EECS 470 Project #3

- This is an individual assignment. You may discuss the specification and help one another with the SystemVerilog language. The modifications you submit must be your own.
- This assignment is worth 4% of your course grade.
- Due at 11:55pm EDT on Sunday, 9th February, 2020. *Late submissions are not accepted.*
- **This assignment is considerably more work than programming assignments 1 and 2. Do not leave it until the last minute!**

---

1 Introduction

VeriSimpleV is a simple pipelined implementation of a subset of the RISC-V instruction set architecture written in synthesizable, behavioral SystemVerilog. The structure of this implementation is very similar to the MIPS pipeline covered in the text. We have provided you with a base version, which has absolutely no hazard detection logic. This version of VeriSimpleV inserts 4 invalid instructions (stalls) after every instruction to remove any possibility of a hazard.

2 Assignment

Your assignment will be to modify the provided implementation of VeriSimpleV to handle hazards and forwarding. You will need to begin by modifying the `if_stage.sv` file to issue valid instructions so that more than one is in the pipeline at a time.

Your solution is subject to the following restrictions:

- Branches should resolve in the stage in which they are currently resolved.
- All forwarding must be to the `EX` stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the `ID` stage, meaning the dependent instruction should wait in the `ID` stage. Obviously the instruction following the stalling instruction in the `ID` stage will need wait in the `IF` stage. Thus, if you need to insert an invalid instruction (a stall), it must appear in the `EX` stage.
- If you wish to insert a `noop` you must also invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the `load/store` go and have the `IF` stage wait for the bus to be free.

You will need to add logic to handle all types of hazards: structural, control and data. You will also need to add logic to forward data to avoid data hazards, where possible within the limitations above, and add stalls if and only if there is a data hazard that cannot be resolved by forwarding. You should predict branches as not taken and squash if incorrect. Verify that your improved pipeline produces the same results as our provided version.

Your improved pipeline will be tested in synthesis as well as simulation, so you should test it that way as well. Your submission will be graded automatically by comparing the files output by the provided testbench, which include the contents of the pipeline, the contents of memory and the CPI.

2.1 Hints

- 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!

3 Project Files

For this project, you are provided with most of the code and the entire build and test system. **The source files are available at this repository**. Here is a quick introduction to what you've been provided and how it's structured.

The VeriSimpleV pipeline is broken up into 7 files in the `project3/verilog/` folder. There are 5 files which correspond to the pipeline stages (`project3/verilog/if_stage.sv`, etc.); the register file module is separated into the `project3/verilog/regfile.sv` file and instantiated by the ID stage; and the stages are tied together by the pipeline module, which can be found in the project3/verilog/pipeline.v file.

The `project3/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 nonsynthesizable verilog can be found in the `project3/testbench/` folder. Note that the memory module defined in the `project3/testench/mem.sv` file is nonsyn- thesizable.

Testing this project is less about the testbench and more about the testcases. Now that you've moved up to a complete processor design, testing requires running programs. You have been provided with a set of testcases in the `project3/test_progs/` folder, written in RISC-V assembly or C language. To run one of them, you first have to assemble it into machine code, which is done using `riscv64-unknown-elf-gcc`. The rules for compiling testcases in the Makefile are as follows:

```
SOURCE = test_progs/sampler.s
...
GCC = riscv64-unknown-elf-gcc
...
program:  compile disassemble hex
   @:
...
assembly:  assemble dissemble hex
   @:
```

`make assembly` reads the code in from the `project3/test_progs/sampler.s` file and writes the assembled machine code out to the `project3/program.mem` file, which is then read into memory by the testbench. To compile another test case, you need to override the `SOURCE` variable in the Makefile. To comile `project3/test_progs/*.c` testcases, `make program` should be used. Finally, once you have an assembled program ready to test, you can run the VeriSimpleV the same way you've run every other project so far, with the make command. If you need to run a little more interactively, to see where a particular instruction is in the pipeline for instance, you have been provided with a visual debugger, which can be run using the `make vis_simv` command. Play around with this. It will make the project much easier.

4 Submission

You will submit this project the same way you've submitted the last two, using the submission script. In addition, you will need to create a **private** repository on Bitbucket with the name `eecs470_f19_project3_{uniquename}`, and grant Admin access to the account `eecs470staff@umich.edu`. After you create the repo, you will need to check your code into the repository.