ENEE446: Project 1 - Pipeline Simulator

Spring 2021

Deadline: March 31, 2021, at 11:59pm

In this assignment, you will become familiar with the RISC-V integer and floating point pipelines. You will create a cycle-accurate simulator of an in-order RISC-V processor, called `pipesim`, that supports 32-bit integer and floating point operations. Your simulator will be very **basic**: it will model all the necessary pipeline interlocks to enforce correct pipeline execution, but it will not support out-of-order execution, data forwarding, nor branch prediction.

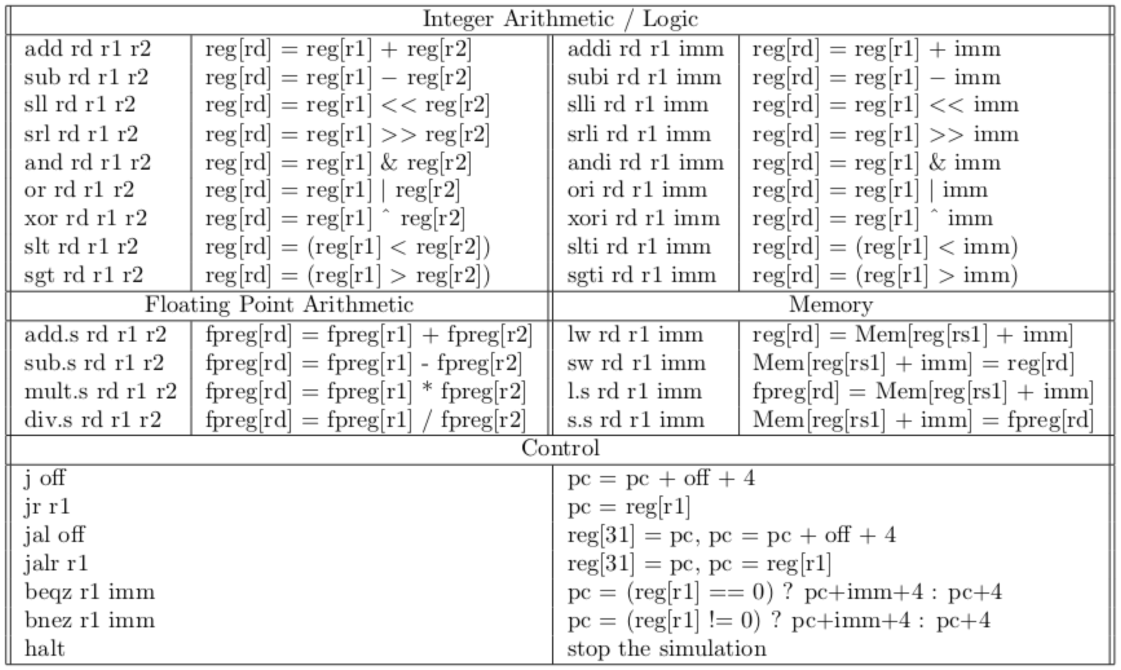
1 Files

Please download the project files from **"Project1"** folder on ELMS. There are 18 files: `asm.c`, `main.c`, `pipeline.c`, `pipeline.h`, `fu.c`, `fu.h`, `output.c`, `output.h`, `Makefile`, `io_pipe.fu`, `simple.s`, `simple.io.out`, `vect.s`, `vect.io.out`, `newton.s`, `newton.io.out`, `cos.s`, and `cos.io.out`.

`asm.c` is an assembler for the RISC-V-like ISA which your simulator will implement (more about the assembler and the ISA later). `main.c, pipeline.c, pipeline.h, fu.c, fu.h, output.c, and output.h` are the simulator source files. Much of the simulator's internals, like the functional units and the simulator's output generator, have been implemented already and are provided in these files. Your assignment is to build the main control module of the pipeline (most of the changes will go in **pipeline.c**). `Makefile` is a unix make file which will produce the binaries `asm` and `pipesim`. `io_pipe.fu` is a functional unit configuration file (more about this later).

In the remaining files, we have provided **four** example assembly programs. `simple.s` performs a simple arithmetic sequence, `vect.s` performs a vector operation, `newton.s` finds the left-most root of a quadratic expression using Newton's method of root approximation, and `cos.s` estimates the cosine of "1" using a Taylor series expansion with 10 iterations. `simple.io.out, vect.io.out, newton.io.out, and cos.io.out` contain the output for these 4 benchmarks on a properly functioning simulator.

2 RISC-V ISA



Integer Arithmetic / Logic			
add rd r1 r2	reg[rd] = reg[r1] + reg[r2]	addi rd r1 imm	reg[rd] = reg[r1] + imm
sub rd r1 r2	reg[rd] = reg[r1] − reg[r2]	subi rd r1 imm	reg[rd] = reg[r1] − imm
sll rd r1 r2	reg[rd] = reg[r1] << reg[r2]	slli rd r1 imm	reg[rd] = reg[r1] << imm
srl rd r1 r2	reg[rd] = reg[r1] >> reg[r2]	srli rd r1 imm	reg[rd] = reg[r1] >> imm
and rd r1 r2	reg[rd] = reg[r1] & reg[r2]	andi rd r1 imm	reg[rd] = reg[r1] & imm
or rd r1 r2	reg[rd] = reg[r1] | reg[r2]	ori rd r1 imm	reg[rd] = reg[r1] | imm
xor rd r1 r2	reg[rd] = reg[r1] ˆ reg[r2]	xori rd r1 imm	reg[rd] = reg[r1] ˆ imm
slt rd r1 r2	reg[rd] = (reg[r1] < reg[r2])	slti rd r1 imm	reg[rd] = (reg[r1] < imm)
sgt rd r1 r2	reg[rd] = (reg[r1] > reg[r2])	sgti rd r1 imm	reg[rd] = (reg[r1] > imm)
Floating Point Arithmetic		**Memory**	
add.s rd r1 r2	fpreg[rd] = fpreg[r1] + fpreg[r2]	lw rd r1 imm	reg[rd] = Mem[reg[rs1] + imm]
sub.s rd r1 r2	fpreg[rd] = fpreg[r1] - fpreg[r2]	sw rd r1 imm	Mem[reg[rs1] + imm] = reg[rd]
mult.s rd r1 r2	fpreg[rd] = fpreg[r1] * fpreg[r2]	l.s rd r1 imm	fpreg[rd] = Mem[reg[rs1] + imm]
div.s rd r1 r2	fpreg[rd] = fpreg[r1] / fpreg[r2]	s.s rd r1 imm	Mem[reg[rs1] + imm] = fpreg[rd]
Control			
j off		pc = pc + off + 4	
jr r1		pc = reg[r1]	
jal off		reg[31] = pc, pc = pc + off + 4	
jalr r1		reg[31] = pc, pc = reg[r1]	
beqz r1 imm		pc = (reg[r1] == 0) ? pc+imm+4 : pc+4	
bnez r1 imm		pc = (reg[r1] != 0) ? pc+imm+4 : pc+4	
halt		stop the simulation	

Figure 1: Instructions from the RISC-V ISA supported by the simulator.

You will be simulating the RISC-V-like ISA from Hennessy & Patterson, with some key differences. First, instead of a 64-bit architecture you will implement a **32-bit** architecture only. In other words, all registers and data paths are 32 bits wide, and all instructions will operate on 32-bit operands. Within this 32-bit data size, you will support both signed and unsigned fixed point data types as well as single-precision floating point data types. Second, you will only support a subset of the RISC-V ISA. A summary of these instructions is given in **Figure 1**, including the assembly notation and the C-expression for each instruction. In addition to the arithmetic/logical instructions listed at the top of Figure 1, you will also support **unsigned** versions of `add, addi, sub, subi, slt, slti, sgt,` and `sgti`. (The mnemonic for these unsigned instruction versions will include the letter "u", for example `addu` and `addui`). Finally, although all immediate values in RISC-V's branch and jump instructions are left-shifted by 2 (see Hennessy & Patterson), your control instructions **SHOULD NOT** perform the left shift.

Notice all the instructions in Figure 1 except for one exist in the normal RISC-V ISA. The instruction we've added is `halt`. As its name implies, when your simulator executes a `halt` instruction, it should terminate the simulation. For more information about RISC-V, consult Section A.9 of Hennessy & Patterson.

3 asm: An Assembler for the RISC-V ISA

We have provided an assembler, `asm.c`, that supports all the instructions in Figure 1. (Actually, it supports even more instructions, but you can ignore any instruction that hasn't been described in Section 2). The `asm.c` file is fully functional, and you will **not** need to make any modifications to this file. Simply use the `Makefile` to make the binary `asm` from the `asm.c` source file.

The format for assembly programs is very simple. A valid assembly program is an ASCII file in which each line of the file represents a single instruction, or a data constant. The format for a single line of assembly code is:

```
label<tab>instruction<tab>field0<tab>field1<tab>field2<tab>comments
```

Here's a simple example of a full assembly program:

label	**instr**	**field0**	**field1**	**field2**	**comments**
	`addi`	`R1`	`R0`	`#5`	`load reg[1] with 5`
`start`	`subi`	`R1`	`R1`	`#-1`	`decrement reg[1]`
	`l.s`	`F0`	`R0`	`var1`	`loads fpreg[0] with value stored in var1`
	`add.s`	`F2`	`F0`	`F0`	`double fpreg[0]`
	`s.s`	`F2`	`R0`	`var1`	`put fpreg[2] back`
	`beqz`	`R1`	`done`		`goto done when reg[1]==0`
	`j`	`start`			
`done`	`halt`				
`var1`	`.df`	`32.0`			`Declare a variable, initialized to 32`

The leftmost field on a line is the label field which indicates a symbolic address. Valid labels contain a maximum of 6 characters and can consist of letters and numbers. The label is optional (the tab following the label field is not). After the optional label is a tab. Then follows the instruction field, where the instruction can be any one of the assembly-language mnemonics described in Section 2. After another tab comes three fields, also separated by tabs. Each field represents either a register identifier or an immediate value. Register identifiers begin with the letter "R" or "F" as in "R0" or "F0" for integer and floating point registers, respectively. Immediates are specified as a constant decimal value preceded by a "#" sign as in "#5". For the memory and control instructions, the `imm` field can either be a decimal value preceded by the "#" sign, or a label can be used as in "var1". After the last field is another tab, then any comments. The comments end at the end of the line.

In addition to instructions, lines of assembly code can also include directives for the assembler. A directive tells the assembler to put a constant value into the place where an instruction would normally be stored. We support two directives, "`.dw`" and "`.df`", for storing integer and floating point constants, respectively.

4 Makefile

To build the entire project, run `make` in your terminal.

Note: This works in **Linux/OSX** environment, but not in Windows (natively). Please SSH to your GLUE account if you have trouble executing '`make`' commands. Once the project compiles, use `pipesim` command, as described in Section 5.2.1.

5 Pipeline Simulator

Your assignment is to create a cycle-accurate simulator of an in-order RISC-V processor that supports 32-bit integer and floating point operations. We have provided some code to get you started in `fu.c` and `main.c`. This code will make your life easier since you won't have to write the whole simulator from scratch, and it will enforce some coding disciplines that ensure you actually simulate the details of the processor pipeline. The following sections describe both the code we provide and the code you will write in greater detail.

5.1 Pipeline Structure

We have provided pipeline register data structures specified in the file, `pipeline.h`. To enforce proper pipeline simulation, you should use these structures in your simulator. Here is the main processor structure, `state_t`:

```
typedef struct _state_t {
    /* memory */
    unsigned char mem[MAXMEMORY];
    /* register files */
    rf_int_t rf_int;
    rf_fp_t rf_fp;
    /* pipeline registers */
    unsigned long pc;
    if_id_t if_id;
    fu_int_t *fu_int_list;
    fu_fp_t *fu_add_list;
    fu_fp_t *fu_mult_list;
    fu_fp_t *fu_div_list;
    wb_t int_wb;
    wb_t fp_wb;
    int fetch_lock;
} state_t;
```

Both the integer and floating point pipelines pass through four types of pipeline stages: fetch, decode, execute, and writeback. The "pc" field and "if_id" structure contain the pipeline registers for the fetch and decode stages, respectively. These pipeline registers are shared by both integer and floating point pipelines, so only one instruction (integer or floating point) can be fetched and decoded per cycle. The "fu_int_list" pointer points to the pipeline registers for the integer execute stage(s), and the "fu_add/mult/div_list" pointers point to the pipeline registers for the floating point add, multiply, and divide execute stages. (We will discuss these pipeline registers further in Section 5.2). And finally, the "int_wb" and "fp_wb" structures contain the pipeline registers for the writeback stages of the integer and floating point pipelines, respectively. At most one integer and one floating point instruction (i.e., two instructions total) can writeback per cycle.

In addition to the pipeline registers, the processor state also contains memory, "mem", and two register files - one for the integer pipeline, "rf_int", and another for the floating point pipeline, "rf_fp". Each register file contains an array of registers. Notice the integer registers are of type "int_t", as defined in `fu.h`, which allows for a signed or unsigned interpretation of each register's contents through a "union" construct. This permits both signed and unsigned arithmetic operations on the integer register data.

5.2 Simulator Loop, Functional Units, and Output Generator

Included in the download files are three modules that contain code we provide: main.c, fu.c, and output.c. This section describes the functionality implemented in these modules. Note, while the code we provide is fully functional, you are welcome to modify most of it. In fact, to complete the assignment, you will need to make at least a few modifications to our code in order to integrate it with your code. The only code you **cannot** modify is the **output** generator code which we will discuss in Section 5.2.3.

5.2.1 main.c

`main.c` contains the main simulator loop. This loop calls the `fetch`, `decode`, `execute`, and `writeback` functions you will provide to simulate each type of pipeline stage (see Section 5.3). One iteration through the main simulator loop advances each pipeline stage by 1 cycle, and corresponds to 1 simulated processor cycle. Notice the pipeline stages are called in **reverse** order. This ensures each instruction propagates only 1 pipeline stage per cycle. (Calling the stages in forward order would allow a new instruction to propagate all the way down the pipeline in 1 iteration). `main.c` also contains a function to parse the command line arguments. The simulator expects **two** arguments: the assembled program file preceded by the flag "-b", and the functional unit configuration file preceded by the flag "-o".

5.2.2 fu.c

`fu.c` contains a significant amount of code that implements the integer/floating point functional units and the simulation of their execute stages. The functional unit implementation consists of 4 types of routines. First, `state_create` reads a functional unit configuration (FU config) file, and creates and intializes the functional units accordingly. We have provided an FU config file, called `io_pipe.fu`. Each line in the FU config file specifies a functional unit to be simulated, providing the name and type of functional unit. There are four valid functional unit types: INT, ADD, MULT, and DIV (each type has a corresponding linked list in the pipeline register data structure described in Section 5.1). ADD executes the `add.s` and `sub.s` instructions, MULT executes the `mult.s` instruction, and DIV executes the `div.s` instruction. All remaining instructions are of type integer and execute in INT-type functional units. Each line in the FU config file also specifies a list of numbers, one for each pipeline stage that specifies the number of cycles in that stage. For example, the floating point multiplier in `io_pipe.fu` contains 4 1-cycle stages.

Second, an instruction can be "issued" into one of the functional units via the `issue_fu_int` or `issue_fu_fp` routines. The former is used for integer instructions, and the latter for floating point instructions. These routines return "0" if the instruction is successfully issued; otherwise, they return "-1" indicating no functional units of that type are free (i.e., a structural hazard). Third, the functional units can be advanced one cycle by calling the `advance_fu_int` and `advance_fu_fp` routines. These routines move previously issued instructions through the functional units' execute stages. When execution of an instruction completes, the instruction is placed into the appropriate writeback stage pipeline register, `int_wb` or `fp_wb`, depending on the instruction's type (integer or floating point, respectively). Finally, the `fu_int_done` and `fu_fp_done` routines test for instructions still in-flight in functional units of type integer or floating point, respectively, and return FALSE when one or more instructions are still in flight or TRUE when no instructions are in flight.

In addition to implementing the functional units, `fu.c` also provides a routine for decoding instructions, `decode_instr`, which takes two arguments, an instruction and a pointer to a flag (`use_imm`), and returns a pointer into the instruction table defined at the top of `fu.c`. Each instruction table entry provides 4 pieces of information: name, group number, operation number, and data type. The group number specifies a group of instructions from Figure 1 to which the instruction belongs. `FU_GROUP_INT` contains the instructions in the "Integer Arithmetic / Logic" portion of Figure 1. Within this group, the `use_imm` flag specifies whether the instruction uses an immediate value (i.e., the right half of the "Integer Arithmetic / Logic" portion of Figure 1), or whether the instruction does not use an immediate value (i.e., the left half of the "Integer Arithmetic / Logic" portion). `FU_GROUP_ADD`, `FU_GROUP_MULT`, and `FU_GROUP_DIV` contain the instructions in the "Floating Point Arithmetic" portion of Figure 1 (note that both `add.s` and `sub.s` belong to `FU_GROUP_ADD`). `FU_GROUP_MEM` contains the instructions in the "Memory" portion of Figure 1, and `FU_GROUP_BRANCH` contains the instructions in the "Control" portion of Figure 1, excluding the `halt` instruction which has its own group, `FU_GROUP_HALT`. (We also create a special group for NOP instructions; see Section 5.3.5). Within a group, the operation number specifies the actual decoded instruction. Finally, the data type specifies either integer (`DATA_TYPE_W`) or floating point (`DATA_TYPE_F`). We have provided the routine, `perform_operation`, which shows an example use of `decode_instr`. In `perform_operation`, the `add` instruction is fully decoded and implemented.

Notice `decode_instr` uses two macros, `FIELD_OPCODE` and `FIELD_FUNC`, to extract opcode fields from instructions. These macros are defined in `fu.h` along with others that facilitate extraction of register specifiers (`FIELD_R1/R2/R3`), extraction and sign extension of 16-bit signed and unsigned immediates (`FIELD_IMM` and `FIELD_IMMU`, respectively) and 26-bit signed offsets (`FIELD_OFFSET`).

5.2.3 output.c

`output.c` contains the output generator routine, `print_state`, which is called once every iteration of the main simulation loop. This routine (along with other routines in `output.c`) dumps the state of memory, the register files, and the pipeline registers. Your final submitted simulator should use these routines in their **unmodified** form; otherwise, we will not be able to grade your simulator. Of course, you can make modifications to these routines during debugging, but be sure to remove your modifications before submitting your simulator.

5.3 Your code

Your job is to write the routines that simulate the pipeline stages in `pipeline.c`: `fetch, decode, execute,` and `writeback`. You will also need to modify some of the code we provide to properly integrate your code.

`fetch` should go to memory and fetch the current instruction specified by the program counter, and place the fetched instruction into the `if_id` pipeline register. `decode` should examine the instruction in the `if_id` pipeline register, decode the instruction, and determine whether it can issue into the functional units. As long as the instruction has no data, control, or structural hazards (described below), it should issue; otherwise, it should **stall**. `execute` should advance all the functional units by one cycle. And `writeback` should write the result of any instruction in the `int_wb` or `fp_wb` pipeline registers into the register file. The following describes in detail some of the issues your code must deal with.

5.3.1 Dealing with Data Hazards

In this simulator, you will **stall** all data hazards. (Although stalling leads to poor performance, the goal for this project is to implement simple mechanisms only). Data hazards should be detected and stalled in the decode stage.

There are two types of data hazards you need to detect and stall: **RAW** and **WAW**. (WAR hazards cannot occur since your pipeline does not perform late reads). A RAW hazard occurs when the instruction in decode reads a register written by an earlier instruction still executing in a functional unit. If a RAW hazard is detected, the instruction in decode must stall until the earlier instruction reaches its writeback stage. We will assume register writes happen on the first half of the clock cycle while register reads happen on the second half of the clock cycle, so when the earlier instruction reaches writeback, the dependent instruction in decode can issue on the same cycle. (In fact, since the pipeline is simulated in reverse order, writeback is performed before decode within the same simulated cycle).

A WAW hazard occurs when the instruction in decode writes a register written by an earlier instruction still executing in a functional unit, and the number of execute cycles remaining for the earlier instruction is larger than the latency of the instruction in decode. (If the instruction in decode were to issue immediately, the writes would reorder). If a WAW hazard is detected, the instruction in decode must stall until the number of execute cycles remaining for the earlier instruction is **equal** to the latency of the instruction in decode.

Note: Your simulator should stall WAW hazards for the **minimum** number of cycles only. (For example, you should not stall until the earlier instruction reaches its writeback stage). Furthermore, you can assume the integer pipeline is a fixed pipeline. Since WAW hazards can only occur in variable pipelines, WAW hazards will never occur through integer registers, only through floating point registers.

5.3.2 Dealing with Control Hazards

In this simulator, you will **stall** all control hazards. When a control instruction enters the decode stage and issues (after resolving any data and structural hazards), you should immediately set the `fetch_lock_flag` in the `state_t` structure to TRUE. This prevents fetch and decode on subsequent cycles (see the simulator loop in `main.c`), though you should allow fetch of the sequential successor immediately following the control instruction. When the control instruction enters writeback, you should set `fetch_lock` back to FALSE. So, the processor stalls for the entire time the control instruction is in-flight. If the control instruction is not taken, you should not update the fetch stage. However, if the control instruction is taken, you must set the program counter to the target address and squash the instruction in `if_id` (i.e., the sequential successor immediately following the control instruction that was incorrectly fetched).

5.3.3 Dealing with Structural Hazards

In addition to data and control hazards, there are **two** types of structural hazards you must **stall**. Once again, detection and stall should occur in decode. First, you should stall if there is no functional unit available for the instruction in decode. This occurs whenever you try to issue an instruction, but `issue_fu_int` or `issue_fu_fp` return "-1". Second, you must ensure that at most 1 floating point and 1 integer instruction `writeback` on the same cycle. A structural hazard of this kind occurs when the number of execute cycles remaining in *any* earlier instruction still executing in a functional unit is one less than the latency of the instruction in decode, and both instructions write to the same register file. This kind of structural hazard can only occur in a variable pipeline (similar to the WAW hazard). As a result, structural hazards in the writeback stage need only be checked for the floating point register file.

5.3.4 Executing Instructions

The functional unit models described in Section 5.2.2 only simulate the timing of instructions; you must implement the function performed by each instruction separately. We have started this for you in `perform_operation`, but you must provide the majority of the code. One question is when should `perform_operation` occur? Since the code in `output.c` only prints the instruction field of each pipeline register, you can call perform operation at any time. One possibility is to call it in the decode stage upon instruction issue, and modify the functional units to carry the result of the instruction all the way to writeback. Another possibility is to call `perform_operation` within the functional units themselves, perhaps in the first execute stage. Yet another possibility is to carry the operands for the instruction through the functional units and call `perform_operation` during writeback. It's up to you.

5.3.5 NOP Instruction

The NOP instruction is needed to initialize the pipeline register data structures, and to replace or "squash" the instruction in the `if_id` pipeline register following a taken branch. You should use "0" (i.e., an instruction with all 0s) as a NOP instruction. The instruction table in `fu.c` places the NOP instruction into a special group, `FU_GROUP_NONE`. Any instruction with this group number should be treated as a NOP. Since NOPs do nothing, there is no need to execute them. Hence, you can throw away any NOP instruction encountered in the decode stage. This saves having to execute and writeback NOPs, though you will need to fetch and decode them.

5.3.6 Halting

When your simulator encounters the `halt` instruction, it should end the simulation. Notice, however, any earlier instruction preceding the halt should be allowed to complete. So, once you encounter a `halt` instruction in the decode stage, you can stop fetching and decoding; however, you should keep calling `execute` and `writeback` until all in-flight instructions drain from the pipeline and writeback. (Use the `fu_int_done` and `fu_fp_done` routines to detect when all instructions have drained from the functional units).

6 Deliverables

Please submit a compressed file that contains all the following files to ELMS.

- **Source code**: Your code should contain comments on what a function or a block does. Also, please make sure that it is compatible with Glue.
- **Output** of your simulator on each benchmark: Please name your files `simple.out`, `vect.out`, `newton.out`, and `cos.out` respectively.
- A **report** in PDF that illustrates what you have done: Your report does not have to be elaborate, but it should cover everything you did.

7 Grading

7.1 Baseline (100%)

- Report (10%)
- Benchmarks (90%)
  - `simple.s` (15%)
  - Others (25% for each)

Your assignment will be graded by comparing the output of your simulator on each of the four benchmarks against the output files `simple.io.out, vect.io.out, newton.io.out,` and `cos.io.out`. As a result, your goal should be to design your simulator such that you get the **exact** same output as the output files we have provided.

7.2 Bonus (10%)

Beyond the baseline, if you could perform certain improvements/optimizations to your simulator, you will be given some extra points (up to 10%) for this project.

Example improvements/optimizations you could perform (but not limited to):

1. Make your simulator support **data forwarding** and evaluate its effect.
2. Make your simulator support **branch prediction** and evaluate its effect.
3. Make your simulator support **out-of-order execution** and evaluate its effect.
4. **Rewrite** (e.g., loop unrolling) the provided assembly programs and evaluate its effect.

Note: Please make sure to highlight what you have done for bonus in your report.

7.3 Academic Honesty

Do not allow any other student to see any of your code. You may however discuss the assignment in general terms, with the other students. If copying or excessive collaboration is detected in your submissions, the matter will be referred to the Student Honor Counsel.