CSEE 4824 Project #3

Note:

- This is an individual assignment. While you may discuss the specification and help one another with the SystemVerilog language, your solution – particularly the designs you submit – must be your own.
- The project milestone is due by **Thursday, February 22nd** (worth 5% of the project grade)
- This project is due by **Thursday, February 29th**
- Project 3 is considerably more work than projects 1 and 2. Do not leave it until the last minute!

1 Introduction

In this project you will be implementing the hazard and forwarding logic for a RICS-V, pipelined, 5-stage processor: the VeriSimpleV processor. This will help prepare you for the final project, where you will work on a team to extend VeriSimpleV to use one of the out-of-order implementations discussed in class, either a Pentium 6 or MIPS R10K based design.

For this project there is an added milestone – worth 5% of your project grade – that hopes to ensure you do not leave the project until the last minute. The milestone is due by **Thursday, February 22nd**.

2 The VeriSimpleV Processor

The VeriSimpleV processor is a 5-stage pipelined implementation of the 32-bit integer subset of the RISC-V instruction set architecture (ISA). It is written in synthesizable, behavioral SystemVerilog. VeriSimpleV is based on the 5-stage pipeline covered in class and has Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback stages (IF, ID, EX, MEM, WB).

But the processor is unfinished! The provided implementation *has no hazard detection logic!* To accommodate this, the provided processor only allows one instruction in the pipeline at a time to be absolutely certain there are no hazards. The provided processor is correct, and it will produce the correct output for all programs, but it has a miserable CPI of 5.0.

Note on clock period: The Makefile is set with a clock period of 30.0ns. This is both because of the full 32-bit multiplication in EX stage's `ALU` module, and because we want to speed up synthesis by giving `dc_shell` a break. You will use the pipelined multiplier from project 2 in the final project to decrease your processor's clock period. However, you should not change the multiplier or clock period for project 3.

3 Assignment

Your assignment is to modify the provided VeriSimpleV implementation to remove the stalls and implement hazard and forwarding logic as described in lecture and the text.

Implement the following hazard and forwarding logic:

Structural Hazards Access to memory is shared between the IF and MEM stages. Stall IF and let MEM have priority if there is a conflict.

This is the only hazard required for the milestone.

Control Hazards Predict all branches as not taken. Detect taken branches and squash any predicted instructions.

Data Forwarding Most data dependencies should have their values forwarded into the EX stage (even if the data aren't used until a later stage)

Data Hazards Not all data dependencies can be forwarded. Stall any instructions whose data will not be ready in time (Hint: only one type of instruction causes these hazards)

Your solution is subject to the following restrictions:

- Branches should resolve in the same stage as they are currently resolved.
- When stalling an instruction, be sure to set both the valid bit to 0 and the instruction to a `` `NOP ``.
- Data Hazard stalling should occur in the ID stage (since data are forwarded to EX). Instructions in the IF stage will need to wait on ID, and stalls (invalidated instructions) should appear in the EX stage.

Your solution will be graded in simulation on the criteria of memory correctness and correct CPI. You must have the correct memory output (relative to the provided processor) to get points for either, and you must match the exact CPI of a correct processor to get the CPI points.

Some example correct outputs (including the non-evaluated `.ppln` file output) are provided in the `correct_out/` folder.

3.1 Getting Started

Start the project by removing the provided stalling behavior, implement structural hazards for the milestone, then implement the rest of the hazard and forwarding logic.

The provided stalling behavior is set in the `verilog/pipeline.sv` file. You should open the file and find the `always_ff` block where the `next_if_valid` signal is set. This is the start of a `valid` bit which is passed between the stages along with the instruction, and it starts at 1 in the IF stage due to `next_if_valid`. The `next_if_valid` signal is currently set to read the valid bit from the WB stage, so will insert 4 invalid instructions between every valid one.

Start by the setting `next_if_valid` to always equal 1.

3.2 Hints

- Be careful with forwarding and register 0. the testcase and computer.
- There is a *lot* of SystemVerilog here; take your time looking it over. Try to understand how it works before you modify it. The slides from Finite State Machine - Lab 3 will also help walk you through it.
- Start this process early!

4 Project Files

For this project, you are provided with most of the code and the entire build and test system. This is a quick overview of the Makefile and the verilog files you will be editing. See the lab 4 slides for an extended discussion.

The VeriSimpleV pipeline is broken up into 9 files in the `verilog/` folder. There are 2 headers: `sys_defs.svh` defines data structures and typedefs, and `ISA.svh` define decoding information used by the ID stage. There are 5 files for the pipeline stages: `stage_{if,id,ex,mem,wb}.sv`. The register file is in `regfile.sv` and is instantiated inside the ID stage. Finally, the stages are tied together by the pipeline module in `pipeline.sv`.

The `verilog/sys_defs.svh` file contains all of the `typedef's` and `` `define's `` that are used in the pipeline and testbench. The testbench and associated non-synthesizable verilog can be found in the `test/` folder. Note that the memory module defined in the `test/mem.sv` file is non-synthesizable.

4.1 Running Programs with the Makefile

Now that you've moved up to a complete processor design, testing is less about the testbench and more about the programs. You have been provided with many assembly and C code programs in the `programs/` folder.

To run a program on the processor, run `make <my_program>.out`. This will assemble a RISC-V `*.mem` file which will be loaded into `mem.sv` by the testbench, and will also compile the processor and run the program.

All of the "`<my_program>.abc`" targets are linked to do both the executable compilation step and the `.mem` compilation steps if necessary, so you can run each without needing to run anything else first.

`make <my_program>.out` should be your main command for running programs: it creates the `<my_program>.out`, `<my_program>.wb`, and `<my_program>.ppln` output, writeback, and pipeline output files in the `output/` directory. The output file includes the status of memory and the CPI, the writeback file is the list of writes to registers done by the program, and the pipeline file is the state of each of the pipeline stages as the program is run.

The following Makefile rules are available to run programs on the processor:

```
# ---- Program Execution ---- #
# These are your main commands for running programs and generating output
make <my_program>.out      <- run a program on simv
                              generate *.out, *.wb, and *.ppln files in 'output/'
make <my_program>.syn.out  <- run a program on syn_simv and do the same

# ---- Executable Compilation ---- #
make simv      <- compiles simv from the TESTBENCH and SOURCES
make syn_simv  <- compiles syn_simv from TESTBENCH and SYNTH_FILES
make *.vg      <- synthesize modules in SOURCES for use in syn_simv
make slack     <- grep the slack status of any synthesized modules

# ---- Program Memory Compilation ---- #
# Programs to run are in the programs/ directory
make programs/<my_program>.mem  <- compile a program to a RISC-V memory file
make compile_all                <- compile every program at once (in parallel with -j)

# ---- Dump Files ---- #
make <my_program>.dump  <- disassembles compiled memory into RISC-V assembly dump files
make *.debug.dump       <- for a .c program, creates dump files with a debug flag
make dump_all           <- create all dump files at once (in parallel with -j)

# ---- Verdi ---- #
make <my_program>.verdi     <- run a program in verdi via simv
make <my_program>.syn.verdi <- run a program in verdi via syn_simv

# ---- Visual Debugger ---- #
make <my_program>.vis  <- run a program on the project 3 vtuber visual debugger!
make vis_simv          <- compile the vtuber executable from VTUBER and SOURCES

# ---- Cleanup ---- #
make clean            <- remove per-run files and compiled executable files
make nuke             <- remove all files created from make rules
```

Figure 1: Reference table of Make targets

5 Submission

Put the following files in a zip and submit it to CourseWorks:

- `verilog/pipeline.sv`
- `verilog/regfile.sv`
- `verilog/stage_if.sv`
- `verilog/stage_id.sv`
- `verilog/stage_ex.sv`
- `verilog/stage_mem.sv`
- `verilog/stage_wb.sv`
- `verilog/sys_defs.svh`
- `verilog/ISA.svh`