The two-cycle and four-cycle implementations achieve 0.5 and 0.25 IPC respectively.

The pipelined implementation of the processor will achieve somewhere between 0.5 IPC and 1.0 IPC. Since the only thing keeping your processor from an IPC of 1.0 is branch misprediction, the actual IPC of your processor depends only on the accuracy of your PC+4 branch predictor.

Exercise 5 (3 Points): What is the IPC for the two-stage pipelined processor for each benchmark tested by the `smips` script? (Just the programs that report instructions and cycles). What is the accuracy of the PC+4 branch predictor for each benchmark? Write your answers in discussion.txt.

When you are done with your lab, commit all the changes using `git commit -am "Done with lab 4"` and push your changes to the course locker for grading using `git push`.