Lab 5: Pipelining an SMIPSv2 Processor: Part I

6.375 Laboratory 5
Assigned: March 8, 2013
Due: March 15, 2013

1 Introduction

In this laboratory assignment and the next you will be provided with an unpipelined two stage SMIPSv2 processor in Bluespec which you must pipeline to achieve good performance. Obtaining good performance requires a solid understanding of how Bluespec schedules rules, something you should be an expert at by the time you complete these labs. For this first lab your task is to change the magic memory provided with a more realistic multicycle memory and decouple the fetch stage from the rest of the pipeline, handling control hazards appropriately. The next lab will complete the pipelining of the processor to achieving adequate performance.

This lab handout describes the processor infrastructure, including how to build and run the processor to determine if it functions correctly and how well it performs, advice on how to debug the processor, the initial unpipelined processor design, and steps you should take to pipeline the unpipelined processor.

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 SMIPSv2 processor in simulation and on the FPGA. This section describes that infrastructure.

Appendix A includes a reference on the SMIPSv2 instruction set your processor supports.

2.1 Getting Started

Update your local repository with the code from the lab 5 harness. Add the 6.375 course locker, source the setup script, navigate to the directory which contains the `audio/` folder from previous labs, download `lab5-harness.tar.gz` from the course website, and run

```
$ tar xf lab5-harness.tar.gz
```

This will create a directory called `smips` with the code for this lab and the next. To save your changes to your local repository, run

```
$ git add smips
$ git commit -m "lab5 initial checkin"
```

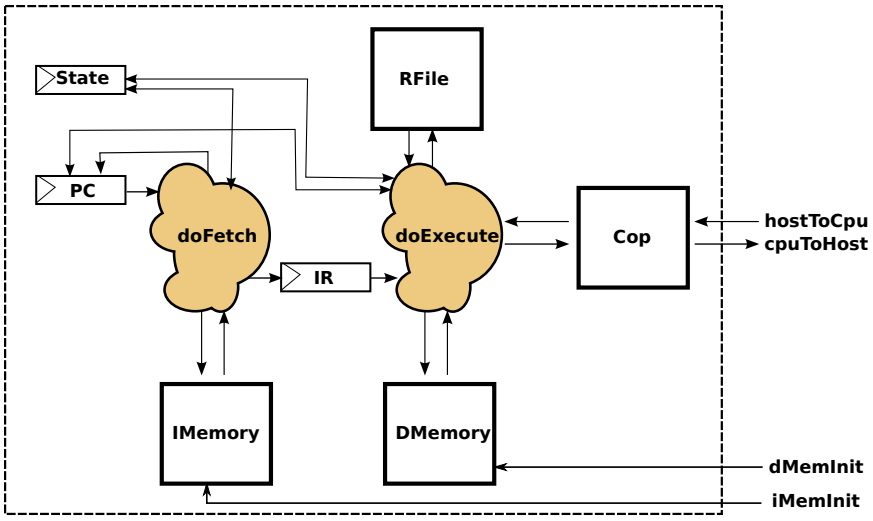

Figure 1: Original Microarchitecture

2.2 The Source Code

Figure 1 shows the processor core described by the source code in the `src/` directory. The processor uses magic memory for instruction and data memory. The processor communicates to the host through a coprocessor which tracks cycle and instruction counts, among other things.

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

`AddrPred.bsv` Implementation of a simple next-pc address predictor.

`Cop.bsv` Implementation of the coprocessor module.

`DMemory.bsv` Implementation of the data memory. You will be asked to update this file in this lab.

`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. You will be asked to update this file in this lab.

`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` The actual processor. The processor provided is a two stage unpipelined processor shown in figure 1. This is where much of your modifications should be made for this lab.

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

`RFile.bsv` Implementation of the register file.

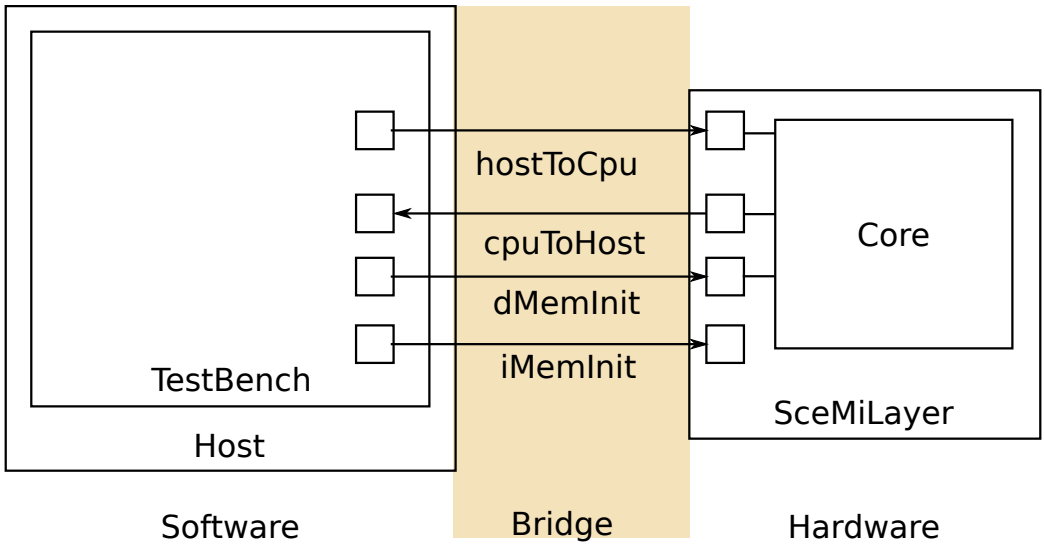

Figure 2: SceMi Setup

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

`Types.bsv` Common types.

2.3 The SceMi Setup

Figure 2 shows the SceMi setup for the lab. The SceMiLayer instantiates the core shown in figure 1 and SceMi ports for the core'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. 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 SMIPSv2 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 by the end of these two labs.

2.6 Using the Test Bench

Our SceMi test bench is software run on the host processor which interacts with the SMIPSv2 processor over the SceMi link, as shown in figure 2. The test bench loads a program for the SMIPSv2

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 SMIPSv2 processor.

To run the test bench, first build the project as described in section 2.4 and compile the SMIPSv2 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 SMIPSv2 program could either fail, or pass, as determined by a value in the toHost register in the SMIPSv2 Processor, which is set by the running SMIPSv2 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 SMIPSv2 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. In this part of the lab you are asked to change the memory interface to a request/response interface implemented using Block RAMs on the FPGA.

There are a number of reasons for this exercise. It will hopefully help familiarize you with the processor code, it introduces you to Block RAMs, which will likely be useful in your final projects, and it is required to make the design run on the FPGA.

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.

The Bluespec library includes support for using Block RAMs in the BRAM package. This support is described in detail in section C.1.5 of the Bluespec reference guide. The Bluespec BRAM modules have a request/response interface. For example, to instantiate a BRAM in your Bluespec design which stores elements of type `DataType` and uses an address of type `AddrType`, import the `BRAM` package and write something like:

```
module mkFoo();
    BRAM_Configure cfg = defaultValue;
    BRAM1Port#(AddrType, DataType) bram <- mkBRAM1Server(cfg);
    ...
```

The width of the address and data types determine the size of Block RAM instantiated.

The request type for a BRAM is described in the reference guide. It includes a boolean flag indicating whether the request is a read request or a write request, the address to read from or write to, the data to write, and whether to give a response on write or not. For example, to write the value 42 to address 7, you might issue a request as:

```
    rule foo ();
        bram.portA.request.put(BRAMRequest {
            write: True,
            responseOnWrite: False,
            address: 7,
            datain: 42});
        ...
```

The response type of the BRAM interface is the data read for read requests.

3.2 Updating the Memory

To use Block RAMs for the instruction and data memories requires the memory interface change to a request/response style. For example, a new interface for the data memory which would be appropriate is:

```
interface DMemory;
```

```
    interface Put#(MemReq) req;
    interface Get#(MemResp) resp;
    interface MemInitIfc init;
endinterface
```

Problem 1: Change the instruction memory (`IMemory.bsv`) and data memory (`DMemory.bsv`) to use Block RAMs and a request/response interface like the one shown above (The file `MemInit.bsv` contains a MemInit block for BRAMs you may find useful). You will have to update the processor implementation (`Proc.bsv`) to use the new request/response interface. This may require adding more stages to the processor. After changing the memories, verify the processor still works in simulation.

Problem 2: Run your processor design on the FPGA. You will likely need to introduce more stages in the processor to break up long critical paths and meet timing. Can you tell in the synthesis how much of the Block RAM resources on the FPGA your design is consuming?

4 Handling Control Hazards

To efficiently use hardware, all stages of the processor should execute concurrently. As a first step in this direction, you should decouple the fetch stage from the rest of the pipeline so fetch can run concurrently with the rest of the pipeline.

As discussed in the lectures, when fetch is decoupled from the rest of the pipeline, there arises the possibility of control hazards. The fetch stage must predict what the next instruction is before the execute stage has computed the actual next instruction. If the fetch stage predicts incorrectly, the incorrect instructions it fetched must not be executed.

Problem 3: Decouple your fetch stage from the rest of the pipeline. Use the epoch scheme discussed in lecture to properly handle the control hazards that arise. Use the `mkPcPlus4` address predictor provided in `AddrPred.bsv` to predict the next instruction to fetch. Pass the predicted pc to the `exec` function, and check the returned `eInst.mispredict` to identify when mis-prediction occurs. Verify your process still works correctly in simulation.

If you have properly decoupled your fetch stage from the rest of the pipeline, the `doFetch` rule should no longer be guarded with `stage == Fetch`, and the `doFetch` rule should be concurrently schedulable (not conflicting) with the rules for the remaining stages in the pipeline. You may need to introduce Ehrs in your design to achieve this.

5 Handling Data Hazards

The next lab will ask you to decouple the rest of the stages of your processor so it is fully pipelined. You will need to handle data hazards using the scoreboard approach described in lecture. Though you are not asked to handle data hazards for lab5, if you have free time left over this week, it would be wise to get an early start on lab 6 and handle data hazards in your pipeline.

6 What to Turn In

When you are done updating the memories to use block rams and have updated your processor to handle control hazards, check in your code via git. For example:

```
smips$ git add -u .
smips$ git commit -m "Lab 5 submission"
smips$ git push
```

A SMIPSv2 Instruction Set

31–26	25–21	20–16	15–11	10–6	5–0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100011	base	dest	signed offset			LW rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-Type Computational Instructions						
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-Type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset
System Coprocessor (COP0) Instructions						
010000	00000	dest	cop0src	00000	000000	MFC0 rt, rd
010000	00100	src	cop0dest	00000	000000	MTC0 rt, rd

Figure 3: SMIPSv2 Instruction Set