# ECE4700J Computer Architecture

Summer 2024

## Lab #3 RISCV Five-Stage Pipeline Optimization

Due: 11:59pm (Beijing time) Jun. 14th, 2023

**Logistics:**

- This lab is an individual exercise.
- Please use the discussion board on Piazza for Q&A.
- All reports and code (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- Though this is a two-week assignment, it is considerably much more work than lab assignments 1 and 2. Do not leave it until the last minute!

## Contents

# 1 Overview

In this lab, you will study how to design a RISCV 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 RISCV toolchains, write your own test cases and use your processor to run your own program.

# 2 Software Environment Setup

In this lab, the real testcases are to be generated by yourselves. Now that you've moved up to a complete processor design, testing requires running programs. You have been provided with a set of testcase source codes 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".

## 2.1 Toolchain Installation

To successfully build your memory file, you first need to install a set of RISCV toolchains. (gcc, objdump, as, elf2hex for RISCV). The details in the provided `Makefile` in the starter files 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: https://github.com/riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC, and follow its instruction. You may also refer to http://www.lujun.org.cn/?p=4257 for instructions. Note that what **our target should be "riscv32-unknown-elf"**.

### General Installation Commands

```
### For Linux/Ubuntu WSL shell
# Step 1: Clone the Toolchain Repo
git clone https://github.com/riscv/riscv-gnu-toolchain

# Step 2: 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

# Step 3: 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: https://github.com/sifive/elf2hex: Converts ELF files to HEX files that are

suitable for Verilog's readmemh. Note that **our target should be riscv32-unknown-elf**.

### General Installation Commands

```
### For Linux/Ubuntu WSL shell
# Step 4: Clone the elf2hex Repo
git clone git://github.com/sifive/elf2hex.git
cd elf2hex

# Step 5: Configure targeting rv32
autoreconf -i
./configure --prefix=/opt/elf2hex --target=riscv32-unknown-elf

# Step 6: Compile and Install
make
sudo make install
```

Also, remember to include the path where you install them in your PATH variable so that your shell can know where it is. You can do this by using the "export" command and 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 by `source ~/.bashrc`.

Then you should be able to directly use your toolchains.

## 2.2 Compile and Assemble Testcases

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, the rules for compiling testcases in the `Makefile` are as follows:

Build the memory file for assembly sources (`.s` file):

```
make assembly
```

Build the memory file for C-program sources (`.c` file):

```
make program
```

### How to interpret the output?

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**.

# 3 Optimize the VeriSimpleV Pipeline!

## 3.1 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 provided under in CANVAS. Here is a quick introduction to what you've been provided and how it's structured.

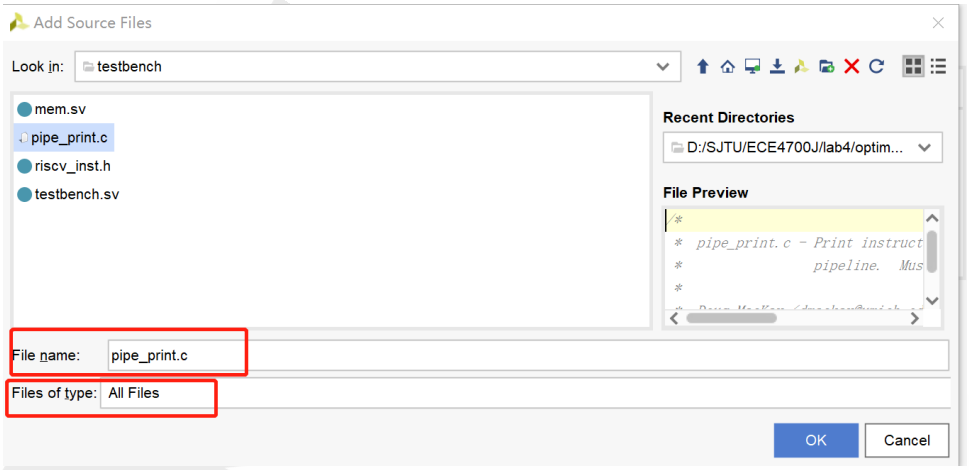
After you unzip the starter file, the VeriSimpleV pipeline is broken up into 7 files in the `baseline/verilog/` folder. There are 5 files which correspond to the pipeline stages (`baseline/verilog/if_stage.sv`,`baseline/verilog/id_stage.sv`, etc.); the register file module is separated into the `baseline/verilog/regfile.sv` 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.

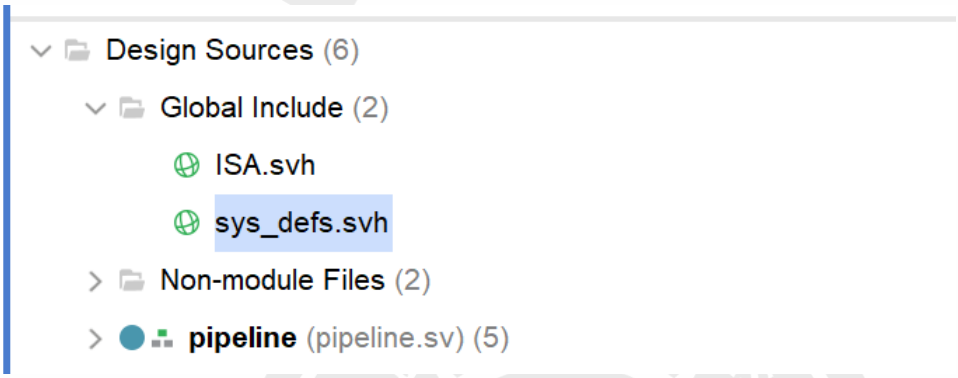
The `optimization/` folder initially just contains exactly the same contents as in the `baseline/` folder. When you do your optimizations and modify the source code, **you should work in the `optimization/` directory**. **Do not modify Verilog code in `baseline/` directory**. The baseline can also serve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

## 3.2 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 (See the picture below).




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.

## 3.3 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 (i.e. not a single-cycle processor).

Your solution is subject to the following restrictions:

- Branch judgement is done in EX stage and instruction "flush" should be done immediately in the next cycle.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.

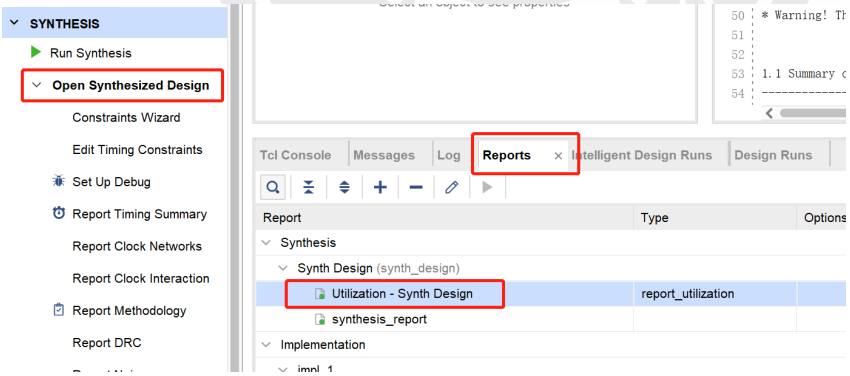
- Any stalling due to data hazards must occur in the ID stage, meaning the dependent instruction should wait in the ID stage. Obviously the instruction following the stalling instruction in the ID stage will need wait in the IF stage. Thus, if you need to insert an invalid instruction (a stall), it must appear in the EX stage.
- If you wish to insert a nop 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, 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. 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)



Project Summary | Device | testbench.sv | Utilization - Synth Design - synth_1

D:/SJTU/ECE4700J/lab4/optimization/project_1/project_1.runs/synth_1/pipeline_utilization_synth.rpt

| 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 |

## 3.4 Hints for Testings and Debuggings

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[ 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 complier, 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 does.

# 4 Deliverables

Please compress all the following files into a single `.zip` file named with your student ID, e.g. ECE4700lab3_519370910000.zip

- Section 3.3: `optimization/sys_defs.sv` file.
- Section 3.3: All the files under directory `optimization/verilog` (if you declare your own modules for the pipeline like `hazard_detect.sv` etc, do include those files in this directory)

**Please don't edit other provided files! They will not be used during grading.**

# 5 Grading policy

| Factors | Percentage |
|---|---|
| Section 3.3 design (fib_rec, rv32_copy) | 80% |
| Section 3.3 design (hidden cases) | 20% |

# References

1. UM-SJTU JI ECE4700J SU 2022 Lab4
2. Umich EECS470 WN 2021 Project3

# Acknowledgement

- Haoyang Zhang (ECE4700J SU 2022 TA)
- Jon Beaumont (University of Michigan)
- Xilinx