Bluespec for a Pipelined SMIPSv2 Processor

6.375 Laboratory 2
February 23, 2009

The second laboratory assignment is to implement a pipelined SMIPSv2 in Bluespec SystemVerilog. As with Lab One, your deliverables for this lab are (a) your working Bluespec code checked into CVS and (b) written answers to the critical thinking questions. The lab is due at the start of class on Friday, February 27^{th}. Again you may submit your question answers on paper in class, or electronically in CVS (plain text or PDF only). As always, you may discuss your design with others in the class, but you must turn in your own work.

The goal of this lab is to familiarize yourself with the Bluespec language and compiler, with particular emphasis on scheduling Guarded Atomic Actions. For this lab we have decided to impose the following restriction: do not use Wires, RWires, PulseWires, or any kind of primitive Wire to complete these assignments. We feel that it is important for you to gain insight into what the Bluespec scheduler is doing and why. These combinational primitives can muddy the picture and make it difficult to reason about your design. After completing this lab you will have a much better understanding of when Wires are and are not appropriate, and will be better equipped to use these constructs in your project.

The SMIPSv2 processor you design in this lab will implement the same ISA as the previous lab, but will be a different microarchitecure. The major difference between this design and the previous lab is the presence of Instruction and Data caches. Even on a cache hit, data will not be returned until the following cycle. This means that the processor must be split into a 4-stage pipeline, as shown in Figure 1. This has several ramifications, including: (a) you must correctly detect dependencies and stall the pipeline when they occur; (b) when a branch is mis-predicted you should not only kill the instructions which were fetched, but also correctly ignore all responses from memory requests these instructions have made; (c) FIFOs must be sized appropriately and have good scheduling properties for your design to achieve high throughput.

Before proceeding you should do *Tutorial 8: GAA-to-RTL Synthesis using the Bluespec Compiler*. This will familiarize you with the Bluespec compiler and simulating a simple SMIPSv2 microarchitecture generated from BSV source code. Additonally you should understand all the concepts presented in lecture *Bluespec - Modelling Processors*. The processor in that example, although much simpler than the one presented here, will serve as a good starting point for your design. As with Lab One you should start with the harness at `/mit/6.375/lab-harnesses/lab2-harness.tgz`.

Processor Overview

The design presented in Figure 1 is mostly straightforward: the pcGen stage generates the next instruction address using the PC register, and sends it to the `instReqQ` FIFO. It also writes the current PC + 4 to the `pcQ` for use in branch instructions. The response from the ICache is placed into the `instRespQ` FIFO. Then the Execute stage takes the response out of the FIFO and performs the operation. The result of the execution is placed into the writeback queue `wbQ` (the details of which we will explain in Section). If the instruction was a load or a store the memory request is

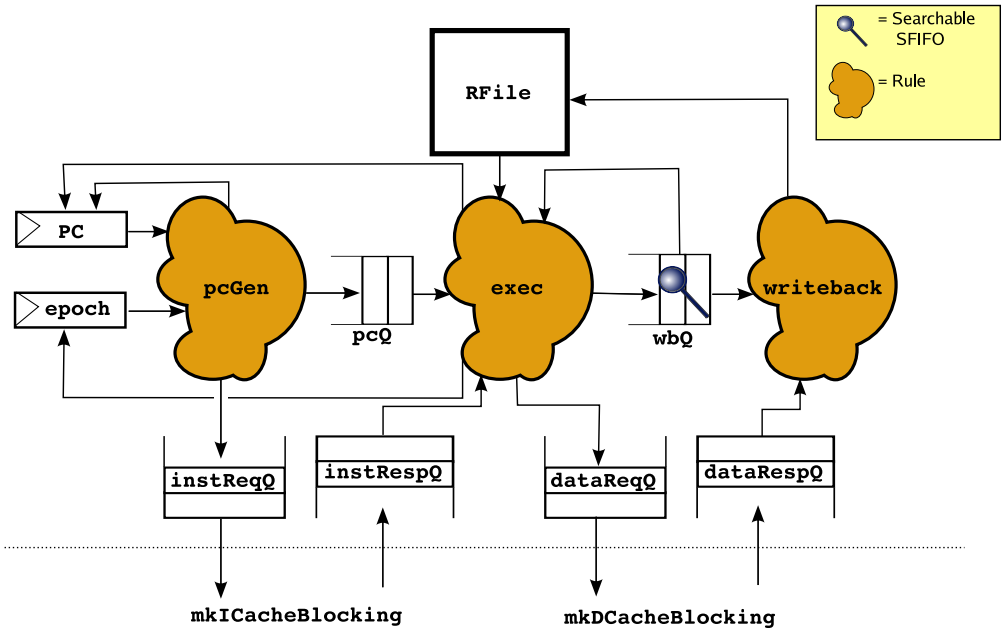


Figure 1: Pipeline Rules for SMIPSv2 Processor

placed into the `dataReqQ` queue. The Writeback stage is responsible for taking the final result from the `wbQ` and the `dataRespQ` and actually updating the register file.

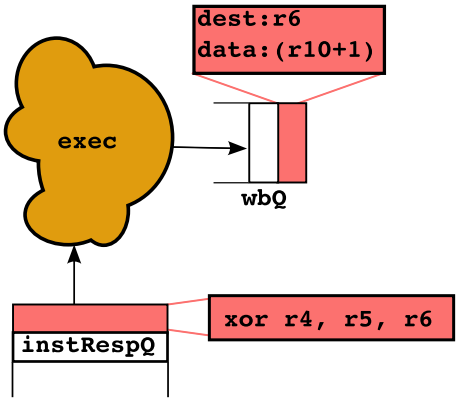
For your design you will probably want one rule for each stage, so a `pcGen`, `execute`, and `writeback` rule. (Plus a rule for dealing with the epoch as we will explain in Section .) The interaction of these rules will be key, so carefully craft their predicates and actions. When is it safe to execute the next instruction? What actions should the writeback rule perform? Careful thought about the *intent* of each rule will aid you during implementation.

Stalling the Pipeline

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`, which is being written by instruction A. This means we need to stall until the writeback of A is complete. We will implement this using `SFIFO`, a searchable FIFO. `SFIFO` is just like a normal Bluespec FIFO except it has the following interface:

```
interface SFIFO#(type any_T, 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 has 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 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 condider what the result of the execute stage should be. Well, if the instruction was an ALU op, 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 acknowledgement 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 thing we're searching). The function should return true if the search value is "found" in the FIFO.

```
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 (IProc);
... //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
```

NOTE: This rule predicate is not quite complete, as we will learn in the next section.

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("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;
     ...
```

Working with BSVCLib

In Figure 1 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 a library of predefined FIFOs with various properties. In BSVCLib, located at `/mit/6.375/install/bsvclib`, we've provided a library of useful Bluespec modules, including SFIFO and a Bypass FIFO that you should use in your design. To use SFIFO import the `SFIFO` package. To use the Bypass FIFO, import `BFIFO`.

The properties of various FIFOs are given in Figure 2. When choosing a FIFO remember to consider both its size, and its schedule. 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. you want 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 provide 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

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

later. You should ensure that your design is functionally correct with normal FIFOs before you attempt to introduce Bypass FIFOs.

You may notice that in the lab handout, we use constructs such as `mkCBusWideRegW` to wrap certain registers, like the to/from host interface. This wrapping mechanism allows us to build a concise bus interface for the Lab 2 test harness, but you should ignore them for now. However, we will use the CBus mechanism later, in Lab 3, to develope some useful debugging tools.

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Correctness First, Performance Second

Design for correctness first, then worry about performance. Initially you should implement the design using the standard mkFIFO and mkSFIFO queues. Don't worry about the stalling logic yet, just get the structure correct. Pay special attention to the writeback queues `wbDataQ` and `wbDestQ`. When should the execute rule write into which queue? What assumptions will the writeback rule make about their contents.

After all the queues are in place implement the epoch, making the fetch send the appropriate tag to the memory system. Then change the execute rule to use the writeback queues instead of updating the register file itself. For now make execute use a conservative stall signal which always stalls at least once. Implement the writeback rule, taking into consideration the different situations: Load Responses, Store Acks, etc.,

At this point your design should be able to pass `make run-asm-tests`. Even such a naive design should show an improvement in IPC on the benchmarks over `smips-4mcycle`.

Then, once your system is complete, begin to improve performance iteratively. First implement the real stall signal using `SFIFO`. Then examine the schedule and see which rules are blocking each other. What can you do about that? Is this a place where a different FIFO could affect things? After making a change (particularly a change which involves adding (mkBFIFO1) make sure to

rerun `make run-asm-tests` to ensure correctness.

Tip 2: Check the Scheduler Output

As you begin to refine your design, pay close attention to the output of the Bluespec compiler (logged in the file `build/out.log`). Sometimes rule conflicts can point to a bug in the design. For example, early in my implementation I had a conflict between execute and writeback rules, as it turned out, this was because I had a bug where I forgot to change one leg of the execute case statement, so it was still updating the register file directly. After viewing the schedule I realized that something must be wrong.

This step 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. Make sure you understand all of this before you begin using the Bypass FIFO!

Tip 3: Observe rule CAN_FIRE and WILL_FIRE signals

As you refine, a good strategy is to run a single benchmark and observe the waveform. For each rule the compiler outputs a CAN_FIRE signal and a WILL_FIRE signal. While the scheduler output can give you a sense of static priorities, use these signals to observe the dynamic interactions of these rules.

A related tip is to observe the signals for FIFO methods. Each FIFO will have wires corresponding to enq, deq, and first, as well as RDY_enq, EN_deq, and so on. Observing when enqueues and dequeues are occurring can be extremely helpful, particularly if your design is deadlocking for some reason.

Tip 4: Think very carefully about 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 does not have any benefit.

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.

Critical Thinking Questions

As with Lab One, the primary deliverable for this lab assignment is your working Bluespec source code for the popelined SMIPSv2 processor. In addition, you should prepare written answers to the following questions and turn them in at the beginning of class on the due date. Qestion 2 involves adding a branch predictor, as you did in Lab One. Again, this may require significant design work. Again, do not worry if you are unable to finish the branch predictor question. As long as you make a reasonable effort you will not be penalized.

Question 1: Design Choices

Discuss and motivate any design choices you made. Is there any way in which your implementation differs from the diagram in Figure 1? 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 2: Adding a Branch Predictor

As with lab one, create a version of your design which adds a branch predictor and a branch history table. Similarly with Lab One, you should attempt this question after all other questions and tasks are finished. The purpose of this task is to think about designing modules in Bluespec from scratch, and then using them in a design.

This will encompass:

- Creating a new interface for the branch predictor.
- Creating a module which provides that interface.
- Changing your design to use this new module properly.

What was the effect does this new module have on your design's schedule? On the overall throughput of your processor?

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 labelling 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.