ENSC 254 Final Project

Important Logistics:

- The final project weighs 25% of the final marks. It includes 60 points in total, which will be scaled to 25% of the final marks.
- The final project will be done and graded per 2-student group. You can select your new teammate on the course Canvas, or you can continue working with your current lab teammate. You must select your teammate by July 11th.
- Please make sure your code can compile and run correctly on the lab computers (i.e., FAS-RLA Linux computers). If your code cannot be compiled on the lab computers, then you get 0 marks for the final project. If your code can compile, but cannot run correctly, then you only get the points where your code runs correctly on the lab computers.

Introduction of the Final Project:

The purpose of this final project is to enhance your understanding of the hardware architecture design of the RISC-V CPU, mainly focusing on its pipelined datapath and control logic implementation in the hardware, and its cache organization. Specifically, you will develop a cycle-accurate simulator of a RISC-V CPU, which is built on top of your lab 2-4. A cycle-accurate simulator is an essential tool in computer architecture for detailed and precise modeling, analysis, and optimization of hardware systems at the cycle level. By providing detailed cycle-by-cycle information, such simulators allow computer architects to identify performance bottlenecks and optimize the hardware architecture design. Some widely used cycle-accurate simulators by computer architects include gem5, GEMS, Multi2Sim, Sniper, to name just a few.

To help you stay on track of this challenging (but fun) final project, we have divided it into two milestones, with a suggested due date for each milestone.

We will first release milestone 1 and then gradually release the following milestones.

Milestone 1: [25 points] Basic pipeline without hazard detection/resolving

- Simulate a basic (perfect) 5-stage single-issue pipeline as taught in Lecture 8-10
- Assume there is no pipeline hazard in the instruction stream; we have manually inserted nops in the instruction stream to achieve this
- For any given cycle, simulate which instruction is performed at which pipeline stage; you need to include a global counter to simulate the clock cycles

Milestone 2: [35 points] Full pipeline with hazard detection/resolving

- Revise the simulator from milestone 1 to model a full 5-stage single-issue pipeline
- The instruction streams have all sorts of data/control hazards as taught in Lecture 8-10, which need to be detected and resolved in the pipeline
- For any given cycle, for each pipeline stage, simulate which instruction is performed, the detection of any data or control hazard, and the resolving of such hazard with stalling and/or forwarding techniques, and how many cycles of stalls (total number of stalls) are there in the pipeline execution.

General Grading Logistics:

To succeed in the final project, we highly encourage you to work closely in a 2-student team. While you can divide the workload by milestones or the implementation of different pipeline stages, based on the extensive discussions between TAs and myself, it's very hard to grade based on what you individually did: any individual team member's failure may lead to the failure of the entire project, just like real projects in the industry. So, for the final project, by default, each team member will get the same points, and we will not provide WorkDistForGrading.csv as done in lab assignments. However, if some of the team members didn't do their assigned jobs, please DOCUMENT it and we will handle it case by case.

Framework Code:

Milestones 1 and 2 will be built on top of your lab 2 and lab 3 code. In the project.zip file, we have provided you a modified framework on top of your lab 2-4 code to build a cycle-accurate simulator; note you will need to copy utils.c, disasm.c, and emulator.c from your own lab 2-3 code, and cache.c from your own lab 4 code, to replace the files in the provided code.

Other than the files you worked on in lab 2-4, please pay attention to the following important files on which you will be working for the project.

1. pipeline.c: This file contains the top function of your cycle-accurate simulator: cycle_pipeline. You can also include your stage execution functions (described below) inside this file.
2. pipeline.h: Header file to declare all the data structures and some function prototypes that you are using for the simulator. We have provided the skeleton code; you have to complete these data structures.
3. stage_helpers.h: Header file to define helper functions that you will be using inside the stage execution functions.
4. utils.c, disasm.c, and emulator.c: Please replace these files with your own version from lab 2 and 3. You can make further changes if necessary. Before replacing disasm.c, back up its modified ***decode_instruction*** function (with extra error checking); after replacing disasm.c with your own version, replace the ***decode_instruction*** function with our modified version which you have backed up.
5. cache.c: Please replace this file with your own version from lab 4. You can make further changes if necessary.
6. cache.h: DO NOT replace this file with your own version from lab 4. You need to change the cache configurations in this file only.

Important: You must only modify the above files for the project. When we do the auto grading, we will only copy the above files into our work directory. Therefore, any changes you make to other files will not be considered.

In the new framework provided, you can use the following flags to enable different modes (i.e., disassembler, emulator, simulator, etc.).

-d: run the disassembler

-v: initialize the register file to value 4 except x0

-r: enable dumping register file (Note: for the cycle-accurate simulator, in addition to this flag, you need to enable DEBUG_REG_TRACE in pipeline.h as described below)

-i: enable interactive mode for the emulator

-t: enable interactive mode for the emulator along with the disassembler where it prints each instruction

-e: run program until completion via exit with ecall

-s: enable cycle-accurate simulator

-f: enable hazard detection and resolving in the simulator (this should be used along with -s)

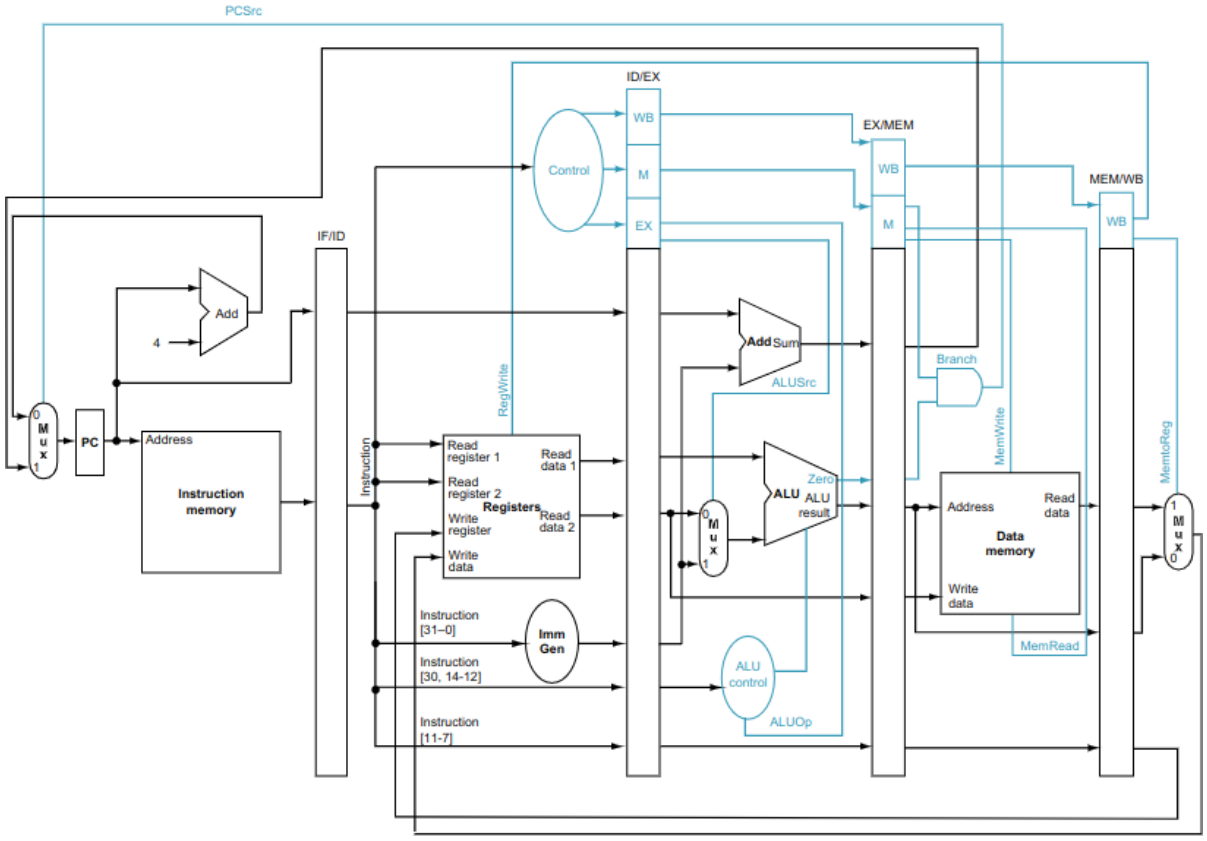
-c: enable cache simulation in the simulator (this should be used along with -s)

-m: enable emulator

-p: enable memory dump after the simulation

Milestone 1: [25 points] Basic pipeline without hazard detection/resolving

Pipeline Diagram



Milestone 1 Overview:

The main goal in this part is to implement a basic cycle-accurate simulator for a single-issue RISC-V pipeline in C++ without hazard detection or resolving. The simulator will accurately track the execution of instructions through each stage of the pipeline and the transitions between stages, modeling how an ideal processor handles instruction execution.

An overview of the 5-stage pipeline of RISC-V architecture is given in the above figure (Lecture 8). It has the following stages communicating via pipeline registers.

1) Instruction Fetch (IF) - Fetches instructions from the instruction memory.
2) Instruction Decode (ID) - Decodes instructions and reads from the register file.
3) Execute (EX) - Performs arithmetic or logical operations, calculates memory addresses or branch targets.
4) Memory (MEM) - Accesses data memory for loading and storing instructions.
5) Write Back (WB) - Writes results back to the register file.

The simulator will model these five stages of a RISC-V pipeline. Each stage will interact through pipeline registers that help in transitioning the instruction's state from one stage to the next.

Milestone 1 Implementation Details:

Pipeline Registers and Pipeline Register Pairs

- All the signals associated with the pipeline registers are shown in the figure in black color. This figure is a copy of Fig. 4.53 from the textbook, i.e., slide 53 of Lecture 8.
- Each *pipeline register* struct ('ifid_reg_t', 'idex_reg_t', 'exmem_reg_t', 'memwb_reg_t') defined in *pipeline.h* encapsulates the data passed between two successive pipeline stages.
- Each *pipeline register pair* struct ('ifid_reg_pair_t', 'idex_reg_pair_t', 'exmem_reg_pair_t', 'memwb_reg_pair_t') defined in *pipeline.h* contains a pair of the pipeline registers, 'inp' and 'out':
  - 'inp': The current pipeline stage's input ('inp') is based on the output ('out') of the preceding pipeline stage from the previous cycle.
  - 'out': The current pipeline stage's output ('out') is used as the input ('in') of the following pipeline stage in the next cycle.

Pipeline Wires

Pipeline wires can be recognized by the blue sections in the figure. They carry important control information to/from the pipeline registers and help implement important combinational logic between non-successive stages. All the wires for the combinational logic that go across different stages (example: branching) should be added to the *pipeline_wires_t* struct in *pipeline.h* so that it can be passed as an argument to individual stages easily. Note that the blue sections in the pipeline registers should be added to the pipeline register structs.

Pipeline Stages

You should write five separate functions to handle the process happening inside each pipeline stage. For the exact functionality of each function, please follow the figure and the textbook (Lectures 8-10). A summary of these functions (pipeline.h and pipeline.c) is given below with the function names.

1) **stage_fetch**: This function has access to instruction memory, PC, and pipeline wires. It works on fetching instructions from the instruction memory, updating PC accordingly, and writing data to the ifid_reg (of ifid_reg_t type).
2) **stage_decode**: This function has access to ifid_reg, register file and pipeline wires. It works on decoding the instruction, reading the register file for rs1 and rs2 (if necessary), generating imm, and updating idex_reg (of idex_reg_t type type).
3) **stage_execute**: This function has access to idex_reg and pipeline wires. As shown in the figure, it works on executing ALU and passing the results to the exmem_reg (of exmem_reg_t type).
4) **stage_mem**: This function has access to exmem_reg, pipeline wires, and data memory. It works on accessing the data memory and passing down the values to memwb_reg (of memwb_reg_t type).
5) **stage_writeback**: This function has access to memwb_reg, pipeline wires, and register file. It is working on writing the results to the destination register (rd).

Simulation Cycle

The crux to building a cycle-accurate simulator is to have a functional specification on all processes (i.e., pipeline stages) that happen in a clock cycle. We have created a wrapper for this functional specification and named it 'cycle_pipeline' (in pipline.c) with the intent that each call to this function must simulate exactly one clock cycle of the pipeline states.

Within this wrapper, we call all five pipeline stages (functions mentioned above).

For each cycle, simulate the pipeline operation by calling:

- Each stage_* function reads from the output ('out') of the pipeline register pair from its preceding pipeline stage, performs its designated operations, and writes the input ('out') of the pipeline register pair for its following pipeline stage.
- After all pipeline stages have been processed, the 'inp' versions of all pipeline register pairs are copied to their 'out' versions to be used in the next cycle.
- Increment the 'total_cycle_counter' at the end of each cycle. This variable is used to track the number of cycles that it has simulated.

This will give you a basic implementation of the pipeline without hazard detection/resolving.

Note to disasm.c

You have to change your ***decode_instruction*** function in your disasm.c file to append the following code at its beginning. The reason to do this is because the pipeline will be uninitialized for the first 4 cycles, and so the call to parse_instruction would fail without the following code.

```
if(instruction_bits == 0)
{
   printf("\n");
   return;
}
```

In the project.zip file we have provided you with an updated 'decode_instruction' function (in disasm.c) with this code added. Therefore, make sure to keep this function 'decode_instruction' unchanged when you move your own lab 2/3 code to replace disasm.c.

Your TODOs:

Please check pipeline.h, pipeline.c, and stage_helpers.h, search "YOUR CODE HERE"; this is where you have to fill in your code, to make the whole pipeline work. Have fun :-)

Milestone 1 Testing

During milestone 1 testing, we use the following outputs to check the accuracy of the simulator.

1. Cycle counter
2. A print indicating the instruction that is processed in each pipeline stage during every cycle. The printing format is like this:
   [STAGE]: Instruction [INSTR_HEX]@[INSTR_ADDRESS]: INSTR

3. Similar register trace that we used for lab 3. Instead of dumping registers for every instruction, we will be dumping registers for every cycle in the simulator.

A sample output is as follows.

```
v==============Cycle Counter =     5==============v

[IF ]: Instruction [00474d33]@[00001014]: xor   x26, x14, x4
[ID ]: Instruction [007ecd33]@[00001010]: xor   x26, x29, x7
[EX ]: Instruction [00000013]@[0000100c]: addi  x0, x0, 0
[MEM]: Instruction [00000013]@[00001008]: addi  x0, x0, 0
[WB ]: Instruction [00000013]@[00001004]: addi  x0, x0, 0
r 0=00000000 r 1=00000000 r 2=000effff r 3=00003000
r 4=00000000 r 5=00000000 r 6=00000000 r 7=00000000
r 8=00000000 r 9=00000000 r10=00000000 r11=00000000
r12=00000000 r13=00000000 r14=00000000 r15=00000000
r16=00000000 r17=00000000 r18=00000000 r19=00000000
r20=00000000 r21=00000000 r22=00000000 r23=00000000
r24=00000000 r25=00000000 r26=00000000 r27=00000000
r28=00000000 r29=00000000 r30=00000000 r31=00000000
```

Important: You must **ENABLE DEBUG_REG_TRACE and 'DEBUG_CYCLE' in pipeline.h.** This enables the dumping of the above information. Also make sure you **DISABLE ALL OTHER PRINTS** before generating this output.

There are five tests given for the milestone 1 evaluation. There is a script called ***test_simulator_ms1.sh*** to execute all the commands.

- R-type instructions (3 points)

This input will help you to verify basic pipeline functionality and correctness of your ALU instructions.

Commands to run:

./riscv -s -v ./code/ms1/input/R/R.input > ./code/ms1/out/R/R.trace

diff ./code/ms1/ref/R/R.trace ./code/ms1/out/R/R.trace

- I-type instructions except load instructions (3 points)

This input will help you to verify whether your simulator works correctly with the immediate value generation and perform ALU instructions with the immediate.

./riscv -s -v ./code/ms1/input/I/I.input > ./code/ms1/out/I/I.trace

diff ./code/ms1/ref/I/I.trace ./code/ms1/out/I/I.trace

- Load instructions and S-type instructions (3 points)

This input will help you to verify how your simulator works with memory accesses.

./riscv -s -v ./code/ms1/input/LS/LS.input > ./code/ms1/out/LS/LS.trace

diff ./code/ms1/ref/LS/LS.trace ./code/ms1/out/LS/LS.trace

- Multiply testcase (3 points)

This is an actual testcase generated based on an actual program. This includes a mix of instructions including UJ-type and SB-type instructions.

./riscv -s -v ./code/ms1/input/multiply.input > ./code/ms1/out/multiply.trace

diff ./code/ms1/ref/multiply.trace ./code/ms1/out/multiply.trace

- Random testcase (3 points)

This is an actual testcase generated based on an actual program. This includes a mix of instructions including UJ-type and SB-type instructions.

./riscv -s -v ./code/ms1/input/random.input > ./code/ms1/out/random.trace

diff ./code/ms1/ref/random.trace ./code/ms1/out/random.trace

Milestone 1 Marking (25 points)

- Milestone 1 contains 25 points of the project.
- 15 points are given for the successful completion of the above tests. There are five tests, each carries 3 points. For each test, if the entire test passes (i.e., match with the reference output), you will earn all 3 points; otherwise, you will get 0 points for the test. No partial points will be given.
- 5 points are given for the project report. In this part of the report, please describe your implementation including the following information.
    - How do you generate control logic in the ID stage?
    - How is the 'mux' shown in the IF stage implemented?
    - How is the 'mux' shown in the EX stage implemented?
    - A table of the 'alu_op' values based on the instruction
    - A table of 'alu_control' values based on the instruction
    - What is the logic behind the implementation of your gen_branch function?
- 5 points are given for the demo of this part. Earning these points depends on how confident you will explain and answer questions about your work related to this part during the demo.