6.S195: Lab 6
6-Stage SMIPS Pipeline with Simple Branch Predictor

October 24, 2013

Due: Sunday November 3, 2013

Note: This lab uses a different infrastructure than the previous SMIPS lab in order to compile to FPGAs. Future SMIPS labs will use this same infrastructure.

1 Introduction

In this lab you will be provided a two stage pipelined implementation of a SMIPS processor, and you will be further pipelining the processor into a six stage pipeline and adding a simple branch target buffer (BTB) as a branch predictor. This 6 stage pipeline will be your starting point for more advanced branch predictors and memory caches in later labs.

This lab also introduces the flow to compile Bluespec designs for an FPGA. By compiling your designs with FPGA synthesis tools, you can get metrics about your design's performance such as area and clock speed. This will allow you to better compare the performance of two different designs.

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. This section describes that infrastructure.

2.1 Getting Started

The code for this lab is in the lab6.git repo. You can clone this repo to a local directory by running the following command.

```
$ git clone $GITROOT/lab6.git
```

This will create a directory called `lab6` with the code for this lab.

2.2 The Source Code

The included processor has a 2 cycle pipeline connected with conflict-free fifos. Currently the processor uses memories implemented as massive register files. This implementation will not fit on an FPGA, so these memory implementations will be replaced with a memory module that will be synthesized to on board ram in the FPGA.

The source code implementing the processor and all of its components is split into files in the `src/` directory as follows.

`AddrPred.bsv` Implementations of a simple pc plus 4 address predictor and a branch target buffer 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.

`FPGAMemory.bsv` Implementation of a memory module that will be synthesized to internal FPGA RAM. This memory module will be used for both instruction and data memory.

`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.

`Proc.bsv` A two cycle pipelined implementation of the SMIPS processor. You will be modifying this file during lab to make your 6 stage pipeline.

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

`RFile.bsv` Implementation of the register file.

`Scoreboard.bsv` Implementation of a scoreboard for handling data hazards.

`Types.bsv` Common types.

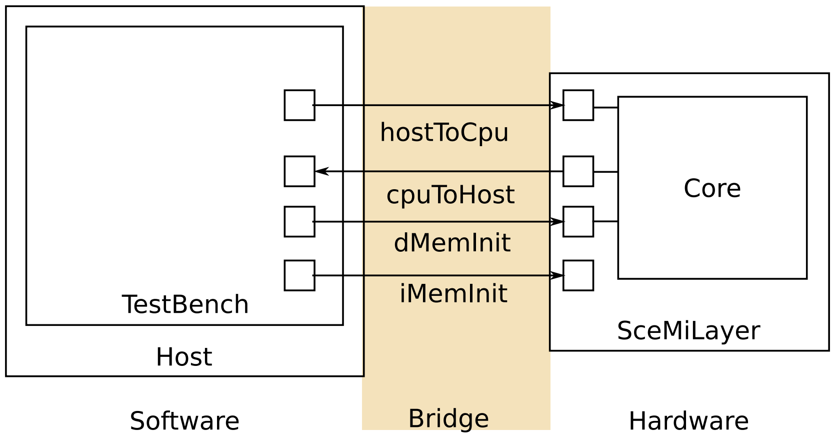

Figure 1: SceMi Setup

2.3 The SceMi Setup

Figure 1 shows the SceMi setup for the lab. The SceMiLayer instantiates the processor from `Proc.bsv` 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.

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.

2.4 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` from the `scemi/sim/` directory.

The file `scemi/sim/sim.bspec` is a Bluespec Workstation project file that can be used to build the project from within the Bluespec Workstation rather than building from the command line. This project can also be used to to run the schedule analysis tool in the Bluespec GUI. To build the project in the Workstation, run from the `scemi/sim/` directory:

```
bluespec sim.bspec&
```

This opens up the Workstation. From there you can compile and link to generate the two executables `bsim_dut` and `tb`. The executable `bsim_dut` simulates the hardware; `tb` is the test bench.

The `fpga/` directory contains its own `project.bld` and `fpga.bspec` for building the project for the FPGA. Building for the FPGA includes running synthesis, map, and place-and-route, and takes on the order of an hour to complete. When the build has completed, the FPGA can be programmed using the `programfpga` command on the FPGA servers. Building for FPGA also creates a `tb` executable for the test bench. Note: you will not be able to successfully synthesize for the FPGA until after completing problem 1 of this lab.

In order to use the xilinx tools, the xilinx settings must be sourced. The course locker includes a script called `xilinx` which runs a command with the xilinx settings sourced. For example, to build for the FPGA:

```
fpga$ xilinx build -v xupv5_dut
     (... wait a long time ...)
fpga$ build -v tb
```

2.5 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 command:

```
programs$ make
```

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. Running the assembly tests is a good way to check for errors in your processor implementation and to narrow down which instructions are the source of any problems.

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.

2.6 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 a single command line argument which is the `.vmh` file with the program to run on the SMIPS processor.

To run the test bench, first build the project as described in section 2.4 and compile the SMIPS programs as described in section 2.5. For simulation the executable `bsim_dut` will be created, which should be running when you start the test bench. For the FPGA you should first program the FPGA with `programfpga` on one of the FPGA servers, and wrap the call to `tb` in `runtb`.

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

```
sim$ ./bsim_dut 2> qsort.err > qsort.out &
sim$ ./tb ../../programs/build/qsort.bench.vmh
```

To run the qsort benchmark on the FPGA, assuming the design has been synthesized already, you could use the commands:

```
fpga$ programfpga
fpga$ runtb ./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.

In simulation the test bench can also be run from the Bluespec Workstation. By default the qsort benchmark is run in the workstation. To change which program to run on the SMIPS processor in the workstation go to `Project->Options`, choose the `Sce-Mi` tab and change the command line arguments to `tb` in the `simulate command` field. When simulating from the workstation, output from the `bsim_dut` is redirected to `bsum_dut.out` and `bsum_dut.err`.

For your convenience, we have provided scripts `run_assembly` and `run_benchmarks` in the `sim/` and `fpga/` directories which run all of the *compiled* assembly tests and benchmarks respectively. The scripts in the `fpga/` directory require the FPGA has already been programmed.

3 The Memory Interface and Block RAMs

The processor implementation you are provided makes use of memories with combinational reads. These are modeled in `src/IMemory.bsv` and `src/DMemory.bsv` using RegFiles. Each of these memories is too large to fit in the FPGA as they are implemented. Fortunately the FPGA has some built in RAM that can be used, but the processor needs to be changed in order to use it because the reads are sequential. That means read data is available the cycle after the read request. This is just like the delayed memory used for the 4 cycle processor in the first SMIPS lab.

3.1 Block RAMs

The registers you instantiate in your hardware designs are mapped to slice registers on the FPGA. Slice registers are not well suited for memories because they are not very dense and require large muxing logic to select the data.

Block RAMs are on-chip memory resources available to use on the FPGA. They are much denser than slice registers and have built-in logic to perform the address decoding. To read from a Block RAM, you give it the address to read from, and the data will be available *on the next cycle*. The XUPv5 FPGAs we are using have about 600K bytes worth of Block RAM storage.

Another form of memory storage available on the XUPv5 is DRAM. In contrast to Block RAMs, DRAM is off-chip memory. DRAM has a capacity of 1G bytes worth of storage, but may take 10s of cycles to access. In practice, Block RAMs are much easier to use than DRAM.

The RegFiles used to model the memory in `IMemory.bsv` and `DMemory.bsv` use 16 bit addresses. This was chosen specifically so that the 2 memories, with 64K words, and 4 bytes per word take up only $2 * 64K * 4 = 512K$ bytes and can be implemented on the FPGA using Block RAMs. Fortunately this amount of memory is adequate for all the assembly tests and benchmark programs.

A module has already been written for you that implements block RAMs that can be used for instruction and data memory. The file `FPGAMemory.bsv` contains the interface and implementation of mkFPGAMemory. The interface is shown below:

```
interface FPGAMemory;
    method Action req(MemReq r);
    method ActionValue#(MemResp) resp;
    interface MemInitIfc init;
endinterface
```

The interface is identical to the DelayedMemory from the first SMIPS lab except this interface contains a MemInitIfc sub interface. MemInitIfc interfaces are used to fill up the memories through the scemi interface. Like the delayed memory, this memory is designed to only give responses on reads, not writes.

4 Lab Exercises

When you are done with the exercises in this section, you will have a six stage pipelined SMIPS processor with a BTB branch predictor that can be synthesized on an FPGA. These exercises will guide you by introducing incremental changes to the processor to get to the final goal.

Exercise 1 (30 Points): Split the two stage pipeline in `Proc.bsv` to a six stage pipeline covering instruction fetch, decode, register fetch, execute, memory, and write back. This will reduce the critical path so your processor can function with the default clock frequency provided by the scemi interface. This processor will not fit on an fpga yet since it does not use block RAM yet.

To test this processor run the following:

```
$ cd scemi/sim
$ build -v
$ ./run_assembly
    ... check that all tests print PASSED ...
$ ./run_benchmarks
    ... check that all tests printed the number of cycles
         and instructions and printed PASSED ...
```

This tells you your processor is functional, but it does not tell you if the rules are firing concurrently as you want them to. To check what your processor is doing each cycle you can add `$display` statements to each your rules. You can see the output from these statements by building your processor and running `run_assembly_loud` and `run_benchmarks_loud`. If you see your rules firing in parallel and no unexpected bubbles, then your processor works!

When you are done with this part, turn it in and create a checkpoint for yourself by running the following git command in the `lab6` folder:

```
$ git commit -am "Finished Exercise 1"
$ git push
```

Exercise 2 (10 Points): Replace the memories with block RAM memories found in `FPGAMemory.bsv`. Use the module mkFPGAMemory to create two block RAMs, an instruction one and a data one. Refer to section 3 for more information about this new memory module. You can test the processor by running the same commands as you ran for the previous exercise.

When you are done with this part, turn it in and create a checkpoint for yourself by running the following git command in the `lab6` folder:

```
$ git commit -am "Finished Exercise 2"
$ git push
```

Exercise 3 (10 Points): Now it is time to replace the pc + 4 branch predictor with one shown in class: the branch target buffer. The file `AddrPred.bsv` has an implementation for mkBtb that implements the BTB for you. Replace the pc + 4 branch predictor with the BRB and connect all the needed predictor update and mispredict signals for the predictor. You can test the processor by running the same commands as you ran for the previous exercises.

When you are done with this part, turn it in and create a checkpoint for yourself by running the following git command in the `lab6` folder:

```
$ git commit -am "Finished Exercise 3"
$ git push
```

5 FPGA Synthesis

At this point, your processor will be *almost* ready for FPGA synthesis. Because of the way the fifos and the scoreboards are implemented, their sizes need to be powers of two. If they are not powers of two, xilinx's tools can't create hardware for them. At this point, change the fifo and scoreboard sizes so they are powers of two if they aren't already.

Once you are done sizing your fifos and scoreboard, you can run FPGA synthesis by running the following command:

```
$ cd scemi/fpga
$ xilinx build -v xupv5_dut
    ... wait for a long time (~1 hour) ...
```

This command takes a lot of computing resources, so try to run it on a machine with little load. You can check the load of a machine with the `top` command.

5.1 FPGA synthesis reports

The Xilinx tools output reports to the `xilinx/` directory from which we can learn the area and critical path of our design. The file `xilinx/mkBridge.srp` is the synthesis report. It contains summary information about how many slices our design takes up and the critical paths in our design. The Device utilization summary near the end of the file shows the resources our design uses. The Timing summary lists the critical path and its length for each clock in the design. The clock for our processor is called `scemi_clk_port_clkgen/current_clk1`.

The file `xilinx/mkBridge.par` is the place-and-route report. If the design fails to meet timing constraints, it is reported in this file. You should always verify near the end of `xilinx/mkBridge.par` that it says "All constraints were met". It is not always obvious if not all timing constraints are met, so look carefully. If place-and-route fails to meet all the timing constraints, you can look at the synthesis report `xilinx/mkBridge.srp` as described above to understand what the failing critical path is. You can also get more detailed information about the failing timing constraint by reading `xilinx/mkBridge.twr`, which gives a detailed description of what went wrong for any timing constraints not met.

Finally, we can get additional information about which parts of our design take which resources from the file `xilinx/mkBridge_map.mrp`, which shows a detailed report of resource utilization by hierarchy in the section Utilization by Hierarchy. This shows the resources used for each synthesized module in the design, hierarchically.

Exercise 4 (10 Points): Synthesize your processor for an FPGA and look at the output logs in the `xilinx/` folder. Answer the following questions in `lab6/discussion.txt`:

1. Did your design meet all the constraints when synthesizing? Copy the relevant lines from the appropriate report.
2. What clock frequency can your design run at? Copy the relevant lines from the appropriate report.
3. Using the clock frequency from question 2, calculate the IPC and IPS (instructions per second) for each benchmark using the simulation results from `run_benchmarks`.
4. Where is the critical path in your design? Copy the relevant lines from the appropriate report.
5. What (if anything) could you do to further reduce your critical path?

5.2 Running on an FPGA

If the design compiled without errors, then your design met the timing requirements and should work if programmed to an FPGA. If you want to try running it, log into a vlsifarm computer with an available FPGA board (vlsifarm-03.mit.edu, vlsifarm-04.mit.edu, and vlsifarm-06.mit.edu) and run the following: **Update 10/31/2013:** The FPGA boards are currently hung up. Once I get them up and running, I'll let you know. Also, you need to update the script `run_assembly` to include `runtb` before `./tb` just like `run_benchmarks`.

```
$ cd scemi/fpga
$ build -v tb
$ programfpga
$ ./run_assembly
$ ./run_benchmarks
```

You should see the same output as simulation if the FPGA is working.

6 Submitting the lab

When you are done with this lab, commit all your changes and push your changes back to be graded by running the following commands:

```
$ git commit -am "Done with lab 6"
$ git push
```