Due date: Thursday 5/17 11:59:59pm EDT (FINAL DEADLINE – NO LATE DAYS ALLOWED).

Special Instructions for Turning in Project: (Updated May 8th) After each exercise in this project (other than the Lab 7 part III exercise), commit your changes with a descriptive commit message like “Finished exercise 1” and push your changes. If you would like us to grade your project early to see if you got the necessary number of points for your desired grade, please send an e-mail to the course staff saying so.

Grading Note: In order to pass 6.S084 you must complete all labs and the final project. To “complete” the final project, you must complete the first three exercises of this final project. The point cutoffs for assigning grades for the course will be calculated using all labs, quizzes, and the first three exercises of this final project. The remainder of the final project give you an opportunity to get the necessary points for a higher grade.

Handout Version: 1.3 (5/8/2018)

Pipelined Processor Project

Note: The git repo for this project is called project.git and is in the same folder as the lab repos.

1 Processor Pipelining

The CEO of GATE Co. has given you a 4 stage pipelined processor and has asked you to improve the performance before they are used in the newest line of GATE Co.’s elevators.

You initially noticed three major issues with the processor:

1. Not all of the pipeline stages fire together.
2. The clock period is very long.
3. There are a lot of lost cycles due to mispredicted next PCs.

1.1 Firing Pipeline Stages Together

Currently the Fetch stage conflicts with the Execute stage since each stage writes to the pc register. You can avoid this conflict in the common case by splitting the Execute stage into two separate rules: one rule for instructions that don’t have to redirect the PC, and one rule for instructions that do have to redirect the PC.

Exercise 1 (5 Points): Split the doExecute rule into two rules to get the Execute and Fetch pipeline stages firing concurrently when Execute is not redirecting the PC. To make sure the pipeline stages are firing concurrently as intended, use test.sh to run the pipetests and look at the output of the schedule monitor (see Section 3.1 for more information on the schedule monitor).

Hint: To do this, you will need to define one or more new functions to compute the guards for the split rules.
1.2 Shortening the Clock Period

The provided processor implements the RISC-V multiply instruction `mul rd, rs1, rs2`. This instruction takes the 32-bit values in `rs1` and `rs2` and multiplies them together, storing the bottom 32-bits of the result in `rd`. The current implementation of the multiplier is combinational, and it is causing the clock period to be very long.

To reduce the clock period, you have decided to replace the combinational multiplier with a sequential multiplier.

**Exercise 2 (5 Points):** Make the multiply instruction use the provided radix-4 Booth Multiplier found in `Multiplier.bsv` instead of the combinational multiplier resulting from writing `rVal1 * rVal2`. To see if the `mul` instruction is working as expected, check the output from the Mandelbrot benchmark and the `mul fullasmtest` (they are the only programs in the `sw` folder that use the `mul` instruction).

1.3 Improving Next PC Predictions

Currently the Fetch stage always predicts the next PC to be PC + 4. This is correct for most instructions, but it causes all jumps and taken branches to be mispredicted. To improve the performance of the processor, you decided to add a Branch Target Buffer (BTB) to the processor.

**Exercise 3 (5 Points):** Create a 16 entry BTB to improve the next address predictor. Add the BTB as a next address predictor for your processor. Replace the `pc + 4` prediction with a prediction from the BTB, and make sure to train the BTB in case of jumps and mispredicted branch instructions.

*Hint:* Your BTB should be made out of normal registers, not SRAM. To implement an array of normal registers, you can use a `Vector` of registers as seen in `mkRFile2R1W` in `common.bsv`. The first parameter of `Vector` is the size of the vector. Be careful to make sure the size of your vector and the number of bits in your index are correct.

2 Improving Your Processor

*This part of the project is optional, see the note on grading at the beginning of the handout for details.*

One day you get to work in the morning and you find that the company has been raided by the FBI and the CEO has been arrested. There are some rumors around the office that he was corrupt, but you had seen no evidence of it. Since the CEO is gone, you are free to do what you want to do. You can either continue your work as usual, or go in your own direction to make improvements to the processor.

For this part of the project, choose one of the following two options.

2.1 Option 1: Continue your work as usual

You decide to just keep working on your previous work.

**Exercise 4 (5 Points):** Improve the bypassing of the processor by adding the following two bypassing paths:

1. Bypass from the writeback stage to the decode stage through the register file to remove a dead cycle.
2. Bypass from the redirect execute rule to the fetch rule to remove a dead cycle for mispredicted branches.

You will need to use the modules defined in the file `Bypassing.bsv` provided on the lab section of the course website (This includes a definition of `mkEhr`). You will also have to use an alternate scoreboard found in `Scoreboard.bsv`.

**Exercise 5 (10 Points):** Complete part III of Lab 7. You should check it in as part of your lab 7 and get it checked off before the due date of this project.
2.2 Option 2: Work on a project of your choice

You decide to work on your own project.

Exercise 6 (Up to 15 Points): Perform your own improvement to the processor. All improvements must be proposed to and approved by the course staff through their e-mail list.
3 Appendix

3.1 Pipeline Visualization with ScheduleMonitor

The provided processor contains a ScheduleMonitor module to provide a visual representation of which pipeline rules are firing each cycle. This module is simulation-only and produces no actual hardware. For a fully pipelined processor with no data or control hazards, this module may produce an output similar to the one below:

```
fetch
decode
  execute
    writeback
F___
FD__
FDE_
FDEW
FDEW
FDEW
FDEW
```

The names at the top are the names of each of the columns. These correspond to pipeline stages. The rows below correspond to what is happening in each clock cycle. The first row F___ means in the first clock cycle only fetch fired. The fifth row FDEW means in the fifth clock cycle all 4 stages of the pipeline fired concurrently.

There are four other letters that may appear as output from the ScheduleMonitor integrated with the provided initial code:

- x - An instruction was killed in-place in the specified stage.
- s - An instruction stalled in the decode stage due to a data hazard.
- p - An instruction was poisoned in the specified stage, or a poisoned instruction was passed into the specified stage.
- R - The execute stage fired and redirected the fetch stage due to a mispredicted next pc.

Using ScheduleMonitor

The module constructor for ScheduleMonitor (mkScheduleMonitor) takes in a File object (either stdout, stderr, or an opened text file) and a vector of pipeline stage names. The order of names in this vector determines the order of the columns in the output. The line below instantiates a ScheduleMonitor for a 5 stage pipeline that prints to stdout.

```
ScheduleMonitor monitor <- mkScheduleMonitor(stdout, vec("fetch", "decode",
  "execute", "writeback"));
```

```c
rule fetch;
  // do rest of fetch
  monitor.record("fetch", "F);
endrule
```

```c
rule decode;
  // do rest of decode
  if(...) begin
    // not stalling
```
monitor.record("decode", "D");
else if (...) begin
  // stalling
  monitor.record("decode", "s");
end else begin
  // killed
  monitor.record("decode", "x");
end
endrule

rule execute;
  // do rest of execute
  if (...) begin
    // not poisoned and not redirecting
    monitor.record("execute", "E");
  else if (...) begin
    // not poisoned and redirecting
    monitor.record("execute", "R");
  else begin
    // poisoned
    monitor.record("execute", "p");
  end
endrule

rule writeback;
  // do rest of writeback
  if (...) begin
    // not poisoned
    monitor.record("writeback", "W");
  else begin
    // poisoned
    monitor.record("writeback", "p");
  end
endrule

The record method of ScheduleMonitor writes a character in the specified column of the pipeline schedule diagram. Typically the first letter of the pipeline stage is written in the column when the stage fires normally, but the above code uses some other letters to show special conditions.

### 3.2 cycle and instret CSRs

The initial processor for this project includes two read-only CSRs for use as performance counters. The cycle CSR counts the number of clock cycles that have elapsed since simulation started. The instret CSR counts the number of instructions that have retired.

Since these are read-only CSRs, we only support the csrr pseudoinstruction for these two instructions. A csrr to cycle decodes to an instruction with an iType of RDCYCLE. A csrr to instret decodes to an instruction with an iType of RDIINSTRET.

To properly implement these CSRs we need to ensure the following holds:

- There is a separate rule for incrementing the cycle counter by one. The rule must not conflict with any other rules in order to get an accurate cycle count (< and > scheduling relations are okay).

- The instret counter must be incremented by one in the write back stage for each non-poisoned instruction. A mispredicted branch should still increment instret, but all the following wrong-path instructions should not increment instret.
• In the writeback stage, before writing to the register file, you should set data to \texttt{cycle} or \texttt{instret} if the iType of the current instruction is \texttt{RDCYCLE} or \texttt{RDIINSTRET} respectively.

These CSRs are used to produce the cycle and instruction counts printed at the end of the benchmarks.