# CMPE344 Final Project III Report

**Ömer Faruk Deniz, Özdeniz Dolu**

2016400003, 2015400009

Computer Engineering Department
Boğaziçi University

February 17, 2021

# Contents

# 1 Introduction

## 1.1 Problem Description

In this project, we are expected to create a RISC-V pipelined datapath simulator on a high level programming language. Simulation is expected to implement hazard control and forwarding units as discussed throughout the lectures and in the course textbook. Simulation is also expected to keep data on CPI, total clock cycles, total number of stalls, stalls caused by instructions and prepare an output report on the execution using that data. Figure 1.1 is the diagram for the pipelined datapath structure taken from the course textbook. The simulator is also expected to simulate the instructions and, or, sub, add, beq, sd and ld.

GOOGLE DRIVE LINK FOR THE CODE

## 1.2 Our Approach

We have decided to write this simulator in the programming language Python. We aimed create a simulator structure that is strongly correlated with both the data and logical structure of the datapath. Simulator program, via a console argument, takes a plain text file that contains assembly code and some data initialization and fill the code in its virtual memory. Our simulator can run add. sub, and, or, beq, ld and sd instructions. Our simulator has variables for all the registers of RISC-V ISA and also for registers like ID/EX etc. We used the standard data structures of Python such as dictionaries and lists to simulate registers and memory. And to simulate the execution, we have defined methods for each phase of the datapath that are continuously executed one by one until the program counter reaches the end of the code. Those methods(for example a method for simulating the ID phase), during their execution, update the related registers according to their function. During the execution, simulator keeps all the related statistics to create a final report.

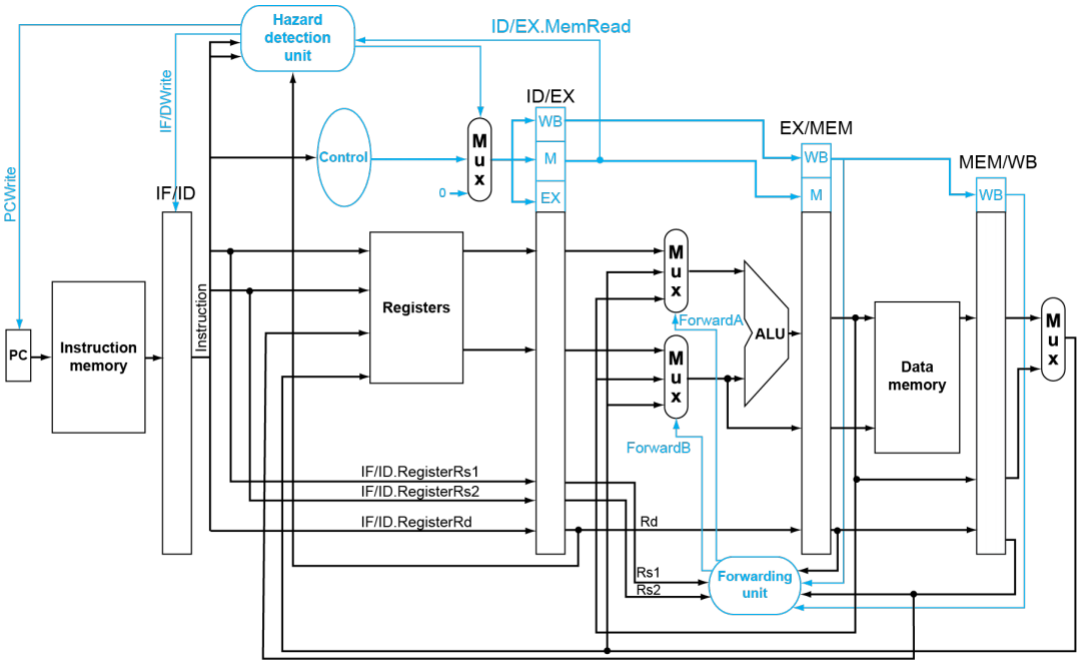


Figure 1.1: Pipelined datapath diagram taken from course textbook

# 2 Implementation and Modules

## 2.1 Input Format

Simulator expects a path to a plain text file given as a first argument when running the code(see Chapter 3 for more details on running the simulator). Following segment is an example program.txt file.

```
1  x1=7
2  x2=18
3  m[5]=5
4  ----
5  add x5,x1,x5
6  sub x1,x3,x2
7  ld x1,0(x5)
8  beq x1, x3, equal
9  equal:
10 add x6, x7, x8
```

In our input format, lines that come before the — separator line(line 4 in the above segment) are used for initializing the registers and memory of the simulator. This allows us to start the simulation in a more flexible set of states as by default all the memory and the registers are initialized at 0. Line 3 is a format for initalizing the memory. (addresses start at 0 and we are assuming double word addressing.)

The lines that come after the — separator line is the assembly code. Simulator can only process the instructions and, or, add, sub, beq, ld and sd. We are checking for excess whitespaces or similar formatting problems but still it is best to comply with the format shown in above segment.

## 2.2 Modules

### 2.2.1 Simulator Class (simulator.py)

**Constructor**

We have created a class for the simulation that keeps all the relevant data in the memory. Following code segment contains its constructor method. As can be seen in the first lines of the constructor method, registers and memory are kept as Python lists of corresponding sizes.

At the lines 24 to 30, one can see the initialization of the registers IF_ID, ID_EX, EX_MEM, MEM_WB. These registers are kept as Python dictionaries and they are the main channel of

communication between the different stages of the pipelined execution. At every stage, the corresponding methods for that stage change the values that are written in these variables. They are intended to simulate the datapath circuitry in a one to one correspondance with the Figure 1.1

This class also contains many more helper and/or main methods that we have omitted in this section due to inconvenience. They can be fully accessed through the code files.

```
1  class Simulator:
2      def __init__(self, program_path):
3          self.REGISTERS = [0] * 32 # REGISTER FILE
4          self.MEMORY = [0] * 1000 # MEMORY
5          self.PC = 0 # PROGRAM COUNTER
6          self.CLOCK = 1 # CLOCK
7          self.WORD_LEN = 4
8          self.FLUSH = False
9
10         self.FINISHED_INSTRUCTION_COUNT = 0
11         init_lines, self.INSTRUCTION_NAMES, self.INSTRUCTION_MEMORY =
               get_program(program_path) # read program
12         self.parse_inits(init_lines) # parse register and memory
               init commands
13
14         self.NOP_INSTRUCTION = get_nop_instruction() # get nop
               instruction which has all fields 0
15         self.NOP_CONTROL = get_nop_control() # all fields are 0
16         self.ALL_STAGES_NOP = True # used to check if all stages are
               having NOP instructions
17
18         self.stalls = {} # dictinary that holds instruction name and
               their stall counts
19         self.STALL_OCCURRED = False # flag to check stall occurred in
               the current clock
20
21         self.INSTRUCTIONS_IN_PIPELINE = ['NOP'] * 5 # name of
               instructions in the pipeline, used only for reporting
               purposes
22
23         # fields of the stage registers and their initial values for
               NOP instructions
24         self.IF_ID = {"PC": 0, "instruction": self.NOP_INSTRUCTION}
25         self.ID_EX = {"PC": 0, "rs1_data": 0, "rs2_data": 0,
26             "imm_gen_offset": 0, "funct_for_alu_control": "0000", "rd
                   ": 0,
27             "control": self.NOP_CONTROL, "rs1": None, "rs2": None}
28         self.EX_MEM = {"PC_plus_OFFSET": 0, "ALU_zero": 0, "
               ALU_result": 0, "rs1_data": 0, "rd": None,
```

```
29                "control": self.NOP_CONTROL}
30            self.MEM_WB = {"read_from_memory": 0, "ALU_result": 0, "rd":
                  None, "control": self.NOP_CONTROL}
```

### run() method

This is the main method that runs the simulation in a loop. Following is the code segment for the definition of this method from simulator.py. As can be seen in the line 6, while loop continues to execute the simulation until the PC reaches to end of the program or all of the stages have a NOP instructions in them. Former condition checks if the simulation is finished with all the instructions and the latter condition makes sure that no instruction is left in anywhere in the pipeline.

In the lines 9 to 13, the simulation is happening as all the stages are run one by one, one after each other. All the stage outputs are in the format of their corresponding registers.

The condition in the line 24 is to make sure the pipeline is flushed if a branching has occurred as if it is the case that branch condition is met, FLUSH variable will be set to True. We haven't implemented a branch prediction unit, therefore we need to flush every time.

```
1     def run(self):
2         print(f"------STATUS AT THE BEGINNING------")
3         print(*self.INSTRUCTIONS_IN_PIPELINE, sep=' | ')
4         self.print_status()
5         # while PC is valid and not all STAGES are filled with NOPs
6         while(self.PC < len(self.INSTRUCTION_MEMORY) or not self.
              ALL_STAGES_NOP):
7             PC_running = self.PC
8             # run each stage separately before updating the stage
                  registers
9             self.run_WB()
10            output_for_MEM_WB = self.run_MEM()
11            output_for_EX_MEM = self.run_EX()
12            output_for_ID_EX = self.run_ID()
13            output_for_IF_ID = self.run_IF()
14
15            # update the stage registers
16            if not self.STALL_OCCURRED:
17                self.IF_ID = output_for_IF_ID
18            else: # self .IF_ID should be preserved if stall is
                  occurred and instruction is fetched
19                self.STALL_OCCURRED = False
20            self.ID_EX = output_for_ID_EX
21            self.EX_MEM = output_for_EX_MEM
22            self.MEM_WB = output_for_MEM_WB
```

```
            # fill stage registers with NOP
            if self.FLUSH:
                self.PC = self.PC_plus_OFFSET
                self.IF_ID['instruction'] = self.NOP_INSTRUCTION
                self.ID_EX['control'] = self.NOP_CONTROL
                self.EX_MEM['control'] = self.NOP_CONTROL
                self.FLUSH = False


            print(f"-----STATUS AT THE END OF CLOCK = {self.CLOCK}-----")
            print(*self.INSTRUCTIONS_IN_PIPELINE, sep=' | ', end="")
            print(f" is run at PC = {PC_running}")
            self.print_status()
            #break
            self.CLOCK += 1
            # if all registers are full of control values of zero, namely NOPs update the flag
            if ((self.IF_ID['instruction'] == self.NOP_INSTRUCTION
                and self.ID_EX['control'] == self.EX_MEM['control'])
                and
                 (self.EX_MEM['control'] == self.MEM_WB['control'] and
                     self.MEM_WB['control'] == self.NOP_CONTROL)):
                self.ALL_STAGES_NOP = True
            else:
                self.ALL_STAGES_NOP = False
        self.CLOCK -= 1
        self.print_final_report()
```

### run_IF() method

This is the method that simulates the IF stage. Following is a code segment that contains its definition from the simulator.py file. The method simply fetches a new instruction from the instruction memory if the PC hasn't reached the end of the file. Members of the INSTRUCTION_MEMORY contains the instructions as dictionaries with field such as rs1, rs2, rd etc. The return value of this function has the same structure as the IF_ID register as it is going to be used to update that register.

```
    def run_IF(self):
        # if not all instructions are entered the pipeline
        if self.FLUSH:
            self.INSTRUCTIONS_IN_PIPELINE = ['NOP'] + self.INSTRUCTIONS_IN_PIPELINE[:-1]
            return {'PC':self.PC, 'instruction': self.NOP_INSTRUCTION}
        if self.PC < len(self.INSTRUCTION_MEMORY) and not self.STALL_OCCURRED:
```

```
7               new_instruction = self.INSTRUCTION_MEMORY[self.PC]
8               self.INSTRUCTIONS_IN_PIPELINE = [self.INSTRUCTION_NAMES[
                    self.PC // self.WORD_LEN]] + self.
                    INSTRUCTIONS_IN_PIPELINE[:-1]
9               PC = self.PC
10              self.PC += self.WORD_LEN
11              return {'PC':PC, 'instruction': new_instruction, 'rs1':
                    new_instruction['rs1'], 'rs2': new_instruction['rs2']}
12          else: # add  NOP to the pipeline
13              if not self.STALL_OCCURRED:
14                  self.INSTRUCTIONS_IN_PIPELINE = ['NOP'] + self.
                        INSTRUCTIONS_IN_PIPELINE[:-1]
15              return {'PC':self.PC,'instruction': self.NOP_INSTRUCTION}
```

**run_ID() method**

This is the method that simulates the ID stage. Following is a code segment that contains its definition from the simulator.py file. The method starts by accessing the instruction via the IF_ID register. It uses some helper functions from the instruction.py file to get the related control values of that instruction. A hazard control unit for load use hazard is also implemented in this method as it can be seen at the line 18 and onward. This hazard can only be solved by adding a stall to the pipeline. The output format of this method has the same structure as ID_EX register as it is going to be used to update that register.

```
1   def run_ID(self):
2       # read from stage registers
3       instruction = self.IF_ID['instruction']
4       PC = self.IF_ID['PC']
5
6       # operate
7       control = get_control_values(instruction) # calculate control
8       imm_gen_offset = instruction['immed'] # sign extend the
            offset
9       rs1 = instruction['rs1']
10      rs2 = instruction['rs2']
11      rs1_data = self.read_register(rs1)
12      rs2_data = self.read_register(rs2)
13      # will be used for setting ALU control in EX
14      funct_for_alu_control = get_funct_for_alu_control(instruction
            )
15      rd = instruction['rd']
16
17      # check for hazard
18      if self.ID_EX['control']['MemRead'] and ((self.ID_EX['rd'] ==
            self.IF_ID['instruction']['rs1']) or (self.ID_EX['rd'] ==
            self.IF_ID['instruction']['rs2'])):
```

```
19              self.STALL_OCCURRED = True
20              self.INSTRUCTIONS_IN_PIPELINE = [self.
                    INSTRUCTIONS_IN_PIPELINE[0]] + ['NOP'] + self.
                    INSTRUCTIONS_IN_PIPELINE[1:-1]
21              stall_instruction = self.INSTRUCTIONS_IN_PIPELINE[2] #
                    instruction in the ex stage causes the hazard
22              # if already defined
23              if stall_instruction in self.stalls:
24                  self.stalls[stall_instruction] += 1
25              else:
26                  self.stalls[stall_instruction] = 1
27              # return the output to be written to ID_EX
28              return {"PC": 0, "rs1_data": 0, "rs2_data": 0,
29              "imm_gen_offset": 0, "funct_for_alu_control": "0000", "rd
                    ": 0,
30              "control": self.NOP_CONTROL, "rs1": None, "rs2": None}
31          else:
32              # return the output to be written to ID_EX
33              return {"PC": PC, "rs1_data": rs1_data, "rs2_data":
                    rs2_data,
34              "imm_gen_offset": imm_gen_offset, "funct_for_alu_control"
                    : funct_for_alu_control, "rd": rd,
35              "control": control, "rs1": rs1, "rs2": rs2}
```

**run_EX() method**

This is the method that simulates the EX stage. Following is a code segment that contains its definition from the simulator.py file. The method starts by accessing the relevant data via the ID_EX register. As can be seen starting from the line 16, method contains a forwarding unit to address EX and MEM hazards. After deciding on the hazard type by logical operations, method updates the corresponding forwarding unit bits. At the line 64 and onward it executes the EX stage by using the values that came from ID_EX register or forwarded values if there are any. The output format of this method has the same structure as EX_MEM register as it is going to be used to update that register.

```
1       def run_EX(self):
2           # read from stage registers
3           control = self.ID_EX['control']
4           PC = self.ID_EX['PC']
5           rs1_data = self.ID_EX['rs1_data']
6           rs2_data = self.ID_EX['rs2_data']
7           imm_gen_offset = self.ID_EX['imm_gen_offset']
8           funct_for_alu_control = self.ID_EX['funct_for_alu_control']
9           rs1 = self.ID_EX['rs1']
10          rs2 = self.ID_EX['rs2']
11          rd = self.ID_EX['rd']
```

```
12
13          # operate
14          # forwarding unit
15          # EX Hazard: pg 300 in the book
16          ForwardA = "00"
17          ForwardB = "00"
18          if (self.EX_MEM['control']['RegWrite'] and (self.EX_MEM['rd']
                != 0) and (self.EX_MEM['rd'] == self.ID_EX['rs1'])):
19              ForwardA = "10"
20          if (self.EX_MEM['control']['RegWrite'] and (self.EX_MEM['rd']
                != 0) and (self.EX_MEM['rd'] == self.ID_EX['rs2'])):
21              ForwardB = "10"
22
23          # MEM Hazard: pg 301 in the book
24          if (self.MEM_WB['control']['RegWrite'] and (self.MEM_WB['rd']
                != 0) and not(self.EX_MEM['control']['RegWrite']
25              and (self.EX_MEM['rd'] != 0) and (self.EX_MEM['rd'] ==
                    self.ID_EX['rs1'])) and (self.MEM_WB['rd'] == self.
                    ID_EX['rs1'])):
26              ForwardA = "01"
27          if (self.MEM_WB['control']['RegWrite'] and (self.MEM_WB['rd']
                != 0) and not(self.EX_MEM['control']['RegWrite']
28              and (self.EX_MEM['rd'] != 0) and (self.EX_MEM['rd'] ==
                    self.ID_EX['rs2'])) and (self.MEM_WB['rd'] == self.
                    ID_EX['rs2'])):
29              ForwardB = "01"
30
31          ALU_control = get_alu_control(str(control['ALUOp1'])+str(
                control['ALUOp0']), funct_for_alu_control)
32          PC_plus_OFFSET = PC + 2 * imm_gen_offset # calculate PC
                offset
33          ALU_result = None
34          param1 = 0
35          param2 = 0
36          if ForwardA == "00":
37              param1 = rs1_data
38          elif ForwardA == "10":
39              rs1_data = self.EX_MEM['ALU_result']
40              param1 = rs1_data
41          elif ForwardA == "01":
42              if self.MEM_WB['control']['RegWrite']:
43                  read_from_memory = self.MEM_WB['read_from_memory']
44                  ALU_result = self.MEM_WB['ALU_result']
45                  if self.MEM_WB['control']['MemToReg']: # ld: write
                        the value at rs2+offset to rs1, else do not write
                        to reg
```

```
46                      param1 = read_from_memory
47                  else: # r-type: write the ALU_result to rd
48                      param1 = ALU_result
49
50          if ForwardB == "00":
51              param2 = rs2_data
52          elif ForwardB == "10":
53              rs2_data = self.EX_MEM['ALU_result']
54              param2 = rs2_data
55          elif ForwardB == "01":
56              if self.MEM_WB['control']['RegWrite']:
57                  read_from_memory = self.MEM_WB['read_from_memory']
58                  ALU_result = self.MEM_WB['ALU_result']
59                  if self.MEM_WB['control']['MemToReg']: # ld: write
                        the value at rs2+offset to rs1, else do not write
                        to reg
60                      param2 = read_from_memory
61                  else: # r-type: write the ALU_result to rd
62                      param2 = ALU_result
63
64          if control['ALUSrc'] == 0: # r-format or beq
65              ALU_result = perform_ALU_operation(ALU_control, param1,
                    param2)
66          elif control['ALUSrc'] == 1: # ld, sd
67              if control['MemWrite'] == 1: # sd: rs2_data+offset
68                  ALU_result = perform_ALU_operation(ALU_control,
                        param2, imm_gen_offset)
69              else: # ld: rs1_data+offset
70                  ALU_result = perform_ALU_operation(ALU_control,
                        param1, imm_gen_offset)
71          ALU_zero = ALU_result == 0
72
73          # return the output to be written to EX_MEM
74          return {"PC_plus_OFFSET": PC_plus_OFFSET, "ALU_zero":
                ALU_zero,
75              "ALU_result": ALU_result, "rs1_data": rs1_data, "rd": rd,
                    "control": control}
```

**run_MEM() method**

This is the method that simulates the MEM stage. Following is a code segment that contains its definition from the simulator.py file. The method starts by accessing the relevant data via the EX_MEM register. It then proceeds to execute the MEM stage by checking if the instruction is a branch with branch condition met, sd with MemWrite set or ld with MemRead set. It creates a flush in the case of branch condition met and otherwise, it executes the corresponding memory operations and creates an output. The output format of this method has the same structure as

MEM_WB register as it is going to be used to update that register.

```
    def run_MEM(self):
        # read from stage registers
        control = self.EX_MEM['control'] # read control from previous stage
        PC_plus_OFFSET = self.EX_MEM['PC_plus_OFFSET']
        ALU_zero = self.EX_MEM['ALU_zero']
        ALU_result = self.EX_MEM['ALU_result']
        rs1_data = self.EX_MEM['rs1_data']
        rd = self.EX_MEM['rd']

        # operate
        if control['Branch'] and ALU_zero: # if a branch instruction and rs1_data-rs2_data == 0
            self.PC_plus_OFFSET = PC_plus_OFFSET
            # flush instructions in the IF, ID, EX when MEM is executing
            self.FLUSH = True
            self.INSTRUCTIONS_IN_PIPELINE = ['NOP', 'NOP'] + self.INSTRUCTIONS_IN_PIPELINE[2:]
            stall_instruction = self.INSTRUCTIONS_IN_PIPELINE[2]
            if stall_instruction in self.stalls:
                self.stalls[stall_instruction] += 3 # since flush adds two stalls to the pipeline
            else:
                self.stalls[stall_instruction] = 3
        if control['MemWrite']: # sd, will write to memory
            self.MEMORY[ALU_result] = rs1_data

        read_from_memory = None
        if control['MemRead']: # ld, will write to register file
            read_from_memory = self.MEMORY[ALU_result]

        # return the output to be written to MEM_WB
        return {"read_from_memory": read_from_memory, "ALU_result": ALU_result, "rd": rd, "control": control}
```

### run_WB() method

This is the method that simulates the WB stage. Following is a code segment that contains its definition from the simulator.py file. The method starts by accessing the relevant data via the MEM_WB register. It then proceeds to execute WB stage by checking the values of RegWrite and MemToReg values to decide if the instruction is ld or an R-type instruction and writes to corresponding registers accordingly.

```
1         def run_WB(self):
2         # read from stage registers
3         control = self.MEM_WB['control'] # read control from previous
              stage
4         read_from_memory = self.MEM_WB['read_from_memory']
5         ALU_result = self.MEM_WB['ALU_result']
6         rd = self.MEM_WB['rd']
7         # operate
8         if control['RegWrite']:
9             if control['MemToReg']: # ld: write the value at rs2+
                  offset to rs1, else do not write to reg
10                self.write_to_register(rd, read_from_memory)
11            else: # r-type: write the ALU_result to rd
12                self.write_to_register(rd, ALU_result)
13        if control != self.NOP_CONTROL:
14            self.FINISHED_INSTRUCTION_COUNT += 1
```

### 2.2.2 Helper Functions (simulator.py)

Simulator class contains many helper functions that are related to memory and register operations, stage executions and output printing. We omitted the code segments because they are long and redundant in many cases. Following is a list of helper functions in this file, briefly explained.

- parse_inits(self, init_lines): This method parses and executes the register and memory initialization lines of the input program file.
- write_to_register(self, index, value): This method writes the value to the register given as index, unless it is the register x0.
- read_register(self, index): This method reads and returns the value of the given register.
- print_status(self): This method prints the current values of registers and memory to the console but it omits the values that are equal to 0 for convenience.
- print_final_report(self): This method prints the final report on the simulation of the program. It is intended to be used when the execution is finished. It prints the values total number of clock cycles, CPI, total number of stalls, number of stalls caused by specific instructions.

### 2.2.3 Helper Functions (instruction.py)

This module contains many helper functions that are related to instructions, control values, ALU operations, parsing from the input program etc. We omitted the code segments because they are long and redundant in many cases. Following is a list of helper functions in this file, briefly explained.

- get_program(program_path): This method parses the input program file and processes the instruction lines. Restructures all the instructions as an ordered list of dictionaries that contain fields such as rs1, rd, funct7 etc. Returns a list of those restructured instructions.
- perform_ALU_operation(ALU_Control, param1, param2): This method takes a binary string ALU_Control and decides on which operation to execute on parameters, Essentially simulates an ALU unit.
- get_alu_control(ALU_op, funct_for_alu_control): This method returns the ALU_Control binary string used by ALU unit by calculating it using its parameters.
- get_nop_instruction(): This method returns structured fields corresponding to a NOP instruction.
- get_control_values(instruction): This method returns a dictionary containing control values using the given instruction's opcode field.

### 2.2.4 main.py

This module is the runner program for the simulator. It processes the arguments given to the program and initializes the simulator accordingly.

## 2.3 Output Format

The simulator prints all its output to the console as formatted text. It can be printed into a file using terminal specific commands. The program outputs information about the status of the pipeline on every clock cycle and creates a final report containing statistics at the end of the execution. See Section 4 for sample executions and outputs.

# 3 Execution & Dependencies

This simulator is tested and run on Windows 10 and Mac operating systems using Python 3.7. There are no other packages used other than those that come with Python 3 installation. Running the following terminal command on a folder that contains the code files and also a sample input program(described in Section 2.1) should be suffice to start the simulation.

```
1 python3 main.py program.txt
```

# 4 Sample Simulations & Outputs

In this section, we investigate the simulator outputs on a number of different input programs.

## 4.1 EX Hazard Example

As we can see in output, no stalls have been introduced in dealing with this hazard as it is dealt by the forwarding unit. Total number of cycles is 6 as intended since the first instruction runs for 5 cycles and the second instruction tailing it add 1 more cycle to it. We see that final value for x2 is -6 and x19 is 3 which is correct. This example is taken from Chapter 4 Part 1 Slide 39.

Input program:

```
x1=3
x3=9
---
add x19, x0, x1
sub x2, x19, x3
```

Simulator output:

```
-----STATUS AT THE BEGINNING-----
NOP | NOP | NOP | NOP | NOP
x1: 3 x3: 9
-----STATUS AT THE END OF CLOCK = 1-----
add x19, x0, x1 | NOP | NOP | NOP | NOP is run at PC = 0
x1: 3 x3: 9
-----STATUS AT THE END OF CLOCK = 2-----
sub x2, x19, x3 | add x19, x0, x1 | NOP | NOP | NOP is run at PC = 4
x1: 3 x3: 9
-----STATUS AT THE END OF CLOCK = 3-----
NOP | sub x2, x19, x3 | add x19, x0, x1 | NOP | NOP is run at PC = 8
x1: 3 x3: 9
-----STATUS AT THE END OF CLOCK = 4-----
NOP | NOP | sub x2, x19, x3 | add x19, x0, x1 | NOP is run at PC = 8
x1: 3 x3: 9
-----STATUS AT THE END OF CLOCK = 5-----
NOP | NOP | NOP | sub x2, x19, x3 | add x19, x0, x1 is run at PC = 8
x1: 3 x3: 9 x19: 3
-----STATUS AT THE END OF CLOCK = 6-----
```

```
20 NOP | NOP | NOP | NOP | sub x2, x19, x3 is run at PC = 8
21 x1: 3 x2: -6 x3: 9 x19: 3
22
23 -----FINAL REPORT-----
24 Total # of Clock Cycles: 6
25 Cycles per Instruction(CPI): 3
26 No stall occurred.
```

## 4.2 Load-Use Hazard Example

As we can see in the output, a stall has been inserted at the ID stage of load instruction as a load-use hazard has been detected in the run_ID() method. This additional stall has increased the total number of cycles to 7 which makes the CPI 3.5. As we can see the final values of the x1 and x4, the instructions has been executed correctly as the results are as expected from the program. This example is taken from Chapter 4 Part 1 Slide 41.

Input program:

```
1 x2=1
2 m[1]=10
3 x5=4
4 ---
5 ld x1,0(x2)
6 sub x4,x1,x5
```

Simulator output:

```
1 -----STATUS AT THE BEGINNING-----
2 NOP | NOP | NOP | NOP | NOP
3 x2: 1 x5: 4 m[1]: 10
4 -----STATUS AT THE END OF CLOCK = 1-----
5 ld x1,0(x2) | NOP | NOP | NOP | NOP is run at PC = 0
6 x2: 1 x5: 4 m[1]: 10
7 -----STATUS AT THE END OF CLOCK = 2-----
8 sub x4,x1,x5 | ld x1,0(x2) | NOP | NOP | NOP is run at PC = 4
9 x2: 1 x5: 4 m[1]: 10
10 -----STATUS AT THE END OF CLOCK = 3-----
11 sub x4,x1,x5 | NOP | ld x1,0(x2) | NOP | NOP is run at PC = 8
12 x2: 1 x5: 4 m[1]: 10
13 -----STATUS AT THE END OF CLOCK = 4-----
14 NOP | sub x4,x1,x5 | NOP | ld x1,0(x2) | NOP is run at PC = 8
15 x2: 1 x5: 4 m[1]: 10
16 -----STATUS AT THE END OF CLOCK = 5-----
```

```
17 NOP | NOP | sub x4,x1,x5 | NOP | ld x1,0(x2) is run at PC = 8
18 x1: 10 x2: 1 x5: 4 m[1]: 10
19 ------STATUS AT THE END OF CLOCK = 6------
20 NOP | NOP | NOP | sub x4,x1,x5 | NOP is run at PC = 8
21 x1: 10 x2: 1 x5: 4 m[1]: 10
22 ------STATUS AT THE END OF CLOCK = 7------
23 NOP | NOP | NOP | NOP | sub x4,x1,x5 is run at PC = 8
24 x1: 10 x2: 1 x4: 6 x5: 4 m[1]: 10
25
26 ------FINAL REPORT------
27 Total # of Clock Cycles: 7
28 Cycles per Instruction(CPI): 3.5
29 Total # of Stalls: 1
30 Instructions and # of Stalls Caused:
31 ---> ld x1,0(x2): 1
```

## 4.3 Multiple Hazards Example

We see that in this input program there are multiple hazards. At lines 7 and 8 of the input program there is a load-use hazard, at lines 10 and 11 of the program there is another load-use hazard, at lines 8 and 9 there is an EX hazard and at lines 11 and 12 there is an EX hazard. Since EX hazards are dealt by the forwarding unit, no stalls have been introduced by them. We see that the two load-use hazards are dealt by introducing 2 different stalls. We see that the code correctly identified those stalls are resulted by the ld instructions and reported it. Looking at the final values of the registers and the memory, we see that all of the instructions have been executed correctly and the values are as expected from the program. This example is taken from Chapter 4 Part 1 Slide 42.

Input program:

```
1  m[0]=2
2  m[8]=3
3  m[16]=23
4  x4=9
5  ---
6  ld x1, 0(x0)
7  ld x2, 8(x0)
8  add x3, x1, x2
9  sd x3, 24(x0)
10 ld x4, 16(x0)
11 add x5, x1, x4
12 sd x5, 32(x0)
```

Simulator output:

```
1  ------STATUS AT THE BEGINNING------
2  NOP | NOP | NOP | NOP | NOP
3  x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
4  ------STATUS AT THE END OF CLOCK = 1------
5  ld x1, 0(x0) | NOP | NOP | NOP | NOP is run at PC = 0
6  x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
7  ------STATUS AT THE END OF CLOCK = 2------
8  ld x2, 8(x0) | ld x1, 0(x0) | NOP | NOP | NOP is run at PC = 4
9  x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
10 ------STATUS AT THE END OF CLOCK = 3------
11 add x3, x1, x2 | ld x2, 8(x0) | ld x1, 0(x0) | NOP | NOP is run at PC
       = 8
12 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
13 ------STATUS AT THE END OF CLOCK = 4------
14 add x3, x1, x2 | NOP | ld x2, 8(x0) | ld x1, 0(x0) | NOP is run at PC
       = 12
15 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
16 ------STATUS AT THE END OF CLOCK = 5------
17 sd x3, 24(x0) | add x3, x1, x2 | NOP | ld x2, 8(x0) | ld x1, 0(x0) is
       run at PC = 12
18 x1: 2 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
19 ------STATUS AT THE END OF CLOCK = 6------
20 ld x4, 16(x0) | sd x3, 24(x0) | add x3, x1, x2 | NOP | ld x2, 8(x0)
      is run at PC = 16
21 x1: 2 x2: 3 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
22 ------STATUS AT THE END OF CLOCK = 7------
23 add x5, x1, x4 | ld x4, 16(x0) | sd x3, 24(x0) | add x3, x1, x2 | NOP
       is run at PC = 20
24 x1: 2 x2: 3 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23
25 ------STATUS AT THE END OF CLOCK = 8------
26 add x5, x1, x4 | NOP | ld x4, 16(x0) | sd x3, 24(x0) | add x3, x1, x2
       is run at PC = 24
27 x1: 2 x2: 3 x3: 5 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5
28 ------STATUS AT THE END OF CLOCK = 9------
29 sd x5, 32(x0) | add x5, x1, x4 | NOP | ld x4, 16(x0) | sd x3, 24(x0)
      is run at PC = 24
30 x1: 2 x2: 3 x3: 5 x4: 9 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5
31 ------STATUS AT THE END OF CLOCK = 10------
32 NOP | sd x5, 32(x0) | add x5, x1, x4 | NOP | ld x4, 16(x0) is run at
     PC = 28
33 x1: 2 x2: 3 x3: 5 x4: 23 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5
34 ------STATUS AT THE END OF CLOCK = 11------
35 NOP | NOP | sd x5, 32(x0) | add x5, x1, x4 | NOP is run at PC = 28
36 x1: 2 x2: 3 x3: 5 x4: 23 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5
37 ------STATUS AT THE END OF CLOCK = 12------
```

```
38 NOP | NOP | NOP | sd x5, 32(x0) | add x5, x1, x4 is run at PC = 28
39 x1: 2 x2: 3 x3: 5 x4: 23 x5: 25 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5 m
       [32]: 25
40 ------STATUS AT THE END OF CLOCK = 13-------
41 NOP | NOP | NOP | NOP | sd x5, 32(x0) is run at PC = 28
42 x1: 2 x2: 3 x3: 5 x4: 23 x5: 25 m[0]: 2 m[8]: 3 m[16]: 23 m[24]: 5 m
       [32]: 25
43
44 ------FINAL REPORT------
45 Total # of Clock Cycles: 13
46 Cycles per Instruction(CPI): 1.8571428571428572
47 Total # of Stalls: 2
48 Instructions and # of Stalls Caused:
49 ---> ld x2, 8(x0): 1
50 ---> ld x4, 16(x0): 1
```

## 4.4 Branching Example

In this example we see a simple branching code. During execution of the first 2 instructions, we see that simulator detects that the branching condition is met at EX stage and therefore it flushes the remaining 3 instructions in the pipeline and jumps to the label lab1. This of course introduces new cycles due to the flushed stages, and as we see 3 stalls are counted at the end due to the flush. In the end we see that the execution is completed in 10 cycles. Since only 3 instructions are executed in total, the CPI value 3.33 is correct. Looking at the final values of the registers, we see that all of the values are correct and as expected from the code.

Input program:

```
1  x1=7
2  x2=3
3  x3=4
4  ---
5  add x6,x2,x2
6  beq x1,x1,lab1
7  add x7,x3,x3
8  add x10,x3,x3
9  lab2:
10 add x5,x1,x1
11 lab1:
12 add x11,x1,x1
```

Simulator output:

```
1  ------STATUS AT THE BEGINNING------
2  NOP | NOP | NOP | NOP | NOP
3  x1: 7 x2: 3 x3: 4
4  ------STATUS AT THE END OF CLOCK = 1------
5  add x6,x2,x2 | NOP | NOP | NOP | NOP is run at PC = 0
6  x1: 7 x2: 3 x3: 4
7  ------STATUS AT THE END OF CLOCK = 2------
8  beq x1,x1,lab1 | add x6,x2,x2 | NOP | NOP | NOP is run at PC = 4
9  x1: 7 x2: 3 x3: 4
10 ------STATUS AT THE END OF CLOCK = 3------
11 add x7,x3,x3 | beq x1,x1,lab1 | add x6,x2,x2 | NOP | NOP is run at PC
       = 8
12 x1: 7 x2: 3 x3: 4
13 ------STATUS AT THE END OF CLOCK = 4------
14 add x10,x3,x3 | add x7,x3,x3 | beq x1,x1,lab1 | add x6,x2,x2 | NOP is
       run at PC = 12
15 x1: 7 x2: 3 x3: 4
16 ------STATUS AT THE END OF CLOCK = 5------
17 NOP | NOP | NOP | beq x1,x1,lab1 | add x6,x2,x2 is run at PC = 16
18 x1: 7 x2: 3 x3: 4 x6: 6
19 ------STATUS AT THE END OF CLOCK = 6------
20 add x11,x1,x1 | NOP | NOP | NOP | beq x1,x1,lab1 is run at PC = 20
21 x1: 7 x2: 3 x3: 4 x6: 6
22 ------STATUS AT THE END OF CLOCK = 7------
23 NOP | add x11,x1,x1 | NOP | NOP | NOP is run at PC = 24
24 x1: 7 x2: 3 x3: 4 x6: 6
25 ------STATUS AT THE END OF CLOCK = 8------
26 NOP | NOP | add x11,x1,x1 | NOP | NOP is run at PC = 24
27 x1: 7 x2: 3 x3: 4 x6: 6
28 ------STATUS AT THE END OF CLOCK = 9------
29 NOP | NOP | NOP | add x11,x1,x1 | NOP is run at PC = 24
30 x1: 7 x2: 3 x3: 4 x6: 6
31 ------STATUS AT THE END OF CLOCK = 10------
32 NOP | NOP | NOP | NOP | add x11,x1,x1 is run at PC = 24
33 x1: 7 x2: 3 x3: 4 x6: 6 x11: 14
34
35 ------FINAL REPORT------
36 Total # of Clock Cycles: 10
37 Cycles per Instruction(CPI): 3.3333333333333335
38 Total # of Stalls: 3
39 Instructions and # of Stalls Caused:
40 ---> beq x1,x1,lab1: 3
```

## 4.5 MEM Hazard Example

This is a MEM hazard example. Since the forwarding unit is dealing with the MEM hazard, we see that no stalls are introduced. Therefore, the execution ends in 7 cycles as expected. We see that the final values of the registers are as expected from the correct execution. Which means that the forwarding unit has done its job properly.

Input program:

```
x2=3
m[3]=7
x4=5
---
ld x1, 0(x2)
add x3,x2,x2
add x4, x1, x4
```

Simulator output:

```
-----STATUS AT THE BEGINNING-----
NOP | NOP | NOP | NOP | NOP
x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 1-----
ld x1, 0(x2) | NOP | NOP | NOP | NOP is run at PC = 0
x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 2-----
add x3,x2,x2 | ld x1, 0(x2) | NOP | NOP | NOP is run at PC = 4
x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 3-----
add x4, x1, x4 | add x3,x2,x2 | ld x1, 0(x2) | NOP | NOP is run at PC
    = 8
x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 4-----
NOP | add x4, x1, x4 | add x3,x2,x2 | ld x1, 0(x2) | NOP is run at PC
    = 12
x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 5-----
NOP | NOP | add x4, x1, x4 | add x3,x2,x2 | ld x1, 0(x2) is run at PC
    = 12
x1: 7 x2: 3 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 6-----
NOP | NOP | NOP | add x4, x1, x4 | add x3,x2,x2 is run at PC = 12
x1: 7 x2: 3 x3: 6 x4: 5 m[3]: 7
-----STATUS AT THE END OF CLOCK = 7-----
NOP | NOP | NOP | NOP | add x4, x1, x4 is run at PC = 12
```

```
x1: 7 x2: 3 x3: 6 x4: 12 m[3]: 7

------FINAL REPORT------
Total # of Clock Cycles: 7
Cycles per Instruction(CPI): 2.3333333333333335
No stall occurred.
```

## 4.6 Double Data Hazard Example

This is a double data hazard example. There is a MEM hazard between lines 5 and 7 and there is an EX hazard between the lines 6 and 7. Correct forwarding action is to give priority to the EX hazard and forward the ALU result from the line 6 to the line 7. We see in the output that indeed, forwarding unit of our simulator is correctly forwarding as the final values of the registers are correct and as expected. This example is taken from the Chapter 4 Part 2 Slide 30.

Input program:

```
x2=2
x3=3
x4=4
---
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

Simulator output:

```
------STATUS AT THE BEGINNING------
NOP | NOP | NOP | NOP | NOP
x2: 2 x3: 3 x4: 4
------STATUS AT THE END OF CLOCK = 1------
add x1,x1,x2 | NOP | NOP | NOP | NOP is run at PC = 0
x2: 2 x3: 3 x4: 4
------STATUS AT THE END OF CLOCK = 2------
add x1,x1,x3 | add x1,x1,x2 | NOP | NOP | NOP is run at PC = 4
x2: 2 x3: 3 x4: 4
------STATUS AT THE END OF CLOCK = 3------
add x1,x1,x4 | add x1,x1,x3 | add x1,x1,x2 | NOP | NOP is run at PC =
    8
x2: 2 x3: 3 x4: 4
------STATUS AT THE END OF CLOCK = 4------
NOP | add x1,x1,x4 | add x1,x1,x3 | add x1,x1,x2 | NOP is run at PC =
    12
```

```
15 x2: 2 x3: 3 x4: 4
16 ------STATUS AT THE END OF CLOCK = 5------
17 NOP | NOP | add x1,x1,x4 | add x1,x1,x3 | add x1,x1,x2 is run at PC =
       12
18 x1: 2 x2: 2 x3: 3 x4: 4
19 ------STATUS AT THE END OF CLOCK = 6------
20 NOP | NOP | NOP | add x1,x1,x4 | add x1,x1,x3 is run at PC = 12
21 x1: 5 x2: 2 x3: 3 x4: 4
22 ------STATUS AT THE END OF CLOCK = 7------
23 NOP | NOP | NOP | NOP | add x1,x1,x4 is run at PC = 12
24 x1: 9 x2: 2 x3: 3 x4: 4
25
26 ------FINAL REPORT------
27 Total # of Clock Cycles: 7
28 Cycles per Instruction(CPI): 2.3333333333333335
29 No stall occurred.
```