Lab 6: Pipelining an SMIPSv2 Processor: Part II

6.375 Laboratory 5
Assigned: March 11, 2011
Due: March 18, 2011

1 Introduction

This laboratory assignment continues the previous lab, improving the performance of your elastic SMIPSv2 pipeline. Obtaining good performance requires a solid understanding of how Bluespec schedules rules, something you should be an expert at by the time you complete this lab. Your task is to produce a pipelined design which functions correctly and achieves adequate performance in simulation and on the FPGA.

2 Achieving Pipeline Performance

If you run the benchmarks now you should see IPC numbers around those listed in table 1. This is a good improvement over the original unpipelined processor but is still a ways away from the best possible IPC of 1. If you look at the processor traces you'll see the rules do not always fire when you expect. To fix this you will have to understand how the rules are scheduled and choose implementations of your FIFOs and RegFile which have the right scheduling properties.

One reason your rules may not be firing when expected is if there is a conflict between the rules. Take for example the exec and writeback rules as they are now. The exec rule reads values from the register file and the writeback rule writes values to the register file. The register file requires reads happen before writes in the same cycle, so the exec rule must logically happen before the writeback rule when they are both enabled in the same cycle. At the same time, however, the exec rule calls the deq method of the wbQ and the writeback calls the enq method of the wbQ. The scheduling behavior of the mkSFIFO module is that enq must happen before deq, which says writeback must happen before exec when they are both enabled in the same cycle. This contradicts with the ordering requirement from the register file, meaning there is no logical ordering which satisfies the scheduling requirements, so the two rules conflict and cannot execute concurrently.

Consider now the size of the pcQ. If the pcQ is too small relative to the latency of the instruction cache, the pcQ will be full by the time the first response comes back from the instruction cache. If the pcQ is full the pcGen and exec rules might not be able to fire concurrently if the FIFO chosen for the pcQ doesn't allow simultaneous enqueue and dequeue when full.

Benchmark	IPC
median	0.38
multiply	0.38
qsort	0.42
towers	0.39
vvadd	0.39

Table 1: TA's IPC after removing stage register completely

2.1 Avoiding RWires

While we have introduced wire constructs explicitly in the lectures, it is generally not a good idea to use them directly in your designs. If you feel inclined to use wires, make sure you have a good reason for using them and always wrap the use of the wire in a small safe module. Do not directly instantiate RWires in your mkProc module! For example, rather than using an RWire directly, use library modules such as BypassFIFO or LFIFO, or wrap the RWire inside a small module as we will do for the Bypassed Register File.

2.2 Schedule Analysis in the Workstation

The Bluespec Workstation provides Schedule Analysis tools which will be very helpful in understanding why rules don't fire when you expect them to.

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 `mkDutWrapper` module, which the processor is a part of. The rule names may change slightly depending on the synthesis boundaries, but will always end with the original string which appeared in your BSV source. 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.

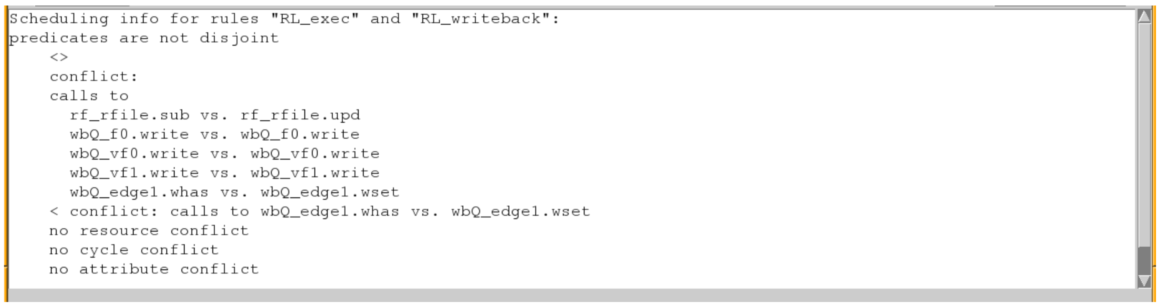
The rules are listed in the rule order tab in their logical order. This is the final global ordering of all the rules. Let's review a few scheduling terms before discussing how exactly to interpret this order. The term **urgency** refers to the relative priority given to two conflicting rules by the 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 sequentially 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. If the two rules are conflicting or mutually exclusive, their order is meaningless, so the compiler chooses an arbitrary order. Relative urgency and earliness can be set by using the pragmas descending_urgency and execution_order, which are described in the Bluespec reference guide.

You can get more information about how two rules are related by going to the Rule Relations tab in the Schedule Analysis window. For example, select the exec rule for Rule 1 and the writeback rule for Rule 2 and click Analyze. The analysis window will report something like that shown in figure 1.

Those items listed under the `<>` indicate reasons why the rules are not conflict free. For example, the first item

```
rf_rfile.sub vs. rf_rfile.upd
```

says the exec rule calls the `sub` method of the register file while the writeback calls the `upd` method of the register file. This places a restriction on the ordering of the exec and writeback rules. It is okay to have items listed under `<>` as long as they all require the same ordering constraint.



```
Scheduling info for rules "RL_exec" and "RL_writeback":
predicates are not disjoint
    <>
    conflict:
    calls to
      rf_rfile.sub vs. rf_rfile.upd
      wbQ_f0.write vs. wbQ_f0.write
      wbQ_vf0.write vs. wbQ_vf0.write
      wbQ_vf1.write vs. wbQ_vf1.write
      wbQ_edge1.whas vs. wbQ_edge1.wset
    < conflict: calls to wbQ_edge1.whas vs. wbQ_edge1.wset
    no resource conflict
    no cycle conflict
    no attribute conflict
```

Figure 1: Rule Analysis between exec and writeback

Module	Package	Size	enq vs deq	Simultaneous deq, enq?
mkFIFO	FIFO	2	CF	Not empty and not full
mkFIFO1	FIFO	1	CF	Never
mkSizedFIFO(n)	FIFO	n	CF	Not empty and not full
mkLFIFO	FIFO	1	deq < enq	Not empty
mkLSizedFIFO(n)	FIFO	n	deq < enq	Not empty
mkBypassFIFO	SpecialFIFOs	1	enq < deq	Not full
mkSFIFO	SFIFO	2	deq < enq, find	Not empty
mkSFIFO1	SFIFO	1	CF, deq < find	Never
mkSizedSFIFO(n)	SFIFO	n	C, find < enq,deq	Never.

Table 2: Various FIFOs and their properties

Those items listed under the < are reasons the first rule cannot be executed in sequence before the second rule. In this case we see an RWire is being set inside the wbQ which disallows the exec rule to be executed in sequence before writeback.

Taken together this means there is a conflict between the rules and they will never both fire in the same cycle.

2.3 Choosing the Right FIFOs

You have a number of choices to use for your FIFO implementations in the mkProc module. These are listed in table 2 with their properties. Use the schedule analysis tool to understand why your rules aren't firing together and determine what scheduling properties or sizes you need from your FIFOs to resolve any conflicts. Then consult table 2 for the appropriate FIFO.

You should only change the FIFO implementations inside the mkProc module. Assume the FIFOs in the caches are appropriately chosen already. That said, you may still need to consult the FIFO choices inside the caches to make the best decision about which FIFOs to use in the mkProc module.

The mkBypassFIFO is special in that it behaves like a wire if there is simultaneous enqueue and dequeue. This may increase the critical path if you aren't careful. Bypass FIFOs are great for when the data being enqueued either comes directly from or goes directly to a register.

2.4 Bypassing the Reg File

The register file we use has the scheduling property that reads must happen before writes. It may be the case you wish to use a register file with the property that writes happen before reads.

Benchmark	IPC
median	0.613147
multiply	0.630249
qsort	0.753039
towers	0.669303
vvadd	0.666667

Table 3: TA's IPC after bypassing reg file and choosing right FIFOs

Devise a new register file which schedules writes before reads and 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.

The MIPS ISA requires that a read from register r0 always return the value 0. This is why the implementation of mkRFile has the conditional statement

```
    return ( rindx == 0 ) ? 0 : rfile.sub(rindx);
```

instead of directly calling `rfile.sub(rindx);`. Your implementation of the bypassed register file must follow these same semantics. It should always return 0 for register r0, even if a nonzero value was written to register r0 the instruction before.

2.5 Resolving the Conflicts

Choose appropriate implementations for your FIFOs and register file from the choices described above to resolve any remaining conflicts between your pipeline stages. Your IPC after making these changes should be comparable to the TA's IPC numbers listed in table 3.

Remember to rerun the assembly tests too to verify you did not break anything, then check in your code to your local git repository.

3 Implementing a Better Branch Predictor

Currently your pcGen stage guesses the next pc will always be pc+4. We can improve the performance of our processor by implementing a smarter branch predictor. One goal of this task is for you to design and implement your own module from scratch.

1. Define an interface for your branch predictor with the methods we want to have. There should be a way to ask the predictor for its prediction and some way to update the predictor with information about branches taken.
2. Next you should change your processor to use your branch predictor. A good way to start would be to implement a branch predictor which always predicts pc+4 just like the processor does now. Plug that branch predictor into the processor and verify nothing broke and your IPC is the same as it was.
3. Implement a better version of the branch predictor. You are allowed to implement any algorithm you want, but we suggest you keep it simple and use a branch target buffer (BTB) to remember when we have seen a branch instruction and what its results was. Ideally, this is just

Benchmark	IPC
median	0.72
multiply	0.92
qsort	0.79
towers	0.74
vvadd	0.76

Table 4: TA's IPC with Branch Prediction

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.

Switch to using your new branch predictor in your processor and verify everything works and the IPC has improved as expected.

Table 4 shows the IPC of the TA's processor using the 16 element BTB predictor described above. Your processor should have IPC within 5% or better of the TA's IPC numbers.

4 Discussion Questions

Question 1: IPC

List the IPC of your design for each of the provided benchmarks.

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 much your branch prediction rate improved with the new predictor?

Question 2: Design Choices

Discuss and motivate any design choices you made. 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: 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 BypassFIFOs 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.

5 What to Turn In

When you have completed the lab you should check in a final version via git. This should include your 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 git any new source files you may have introduced. For example, if you didn't add any new source files, you could run

```
smips$ git add answers
smips$ git ci -m "Lab 6 final submission"
smips$ git push
```