Lab 6: Pipelining an SMIPSv2 Processor: Part II

6.375 Laboratory 5
Assigned: March 15, 2013
Due: March 22, 2013

1 Introduction

This laboratory assignment continues the previous lab, improving the performance of your SMIPSv2 pipeline. Your task is to produce a fully pipelined design which functions correctly and achieves adequate performance in simulation and on the FPGA. In order to achieve this you will be asked to implement a bypass register file, handle data hazards properly, verify your design is fully pipelined, use a better branch predictor, and meet 100Mhz as reported by synthesis. We then ask you to explore whether a bypassed register file improves the performance of your processor when the critical path is taken into account.

2 Bypassing the Reg File

The register file we use has the property that when reads and writes occur in the same cycle, the value read will be the old value. This means it is up to you to ensure you do not read and write the same register in the same cycle unless you want to read the old value of that register.

When pipelining your processor, it may be the case you wish to use a bypass register file with the property that when reads and writes on the same register occur in the same cycle, the written values are visible to the read in that cycle.

Problem 1: Implement a bypass register file which schedules writes before reads and bypasses written data if a simultaneous read address corresponds to the write address.

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

```
    if(rindx!=0) rfile[rindx] <= data;
```

when writing to the register file. 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.

3 Handling Data Hazards

Problem 2: Pipeline your processor, handling data hazards properly using the scoreboard technique discussed in lecture. The file `src/Scoreboard.bsv` contains an implementation of a scoreboard for you to use. Verify your pipelined processor works correctly in simulation. If you have pipelined your processor properly, you should no longer need the `state` register.

4 Verifying Concurrency in Your Pipeline

For your processor pipeline to be efficient, every stage should be capable of executing concurrently. Sometimes it can be hard to tell if your stages can execute concurrently or if there is some scheduling conflict.

4.1 Schedule Analysis in the Workstation

The Bluespec Workstation provides a Schedule Analysis tool which is 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 `mkProc` module. 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. 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 `doFetch` rule for Rule 1 and the `doExecute` 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 in this specific example:

```
pc.read vs. pc.write
```

says the `doFetch` rule calls the `read` method of the pc register while the `doExecute` calls the `write` method of the pc register. This places a restriction on the ordering of the `doFetch` and `doExecute` rules. It is okay to have items listed under `<>` as long as they all require the same ordering constraint.

Those items listed under the `<` are reasons the first rule cannot be executed in sequence before the second rule. In this case we see that the pc register is written in `doFetch` which disallows the `doExecute` rule to be executed in sequence before `doFetch`.

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

```
Scheduling info for rules "RL_doFetch" and "RL_doExecute":
predicates are not disjoint
    <>
    conflict:
    calls to
      pc.read vs. pc.write
      pc.write vs. pc.read
      pc.write vs. pc.write
      epoch.read vs. epoch.write
    < conflict: calls to pc.write vs. pc.read
    no resource conflict
    no cycle conflict
    no attribute conflict
```

Figure 1: Rule Analysis between `doFetch` and `doExecute`

Problem 3: Verify your processor is fully pipelined, and that all stages can execute concurrently. If your processor is not fully pipelined, change it so it is fully pipelined. You may need to use different kinds of Fifos, Scoreboards, or Reg Files to achieve concurrency, and you may wish to use Ehrs. See `Fifo.bsv` and `Scoreboard.bsv` to see what kinds of Fifos and scoreboards are available for you to use. Verify your design still works correctly in simulation.

5 Using a Better Branch Predictor

Currently your `doFetch` stage guesses the next pc will always be pc+4. We can improve the performance of our processor by using a smarter branch predictor. The file `src/AddrPred.bsv` includes an implementation of a branch target buffer predictor. The `mkBtb` predictor needs to be updated with information about actual branches taken. See the definition of the `Redirect` type in `src/ProcTypes.bsv` for more information on what fields the branch predictor expects to be updated with.

Problem 4: Use the `mkBtb` predictor in your pipeline instead of the `mkPcPlus4` predictor. Verify your processor still works in simulation. If you have used the new branch predictor properly, the IPC of your processor should have increased.

6 Completing the SMIPs Pipeline

Problem 5: Pipeline your design so it can meet 100Mhz according to the synthesis report. You may need to add more pipeline stages to achieve this clock rate. Verify your processor is still fully pipelined (all stages can run concurrently) and works correctly. Then synthesize your design and run it on the FPGA. Note that the Fifo and Scoreboard implementations use the modulo operator and hence are only synthesizable if their size is a power of two.

It was not hard for the TA to achieve a fully pipelined design which synthesis reports can meet 120Mhz. Table 1 shows the IPC of TA's design. You should be able to achieve similar IPC results in your implementation.

Benchmark	IPC
median	0.64
multiply	0.71
qsort	0.66
towers	0.76
vvadd	0.82

Table 1: TA's IPC

7 Design Exploration

IPC measures the performance of your processor in terms of cycles. This fails to take into account the critical path of your design. A processor with lower IPC may perform better in reality if it can be clocked at a higher frequency.

Let IPS be the performance of your processor in instructions per second, assuming the design is run at the frequency reported possible in the synthesis report. For example, if synthesis says your design can run at 108Mhz, and your IPC is x, then IPS = $108M * x$ instructions per second.

Problem 6: Experiment with two different configurations of your pipeline. One which uses the bypassed register file and another which uses the non-bypassed register file. Use the results of your experiments to answer discussion question 3 below.

8 Discussion Questions

Question 1: IPC

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

Question 2: Design Choices

Discuss and motivate any design choices you made. What size and type Fifos, Scoreboard, and register file did you end up using, and why is this a good configuration? In what logical order are your pipeline stages executed?

Question 3: Critical Path/IPC Tradeoff

Report the IPS of your best processor design using the bypassed register file and your best processor design using the non-bypassed register file. Does the bypassed register file improve the real performance of your processor? Why or why not?

9 What to Turn In

When you have completed the lab you should check in a final version via git. This should include your bypassed register file implementation and pipelined processor 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 -u .
smips$ git add answers
smips$ git commit -m "Lab 6 final submission"
smips$ git push
```