



ECE4700J Computer Architecture

Summer 2022

Lab #4 RISC-V Five-Stage Pipeline Optimization

Due: 11:59pm Jun. 26th, 2022 (Beijing Time)

Logistics

- This lab is an individual exercise. You may discuss the specification and help one another with the (System)Verilog language. The modifications you submit must be your own.
- All the design code should be in SystemVerilog.
- There will not be a live demo for this assignment.
- All code and reports (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- This assignment is considerably more work than programming assignments 1, 2 and 3. Do not leave it until the last minute!
- No late submission is allowed for this lab.

Overview

In this lab, you will study how to design a RISC-V five stage pipeline module, and build the software environment for it.

- Study how to write a processor core.
- Study how to handle hazards for a pipeline.
- Be able to build and use some RISC-V toolchains, write your own testcase and use your processor to run your own program.

Assignments

I. Optimize the VeriSimpleV Pipeline (Mandatory)

a) Introduction

VeriSimpleV is a simple pipelined implementation of a subset of the RISC-V instruction set architecture, written in synthesizable, behavioral SystemVerilog. 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.



b) Project Files and Software Environment Setup

i. Project Source 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: [HieronZhang/project-v-open-beta-For-ECE4700J \(github.com\)](https://github.com/HieronZhang/project-v-open-beta-For-ECE4700J). 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 `baseline/verilog/` folder. There are 5 files which correspond to the pipeline stages (`baseline/verilog/if_stage.sv`, etc.); the register file module is separated into the `baseline/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 `baseline/verilog/pipeline.v` file.

The `baseline/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 `baseline/testbench/` folder. Note that the memory module defined in the `baseline/testbench/mem.sv` file is nonsynthesizable.

The `optimization/` folder initially just contains exactly the same contents as in the `baseline/` folder. When you do your optimizations and modify the source code, you should work in the `optimization/` directory. Do not modify Verilog code in `baseline/` directory. The baseline can also serve as the **correct output generator** to help check whether your implementation is correct.

ii. Preparing for the Mem file

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 `baseline/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 `riscv32-unknown-elf-gcc`. The rules for compiling testcases in the Makefile are as follows:

```
# make assembly - Build the memory file for an assembly source
# make program - Build the memory file for a C program
SOURCE = test_progs/rv32_copy.s
```

To generate your memory file (containing machine code), you need to first modify the `SOURCE` variable in Makefile to specify which source program you want to run (by default, the `rv32_copy.s`). The Makefile doesn't have default target, you should run "make program" if your source is a C program and run "make assembly" if your source is an assembly program.

For example, `make assembly` reads the code in from the `baseline/test`



progs/rv32_copy.s file and writes the assembled machine code out to the baseline/program.mem file, which is prepared to be read into memory by the testbench. At the same time, the compiler will also generate some other files:

- program.elf: The ELF file of your program
- program.dump: The dump file of your ELF file, containing PC addresses and corresponding assembly instructions, generated by the disassembler.
- program.debug.dump: The dump file for debugging. It contains your original assembly instructions and their corresponding PC address and layout.

These files (especially the program.debug.dump), are really important for your debugging of this assignment.

To successfully build your memory file, you first need to install a set of RISC-V toolchains. (gcc, objdump, as, elf2hex for RISC-V). The details in Makefile are shown below:

```
GCC = riscv32-unknown-elf-gcc
OBJDUMP = riscv32-unknown-elf-objdump
AS = riscv32-unknown-elf-as
ELF2HEX = riscv32-unknown-elf-elf2hex
```

To install your own RV toolchains, first you need to go to this repo: [riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC \(github.com\)](https://github.com/riscv-collab/riscv-gnu-toolchain), and follow its instruction. You may also refer to this article: [riscv 各种版本 gcc 工具链 编译与安装 | 骏的世界 \(lujun.org.cn\)](https://lujun.org.cn/riscv-各种版本-gcc-工具链-编译与安装). Note that what our target should be riscv32-unknown-elf.

One example is to type these commands in your Linux/Ubuntu WSL shell:

```
# Clone the Toolchain Repo
git clone https://github.com/riscv/riscv-gnu-toolchain

#Build all the prerequisites. This is for Ubuntu
sudo apt-get install autoconf automake autotools-dev curl python3
libmpc-dev libmpfr-dev \
libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev libexpat-dev

#Compile and Install
cd riscv-gnu-toolchain
mkdir build
cd build
../configure --prefix=/opt/riscv32 --with-arch=rv32imc
sudo make -j8
```



Then after you have the gcc, you also need to install the elf2hex, which can be found here: [sifive/elf2hex: Converts ELF files to HEX files that are suitable for Verilog's readmemh. \(github.com\)](https://github.com/sifive/elf2hex). Note that our target should be riscv32-unknown-elf. One example is to type these commands in your Linux/Ubuntu WSL shell:

```
# Clone the elf2hex Repo
git clone git://github.com/sifive/elf2hex.git
cd elf2hex

#Configure targeting rv32
autoreconf -i
./configure --prefix=/opt/elf2hex --target=riscv32-unknown-elf

#Compile and Install
make
sudo make install
```

Also remember to include the path where you install them in your PATH variable (if you didn't install it in your /usr/local/ directory) so that your shell can know where it is. You can do this by modifying your .bashrc file. One example is to add these lines in your ~/.bashrc file (if your riscv32 toolchains and the elf2hex are installed in /opt/riscv32/ and /opt/elf2hex/:

```
export PATH=/opt/riscv32/bin:$PATH
export PATH=/opt/elf2hex/bin:$PATH
```

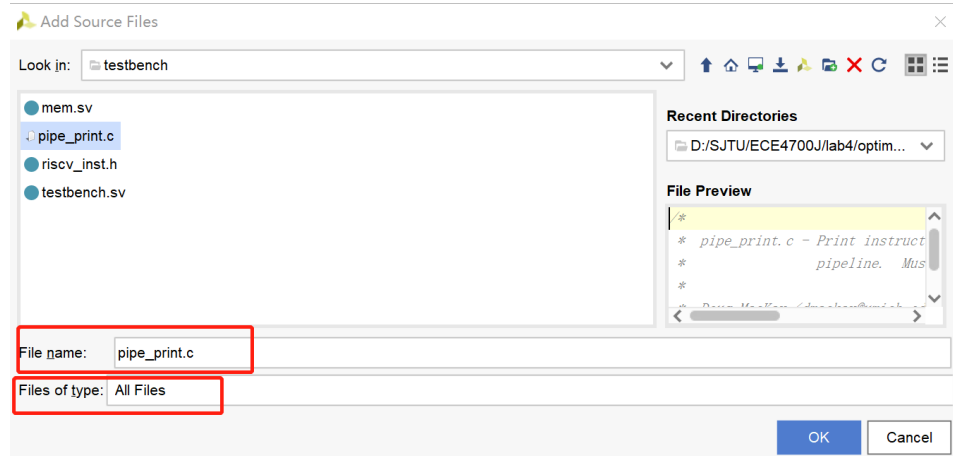
After you modified your .bashrc file, remember to source it:

```
source ~/.bashrc
```

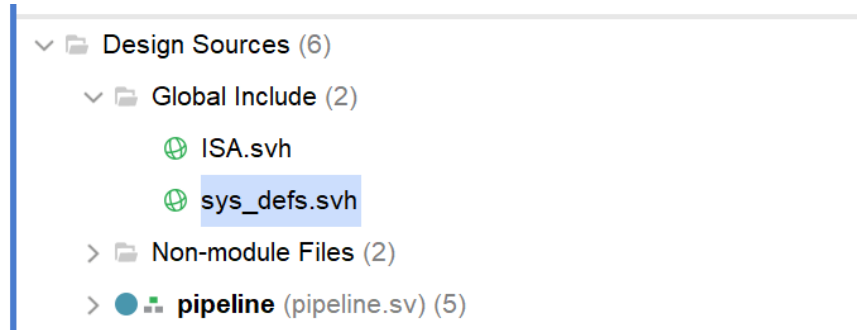
Then you should be able to directly use your toolchains.

iii. Setup Vivado Project

To setup the Vivado project, use all the SystemVerilog files in the baseline/verilog/ directory (if, id, ex, mem, wb, pipeline, regfile), the **ISA.svh** and **sys_defs.svh** as your **design source files**. Use all the files in the baseline/testbench/ directory (testbench.sv, mem.sv, **riscv_inst.h**, **pipe_print.c**), and the memory file (**program.mem**) as your **simulation source files**. To add C files into your Vivado, you need to choose “All files” in the “files of type” option.



Make sure your top module of design is **pipeline**, your top module of simulation is **testbench**. Then make sure to set the **ISA.svh** and **sys_defs.svh** files as **global includes**. Otherwise Vivado will show “syntax error”. This is done by select the files and right-click, select the “Set Global Include”.



Then you can do your behavioral simulation once your **program.mem** file is ready. You may meet this error while simulating: “[USF-XSim-102] Failed to set the LIBRARY_PATH env!”. Just ignore it, as it cannot affect the simulation results.

c) Assignment

Your assignment will be to modify the provided implementation of VeriSimpleV to handle hazards and forwarding. Modify the code in **optimization/** directory to do the optimization. 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.

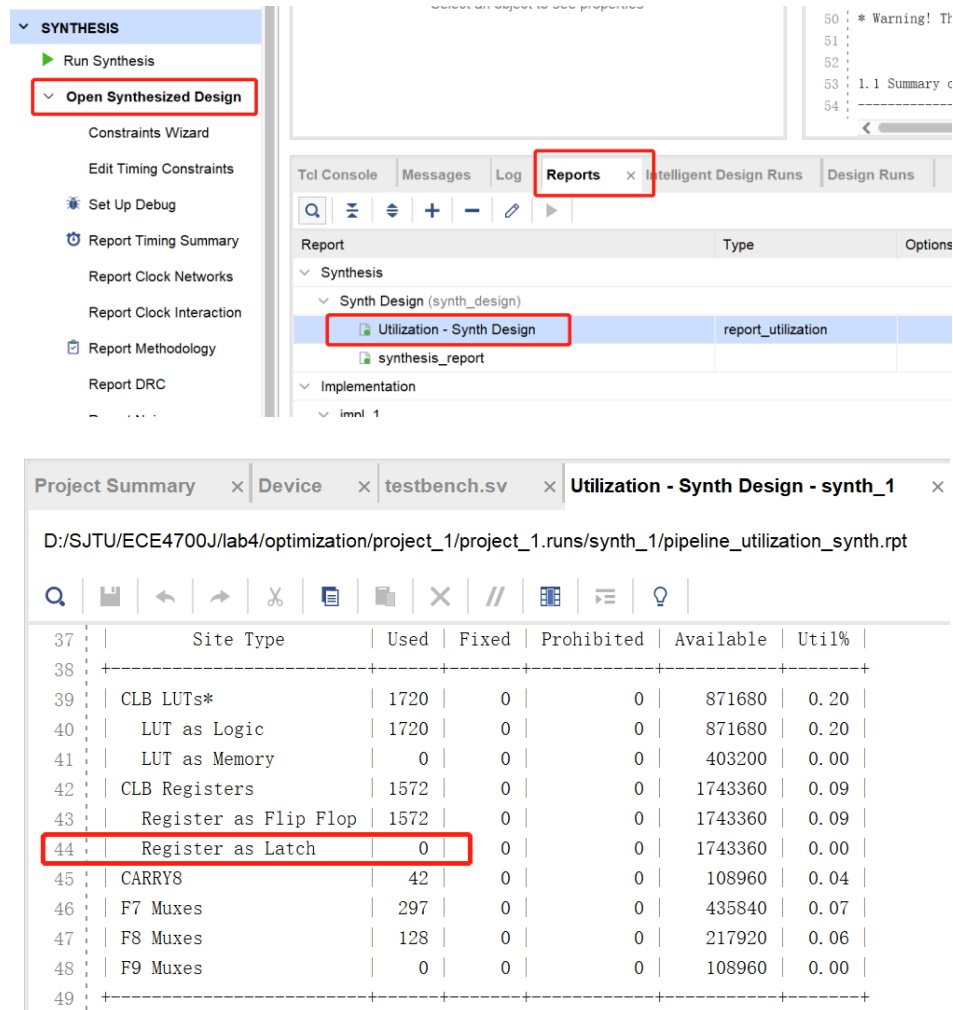
- If you wish to insert a noop you must also invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the load/store go and have the IF stage wait for the bus to be free.

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 submission will be graded automatically by comparing the files output by the provided testbench, which include the contents of the pipeline, the contents of writeback record, memory and the CPI (More details in later section). We provide you a set of C programs and RV assembly programs for you to test your pipeline, however, when grading we use a lot of other hidden cases (which we will not provide). You are encouraged to write your own C programs or RV assembly programs to test your pipeline.

To make your life easier (also mine), we **only require** you to **pass the behavioral simulation of Vivado**. (Actually, if you try to run a post synthesis simulation, Vivado will report ERROR as this: “ERROR: [VRFC 10-2649] an enum variable may only be assigned the same enum typed variable or one of its values [D:/SJTU/ECE4700J/lab4/optimization/testbench/testbench.sv:82]”. This is because Vivado doesn’t support SystemVerilog header files very well. Even if you already use global includes, its compiler still thinks that your definition of a same enum type used at different places is different. To solve this problem, one way is to not use enum typedef in the sys_defs.svh. But for this assignment, we will not require post-syn simulations.)

Besides that, you **must** still ensure that your design **can be successfully synthesized and generates no latches**. The way to check whether your code generates latches is to see the utilization report at “Open synthesized Design” – “Reports” – “Utilization – Synth Design”. Normally on line 44 of that report you will see the number of latches you generated. (See figure below)



The screenshot shows the Synthesis tool interface. On the left, the 'SYNTHESIS' menu is expanded, and 'Open Synthesized Design' is highlighted. In the center, the 'Reports' tab is selected, showing a list of reports. 'Utilization - Synth Design' is highlighted. Below this, the 'Utilization - Synth Design - synth_1' report is open, displaying a table of resource utilization.

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs*	1720	0	0	871680	0.20
LUT as Logic	1720	0	0	871680	0.20
LUT as Memory	0	0	0	403200	0.00
CLB Registers	1572	0	0	1743360	0.09
Register as Flip Flop	1572	0	0	1743360	0.09
Register as Latch	0	0	0	1743360	0.00
CARRY8	42	0	0	108960	0.04
F7 Muxes	297	0	0	435840	0.07
F8 Muxes	128	0	0	217920	0.06
F9 Muxes	0	0	0	108960	0.00

d) Hints

- Be careful with forwarding and register 0.
- 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!

e) Testing and Debugging

When you do simulation of your code & program, first, your testbench will directly generate some outputs (we call them program output). An example program output will look like this:

@@



```
@@
@@           0 Asserting System reset.....
@@
@@           0 : System STILL at reset, can't show anything
@@
@@
@@           100 : System STILL at reset, can't show anything
@@
@@
@@           200 : System STILL at reset, can't show anything
@@
@@
@@           300 : System STILL at reset, can't show anything
@@
@@           360 Deasserting System reset.....
@@
@@
@@@ Unified Memory contents hex on left, decimal on right:
@@@
@@@ mem[  0] = 0000113700000313 : 18927920874259
@@@ mem[  8] = 03f301b300a00f93 : 284573069775736723
@@@ mem[ 16] = 0001220300312023 : 318871260176419
@@@ mem[ 24] = 0081011310412023 : 36311453384384547
@@@ mem[ 32] = 0103229300130313 : 72940033724842771
@@@ mem[ 40] = 10500073fe0292e3 : 1175440000926520035
@@@
@@@ mem[ 4104] = 000000000000000a : 10
@@@ mem[ 4112] = 0000000000000014 : 20
@@@ mem[ 4120] = 000000000000001e : 30
@@@ mem[ 4128] = 0000000000000028 : 40
@@@ mem[ 4136] = 0000000000000032 : 50
@@@ mem[ 4144] = 000000000000003c : 60
@@@ mem[ 4152] = 0000000000000046 : 70
@@@ mem[ 4160] = 0000000000000050 : 80
@@@ mem[ 4168] = 000000000000005a : 90
@@@ mem[ 4176] = 0000000000000064 : 100
@@@ mem[ 4184] = 000000000000006e : 110
@@@ mem[ 4192] = 0000000000000078 : 120
@@@ mem[ 4200] = 0000000000000082 : 130
@@@ mem[ 4208] = 000000000000008c : 140
@@@ mem[ 4216] = 0000000000000096 : 150
@@@
@@@ mem[ 4360] = 000000000000000a : 10
@@@ mem[ 4368] = 0000000000000014 : 20
@@@ mem[ 4376] = 000000000000001e : 30
@@@ mem[ 4384] = 0000000000000028 : 40
@@@ mem[ 4392] = 0000000000000032 : 50
@@@ mem[ 4400] = 000000000000003c : 60
@@@ mem[ 4408] = 0000000000000046 : 70
@@@ mem[ 4416] = 0000000000000050 : 80
@@@ mem[ 4424] = 000000000000005a : 90
@@@ mem[ 4432] = 0000000000000064 : 100
@@@ mem[ 4440] = 000000000000006e : 110
@@@ mem[ 4448] = 0000000000000078 : 120
@@@ mem[ 4456] = 0000000000000082 : 130
```




```
@@@ mem[ 4464] = 000000000000008c : 140
@@@ mem[ 4472] = 0000000000000096 : 150
@@@
@@@
@@          23220 : System halted
@@
@@
@@@ System halted on WFI instruction
@@@
@@
@@ 229 cycles / 131 instrs = 1.748092 CPI
@@
@@ 2280.00 ns total time to execute
@@
```

You need to ensure that **the CPI value** and **all the lines starting with @@@** should be exactly the same as the correct output. (You can use the baseline to generate correct outputs)

Besides the program output, the testbench will also generate two other files: pipeline.out and writeback.out. (They are in the simulation directory of your Vivado project. e.g. project_1/project_1.sim/sim_1/behav/xsim) The pipeline.out is a visualized pipeline output for you to debug your design. The writeback.out is a record of every instruction's writeback. You also need to ensure that **all the contents of your writeback.out file is exactly the same as the correct one**. (Again, you can use the baseline to generate correct outputs)

In your optimization/example_output directory, we provide you some example outputs.

When you want to test your pipeline on different programs, you don't need to recreate a Vivado project. You can just build a new program.mem file using Makefile and then directly run your Vivado simulation again.

You are also highly recommended to use your own testcases. You can write your own C programs or Assembly programs to test your pipeline. Here are some rules for your testcase program:

- Do not use malloc(). Use our own version of malloc: tj_malloc() and tj_free(). They are included in the test_progs/tj_malloc.h.
- Avoid using "/" and "%". Our VerySimpleV ISA is a subset of RV32IM, but it doesn't support divide and remainder.
- Avoid using any system calls.
- **Important:** Since the RISC-V ISA is under developing, its toolchain is also under developing. What we installed is the newest version compiler, however, it sometimes may generate instructions that our baseline pipeline can't recognize (The pipeline is implemented in 2019). If you find that one program caused the baseline pipeline to exit from "illegal instruction", don't worry, it's not your problem. You only need to ensure your optimized pipeline generates the same result (as what we defined) as the baseline do.



f) Submission

You need to submit all the contents of your optimization/ folder. Compress all of them into a tar file, which is named by your name (in lowercase).

e.g. `tar -cvf haoyangzhang.tar optimization/` ”

And submit the tar file via canvas before the deadline.



II. Auto-test Batch Scripts (Optional)

In this optional assignment, you will be designing an auto-test batch script for your pipeline. You will be given an up to 10% bonus to the total score of Lab 1 if you are able to correctly implement the optional assignment. This will be counted towards your final grade.

When I was designing the Lab4 Assignment, I tried to write an auto-test system for you. However, some tricky ERROR message of Vivado stopped me. When I tried to use tcl script to add the sys_defs.svh as design files into my Vivado project, it always fails. The ERROR message is “[Common 17-165] Too many positional options when parsing 'sys_defs.svh'”. Xilinx has some help note for this error message: [46668 - Vivado - ERROR: \[Common 17-165\] Too many positional options when parsing '<name>.tcl', please type '<command> -help' for usage info \(xilinx.com\)](#). However, I don't think this can solve my problem. What's trickier, when I type the same tcl command in the GUI, it works without any error. To make my life easier, I gave up on developing this and leave this problem to you. If anyone can solve this problem and correctly build an auto-test system, you will be given additional 10% points of this assignment.

Your goal is to write some tcl scripts along with some shell scripts or batch scripts (for Windows, .bat file), so that you are able to run all the programs in your test_progs/ directory and show whether your pipeline passes each of these programs, by simply running the auto-test script(either .sh or .bat). Once you finish this, inform the TA via Feishu.

I also put one of my tcl script file and the bat file (which is to call the Vivado and source the tcl script) in the optional/ directory.



JOINT INSTITUTE
交大密西根学院

References:

1. Umich EECS470 WN 2021 Project3

Acknowledgement:

- Jon Beaumont (University of Michigan)



JOINT INSTITUTE
交大密西根学院

Deliverables:

- Individual Deliverables:
 - Submit design files of the Mandatory Assignment: the yourname.tar file via canvas assignment.

Grading Policy

Canvas Files Submission and Correctness – 100%