Lab 5: SMIPS Introduction – Multi-Cycle and Two-Stage Pipeline

Due: Monday October 20, 2014

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 then create two and four cycle implementations driven by memory structural hazards. You will finish by creating a two stage pipelined implementation so fetch and execute are happening in parallel. This two stage pipeline will be the bases for future pipelined implementations.

2 The Processor Infrastructure

A large amount of work has already been done for you in setting up the infrastructure to run, test, evaluate performance, and debug your SMIPS processor in simulation and on the FPGA. The processor designs for this lab cannot be run on FPGAs because of the type of memory used, but the next lab will be able to be synthesized for an FPGA.

2.1 Initial Code

The code provided for this lab has three folders in it: `programs/`, `scemi/`, and `src/`. `programs/` contains SMIPS programs in assembly and C. `scemi/` contains the infrastructure for compiling and simulating the processors. `src/` contains the BSV code for the SMIPS processors.

Within the BSV source folder, there is a folder `src/includes/` which contains the BSV code for all the modules used in the SMIPS processors. These files are briefly explained below.

`Btb.bsv` Implementations of a branch target buffer address predictor.

`Cop.bsv` Implementation of the coprocessor module.

`DMemory.bsv` Implementation of the data memory using a massive register file.

`Decode.bsv` Implementation of the instruction decoding.

`Ehr.bsv` Implementation of EHRs as described in the lectures.

`Exec.bsv` Implementation of the instruction execution.

`Fifo.bsv` Implementation of a variety of FIFOs using EHRs as described in the lectures.

`IMemory.bsv` Implementation of the instruction memory using a massive register file.

`MemInit.bsv` Modules for downloading the initial contents of instruction and data memories from the host pc.

`MemTypes.bsv` Common types relating to memory.

`ProcTypes.bsv` Common types relating to the processor.

`RFile.bsv` Implementation of the register file.

`Types.bsv` Common types.

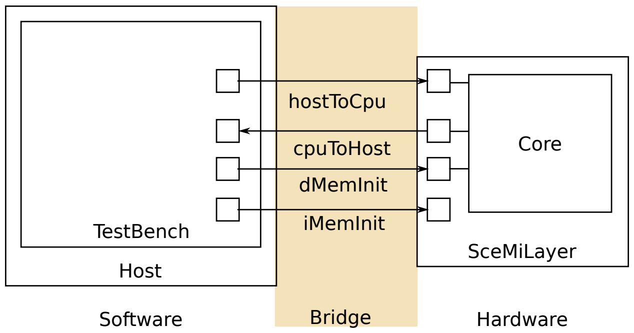

Figure 1: SceMi Setup

2.2 The SceMi Setup

Figure 1 shows the SceMi setup for the lab. The SceMiLayer instantiates the processor from the specified processor BSV file and SceMi ports for the processors's hostToCpu, cpuToHost, iMemInit, and dMemInit interfaces. The SceMiLayer also provides a SceMi port for reseting the core from the test bench, allowing multiple programs to be run on the Processor without reprogramming the FPGA or restarting the bluesim executable.

Source code for the SceMiLayer and Bridge are in the `scemi/` directory. The SceMi link goes over a TCP bridge for simulation and a PCIe bridge when running on the actual FPGA.

2.3 Building the Project

The file `scemi/sim/project.bld` describes how to build the project using the `build` command which is part of the Bluespec installation. Run `build --doc` for more information on the `build` command. The full project, including hardware and testbench, can be rebuilt from scratch by running the command `build -v <proc_name>` from the `scemi/sim/` directory where `<proc_name>` is one of the processor names specified in this lab handout[1]. This will overwrite the executable generated by any previous call to `build`.

Building a processor also generates the Bluespec Workstation project file `<proc_name>.bspec` which can be used to debug the project using the schedule analysis tool or the module browser and waveform viewer.

2.4 Compiling the Assembly Tests and Benchmarks

Our SceMi test bench runs SMIPS programs specified in Verilog Memory Hex (vmh) format. The `programs/` directory contains the source code for a number of assembly tests and benchmark programs you can try out on your processor. A Makefile is provided for compiling the programs to the required `.vmh` format.

To compile all the assembly tests and benchmarks, go to the `programs/` directory and run the `make` command.

This will create a new directory under the `programs/` directory called `build/`, which contains the generated `.vmh` files along with other intermediate results. Compile the assembly tests and benchmarks now.

Those files in the `programs/build/` with the `.asm.vmh` extension are assembly tests. These are microbenchmarks written in assembly which test specific instructions or to give specific performance results and are explained below:

`baseline.asm.vmh` Returns how many cycles it takes to run 100 consecutive NOP instructions[2]. This microbenchmark returns 102 cycles for the one cycle processor because it takes a few cycles to set up the measurement.

[1]Running `build -v` by itself will print an error message containing all valid processor names.
[2]NOP is short for No Operation.

`bpred_*.asm.vmh` Microbenchmarks that contain a lot of branches to test the performance of various branch predictors. These benchmarks return the number of cycles required to run the benchmark. These results should be compared relative to other processors with other branch predictors.

`bpred_bht.asm.vmh` Contains many branches that a branch history table can predict well.

`bpred_btb.asm.vmh` Contains many branches that a branch target buffer can predict well.

`bpred_jal.asm.vmh` Contains many jump-and-link (JAL) instructions.

`bpred_j.asm.vmh` Contains many jump (J) instructions.

`bpred_ras.asm.vmh` Contains many jumps to registers that a return address stack (RAS) can predict well.

`cache.asm.vmh` Tests a cache by writing to and reading from addresses that would alias in a smaller memory.

`smipsv<num>_<op>.asm.vmh` Tests a specific instruction and prints `PASSED` or `FAILED` depending on the results of the test.

It is highly recommended you rerun all the assembly tests after making any changes to your processor to verify you didn't break anything. Also, run the assembly tests when trying to locate a bug, as they will narrow down which instructions are problematic.

Those files in the `programs/build/` directory with the extension `.bench.vmh` are benchmarks which can be used to evaluate the performance of your processor. When completed, the benchmarks print out the total number of instructions executed and the number of cycles required to execute those instructions. Performance is measured in instructions-per-cycle (IPC). The greater the IPC the better. For our pipeline we can never exceed an IPC of 1, but we should be able to get close to it with a good branch predictor and proper bypassing.

2.5 Using the Test Bench

Our SceMi test bench is software run on the host processor which interacts with the SMIPS processor over the SceMi link, as shown in figure 1. The test bench loads a program for the SMIPS processor to execute, starts the processor, and handles `toHost` requests until the processor indicates it has completed, either successfully or unsuccessfully.

The test bench takes `.vmh` files as arguments. These files are compiled SMIPS programs and are loaded and run on the simulated processor in order.

To run the test bench, first build the project as described in section 2.3 and compile the SMIPS programs as described in section 2.4. For simulation the executable `bsim_dut` will be created, which should be running when you start the test bench.

For example, to run the qsort benchmark on the processor in simulation you could use the commands:

```
./bsim_dut > qsort.out &
./tb ../../programs/build/qsort.bench.vmh
```

The test bench outputs the result of the program and statistics. The SMIPS program could either fail, or pass, as determined by a value in the toHost register in the SMIPS Processor, which is set by the running SMIPS program.

For your convenience, we have provided scripts `run_assembly` and `run_benchmarks` in the `sim/` directory which run all of the *compiled* assembly tests and benchmarks respectively.

2.6 Test Bench Output

There are three sources of outputs from SMIPS simulation. These include BSV display statements (both messages and errors), SMIPS print statements, and VCD waveform dumps. Debugging the trickiest bugs sometimes takes all three.

BSV `$display` statements are printed to stdout by `bsim_dut`. BSV can also print to stderr using `$fwrite(stderr, ...)` statements. The scripts `run_assembly` and `run_benchmarks` redirect the stdout of `bsim_dut` to `/dev/null` so you will not see the output of `$display` statements, but you will still see errors printed using `$fwrite`. The above example pipes stdout to `qsort.out` so you can find the output of BSV's display statements there.

SMIPS print statements are handled through moving characters and integers to coprocessor registers. The testbench reads from the coprocessor interface and prints characters and integers to stdout when it sees them.

VCD waveform dumps are saved only when `bsim_dut` is called with the `-V` flag and a file name like this:

```
./bsim_dut -V qsort.vcd > qsort.out &
./tb ../../programs/build/qsort.bench.vmh
```

These waveforms can then be explored using gtkwave from the Bluespec Workstation module viewer.

Exercise 1 (0 Points): Compile the test programs by going to the `programs/` directory and using the `make` command. Compile the one-cycle SMIPS implementation and test it by going to the `scemi/sim/` directory and using the following commands:

```
build -v onecycle
./run_assembly
./run_benchmarks
```

3 Multi-cycle SMIPS Implementations

The provided code, `src/OneCycle.bsv`, implements a one-cycle Harvard architecture[3] SMIPS processor. This processor is only able to do operations in a single cycle because it has separate instruction and data memories, and each memory gives responses to loads in the same cycle. In this portion of the lab you will make two different multicycle implementations motivated by more realistic memory structural hazards.

3.1 Two Cycle Von Neumann Architecture SMIPS implementation

An alternative to the Harvard architecture is the von Neumann architecture[4]. The von Neumann architecture has instructions and data stored in the same memory. 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 (15 Points): Implement a two-cycle SMIPS processor in `TwoCycle.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 by going to the `scemi/sim/` directory and using the following commands:

```
build -v twocycle
./run_assembly
./run_benchmarks
```

[3]The Harvard architecture has separate instruction and data memories
[4]The von Neumann architecture is also called the Princeton architecture

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. In order to get the results of a requested load, you have to call the `resp` action value method in a later clock cycle to get the memory response from the previous read. The `resp` method will not return anything for stores, so it should not be called for them. More details can be found in the source file `DelayedMemory.bsv` in `src/includes/`.

Exercise 3 (15 Points): Implement a four-cycle SMIPS processor in `FourCycle.bsv` as described above. Use the delayed memory module `mem` already included in `FourCycle.bsv` for both instruction and data memory. Test this processor using the following command:

```
build -v fourcycle
./run_assembly
./run_benchmarks
```

4 Two-stage pipeline SMIPS Implementation

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 (30 Points): Implement a two-cycle pipelined SMIPS processor in `TwoStage.bsv` using separate instruction and data memories (with combinational reads, just like the memories from `OneCycle.bsv`). Test this processor using the following command:

```
build -v twostage
./run_assembly
./run_benchmarks
```

4.1 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. IPC is computed by dividing a number of instructions by how many cycles it took to execute those instructions. 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 next address predictor.

Discussion Question 1 (5 Points): What is the IPC for the two-stage pipelined processor for each benchmark tested by the `run_benchmarks` script?

Discussion Question 2 (5 Points): What is the formula to compute the next address predictor accuracy from the IPC? (Hint, how many cycles does it take to execute an instruction when the PC+4 prediction is correct? What about when it is incorrect?) Using this formula, what is the accuracy of the PC+4 next address predictor for each benchmark?

4.2 Next Address Prediction

Now lets use a more advanced next address predictor. One such example is a branch target buffer (BTB). A BTB keeps track of previously used next addresses that were not PC+4, and it is used to compute the next address instead of PC+4 whenever it contains a next PC for the current PC.

`Btb.bsv` contains an implementation of a BTB. Its interface has two methods: `predPc` and `update`. The method `predPc` takes the current program counter and it returns a prediction. The method `update` takes a program counter and the next address for the instruction at that program counter and adds it as a prediction if it is not PC+4.

The `predPc` method should be called in the first pipeline stage, and the `update` method should be called in the last pipeline stage. Since `update` takes in the program counter for the current instruction, you will need to store the PC for the instruction that is currently in the execute stage of the pipeline. You will also need to use this PC in the `Exec` function so it can compute branch targets correctly (PC-4 will not work anymore to compute the PC of that instruction).

The `mispredict` field of `ExecInst` will be very useful here.

Exercise 5 (10 Points): Add a BTB to your two-cycle pipelined SMIPS processor and save the results in `TwoStageBTB.bsv`. Test this processor using the following command:

```
build -v twostagebtb
./run_assembly
./run_benchmarks
```

Discussion Question 3 (5 Points): What is the IPC for the two-stage pipelined processor with a BTB for each benchmark tested by the `run_benchmarks` script? How much has it improved over the previous version?

Discussion Question 4 (5 Points): How does adding the BTB change the performance of the `bpred_*` microbenchmarks? (Hint: the number of cycles for `bpred_btb` should go down.)

Bonus Discussion Questions

Discussion Question 5 (5 Bonus Points): Look at the assembly source for the `bpred_*` benchmarks and explain why each benchmark improved, stayed the same, or got worse.

Discussion Question 6 (5 Bonus Points): How would you improve the BTB to improve the results of `bpred_bht`?