Lab 5: A Pipelined SMIPSv2 Processor

6.375 Laboratory 5
Assigned: March 5, 2010
Due: March 19, 2010

1 Introduction

In this laboratory assignment you will implement a fully pipelined SMIPSv2 processor in Bluespec SystemVerilog. You are responsible for a design which works in both simulation and on the FPGA as well as written answers to critical thinking questions.

The main goal of this lab is to understand how your designs are scheduled, and we hope that after completing it, you will have a much better understanding of how rule scheduling works in Bluespec. In addition to answering the critical thinking questions, you will need to make sure your design meets certain performance criteria (instructions-per-cycle on given benchmarks). Efficient execution of your design will be unattainable without the correct schedule, so you will need to think deeply about the rule interactions; you may want to review the lectures on this topic before proceeding.

While we've introduced wire constructs explicitly in the lectures, it is generally not a good idea to use them directly in your designs. Instead use the tuned performance library modules (*i.e.,*LFIFO) some of which are part of the standard Bluespec library, and some of which we have created specifically for use in this lab. This will allow you to reason about the correctness of your design using the default modules, and then to swap them out for ones which change the scheduling restrictions (often at the expense of adding combinational paths). More details on these modules will be given later in the lab handout.

As the starting point, you are given a fully functional multicycle SMIPSv2 (a simplified version of the MIPS ISA) processor. This design has three separate rules with mutually exclusive guards that each implement one of the three stages of the processor pipeline. This design takes three cycles to process an instruction and only one instruction is active in the pipeline at any given time. Your task is to fully pipeline the design by adding the necessary hardware to allow multiple instructions to execute concurrently, and to add the required guards/stalling logic to make sure that instructions will execute correctly. In addition, you will need to create a simple BTB branch predictor to further improve your processor's IPC.

The full instruction set for the SMIPSv2 architecture is listed in Section 9. Though you have been provided with functions to decode the instruction words, this reference may still be of some use to you when implementing the stall logic.

2 Processor Overview

The code used in this lab is completely independent from the previous labs we have done. The new code is available in the lab5 harness.

1. Extract the code from the lab5 harness and add it to your subversion repository. Remember to source the setup script first. Navigate to the directory which contains your lab1 through lab4 folders and run

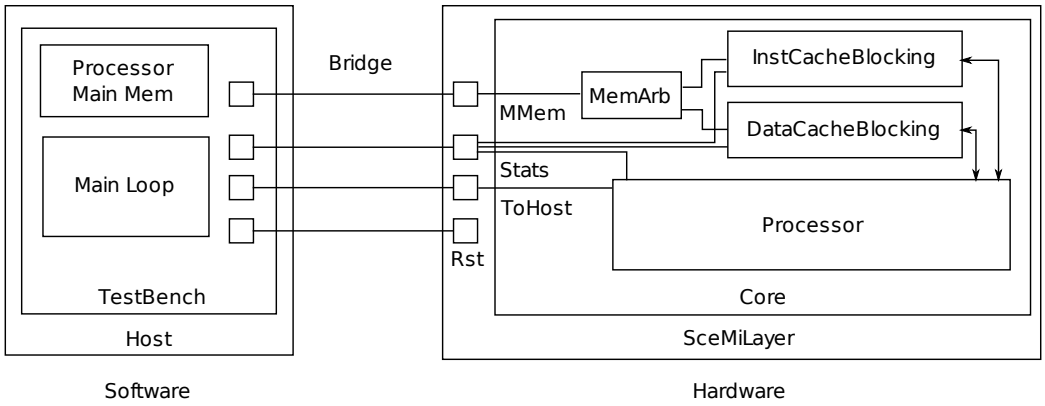


Figure 1: Lab Setup

```
% tar -xzvf /mit/6.375/lab-harnesses/lab5-harness.tar.gz
% svn add lab5
```

2. Check the code into subversion

```
lab5% svn ci lab5 -m "Lab5 Initial Checkin"
```

Figure 1 shows all the components used in this lab. The hardware consists of a core with processor, data and instruction caches. Main memory for the processor is implemented in the software test bench for convenience, with a memory arbitrator in hardware to coordinate requests from the instruction and data caches. The Sce-Mi link also provides a way for the software test bench to access statistics about the processor and caches, a way for the software to read the `cp0_tohost` register which resides in the SMIPSv2 processor, and a way to reset the core to run different programs on it.

The `src/` directory contains source for the SMIPSv2 core. Source for the software test bench is in `tb/`, and the Sce-Mi specific code is in `scemi/`. We have also provided some tools for running tests and benchmarks in simulation in the `tools/` directory. The directory `sim/` contains a Bluespec Workstation project for simulation and `fpga/` contains a Bluespec Workstation project for fpga just as in lab4.

A sketch of the original multicycle processor microarchitecture is given in Figure 2. You can see that there is a stage register which tracks the state of the processor. Originally, the state of the processor is "P" (pcgen) indicating that a request is sent to the instruction cache for the memory (instruction) at the address stored in the program counter. The pcgen rule then sets the stage to "X" (execute), which allows the execute rule to fire once the instruction has been returned. The instruction is decoded, registers are fetched, and the instruction is executed. If the instruction is an ALU instruction, the registers will be written, and the stage will be set back to "P". If it is a memory instruction, the memory request will be sent to the data cache and the stage will be set to "W" (writeback). If it is a branch instruction, the program counter will be updated and the stage will be set to "P".

3 Running the Processor

The `sim/` directory contains a Bluespec Workstation project you can use to simulate the processor. The test bench takes as input a program specified in Verilog Memory Hex format, such as the ones

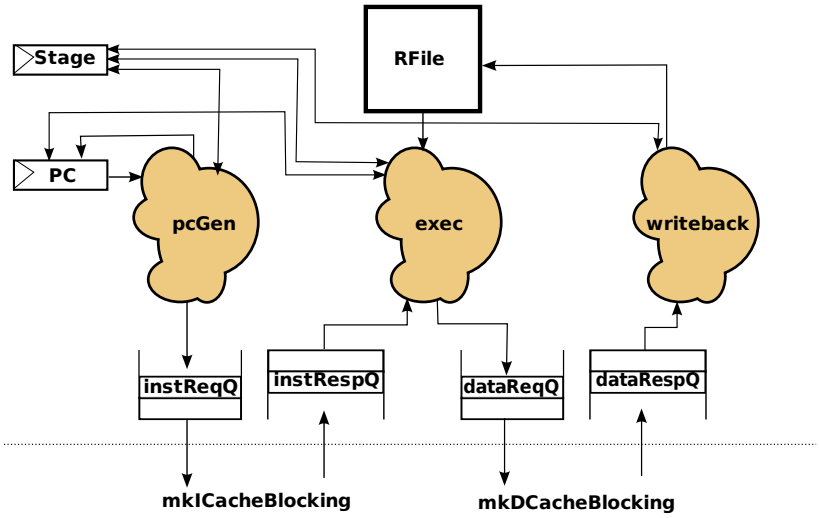


Figure 2: Original Microarchitecture

we provided in `data/`. You can specify which program to use by listing it on the command line when running the test bench. For example

```
./tb ../data/qsort.smips.vmh
```

By default the test bench looks for a file called `program.vmh` in the local directory. You can change the command line used to call the test bench from within the workstation by going to `Project->Options`, the `Sce-Mi` tab and changing the `simulate command` field.

In the `tools/` directory are scripts for running the assembly tests and the benchmarks both in simulation and on the FPGA. These scripts report whether or not the program passed as expected and the IPC for the benchmarks. The scripts for simulation should be run from within the `sim/` directory. For example, to run the assembly tests after building the sim project

```
sim% ../tools/sim-asm-tests
```

And to run the benchmarks and collect IPC information

```
sim% ../tools/sim-smips-bmarks
```

The assembly tests each take a few seconds to run. The benchmarks take more time, somewhere less than 30 seconds. If the assembly test or benchmark is still running after a couple of minutes, the test bench will report the program as timing out.

When you are ready to run your design on the FPGA, use the Bluespec Workstation project file in the `fpga/` directory. After you have compiled, synthesized, and programmed the board you can use the test bench just like you did in simulation. The test bench sends a soft reset to the FPGA every time you run it, so you should not have to reprogram the FPGA to run different programs on the SMIPSv2 processor with the test bench. The scripts `tools/fpga-asm-tests` and `tools/fpga-smips-bmarks` can be used to verify the assembly tests still pass and collect IPC of the benchmarks on the FPGA.

4 Debugging your design

The source for the assembly tests and benchmarks is in the course locker at `/mit/6.375/install/smips-tests/` and `/mit/6.375/install/smips-bmarks` respectively. It may be helpful to look at the source code when debugging problems. We don't anticipate the need to recompile the benchmarks, but if you feel compelled to and need help, feel free to ask the TA for assistance.

Information can be gleaned from your executing processor using a number of means. First of all, we can make use of the statistics recorded in the processor and caches. Statistics gathering is enabled by writing a special register `cp0_statsEn`. This is done using the `MTC0` instruction (used to write to coprocessor registers), which you can see implemented in `Processor.bsv`. Note that stats are enabled using an instruction in the program binary itself, not by a command from the test bench.

As stated earlier, the backing memory for the SMIPSv2 processor is implemented over the Sce-Mi link, which could cause pathologically poor performance on the benchmarks. Luckily our caches are large enough that we can fit the entire working set for each benchmark on the FPGA. We consequently execute the benchmark twice, first to prime the caches, and a second time to collect statistics. You will notice that the benchmarks write a `1` to the `cp0_statsEn` register only after the first run has primed the caches, thus the generated statistics will only reflect the second execution (though the trace will contain both). Note that statistics are collected only for the benchmarks, and not for the assembly tests. Feel free to add more statistics to the core and test bench if you think it will help in debugging.

We have also provided some infrastructure for producing text traces of the processor. If you examine the BSV source for the core you will see various uses of the traceTiny() and the traceFull() functions. These trace functions output a trace tag and some trace data, and use the Bluespec action `$fdisplay` to send the debugging strings to stderr which can be redirected to a file.

The command `bsv-trace.pl` turns this trace output into a clean text trace format with one cycle per line. The script takes a configuration file as input which describes how to transform the trace output. For example, the following commands can be used to run a program on the SMIPSv2 processor and produce a clean trace. Part of the trace output is shown in Figure 3.

```
sim% ./bsim_dut 2> proc.trace & sleep 1 ; ./tb ../data/smipsv1_addiu.S.vmh
sim% bsv-trace.pl /mit/6.375/install/scripts/proc-trace.cfg proc.trace
```

The first column lists the cycle, after which the pc is printed. Following that, is a character corresponding to the current `stage` or rule which is executing in the processor (`P` for `pcgen`, `X`, for `execute`, and `W` for `writeback`). Next is a column corresponding to the state of the instruction cache, followed by a column corresponding to the state of the data cache. The pen-ultimate column displays to the state of the memory arbiter while the last shows the instruction being executed. Look at the implementation of each module which outputs traces for a better idea of the meanings of each field. It is relatively simple to add new trace messages, and to extend the perl scripts to format them appropriately. You may find this useful when pipelining your designs as the version provided may not print out all the information you need.

The test bench sends a reset signal to the core each time it is run, which lasts for a few thousand cycles, so the beginning of the trace will not be very interesting.

Lastly, you will debug this in simulation, so your old friend `$display` will always be there for you.

```
processor stage [     icache           ]  [     dcache           ]  [    mem-arb   ]
 pc       [P|X|W] [req|resp|stage|hit/miss]  [req|resp|stage|hit/miss]  [req0|req1|req2]  exInst
...
...
CYC: 1293  pc=         [ | | ]  [    |    |    | ]  [    |    |    | ]  [    |    |    ]
CYC: 1294  pc=         [ | | ]  [    |    |    | ]  [    |    |    | ]  [    |    |100 ]
CYC: 1295  pc=         [ | | ]  [    |    |R   | ]  [    |    |    | ]  [    |    |    ]
CYC: 1296  pc=         [ | | ]  [    |100 |    |h]  [    |    |    | ]  [    |    |    ]
CYC: 1297  pc=         [ |X| ]  [    |    |    | ]  [    |    |    | ]  [    |    |    ]  bne r2, r3, 0x0002
CYC: 1298  pc=00001018 [P| | ]  [100 |    |    | ]  [    |    |    | ]  [    |    |    ]
CYC: 1299  pc=         [ | | ]  [    |    |    |m]  [    |    |    | ]  [    |    |    ]
```

Figure 3: formatted trace

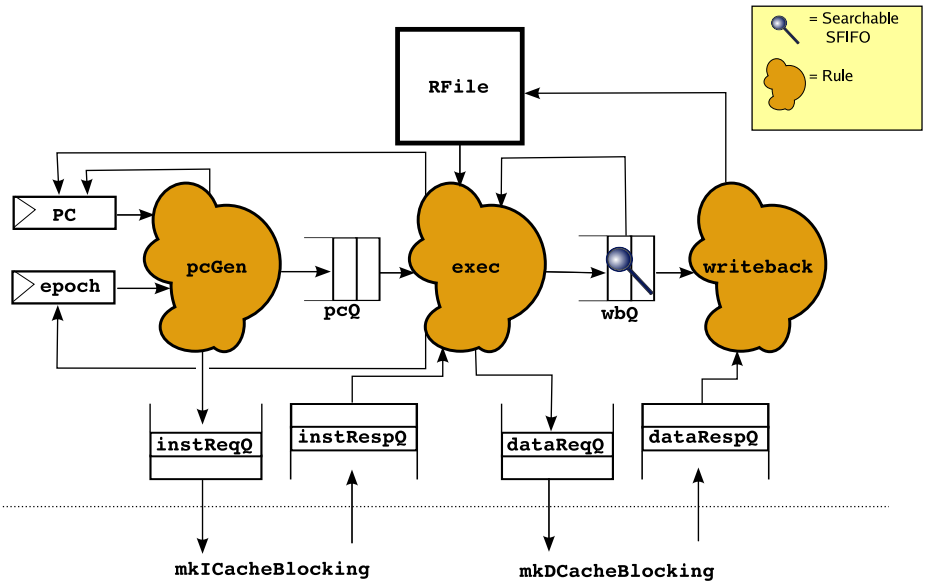


Figure 4: Pipeline Rules for SMIPSv2 Processor

5 Pipelining the Processor

The microprocessor you have been given has both instruction and data caches. These caches use a request-response protocol for both reading and writing, meaning that even on a cache hit, data will not be available until the following cycle. The presence of these two-stage sub pipelines require that your final pipelined microarchitecture consist of at least 3-stages (they may overlap on one stage). The recommended partitioning for your fully pipelined design is shown in Figure 4. This has the benefit of closely matching the 3 cycle rules in the original design. You are welcome to try other pipelining strategies, but you may not change the interface to the memory substructure and must still achieve good pipeline parallelism. Before continuing with the lab, it may be worthwhile to review the processor lectures.

The pipelined microarchitecture presented in Figure 4 is quite straightforward. The pcGen stage performs three functions. First, it sends a request for the current instruction to the instruction cache `instClient`. Secondly, it sends the address information (the pc and epoch) using the `pcQ` to the next stage, and lastly, it predicts the next PC value by setting it to pc + 4 (later we will look at more effective means of prediction).

The execute stage takes the response from the icache, as well as the data from the `pcQ`, looks up the

current value for the registers, and performs the computation for arithmetic and branch instructions. The behavior here diverges slightly, depending on whether the instruction is `ALU`, `BRANCH`, or `MEM`. For `MEM` (memory) instructions, a request is sent to the data cache (`dataClient`), and in the case of a load, the destination register is enqued into the `wbQ` (writeback queue). In the case of an `ALU` instruction, the result is computed and enqued into the `wbQ` along with the destination register. The only effect of `BRANCH` instructions is to update the program counter, so the execute stage need not enqueue any data in the `wbQ` when executing one of these. The last function of the execute stage is to discard mis-predicted instructions.

The writeback stage must complete the execution of `ALU` and `MEM` instructions. `ALU` instructions only require their results to be writtent to a specified register. If the `wbQ` contains a `MEM-load` instruction, it must dequeue the memory response, and write the value to the specified register. If the `wbQ` contains a `MEM-store` instruction, this rule need only dequeue the null `load-response` token from the memory response queue.

For your pipelined design you will probably want to closely follow the microarchitecture in Figure 4, which uses three main rules rules, along with auxiliary rules for dealing with the epoch, which we will explain in in Section 5.2. The interaction of these rules will be key, so carefully craft their predicates and actions. When is it safe to execute the next instruction? What actions should the writeback rule perform? Careful thought about the *intent* of each rule will aid you during implementation.

To achieve full pipelining you will be required to start instructions speculatively. This has several important ramifications:

1. You must correctly detect dependencies and stall the pipeline when necessary. Sometimes, stalls can be avoided by adding bypassing logic (a bypass register file, for example), but when the stall involves memory, there isn't a whole lot that can be done.
2. When a branch is mispredicted you should not only kill instructions which were incorrectly fetched but also ignore the responses from memory requests these instructions have made, but not yet handled. This is accomplished through the use of the `epoch` register and will be discussed in more detail. Remember that there are some instrucions which should never be speculatively executed. Though the issue probably won't arise in our little 3-stage pipeline, think about what might happen if speculatively executed a store to memory.
3. Intermediate state FIFOs should be constructed to have the correct scheduling properties and necessary buffering. This is a primarily a performance consideration: if you do all this work without achieving parallelism, your efforts are largely in vain.
4. You must keep the reads and writes coherent. In many pipelines this is solved by only updating each state element(*e.g.,*the register file) in one stage. If you choose to write in two different stages, you must be careful that the logical order is obeyed. The recommended microarchitecture shown in Figure 4 isolates all register updates to the writeback rule. Deviate from this policy at your own risk!

As part of our policy of "correct then fast", your first step will be to modify your design so that it can correctly handle multiple active instructions.

Although the three rules you have been given are very close to the three stages, you will want to reason think about some simple things before you continue. After you have thought about these cases, you should have a good idea what changes to the state and rules are needed at a high-level. The subsequent subsections will walk you through some of the more tricky details of the stall generation.

1. You will need to use pipeline FIFOs and searchable FIFOs (SFIFOs) to hold intermediate data between stages. Searchable FIFOs are used in order to detect what instructions (more

specifically: which destination registers) are in flight so that the pipeline can be stalled when a hazard is detected. What state does each of the pipeline stages need for each instruction? Where is that generated? What data must you pass between each stages?

2. In the processor in the class lectures, when we discover a mispredicted branch we could immediately kill all of the false-path instruction. However since we've split the instruction read to be multi-cycle, we may have pending responses waiting in the queue. As a result, after you handle a misprediction you will still have to wait for the false-path instructions' memory responses to return. To solve this you will need to mark each instruction using an epoch mechanism. We will discuss more of this later in this document.

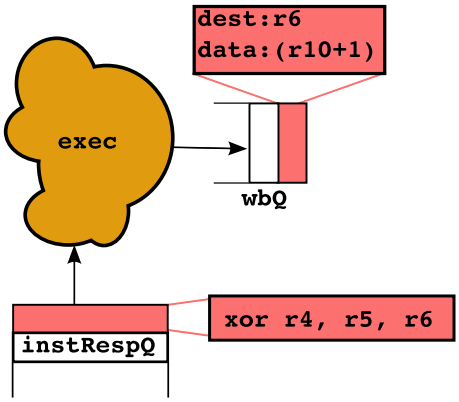
3. In which stage (or stages) do you need to read the register file? When are the instructions read? When must we stall reading to prevent us from reading the wrong value in the register file?

5.1 Building the correct stalling logic

One significant part of this assignment is generating the stall signal for the execute stage. When is it safe to read the register file and execute the next instruction? Essentially, this means detecting Read-After-Write (RAW) hazards. So if our design is executing the following program:

```
A) addiu r6, r10, 1
B) xor   r4, r5, r6
```

The processor will eventually reach the following state:



Instruction B cannot be executed because it reads `r6`, and instruction A is responsible (and has not yet) written the new value back to the register file. This means we need to stall until the instruction A writes the register file. We will implement this using `SFIFO`, a searchable FIFO. with the following interface:

```
interface SFIFO#(type any_T, type search_T);
    //Standard FIFO methods
    method Action enq(any_T data);
    method Action deq();
    method any_T first();
    method Action clear();
    //New SFIFO methods
    method Bool find(search_T searchVal);
    method Bool find2(search_T searchVal);
endinterface
```

Note that SFIFO is just like a normal FIFO with two extra methods: `find()` and `find2()`. These methods take a datatype parameter (the same datatype the FIFO is storing), and return a boolean. Specifically, they return True if the given parameter is present in the FIFO, and False otherwise. `find()` and `find2()` have no implicit condition — they always ready — they will simply return False if the FIFO is empty.

Why does SFIFO include two methods `find()` and `find2()`? This is so you can can search it twice for instructions that have two operands. For instance, in the above example the execute rule can check if `r5` is in the `writebackQ` using `find()`, and `r6` using `find2()`. However if the instruction had been `xori` instead of `xor`, it just would have used `find()` because `xori` just has one register argument.

What type should you store in the writeback queue and how should you search it? First let us consider what the result of the execute stage should be. If the instruction was an ALU operation, then the result should be the destination register and the data to put into it. If it was a Load, then we need the destination register to put the response from memory into. If it was a store then we need to record this fact so we can receive the acknowledgment from memory. Finally let's treat the From Host register specially.

```
typedef union tagged
{
    struct {Bit#(32) data, Rindx dest} WB_ALU;
    Bit#(32) WB_Host;
    Rindx WB_Load;
    void WB_Store;
}
  WBResult deriving (Eq, Bits);
```

Now you must define what it means to search the FIFO by writing a function. This function should take an Rindx (the thing we're looking for) and a WBResult (the elements through which we are searching). The function should return true if the search value is "found" in the given FIFO element.

```
function Bool findf(Rindx searchval, WBResult val);
  //You write this
endfunction
```

When you instantiate the SFIFO, you should pass in the appropriate types and find function. What does it mean to pass in a function to a hardware module in Bluespec? Essentially it means that when the compiler instantiates the module it will do so with the combinational logic you provide. Think of the SFIFO as a black box — a black box with a hole in it. The function you provide fills that hole.

Thus the types of the writeback queue is as follows:

```
//Searchable for stall signal
SFIFO#(WBResult, Rindx)  wbQ  <- mkSFIFO(findf);
```

All in all, the best way to encapsulate the stall signal is probably by writing a function called `stallfunc()`. `stallfunc()` takes an instruction and an SFIFO and returns False if can be executed, and True if it must stall.

So your design will probably look something like this:

```
 function Bool stallfunc(Instr inst, SFIFO#(WBResult, Rindx) f);
   ... //You write this ...
 endfunction
module mkProc (Proc);
 ... //State elements ...
  rule execute (instRespQ.first() matches tagged LoadResp .ld
                &&& unpack(ld.data) matches .inst
                &&& !stallfunc(inst, wbQ));
     ...
     case (inst) matches //Execute the instruction
         tagged LW .it:
     ...
   endrule
```

5.2 Dealing with Branches

You may have noticed that branches are resolved in the Execute stage. Why is this a problem? Because if the branch has been taken (or, with a branch predictor, if the branch has been mispredicted) then the `pcGen` stage has made a memory request for an instruction which we must ignore. (In a design with a non-blocking cache it may even have made more than one.) There are many ways to handle this, but the simplest way is to use an *epoch*.

An epoch is a conceptual grouping of all instructions in-between branch mispredictions. We can track this by having the *pcGen* rule send the epoch as the tag for all load requests:

```
rule pcGen ...
    ...
    instReqQ.enq( Load{ addr:pc, tag:epoch } );
    ...
endrule
```

Note that this is okay because our memory system is in-order, so the tag is essentially unused. If the memory system was allowed to respond out-of-order then we would have to actually create an appropriate tag to differentiate responses. In this case we could devote some bits of the tag field to the epoch, and some to the tag itself. For instance, the eight-bit field could be used to store a three-bit epoch, and a five-bit tag — but we do not need to worry about this for this lab.

When a mispredict occurs we clear all queues which are holding instructions issued after the branch instruction, and increment the epoch. Then we have a separate rule that discards responses from the wrong epoch.

```
rule discard ( instRespQ.first() matches tagged LoadResp .ld
               &&& ld.tag != epoch);
    traceTiny("mkProc" "stage","D");
    instRespQ.deq();
endrule
```

Now when we execute we must check that we can execute the instruction, and that it is from the correct epoch:

```
rule execute (instRespQ.first() matches tagged LoadResp .ld
              &&& ld.tag == epoch
              &&& unpack(ld.data) matches .inst
              &&& !stall(inst, wbQ));
    ...
    Bool branchTaken = False;
    Addr newPC;
    ...
    if (branchTaken)
    begin
      //Clear appropriate FIFOs here
      epoch <= epoch + 1;
      pc <= newPC;
      ...
```

6 Achieving Pipeline Performance

Now that you have a working design, it is time to specialize the state elements (FIFOs and register files) to have the desired scheduling. Below we discuss some of the useful approaches to help you reason about scheduling and the resulting performance.

6.1 Schedule Analysis in the Workstation

As you begin to refine your design, you may wish to take advantage of the schedule analysis facilities provided by the Bluespec Workstation. Sometimes rule conflicts can point to a bug in the design. For example, you could have a conflict between execute and writeback rules, from neglecting to change one leg of the execute case statement, so it was still updating the register file directly. Viewing the schedule and seeing the conflict would make it clear that something must be wrong.

When you have compiled your design you can go to `Window->Schedule Analysis`, which opens up the schedule analysis window. From that window go to `Module->Load` and select `mkProc` as the module to load.

The rule order tab lists all the rules in the `mkProc` module. If you select a rule you can see the rule's predicate, and any blocking rules. Pay close attention to the predicate as you may have invoked a method with an implicit condition which you didn't count on. The predicate listed here includes all lifted implicit conditions.

The rules are listed in the rule order tab in their logical order. This is the final global ordering of all the rules. Let's review a few scheduling terms before discussing how exactly to interpret this order. The term **urgency** refers to the relative priority given to two conflicting rules by the bluespec compiler. If two rules conflict the "more urgent" rule will fire if its guard is true, blocking the firing of the "less urgent" rule. The term **earliness** is used to describe the logical ordering assigned by the bluespec compiler to two rules which don't conflict. If rules A and B are sequential composable, (A before B), then A will appear to fire before B; a will be "earlier" than B, and appear before B in the logical ordering of rules. If A and B are conflict free, the Bluespec compiler makes an arbitrary choice in assigning relative earliness to the two rules. Relative urgency and earliness can be set by using the pragmas descending_urgency and descending_earliness, which are described in the Bluespec user guide.

This information will become even more important as you begin to change your FIFOs to improve throughput. Long FIFOs will tend to "decouple" rules so that they become more independent. Shorter FIFOs will do the opposite, as the rules will interact through the state elements directly.

FIFO Variant	Package	Size	Sched	Comment
`mkFIFO()`	FIFO	2	`deq < enq`	Default. Use this to get your design working. deq and enq may happen simultaneously when contains 1 element
`mkFIFO1()`	FIFO	1	`deq ME enq`	deq if full, enq if empty. Mutually exclusive.
`mkSizedFIFO(`*n*`)`	FIFO	*n*	`deq < enq`	deq and enq may happen simultaneously when neither full nor empty
`mkLFIFO()`	FIFO	1	`deq < enq`	deq and enq may happen simultaneously when full
`mkBFIFO()`	BFIFO	1	`enq < deq`	If enq and deq happen when empty, value is bypassed
`mkSizedBFIFO(`*n*`)`	BFIFO	*n*	`enq < deq`	A larger buffer for when no deq happens
`mkSFIFO()`	SFIFO	1	`deq < find < enq`	Uses SFIFO interface. Properties are like mkFIFO
`mkSizedSFIFO(`*n*`)`	SFIFO	*n*	`deq < find < enq`	Uses SFIFO interface. Properties are like mkSizedFIFO

Figure 5: Properties of various FIFO modules. For all FIFOs: `first < (enq, deq) < clear`.

6.2 Considering different FIFOs

In Figure 4 we represent all FIFOs in the design with the same picture. In reality, in order to achieve good throughput you will need to (a) appropriately size all FIFOs and (b) ensure that they have the correct scheduling properties to ensure maximum concurrency between rules. To this end we are providing you with libraries for the special FIFOs in addition to the standard library FIFOs.

With the exception of SFIFO and BFIFO, all the different flavors of FIFOs listed in the table in Figure 5 are part of the standard bluespec library and are documented in the language reference. SFIFO and BFIFO are implemented in the files `SFIFO.bsv` and `BFIFO.bsv` respectively. You can find these files in the `src/` directory.

The properties of various FIFOs are given in Figure 5. When choosing a FIFO remember to consider both its size, and its scheduling properties. What case do you expect to be the most common? How does the memory latency affect things? Is extra area and an extra cycle of latency worth an improvement in throughput? Be sure to run the benchmark suite and examine which rules fire to check how your change impacts throughput.

Sometimes there may be places where you wish no FIFO existed at all, *i.e.*,a wire. The problem with such a combinational structure is that you must be able to guarantee that your rules will always fire when values are on the wire — no communication should be dropped under any circumstance. Rather than making you reason in such a way for this lab, we are providing you with a safer abstraction: `mkBFIFO1()`, a Bypass FIFO. This FIFO behaves like a wire (`enq()` before `deq()`) as long as both occur. Otherwise the value is buffered in the FIFO, so the `deq()` can occur later. You should ensure that your design is functionally correct with normal FIFOs before you attempt to introduce Bypass FIFOs.

6.3 Sizing FIFOs

Although the scheduling properties of the FIFOs are important, so too is their length. For instance, consider the `pcQ` FIFO. At the beginning of time it starts out empty, then the `pcGen` rule enqueues into it, and makes a memory request. Well, even if the memory request comes back the following cycle because of a cache hit, the response still has to go into the `instRespQ`. Therefore it seems that making the `pcQ` smaller than size 2 will restrict the parallelism of your processor by limiting the number of instructions in flight.

Although it is possible to find a good configuration using an experimental approach, reasoning about the system using high-level knowledge can point you towards the optimum configuration. Always

check the effect of your changes on the generated schedule.

6.4 Adding a Bypass register file

We have one more conflict which needs to be resolved. Consider the interaction between the write-back and execute rules. They interact through two pieces of hardware: namely the register file and the writeback queue. The writeback rule dequeues from the writeback queue, and writes to the register file, while the execute rule reads from the register file, and enqueues (and searches) to the writeback queue. The compiler schedules the SFIFO methods `first` and `dequeue` before the methods `find` and `enqueue`. The register file method `read` is scheduled before `write`. Clearly we have a conflict here: if the execute and writeback rules cannot fire in the same cycle, we will never get full throughput of our pipeline.

The solution here is to devise a new register file which schedules writes before reads, and which bypasses written data if a simultaneous read address corresponds to the write address. Using a pair of RWires and a conflict-free register file (mkRegFileWCF) you should be able to devise a new register file with the behavior described above. In the introduction we discouraged you from using RWires directly in your design. Make an exception here.

You may add new Bluespec source files to the project if you wish, just remember to add them to subversion.

6.5 Adding a Branch Predictor

Now that you have a high-performance design you will now further improve it by improving the manner in which we speculatively fetch instructions. To do this we will add a simple branch predictor to our design or more accurately, we will improve our branch predictor from one that just guesses not taken $pc + 4$ to one which can remember old branch statements.

One purpose of this task is to think about designing modules from scratch and then effectively using them. This should be relatively simple, and can be completed with the following guidelines:

- First we must consider what the methods we want the branch predictor to have. The most obvious case would be that there must be a way to query the predictor to see if we have a prediction to make (other than the default $pc + 4$). Second, we need a way of teaching the predictor what the correct prediction should be. To do this we should have an update method which sends in two address, the address of the branch and the correct next pc value (for this execution of the instruction). Formalize these two methods into the BranchPredictor interface.
- Now that you've determined the external "shape" of your branch predictor it's time to determine the body of the design. While you are allowed to implement any algorithm you want. However, we suggest that you keep it simple and merely use a simple branch target buffer (BTB) to remember when we have seen a branch instruction and what it's result was. Ideally, this is just a big array which holds every possible address and the last prediction (defaulting to $pc + 4$). Of course, you cannot fit 2^{32} address so you will need to keep a cache. A small direct-mapped cache (of say 16 elements) should be pretty good. Remember that the bottom 2-bits of the address will always be zero so you should use the next few bits indexing into the cache.
- Now that you've picked out the behavior you must change the way that your pipeline works. The Fetch rule should make use of the prediction method and the execute rule should update the table when a misprediction is made. This should be quite straightforwards.

- Now as in the pipelining step you should implement your design, **ignoring** the performance of the predictor, and focusing only on it's correctness. Once you believe your design is correct, consider the scheduling of the methods in your design. What order must the methods execute in each cycle for the desired parallelism. Is it okay for predictions not to be made when you send updates to the predictor? If you decide both must happen concurrently, is it okay for the predictor not to observe the new branch update before it makes it's predicition?
- Now that you have completed the design, rerun the benchmarks and see how the design changed. How has the IPC numbers changed?

7 Critical Thinking Questions

Question 1: IPC

List the IPC of your design for each of the provided benchmarks without branch prediction.

Compare the results of your processor with branch prediction to the one without it. How much does the IPC improve on average? Can you estimate how your branch prediction rate improved with the new predictor?

Question 2: Design Choices

Discuss and motivate any design choices you made. Is there any way in which your implementation differs from the diagram in Figure 4? What FIFOs did you end up using, and why is this a good configuration? What is the relationship between your Execute and Writeback rules? Are they conflict-free, sequentially composable, or Conflicting? Why has the scheduler deduced this?

Question 3: Synthesizing Bluespec by Hand

Ben Bitdiddle is writing a 32-bit barrel shifter in Bluespec. The module can take any 32-bit number and shift it right or left by any amount. To minimize area Ben decides to implement the design using a circular shifter and a counter.

Unfortunately, Ben accidentally forgets to write predicates for his rules:

```
typedef enum { Left, Right} Direction;
module mkBarrelShifter (Shifter);
  Reg#(Bit#(32)) r <- mkReg(0);
  Reg#(Bit#(5)) cnt <- mkReg(0);
  Reg#(Direction) dir <- mkRegU();
  rule shiftLeft (True);
    r <= r << 1;
    cnt <= cnt - 1;
  endrule
  rule shiftRight (True);
    r <= r >> 1;
    cnt <= cnt - 1;
  endrule
  method Action shift(Direction d, Bit#(32) data, Bit#(5) amt) if (cnt == 0);
    dir <= d;
    r <= data;
    cnt <= amt;
  endmethod
  method Bit#(32) result() if (cnt == 0);
    return r;
  endmethod
endmodule
```

What schedule will the compiler deduce for Ben's design? (If you must make a choice at some point make it arbitrarily.) What hardware will the compiler generate? Diagram the resulting datapath, clearly labeling which part corresponds to the scheduling logic.

What predicates should Ben have provided? How do they change the schedule? Redraw the correct datapath which will result, again highlighting the scheduling logic.

Note that we understand that you could do this problem by typing the code into the compiler and seeing what happens. You could even base your diagram on the resulting Verilog!

Question 4: Area/Performance Tradeoff

List the increase in FPGA resource usage between the original microarchitecture and your pipelined refinement. Do you think the improved performance (IPC) is worth the increased area? While it is probable that you are able to clock both designs at 50 MHz, it is easy to imagine that your refinement will in reality be able to run at a slower frequency than the original due to the combinational paths introduced through the BFIFOs and SFIFOs. Use the average IPC of both microarchitectures to compute the slowest frequency at which your pipelined refinement must run in order to have better absolute performance (instructions per second), assuming the original can be clocked at 100 MHz.

8 What to Turn In

When you have completed the lab you should check in a final version via subversion. This should include your high performance pipelined processor with bypassed register file and branch prediction functional in simulation and on the FPGA. Also include a file `answers` in the top level lab directory with your answers to the discussion questions. Remember to add to subversion any new source files you may have introduced. For example, if you didn't add any new source files, you could run

```
lab5% svn add answers.pdf
```

```
lab5% svn ci -m "Lab 5 final submission"
```

9 Appendix A

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 6: SMIPSv2 Instruction Set