EECS 470 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 **Monday, February 5th** (worth 5% of the project grade)
- The project is due by **Sunday, February 11th**
- Project 3 is considerably more work than projects 1 or 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, 5-stage pipelined processor: the VeriSimpleV processor. You will also create multiple short Assembly unit tests to expose buggy processors. This will prepare you for EECS 470's 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 Pro (P6) 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 **Monday, February 5th**.

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.

2.1 RISC-V

VeriSimpleV implements a subset of the 32-bit RISC-V ISA. It implements the Base I (*Integer*) and M (multiplication/division) extensions, however we do not test or implement division or remainder instructions. We also do not implement interrupts or other system level instructions, and memory is limited to 64KB. The processor starts with the `PC` at address 0 and will halt upon writeback of a WFI instruction (Wait For Interrupt, which will never come).

We use the the RISC-V toolchain installed on CAEN to test the processor by compiling and running both Assembly (`*.s`) and C (`*.c`) programs found in the `programs/` folder (see the Makefile section below for how to run programs).

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. You will also create multiple short assembly unit tests to expose buggy processors.

3.1 Assembly Unit Tests

For this part you will write short RISC-V Assembly unit tests to expose memory correctness or CPI bugs in buggy processors. Each test must be 15 or fewer total lines (including comments) and must run to completion on a correct processor. It will then be run on a buggy processor and compared to the correct output. If they differ for any of your test cases, then you exposed the bug!

RISC-V assembly instructions look like:

```
addi x1, x0, 5    # register 1  = register 0 + 5
```

You should reference our sample assembly programs in `programs/` to write test cases which expose a buggy processor via either memory correctness or CPI.

You may submit up to five test cases by adding files to the `programs/` folder that match the pattern: `programs/test_[12345].s`

i.e. `programs/test_1.s`

These are only due at the end of the project and not the milestone.

3.2 Hazards and Processor Implementation

Implement the following hazard and forwarding logic in the processor:

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 and synthesis on the criteria of memory correctness and correct CPI. You must have the correct memory output to get any points – the provided processor is already correct – but you are mainly graded on matching your CPI to a correct implementation.

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

4 Project Files

For this project, you are provided with most of the code and the entire build system. This is a quick overview of the Makefile and the verilog files you will be editing.

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

The testbench and associated non-synthesizable verilog exists in the `test/` folder. You should not modify any of the files in `test/` for project 3. Note that the memory module defined in the `test/mem.sv` file is non-synthesizable.

Assembly and C programs are dependent on multiple files that you should not change. Assembly files use a custom Assembly linker (`programs/aslinker.lds`). C programs depend on a custom C linker (`programs/aslinker.lds`), a custom malloc implementation (`programs/tj_malloc.h`), and a C-RunTime file (`programs/crt.s`) which zeroes out registers and calls a C program's `main` function. You should not change any of these.

Lab 4's assignment will have you create a test system to build a "ground truth" from the initial processor that you can compare against as you build your processor.

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` (ex: `make no_hazard.out`). This will both compile your processor in verilog and build the program's initial memory state as a `*.mem` file which will be loaded into `mem.sv` by the testbench to start the program (at `PC=0`).

`make <my_program>.out` should be your main command for running programs: it creates the `<my_program>.out`, `<my_program>.cpi`, `<my_program>.wb`, and `<my_program>.ppln` output, CPI, writeback, and pipeline output files in the `output/` directory.

***.out** – General output and the final state of memory

***.cpi** – The CPI calculation and overall program execution time

***.wb** – The list of writes to registers done by the program

***.ppln** – The state of each of the pipeline stages as the program is run

This is the most useful file for debugging, but is not as useful for final output comparison, since it contains internal details that don't need to match exactly to have a correct implementation.

4.2 Getting Started

We recommend you start the project by removing the provided stalling behavior and implementing structural hazards for the milestone. Then proceed by creating some assembly test cases before implementing the rest of the hazard and forwarding logic.

The stalling behavior is set in the `verilog/cpu.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 setting `next_if_valid` to always equal 1.

Then run a program like `no_hazard` and see what happens!

4.3 Notes

- Be careful with forwarding and register 0.
- Synthesized runs of the pipeline can take a few minutes, depending on 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 Lab 4 will also help walk you through it.
- Start this process early!

5 Submission

To submit your project to the EECS 470 autograder, upload your solution files to the `main` branch of your GitHub repository and run the project submission script for project 3:

```
/afs/umich.edu/class/eecs470/Public/470submit 3
```

Note: This will also run the milestone autograder and return the results in the same email (until the milestone due date – **Monday, February 5th**).

Soon after your submission you will receive an email with a summary of the public output of the autograder. This will contain only the public results of correctness tests and whether the autograder encountered any errors and is not representative of your final grade.

Email the instructors or create a private Piazza post to debug any issues with the autograder.

The following files will be graded:

- `verilog/cpu.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`
- Assembly tests matching the pattern `programs/test_[12345].s`.
  i.e. `programs/test_1.s`
  These are only due at the end of the project and not the milestone.

Appendix: Makefile Targets

The following Makefile targets 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
                              output *.out, *.cpi, *.wb, and *.ppln files
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 Makefile targets