ECE 586 COMPUTER ARCHITECTURE
FINAL PROJECT REPORT
SPRING 2024

MIPS
LITE PIPELINE SIMULATOR

Team 6
Yashaswi Katne
VVSS Lohith
Veekshith Venkatesha Murthy
Rakesh Reddy Kunduru

Main Objective of the Project:

The main Objective of the project is to model a simplified version of the MIPS ISA called MIPS-lite and the in-order 5-stage pipeline. The simulator will take the provided memory image as its input. It will implement two key features:

a) A functional simulator which simulates the MIPS-lite ISA and captures the impact of instruction execution on machine state,
b) A timing simulator which models the timing details for the 5-stage pipeline.

The output of the simulator will include:

a. A breakdown of instruction frequencies in the instruction trace into different instruction types,
b. Final machine state
c. Execution time (in cycles) of the instruction trace on the 5-stage pipeline.

• Given Specifications

a. Instruction Set
   i. Arithmetic Instructions (including signed integers)
   ii. Logical Instructions (including signed integers)
   iii. Memory Access Instructions
   iv. Control Flow Instructions

b. Instruction Format
   i. R - Type

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-Type	opcode	rs	rt	rd	shamt	function
	31 26	25 20	19 16	15 11	10 6	5 0

Used by instructions ADD, SUB, MUL, OR, AND & XOR

ii. I – Type

	6 bits	5 bits	5 bits	16 bits
I- Type	opcode	rs	rt	immediate
	31 26	25 20	19 16	15 0

Used by instructions ADDI, SUBI, MULI, ORI, ANDI, XORI, LDW, STW, BZ, BEQ, JR & HALT.

c. Memory Image
The Simulator will take a memory image as its input.

d. Functional Simulator
Develop a functional simulator which captures the effect of running the simulated program on the simulated machine state.

e. Timing Simulator
Develop a pipeline timing model which captures the flow of instructions through the 5-stge pipeline

• 32-bit MIPS Lite pipeline simulator has 5-stages,

1. Instruction Fetch
2. Instruction Decode
3.Execution
4.Memory
5.Write back

• Implementation

1. High Level Programing language

   ▪ We have chosen to design the MIPS-lite using System Verilog language.

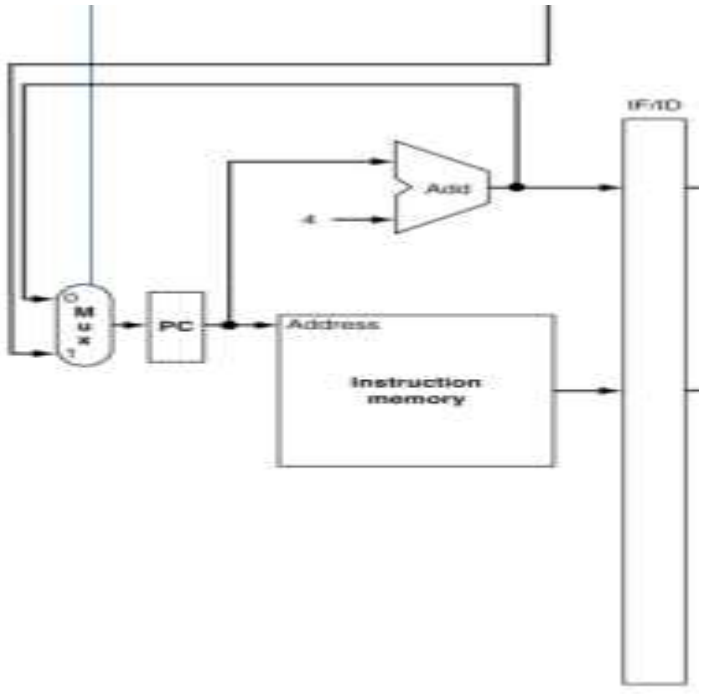
2. Tools Used

   ▪ Questa Sim (Linux)

• Based on the above given specifications we as a team have written the design/RTL code in System Verilog language. So far, we have written the code for two stages of MIPS-lite Pipeline simulator & are as follows:

1. Instruction Fetch
2. Instruction Decode

• For Instruction Fetch stage, we have written code for the components below as per the MIPS pipeline block diagram. We have written all the components in different files & modules as of now.



1. Multiplexer
2. Program Counter
3. Instruction Memory
4. IF/ID pipeline register

In the Instruction Fetch stage, the Instruction being read from memory using the address in the program counter and then begin placed in the IF/IF pipeline register.

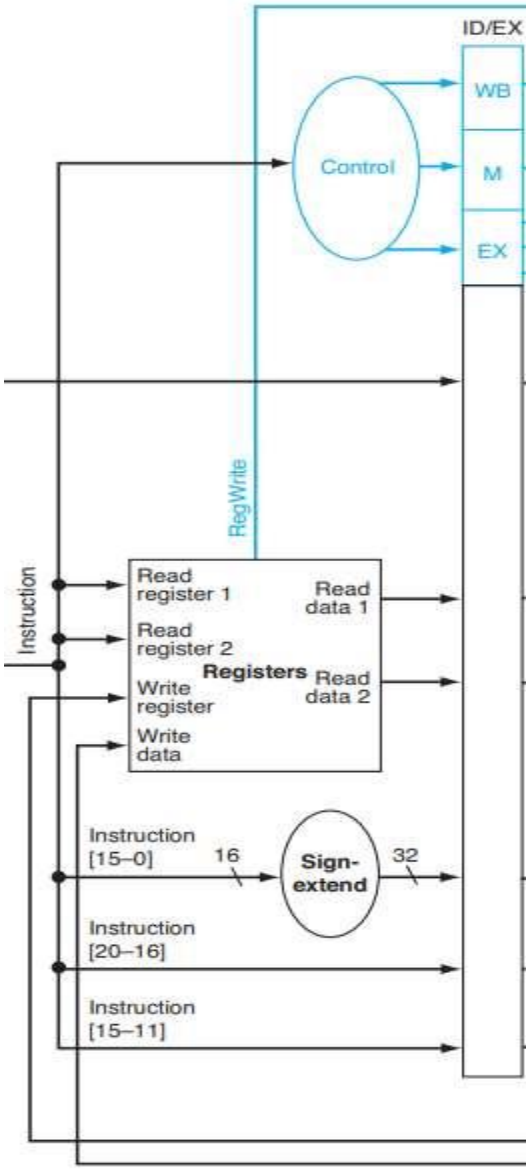
The Program Counter is incremented by 4 and then written back into the program counter to be ready for the next clock cycle.

The Incremented address is also saved in the IF/IF pipeline register in case if it is needed later for the instruction.

• For the Instruction decode stage, we have written code for the components below as per the MIPS Block diagram. All the modules are written separately in different modules and files.

1. Registers
2. Sign Extended
3. ID/EX pipeline register

We haven't written any of the codes that are related to hazards and forwarding as of now, but they are in progress.



In the decode stage at current cycle it will decode the instruction which is fetched from the instruction fetch stage at previous cycle that was stored in IF/ID pipeline register and decode the instruction to know what time of instruction it is by using opcode and function bits.

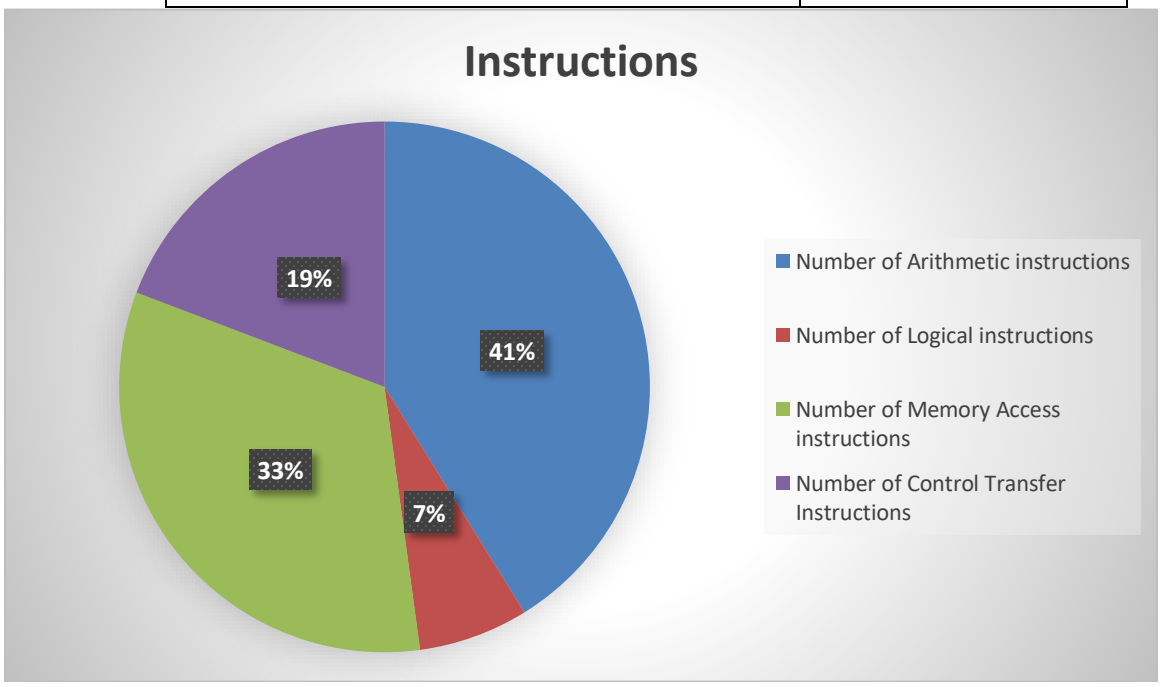
Since the computer cannot determine the type of instruction being fetched, it must be ready for any instruction and send any information that may be required along the pipeline. Instruction decoding and reading from a register file. The 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to access the two registers are supplied by the instruction portion of the IF/ID pipeline register. The ID/EX pipeline register holds all three numbers as well as the PC address that has been incremented. We once more transfer everything that any instruction might require at a later clock period.

• Next Action
We will implement execution, memory & write back stages. and their respective pipeline register and to check the working of MIPS Lite Pipeline Simulator as per the memory image provided.

Simulation Outputs:

1) **Total number of instructions and a breakdown of instruction frequencies for the following instruction types: Arithmetic, Logical, Memory Access, Control Transfer.**

Total number of instructions	911
Number of Arithmetic instructions	375
Number of Logical instructions	61
Number of Memory Access instructions	300
Number of Control Transfer Instructions	175



2) **Final state of program counter, general purpose registers and memory (You only need to include the register and memory locations whose state has changed during the program execution)**

- Final states of Program Counter = 112
- Final states of General-Purpose Registers:

General purpose register names	Final state
R11	1044
R12	1836
R13	2640
R14	25
R15	-188
R16	213
R17	29
R18	3440
R19	-1
R20	-2
R21	-1
R22	76
R23	3
R24	-1
R25	3

- Final state of Memory:

Address = 2400 >> Contents = 2
Address = 2404 >> Contents = 4
Address = 2408 >> Contents = 6
Address = 2412 >> Contents = 8
Address = 2416 >> Contents = 10
Address = 2420 >> Contents = 12
Address = 2424 >> Contents = 14
Address = 2428 >> Contents = 16
Address = 2432 >> Contents = 18
Address = 2436 >> Contents = 29
Address = 2440 >> Contents = 22
Address = 2444 >> Contents = 24
Address = 2448 >> Contents = 26
Address = 2452 >> Contents = 28
Address = 2456 >> Contents = 30
Address = 2460 >> Contents = 32

Address = 2464 >> Contents = 34
Address = 2468 >> Contents = 36
Address = 2472 >> Contents = 38
Address = 2476 >> Contents = 59
Address = 2480 >> Contents = 42
Address = 2484 >> Contents = 44
Address = 2488 >> Contents = 46
Address = 2492 >> Contents = 48
Address = 2496 >> Contents = 50
Address = 2500 >> Contents = 52
Address = 2504 >> Contents = 54
Address = 2508 >> Contents = 56
Address = 2512 >> Contents = 58
Address = 2516 >> Contents = 89
Address = 2520 >> Contents = 62
Address = 2524 >> Contents = 64
Address = 2528 >> Contents = 66
Address = 2532 >> Contents = 68
Address = 2536 >> Contents = 70
Address = 2540 >> Contents = 72
Address = 2544 >> Contents = 74
Address = 2548 >> Contents = 76
Address = 2552 >> Contents = 78
Address = 2556 >> Contents = 119
Address = 2560 >> Contents = 82
Address = 2564 >> Contents = 84
Address = 2568 >> Contents = 86
Address = 2572 >> Contents = 88
Address = 2576 >> Contents = 90
Address = 2580 >> Contents = 92
Address = 2584 >> Contents = 94
Address = 2588 >> Contents = 96
Address = 2592 >> Contents = 98
Address = 2596 >> Contents = 149
Address = 2600 >> Contents = 2
Address = 2604 >> Contents = 4
Address = 2608 >> Contents = 6
Address = 2612 >> Contents = 8
Address = 2616 >> Contents = 10
Address = 2620 >> Contents = 12
Address = 2624 >> Contents = 14
Address = 2628 >> Contents = 16
Address = 2632 >> Contents = 18
Address = 2636 >> Contents = 29

3) Describe the stall conditions in both the "no forwarding" and "forwarding" cases and how long you stalled the pipeline for each stall condition (e.g., in the "no forwarding" case, if a consumer instruction comes right after a producer instruction, then the stall penalty is 2 cycles)

For no forwarding:

1. If the consumer instruction comes immediately after a producer instruction, the stall penalty is 2.

2. If the consumer instruction comes one instruction after a producer instruction and the instruction in between them does not have any stall penalty on it, then stall penalty on consumer instruction is 1.

3. If the consumer instruction comes one instruction after a producer instruction and the instruction in between them have any stall penalty, the stall penalty on consumer instruction is 0.

For forwarding:

1. If the consumer instruction follows right after the producer instruction, then there won't be any stalls hence stall penalty is 0.

2. If the consumer instruction follows right after the producer instruction and if the producer instruction is a LOAD instruction, then the stall penalty is 1.

3. If the instruction is branch and is taken or if the instruction jump register is executed, then the stall penalty is 2.

4) In the case of "no forwarding", the total number of data hazards and the average stall penalty per hazard.

In case of no forwarding from simulation results:

The total number of Data Hazards = 307

The total number of stalls = 544

Average stall penalty per hazard = Total number of stalls/Total number of data hazards

Average stall penalty per hazard = 544/307

Average stall penalty per hazard = 1.804

5) **In the case of "forwarding", the number of data hazards which could not be fully eliminated by forwarding**

   Number of data hazards that could not be fully eliminated = 60

6) **Execution time in terms of number of clock cycles for the "no forwarding" and the "forwarding" scenarios.**

   Total Number of Cycles (without forwarding) >> 1707

   Total Number of Cycles (with forwarding) >> 1213

7) **Speedup achieved by "forwarding" as compared to "no forwarding".**

   Execution time with no forwarding = 1707

   Execution time with forwarding = 1213

   Speedup achieved = (Execution time) no-forwarding / (Execution time) forwarding

   Speedup = 1707/1213

   Speedup = 1.4072

Transcript (ALL THREE MODES MERGED INTO SINGLE TRANSCRIPT):

python .\main.py .\final_proj_trace.txt

**********ECE586 Spring 2024 Final Project Team 6**********

Simulating the MIPS-lite processor with Forwarding: False

***** PC, Memory and Register state *****

Program counter state: 112

Changed memory state:
Address 2400: 2
Address 2404: 4
Address 2408: 6
Address 2412: 8
Address 2416: 10
Address 2420: 12
Address 2424: 14
Address 2428: 16
Address 2432: 18
Address 2436: 29
Address 2440: 22
Address 2444: 24
Address 2448: 26
Address 2452: 28
Address 2456: 30
Address 2460: 32
Address 2464: 34
Address 2468: 36
Address 2472: 38
Address 2476: 59
Address 2480: 42
Address 2484: 44
Address 2488: 46
Address 2492: 48
Address 2496: 50
Address 2500: 52
Address 2504: 54
Address 2508: 56
Address 2512: 58
Address 2516: 89
Address 2520: 62
Address 2524: 64
Address 2528: 66
Address 2532: 68
Address 2536: 70
Address 2540: 72
Address 2544: 74

Address 2548: 76
Address 2552: 78
Address 2556: 119
Address 2560: 82
Address 2564: 84
Address 2568: 86
Address 2572: 88
Address 2576: 90
Address 2580: 92
Address 2584: 94
Address 2588: 96
Address 2592: 98
Address 2596: 149
Address 2600: 2
Address 2604: 4
Address 2608: 6
Address 2612: 8
Address 2616: 10
Address 2620: 12
Address 2624: 14
Address 2628: 16
Address 2632: 18
Address 2636: 29

Register State:
R11: 1044
R12: 1836
R13: 2640
R14: 25
R15: -188
R16: 213
R17: 29
R18: 3440
R19: -1
R20: -2
R21: -1
R22: 76
R23: 3
R24: -1
R25: 3

***** Instruction Counts *****

Total Instruction count: 911
Arithmetic Instruction count: 375
Logical Instruction count: 61
Memory Instruction count: 300
Control Instruction count: 175

***** Timing Simulator *****

Total number of clock cycles: 1707
Number of average stalls: 1.8045602605863191
Total stall count: 554
Total number of hazards: 307

***** Branching Information *****

Total number of branches: 174
Total number of branches taken: 119
Total branch penalties: 238
Average branch penalty: 2.0 cycles

***** Performance of MIPS-lite *****

CPI (Cycles Per Instruction): 1.8737650933040615
IPC (Instructions Per Cycle): 0.533684827182191

Simulating the MIPS-lite processor with Forwarding: True

***** PC, Memory and Register state *****

Program counter state: 112

Changed memory state:
Address 2400: 2
Address 2404: 4
Address 2408: 6
Address 2412: 8
Address 2416: 10
Address 2420: 12
Address 2424: 14
Address 2428: 16
Address 2432: 18
Address 2436: 29
Address 2440: 22
Address 2444: 24
Address 2448: 26
Address 2452: 28
Address 2456: 30
Address 2460: 32
Address 2464: 34
Address 2468: 36
Address 2472: 38
Address 2476: 59
Address 2480: 42
Address 2484: 44
Address 2488: 46
Address 2492: 48
Address 2496: 50

Address 2500: 52
Address 2504: 54
Address 2508: 56
Address 2512: 58
Address 2516: 89
Address 2520: 62
Address 2524: 64
Address 2528: 66
Address 2532: 68
Address 2536: 70
Address 2540: 72
Address 2544: 74
Address 2548: 76
Address 2552: 78
Address 2556: 119
Address 2560: 82
Address 2564: 84
Address 2568: 86
Address 2572: 88
Address 2576: 90
Address 2580: 92
Address 2584: 94
Address 2588: 96
Address 2592: 98
Address 2596: 149
Address 2600: 2
Address 2604: 4
Address 2608: 6
Address 2612: 8
Address 2616: 10
Address 2620: 12
Address 2624: 14
Address 2628: 16
Address 2632: 18
Address 2636: 29

Register State:
R11: 1044
R12: 1836
R13: 2640
R14: 25
R15: -188
R16: 213
R17: 29
R18: 3440
R19: -1
R20: -2
R21: -1
R22: 76
R23: 3

R24: -1
R25: 3

***** Instruction Counts *****

Total Instruction count: 911
Arithmetic Instruction count: 375
Logical Instruction count: 61
Memory Instruction count: 300
Control Instruction count: 175

***** Timing Simulator *****

Total number of clock cycles: 1213
Total stall count: 60
Total number of hazards: 60

***** Branching Information *****

Total number of branches: 174
Total number of branches taken: 119
Total branch penalties: 238
Average branch penalty: 2.0 cycles

***** Performance of MIPS-lite *****

CPI (Cycles Per Instruction): 1.331503841931943
IPC (Instructions Per Cycle): 0.7510305028854081

****

***** Forwarding vs. Non-Forwarding Comparison *****

Total clock cycle count without forwarding: 1707
Total clock cycle count with forwarding: 1213
Speedup due to forwarding: 1.41x
Total IPC lift (%) due to forwarding: 21.73%

****

Source Code:

```
class Instruction(object):
    """
    Represents a CPU instruction with methods to decode it from hexadecimal
format.

    Attributes:
        hex_instr (str): Hexadecimal string representing the instruction.
        opcode_dict (dict): Maps binary opcode strings to their mnemonic.
```

```
        r_type (list): List of mnemonics that are of R-type format.
        hex (str): Hexadecimal representation of the instruction.
        opcode (str): Mnemonic of the opcode.
        type (str): Type of the instruction ('R' for register, 'I' for
immediate).
        rs, rt, rd (int): Source, target, and destination register numbers.
        reg_rs, reg_rt, reg_rd (int): Actual register numbers after decoding.
        imm (int): Immediate value for I-type instructions.
        x_addr (int): Address for branch instructions.
        frwd_rs, frwd_rt (bool): Flags indicating if forwarding is applied to
rs and rt registers.
    """

    def __init__(self, hex_instr):
        """
        Initializes a new Instruction instance.

        Args:
            hex_instr (str): Hexadecimal string of the instruction.
        """
        self.opcode_dict = {
            '000000': 'Add', '000001': 'Addi', '000010': 'Sub', '000011':
'Subi',
            '000100': 'Mul', '000101': 'Muli', '000110': 'Or', '000111':
'Ori',
            '001000': 'And', '001001': 'Andi', '001010': 'Xor', '001011':
'Xori',
            '001100': 'Ldw', '001101': 'Stw', '001110': 'Bz', '001111': 'Beq',
            '010000': 'Jr', '010001': 'Halt'
        }
        self.r_type = ['Add', 'Sub', 'Mul', 'Or', 'And', 'Xor']
        self.hex = hex_instr
        self.opcode = None
        self.type = None
        self.rs = self.rt = self.rd = 0
        self.reg_rs = self.reg_rt = self.reg_rd = None
        self.imm = self.x_addr = None
        self.frwd_rs = self.frwd_rt = False

    def __repr__(self):
        """
        Returns a string representation of the instruction.
        """
        return
f'Instr({self.hex},{self.opcode},R{self.reg_rs}:{self.rs},R{self.reg_rt}:{self
.rt},R{self.reg_rd}:{self.rd},imm:{self.imm},addr:{self.x_addr},rs_f:{self.frw
d_rs},rt_f:{self.frwd_rt})'
```

```
    def decode(self):
        """
        Decodes the instruction from its hexadecimal representation.
        """
        if self.hex is None:
            return
        # Convert hex to a 32-bit binary string
        bin_inst = '{0:032b}'.format(int.from_bytes(bytes.fromhex(self.hex),
byteorder='big', signed=True))
        opcode = bin_inst[:6]
        if opcode in self.opcode_dict:
            self.opcode = self.opcode_dict[opcode]
            if self.opcode in self.r_type:  # Decode R-type instruction
                self.type = 'R'
                self.reg_rs = int(bin_inst[6:11], 2)
                self.reg_rt = int(bin_inst[11:16], 2)
                self.reg_rd = int(bin_inst[16:21], 2)
            else:  # Decode I-type instruction
                self.type = 'I'
                if self.opcode == 'Halt':
                    return
                self.reg_rs = int(bin_inst[6:11], 2)
                if self.opcode == 'Jr':
                    return
                if self.opcode == 'Bz':
                    self.x_addr =
int('{0:016b}'.format(int.from_bytes(bytes.fromhex(self.hex[-4:]),
byteorder='big', signed=True)), 2)
                    return
                self.reg_rt = int(bin_inst[11:16], 2)
                self.imm =
int('{0:016b}'.format(int.from_bytes(bytes.fromhex(self.hex[-4:]),
byteorder='big', signed=True)), 2)
```

```
import os

class Memory(object):
    '''
    A class to simulate memory operations based on a trace file.
    It supports reading and writing both words and bytes to memory locations.
    '''
    def __init__(self, path):
        '''
        Initializes the memory with the contents of a trace file.
```

```
        :param path: The file path to the trace file.
        '''
        self.mem_trace = dict()
        if os.path.exists(path):
            index = 0
            with open(path, 'r+') as trace:
                lines = trace.readlines()
                for line in lines:
                    # Store each line from the trace file into the memory
trace dictionary.
                    # The line's content is stored at 'index', simulating a
memory address.
                    self.mem_trace[index] = line.strip('\n')
                    index += 4  # Increment index by 4 for the next memory
address.

    def read_word(self, index):
        '''
        Reads and returns a word from a specified memory location.

        :param index: The memory address to read from.
        :return: The word at the specified memory address.
        '''
        return self.mem_trace[index]

    def read_byte(self, index):
        '''
        Reads and returns a single byte from a specified memory location.

        :param index: The memory address to read from.
        :return: The byte at the specified memory address.
        '''
        offset = index % 4
        word = self.mem_trace[index - offset]
        # Convert the word from hex to binary, extract the relevant byte, and
return it as an integer.
        byte = int('{0:032b}'.format(int.from_bytes(bytes.fromhex(word),
byteorder='big', signed=True))[offset*8:offset*8+8])
        return byte

    def write_byte(self, index, word):
        '''
        Writes a byte to a specified memory location.

        :param index: The memory address to write to.
        :param word: The byte (in hex) to write.
        '''
```

```
        offset = index % 4
        current_word = self.mem_trace[index - offset]
        current_word_bin =
'{0:032b}'.format(int.from_bytes(bytes.fromhex(current_word), byteorder='big',
signed=True))
        new_byte_bin = '{0:08b}'.format(int.from_bytes(bytes.fromhex(word),
byteorder='big', signed=True))
        # Replace the relevant byte in the current word with the new byte.
        word_write = current_word_bin[:offset*8] + new_byte_bin +
current_word_bin[offset*8+8:]
        self.mem_trace[index - offset] = hex(int(word_write, 2))[2:]

    def write_word(self, index, word):
        '''
        Writes a word to a specified memory location.

        :param index: The memory address to write to.
        :param word: The word to write.
        '''
        self.mem_trace[index] = word

class registers(object):
    '''
    Simulates a set of 32 registers for a processor.
    '''
    def __init__(self):
        '''
        Initializes 32 registers, setting each to 0.
        '''
        self.regs = dict()
        for index in range(32):
            self.regs[index] = 0  # Initialize all registers to 0.

    def read_reg(self, index):
        '''
        Returns the value of a specified register.

        :param index: The register number to read from.
        :return: The value of the specified register.
        '''
        return self.regs[index]

    def write_reg(self, index, word):
        '''
        Writes a value to a specified register.

        :param index: The register number to write to.
        :param word: The value to write to the register.
```

```
        '''
        self.regs[index] = word
```

```
import copy
import memory
from instruction import Instruction

class Processor(object):
    '''
    Simulates a 5-stage pipelined MIPS-lite 32 architecture processor. This
class allows running a trace of instructions
    and printing processor statistics. It supports instruction forwarding to
mitigate data hazards between the EX (Execute)
    and MEM (Memory) stages.

    Attributes:
    - memory_image: An instance of the Memory class containing the instruction
trace.
    - forwarding: Enables full-forwarding if set to True. Defaults to False.
    - debug: Enables verbose logging of pipeline stages if set to True.
Defaults to False.
    '''
    def __init__(self,memory_image,forwarding=False,debug=False):
        '''
        Instantiate a new processor.
        '''
        self.mem_image = copy.deepcopy(memory_image)
        self.original_mem = memory_image
        self.forwarding = forwarding
        self.debug=debug
        self.frwrd_dict = dict()
        self.regs = memory.registers()
        self.data_list =
[0,Instruction(None),Instruction(None),Instruction(None),Instruction(None)]
        self.pc = 0
        self.arith_opcode = ['Add','Addi','Sub','Subi','Mul','Muli']
        self.logic_opcode = ['Or','Ori','And','Andi','Xor','Xori']
        self.ld_opcode = ['Ldw','Stw']
        self.ctrl_opcode = ['Bz','Beq','Jr','Halt']
        self.r_type = ['Add','Sub','Mul','Or','And','Xor']
        self.reg_change={}
        self.ari_count = 0
        self.logic_count = 0
        self.ctrl_count = 0
        self.ld_count = 0
```

```
        self.stall_cycle=0
        self.num_instructions = 0
        self.branch_penalties = 0
        self.branches_taken = 0
        self.total_branches = 0
        self.hazards = 0
        self.st_count = []
        self.Halt = False
        self.cycles=0

    def run(self,):
        '''
        Runs the complete trace of instructions
        '''
        self.cycles = 0
        while self.data_list[4].opcode!='Halt':
            self.Write_back()
            self.Memory_op()
            self.Execute()
            self.Instruction_decode()
            self.Fetch()
            self.data_list[0]=self.pc
            self.cycles += 1
        self.cycles += 1
        return

    def print_stats(self):
        '''
        Prints execution statistics, including cycle count, instruction counts
by type,
        and information on stalls and hazards.
        '''
        print("\nSimulating the MIPS-lite processor with Forwarding: " +
str(self.forwarding) + "\n")
        print("*" * 5 + " PC, Memory and Register state " + "*" * 5 + "\n")
        print("Program counter state: ", self.pc)
        self.print_mem()
        self.print_reg()
        print("\n" + "*" * 5 + " Instruction Counts " + "*" * 5 + "\n")
        print("Total Instruction count: ", self.num_instructions)
        print("Arithmetic Instruction count: ", self.ari_count)
        print("Logical Instruction count: ", self.logic_count)
        print("Memory Instruction count: ", self.ld_count)
        print("Control Instruction count: ", self.ctrl_count)
        print("\n" + "*" * 5 + " Timing Simulator " + "*" * 5 + "\n")
        print("Total number of clock cycles: ", self.cycles)
        if not self.forwarding:
```

```
            print("Number of average stalls: ", sum(self.st_count) /
self.hazards if self.hazards > 0 else 0)
        print("Total stall count: ", sum(self.st_count))
        print("Total number of hazards: ", self.hazards)
        print("\n" + "*" * 5 + " Branching Information " + "*" * 5 + "\n")
        print("Total number of branches: ", self.total_branches)
        print("Total number of branches taken: ", self.branches_taken)
        print("Total branch penalties: ", self.branch_penalties)
        if self.branches_taken > 0:
            print("Average branch penalty: ", (self.branch_penalties /
self.branches_taken), " cycles")
        print("\n" + "*" * 5 + " Performance of MIPS-lite " + "*" * 5 + "\n")
        # Calculate and print CPI (Cycles Per Instruction) and IPC
(Instructions Per Cycle)
        self.cpi = self.cycles / self.num_instructions if
self.num_instructions > 0 else 0
        self.ipc = self.num_instructions / self.cycles if self.cycles > 0 else
0
        print("CPI (Cycles Per Instruction): ", self.cpi)
        print("IPC (Instructions Per Cycle): ", self.ipc)

    def store_stats(self):
        '''
        Stores execution statistics in a dictionary and returns it.
        '''
        stats = {
            "forwarding": self.forwarding,
            "program_counter": self.pc,
            "num_instructions": self.num_instructions,
            "arithmetic_instructions": self.ari_count,
            "logic_instructions": self.logic_count,
            "memory_instructions": self.ld_count,
            "control_instructions": self.ctrl_count,
            "cycles": self.cycles,
            "total_stalls": sum(self.st_count),
            "hazards": self.hazards,
            "total_branches": self.total_branches,
            "branches_taken": self.branches_taken,
            "branch_penalties": self.branch_penalties,
            "average_branch_penalty": self.branch_penalties /
self.total_branches if self.total_branches > 0 else 0,
            "cpi": self.cpi,
            "ipc": self.ipc
        }
        if not self.forwarding:
            stats["average_stalls"] = sum(self.st_count) / self.hazards if
self.hazards > 0 else 0
```

```
        return stats

    def print_mem(self):
        '''
        Prints the state of memory locations that have changed during
execution, each on a new line.
        '''
        keys = [k for k in self.original_mem.mem_trace if
self.mem_image.mem_trace[k] != self.original_mem.mem_trace[k]]
        changed_memory = {k: self.mem_image.mem_trace[k] for k in keys}
        print("\nChanged memory state:")
        for address, value in changed_memory.items():
            print(f"Address {address}: {value}")

    def print_reg(self):
        '''
        Prints the current state of the processor's registers in ascending
order.
        '''
        print("\nRegister State:")
        for reg in sorted(self.reg_change.keys(), key=lambda x: int(x[1:])):
            print(f"{reg}: {self.reg_change[reg]}")

    def _check_target(self,instr):
        '''
        Checks stalls in the Instruction Decode stage.
        Input: instr: object of class Instruction
        '''
        if instr.opcode=='Halt':
            return
        if self.forwarding and ((instr.type=='R') or (instr.opcode in
['Beq','Stw'])):
            for obj in self.data_list[3:]:
                if obj.type=='I' and obj.opcode!='Ldw':
                    if obj.reg_rt==instr.reg_rs:
                        instr.rs = obj.rt
                        instr.frwd_rs = True
                    elif obj.reg_rt==instr.reg_rt:
                        instr.rt = obj.rt
                        instr.frwd_rt = True
                    elif obj.reg_rt==instr.reg_rs==instr.reg_rt:
                        instr.rs = obj.rt
                        instr.frwd_rs = True
                        instr.rt = obj.rt
                        instr.frwd_rt = True
                elif obj.type=='R':
                    if obj.reg_rd==instr.reg_rs:
                        instr.rs = obj.rd
```

```
                        instr.frwd_rs = True
                    elif obj.reg_rd==instr.reg_rt:
                        instr.rt = obj.rd
                        instr.frwd_rt = True
                    elif obj.reg_rd==instr.reg_rs==instr.reg_rt:
                        instr.rs = obj.rd
                        instr.frwd_rs = True
                        instr.rt = obj.rd
                        instr.frwd_rt = True
                elif obj.opcode=='Ldw':
                    if obj.reg_rt==instr.reg_rs:
                        if self.data_list[3:].index(obj)==1:
                            instr.rs = obj.rt
                            instr.frwd_rs = True
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)-1
                            return
                    elif obj.reg_rt==instr.reg_rt:
                        if self.data_list[3:].index(obj)==1:
                            instr.rt = obj.rt
                            instr.frwd_rt = True
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)-1
                            return
                    elif obj.reg_rt==instr.reg_rs==instr.reg_rt:
                        if self.data_list[3:].index(obj)==1:
                            instr.rs = obj.rt
                            instr.frwd_rs = True
                            instr.rt = obj.rt
                            instr.frwd_rt = True
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)-1
                            return

        elif self.forwarding and instr.type=='I':
            for obj in self.data_list[3:]:
                if obj.type=='I'and obj.opcode!='Ldw':
                    if obj.reg_rt==instr.reg_rs:
                        instr.rs = obj.rt
                        instr.frwd_rs =True
                        break
                elif obj.type=='R':
                    if obj.reg_rd==instr.reg_rs:
                        instr.rs = obj.rd
                        instr.frwd_rs =True
```

```
                        break
                elif obj.opcode=='Ldw':
                    if obj.reg_rt==instr.reg_rs:
                        if self.data_list[3:].index(obj)==1:
                            instr.rs = obj.rt
                            instr.frwd_rs = True
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)-1
                            return
        elif (instr.type=='R') or (instr.opcode in ['Beq','Stw']):
            for obj in self.data_list[3:]:
                if obj.type=='I':
                    if obj.reg_rt in [instr.reg_rs,instr.reg_rt]:
                        self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)
                        return
                elif obj.type=='R':
                    if obj.reg_rd in [instr.reg_rs,instr.reg_rt]:
                        self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)
                        return
        elif instr.type=='I':
            for obj in self.data_list[3:]:
                if obj.type=='I':
                    if obj.reg_rt==instr.reg_rs:
                        if self.forwarding:
                            instr.rs = obj.rt
                            instr.frwd_rs =True
                            break
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)
                        return
                elif obj.type=='R':
                    if obj.reg_rd==instr.reg_rs:
                        if self.forwarding:
                            instr.rs = obj.rd
                            instr.frwd_rs =True
                            break
                        else:
                            self.stall_cycle = len(self.data_list[3:]) -
self.data_list[3:].index(obj)
                        return
        return

    def _getSignedNum(self,num, bitLength):
        '''
```

```
        Returns signed number.
        Inputs: num: number
        bitlength: number of bits for signed number
        '''
        mask = (2 ** bitLength) - 1
        if num & (1 << (bitLength - 1)):
            return num | ~mask
        else:
            return num & mask

    def Fetch(self,):
        '''
        Instruction Fetch stage of the processor. Fetches address of
instruction from the program counter.
        '''
        if self.debug:
            print("In IF stage: ",self.data_list[0])
        if self.Halt:
            return
        if self.data_list[0]==None:
            self.data_list[1] = Instruction(None)
            return
        if self.stall_cycle>0:
            self.stall_cycle = self.stall_cycle - 1
            if self.debug:
                print('stall cycles in IF: ',self.stall_cycle)
            return
        for inst in self.data_list[1:]:
            if (self.data_list[0] == inst.x_addr) and inst.opcode=='Stw':
                if self.forwarding:
                    self.data_list[1] = Instruction(hex(inst.rt)[2:])
                    self.pc +=4
                    return
                else:
                    self.hazards += 1
                    self.stall_cycle = len(self.data_list[1:])-
self.data_list[1:].index(inst)
                    self.st_count.append(self.stall_cycle)
        if self.stall_cycle>0:
            if self.debug:
                print('stall cycles in IF: ',self.stall_cycle)
            return
        instr = self.mem_image.read_word(self.data_list[0])
        self.data_list[1] = Instruction(instr)
        self.pc += 4
        return

    def Instruction_decode(self,):
```

```
        '''
        Instruction Decode stage of the processor. Decodes the instruction
from the IF stage.
        '''
        instr = self.data_list[1]
        instr.decode()
        if self.debug:
            print("In ID stage: ",instr)
        if instr.opcode==None:
            self.data_list[2] = instr
            return
        if self.Halt:
            self.data_list[2] = Instruction(None)
            return
        if self.stall_cycle>0:
            if self.debug:
                print('stall cycles in ID: ',self.stall_cycle)
            self.data_list[2] = Instruction(None)
            return
        self._check_target(instr)
        if self.stall_cycle>0:
            if self.debug:
                print('stall cycles in ID: ',self.stall_cycle)
            self.hazards += 1
            self.st_count.append(self.stall_cycle)
            self.data_list[2] = Instruction(None)
            return
        if instr.opcode in self.arith_opcode:
            self.ari_count += 1
        elif instr.opcode in self.logic_opcode:
            self.logic_count += 1
        elif instr.opcode in self.ctrl_opcode:
            self.ctrl_count += 1
        elif instr.opcode in self.ld_opcode:
            self.ld_count += 1
        if instr.opcode=='Halt':
            self.Halt = True
            self.data_list[2] = instr
            return
        if instr.frwd_rs==False:
            instr.rs = self.regs.read_reg(instr.reg_rs) if instr.opcode in
self.logic_opcode else
self._getSignedNum(self.regs.read_reg(instr.reg_rs),32)
        if instr.opcode in ['Jr','Bz']:
            self.data_list[2] = instr
            return
        if instr.frwd_rt==False:
            if instr.opcode in self.r_type:
```

```
                instr.rt =
self._getSignedNum(self.regs.read_reg(instr.reg_rt), 32)
            else:
                if instr.opcode in ['Stw','Beq']:
                    instr.rt = self.regs.read_reg(instr.reg_rt)
        self.data_list[2] = instr
        return

    def Execute(self,):
        '''
        Instruction Execute stage of the processor. It executes the
instruction.
        '''
        instr = self.data_list[2]
        if self.debug:
            print("In EX stage: ",instr)
        if instr.opcode==None:
            self.num_instructions = self.num_instructions
            self.data_list[3] = instr
            return
        else:
            self.num_instructions += 1
        if instr.opcode=='Halt':
            self.data_list[3] = instr
            return
        if self.Halt:
            return
        if instr.opcode in ['Jr', 'Bz', 'Beq']:
            self.total_branches += 1
            branch_taken = False
            if instr.opcode == 'Jr':
                self.pc = instr.rs
                branch_taken = True
            elif instr.opcode == 'Bz' and instr.rs == 0:
                self.pc = self.pc - 8 + 4 * instr.x_addr
                branch_taken = True
            elif instr.opcode == 'Beq' and instr.rs == instr.rt:
                self.pc = self.pc - 8 + 4 * instr.imm
                branch_taken = True

            if branch_taken:
                self.branches_taken += 1
                self.branch_penalties += 2  # Assuming a fixed penalty of 2
cycles for taken branches

            # Clear the pipeline if a branch is taken
            if branch_taken:
                self.data_list[1] = Instruction(None)
```

```
                self.data_list[0] = None
        if instr.opcode=='Add':
            instr.rd = self._getSignedNum(instr.rs + instr.rt,32)
        elif instr.opcode=='Addi':
            instr.rt = self._getSignedNum(instr.rs + instr.imm,32)
        elif instr.opcode=='Sub':
            instr.rd = self._getSignedNum(instr.rs - instr.rt,32)
        elif instr.opcode=='Subi':
            instr.rt = self._getSignedNum(instr.rs - instr.imm,32)
        elif instr.opcode=='Mul':
            instr.rd = self._getSignedNum(instr.rs * instr.rt,32)
        elif instr.opcode=='Muli':
            instr.rt = self._getSignedNum(instr.rs * instr.imm,32)
        elif instr.opcode=='Or':
            instr.rd = instr.rs | instr.rt
        elif instr.opcode=='Ori':
            instr.rt = instr.rs | instr.imm
        elif instr.opcode=='And':
            instr.rd = instr.rs & instr.rt
        elif instr.opcode=='Andi':
            instr.rt = instr.rs & instr.imm
        elif instr.opcode=='Xor':
            instr.rd = instr.rs ^ instr.rt
        elif instr.opcode=='Xori':
            instr.rt = instr.rs ^ instr.imm
        elif instr.opcode=='Ldw':
            instr.x_addr = self._getSignedNum(instr.rs + instr.imm,32)
        elif instr.opcode=='Stw':
            instr.x_addr = self._getSignedNum(instr.rs + instr.imm,32)
        self.data_list[3] = instr
        return

    def Memory_op(self,):
        '''
        Memory stage of the processor. It loads or stores data from the
memory.
        '''
        instr = self.data_list[3]
        if self.debug:
            print('In Mem stage: ',instr)
        if instr.opcode==None:
            self.data_list[4] = instr
            return
        if instr.opcode=='Ldw':
            addr = instr.x_addr
            instr.rt =
int('{0:032b}'.format(int.from_bytes(bytes.fromhex(self.mem_image.read_word(ad
dr)),byteorder='big',signed=True)),2)
```

```
        elif instr.opcode=='Stw':
            addr = instr.x_addr
            data = instr.rt
            self.mem_image.write_word(addr,data)
        self.data_list[4] = instr
        return

    def Write_back(self,):
        '''
        Write Back stage of the processor. It writes the generated data back
to the registers.
        '''
        instr = self.data_list[4]
        if self.debug:
            print('In WB stage: ',instr)
        if instr.opcode==None:
            pass
            return
        if (instr.opcode in ['Stw','Jr','Halt','Bz','Beq']):
            pass
        else:
            if instr.opcode in self.r_type:
                self.reg_change['R'+str(instr.reg_rd)] = instr.rd
                self.regs.write_reg(instr.reg_rd,instr.rd)
            else:
                self.reg_change['R'+str(instr.reg_rt)] = instr.rt
                self.regs.write_reg(instr.reg_rt,instr.rt)
        return
```

```
from processor import Processor
from memory import Memory
import sys
import os

def main():
    if len(sys.argv) < 2:
        print("Usage: python main.py <memory_image_file>")
        sys.exit(1)

    image_file = sys.argv[1]

    # Check if the file exists
    if not os.path.exists(image_file):
        print(f"Error: The file '{image_file}' does not exist. Please provide
a correct file.")
```

```
        sys.exit(1)  # Exit if the file does not exist

    mem_image = Memory(image_file)

    print("\n" + "*" * 10 + "ECE586 Spring 2024 Final Project Team 6" + "*" *
10)
    # Start the processor with forwarding disabled
    p1 = Processor(mem_image, forwarding=False, debug=False)
    p1.run()
    p1.print_stats()
    stats_without_forwarding = p1.store_stats()

    # Start the processor with forwarding enabled
    p2 = Processor(mem_image, forwarding=True)
    p2.run()
    p2.print_stats()
    print("\n**********************************\n")
    stats_with_forwarding = p2.store_stats()

    # Extract total cycles from stats
    cycles_without_forwarding = stats_without_forwarding['cycles']
    cycles_with_forwarding = stats_with_forwarding['cycles']

    # Extract IPC from stats
    ipc_without_forwarding = stats_without_forwarding['ipc']
    ipc_with_forwarding = stats_with_forwarding['ipc']

    # Calculate speedup
    if cycles_with_forwarding > 0:
        speedup = cycles_without_forwarding / cycles_with_forwarding
    else:
        speedup = float('inf')  # Avoid division by zero

    print("*" * 5 + " Forwarding vs. Non-Forwarding Comparison " + "*" * 5 +
"\n")
    print(f"Total clock cycle count without forwarding:
{cycles_without_forwarding}")
    print(f"Total clock cycle count with forwarding:
{cycles_with_forwarding}")
    print(f"Speedup due to forwarding: {speedup:.2f}x")
    print(f"Total IPC lift (%) due to forwarding: {(ipc_with_forwarding -
ipc_without_forwarding) * 100:.2f}%")
    print("\n**********************************\n")

if __name__ == "__main__":
    main()
```