Lab 6: SMIPS 6-stage Pipeline and Branch Prediction

Due: Friday October 31, 2014

1 Introduction

This lab is your introduction to realistic SMIPS pipelines and branch prediction. At the end of this lab, you will have a six-stage SMIPS pipeline with multiple address and branch predictors working together.

2 Additions to the Lab Infrastructure

2.1 New BSV Files

2.1.1 Within `src/includes/`:

`FPGAMemory.bsv` A wrapper for block RAM commonly found on FPGAs. This has an identical interface as the DelayedMemory from the previous lab.

`SFifo.bsv` Two searchable FIFO implementations: one based off of a pipeline FIFO and one based off of a bypass FIFO. Both implementations assume search is done immediately before enq.

`Scoreboard.bsv` Two scoreboard implementations based off of searchable FIFOs. The pipeline scoreboard uses a pipeline searchable FIFO, and the bypass scoreboard uses a bypass searchable FIFO.

`Bht.bsv` An empty file in which you will implement a branch history table (BHT).

2.1.2 Within `src/`:

`TwoStage.bsv` An initial two-stage pipelined SMIPS processor that uses a BTB for address prediction. Compile with `twostage` target.

`SixStage.bsv` An empty file in which you will extend the two-stage pipeline into a six-stage pipeline. Compile with `sixstage` target.

`SixStageBHT.bsv` An empty file in which you will integrate a BHT into the six-stage pipeline. Compile with `sixstagebht` target.

`SixStageBonus.bsv` An empty file in which you can improve the previous processor for bonus credit. Compile with `sixstagebonus` target.

2.2 Testing Improvements

In the previous lab, the command `build -v <proc_name>` (run from the `scemi/sim/` directory) was used to build `bsim_dut` and `tb`. In this lab, this command builds `<proc_name>_dut` instead of `bsim_dut` so switching between processor types does not delete other processor builds.

Simulation scripts now require you to specify the target processor:

```
$ ./run_assembly <proc_name>
$ ./run_benchmarks <proc_name>
```

Simulating a single test requires you to run the correct simulation executable:

```
$ ./<proc_name>_dut -V out.vcd > out.txt &
$ ./tb ../../programs/build/<test_name>.vmh
```

3 Two-Stage Pipeline: `TwoStage.bsv`

`TwoStage.bsv` contains a two-stage pipelined SMIPS processor. This processor differes from the processor you built in the previous lab because this processor uses a FIFO between the two stages.

Discussion Question 1 (10 Points): Debugging practice! If you replace the BTB with a simple `pc + 4` address prediction, the processor still works, but it does not perform as well. If you replace it with a really bad predictor that predicts `pc` is the next instruction for each `pc`, it should still work but have even worse performance because each instruction would require redirection (unless the instruction loops back to itself). If you actually set the prediction to `pc`, you will get errors in the assembly tests; the first one will be from `cache.asm.vmh`. What is the error you get? What is happening in the processor to cause that to happen? Why do not you get this error with any other predictor? How would you fix it? You do not actually have to fix this bug, just answer the questions.

4 Six-Stage Pipeline: `SixStage.bsv`

The six-stage pipeline should be divided into the following stages:

- Instruction Fetch – request instruction from iMem and update PC
- Decode – receive response from iMem and decode instruction
- Register Fetch – read from the register file
- Execute – execute the instruction and redirect the processor if necessary
- Memory – send memory request to dMem
- Write Back – receive memory response from dMem (if applicable) and write to register file

`IMemory` and `DMemory` instances should be replaced with instances of `FPGAMemory` to enable later implementation on FPGA.

Exercise 1 (20 Points): Starting with the two-stage implementation in `TwoStage.bsv`, replace each memory with `FPGAMemory` and extend the pipeline into a six-stage pipeline in `SixStage.bsv`.

Discussion Question 2 (5 Points): What evidence do you have that all stages are able to firing in the same cycle?

Discussion Question 3 (5 Points): In your six-stage pipelined processor, how many cycles does it take to correct a mispredicted instruction?

Discussion Question 4 (5 Points): If an instruction depends on the result of the instruction immediately before it in the pipeline, how many cycles is that instruction stalled?

Discussion Question 5 (5 Points): What IPC do you get for each benchmark?

5 Adding a Branch History Table: `SixStageBHT.bsv`

The branch history table (BHT) is a structure that keeps track of the history of branches and is used in direction prediction. Your BHT should be indexed by a parameterized number of bits taken from the program counter – typically bit n+1 down to bit 2 since bits 1 and 0 will always be zero. Each index should have a two-bit saturating counter. Do not include any valid bits or tags in the BHT; we are not concerned about aliasing in our predictions.

Exercise 2 (20 Points): Implement a branch history table in `Bht.bsv` that uses a parameterizable number of bits as an index into the table.

Discussion Question 6 (10 Points): Planning! One of the hardest things about this lab is properly training and integrating the BHT into the pipeline. There are many mistakes that can be made while still seeing decent results. By having a good plan based on the fundamentals of direction prediction, you will avoid many of those mistakes.

For this discussion question, state your plan for integrating the BHT into the SMIPS pipeline. The following questions should help guide you:

Where will the BHT be positioned in the pipeline? What pipeline stage performs lookups into the BHT? In which pipeline stage will the BHT prediction be used? Will the BHT prediction need to be passed between pipeline stages?

Do you need to add a new FIFO for redirecting using BHT data? Do you need to add a new epoch? How does the fetch stage handle the redirects? Do you need to change anything to the current instruction and its data structures if redirecting?

How will you train the BHT? Which stage produces training data for the BHT? Which stage will use the interface method to train the BHT? Do you need a new FIFO to send training data? For which instructions will you train the BHT?

How will you know if your BHT works?

Exercise 3 (20 Points): Integrate a 256-entry (8-bit index) BHT into the six-stage pipeline from `SixStage.bsv`, and put the results in `SixStageBHT.bsv`.

Discussion Question 7 (5 Points): How much improvement do you see in the `bpred_bht.asm.vmh` test over the processor in `SixStage.bsv`?

Exercise 4 (10 Points): Move address calculation for jump (J) and jump-and-link (JAL) up to the decode stage and use the redirect logic used by the BHT to redirect for these instructions too.

Discussion Question 8 (5 Points): How much improvement do you see in the `bpred_j.asm.vmh` and `bpred_jal.asm.vmh` tests over the processor in `SixStage.bsv`?

Discussion Question 9 (5 Points): What IPC do you get for each benchmark? How much improvement is this over the original six-stage pipeline?

6 Bonus Improvements: `SixStageBonus.bsv`

This section looks at two ways to speed up indirect jumps to addresses stored in registers (JR).

Exercise 5 (10 Bonus Points): Jump-to-Register (JR) instructions have known target addresses in the register fetch stage. Add a redirection path for JR instructions in the register fetch stage and put the results in `SixStageBonus.bsv`. The `bpred_ras.asm.vmh` test should give slightly better results with this improvement.

Most JR instructions found in programs are used as returns from method calls. This means the target address for the jump was written by a previous JAL instruction. Without nested instructions, you can keep track of the last link address for a JAL instruction and jump to that. With nested method calls, you need to keep a stack of addresses on a stack. Push a link address for each JAL and pop a return address for each JR. This stack, known as a return address stack (RAS), is very efficient at predicting target addresses for JR instructions earlier than the register fetch stage.

Exercise 6 (10 Bonus Points): Implement a return address stack and integrate it into your processor. An 8 element stack should be enough. If the stack fills up, the stack pointer should wrap around to the begining, overwriting the oldest data. The `bpred_ras.asm.vmh` test should give even better results with this improvement. If you implemented the RAS in a separate BSV file, make sure to add it to the git repository for grading.