Computer Architecture
ECE_586
Spring 2023
Project Report

MIPS-lite

Project By-
(Team 2)
Sakshi Garge
Jahnvi Mehta
Rutuja Muttha
Muskan Pashine

Introduction:

The objective is to develop a functional simulator. This simulator will analyze an instruction trace, identify different instruction types by decoding their opcodes, and calculate the frequency of each instruction type. By maintaining counters for each instruction type, it will increment the respective counter whenever an instruction of that type is encountered in the trace. This breakdown of instruction frequencies provides insights into the program's behavior and can help identify performance characteristics. The instruction decoder is a critical component of the processor responsible for interpreting instructions fetched from memory. It decodes the binary code, determines the operation based on the opcode and operands, and enables the processor to understand and execute the instructions.

A pipeline simulator is used to simulate the operation of a pipelined processor like MIPS Lite. It models the different stages of the pipeline and simulates the flow of instructions through it. By tracking the processor's state and providing a trace of the program's execution, the simulator allows for performance analysis and identification of bottlenecks. The functional simulator being developed captures the effect of running the program on the simulated machine state. It includes the program counter, general-purpose registers, and memory. The simulator starts with an initialized state, simulates each instruction by updating the machine state accordingly, and continues until it encounters a "HALT" instruction.

By combining the instruction decoding, pipeline simulation, and functional simulation, a comprehensive understanding of program behavior and performance can be achieved. The simulator provides valuable insights for optimizing program execution and improving overall processor performance.

Work Divided:

Rutuja: Functional behavior without any forwarding.
Jahnvi: Functional and Timing behavior without forwarding.
Sakshi & Muskan: Functional and Timing behavior with forwarding.

1.Instructions:

Total number of instructions = 911

Type of Instructions	Total Number of Instructions	Instruction Frequency
Arithmetic	375	41.1626 %
Logical	61	6.6959 %
Memory Access	300	32.9308 %
Control Transfer	175	19.2097 %

2.Final States of Program Counter, Registers and Memory:

PC: 112
R 11 : 1044
R 12 : 1836
R 13 : 2640
R 18 : 3440
R 19 : -1
R 20 : -2
R 22 : 76
R 23 : 3
R 24 : -1
R 14 : 25
R 15 : -188
R 17 : 29
R 16 : 213
R 21 : -1
R 25 : 3
Memory : Address: 2400 , Values: 2
Memory : Address: 2404 , Values: 4
Memory : Address: 2408 , Values: 6
Memory : Address: 2412 , Values: 8
Memory : Address: 2416 , Values: 10
Memory : Address: 2420 , Values: 12
Memory : Address: 2424 , Values: 14

Memory : Address: 2428 , Values: 16
Memory : Address: 2432 , Values: 18
Memory : Address: 2436 , Values: 29
Memory : Address: 2440 , Values: 22
Memory : Address: 2444 , Values: 24
Memory : Address: 2448 , Values: 26
Memory : Address: 2452 , Values: 28
Memory : Address: 2456 , Values: 30
Memory : Address: 2460 , Values: 32
Memory : Address: 2464 , Values: 34
Memory : Address: 2468 , Values: 36
Memory : Address: 2472 , Values: 38
Memory : Address: 2476 , Values: 59
Memory : Address: 2480 , Values: 42
Memory : Address: 2484 , Values: 44
Memory : Address: 2488 , Values: 46
Memory : Address: 2492 , Values: 48
Memory : Address: 2496 , Values: 50
Memory : Address: 2500 , Values: 52
Memory : Address: 2504 , Values: 54
Memory : Address: 2508 , Values: 56
Memory : Address: 2512 , Values: 58
Memory : Address: 2516 , Values: 89
Memory : Address: 2520 , Values: 62
Memory : Address: 2524 , Values: 64
Memory : Address: 2528 , Values: 66
Memory : Address: 2532 , Values: 68
Memory : Address: 2536 , Values: 70
Memory : Address: 2540 , Values: 72
Memory : Address: 2544 , Values: 74
Memory : Address: 2548 , Values: 76
Memory : Address: 2552 , Values: 78
Memory : Address: 2556 , Values: 119
Memory : Address: 2560 , Values: 82
Memory : Address: 2564 , Values: 84
Memory : Address: 2568 , Values: 86
Memory : Address: 2572 , Values: 88
Memory : Address: 2576 , Values: 90
Memory : Address: 2580 , Values: 92
Memory : Address: 2584 , Values: 94
Memory : Address: 2588 , Values: 96
Memory : Address: 2592 , Values: 98
Memory : Address: 2596 , Values: 149

Memory : Address: 2600 , Values: 2
Memory : Address: 2604 , Values: 4
Memory : Address: 2608 , Values: 6
Memory : Address: 2612 , Values: 8
Memory : Address: 2616 , Values: 10
Memory : Address: 2620 , Values: 12
Memory : Address: 2624 , Values: 14
Memory : Address: 2628 , Values: 16
Memory : Address: 2632 , Values: 18
Memory : Address: 2636 , Values: 29

3.Stalls, Hazards and Total Clock Cycles:

Number of stalls in case of without forwarding : 554
Number of stalls in case of forwarding : 60
Number of hazards in case of without forwarding : 307
Number of hazards in case of forwarding : 60
Number of clock cycles in case of without forwarding : 1707 cycles
Number of clock cycles in case of forwarding : 1213 cycles
Average stall penalty in case of without forwarding : 1.8046
Speedup achieved in case of forwarding : 1707 / 1213 : 1.4073

In the case of without forwarding, we have simulated in such a way that 2 stalls are added whenever a producer instruction immediately followd a consumer instruction.

4.Source Code:

a. Functional Simulator:

```
"""
Author: Rutuja Muttha
Computer Architecture_ECE586_Spring2023
Functional Simulator
MIPS-lite

"""


import os

#Define opcode values as per defined in Project_specs

# Arithmetic Instruction
ADD  = 0b000000
SUB  = 0b000010
```

```
MUL  = 0b000100
ADDI = 0b000001
SUBI = 0b000011
MULI = 0b000101

# Logical Instruction
OR   = 0b000110
AND  = 0b001000
XOR  = 0b001010
ORI  = 0b000111
ANDI = 0b001001
XORI = 0b001011

# Memory Access Instruction
LDW  = 0b001100                    # Load the contents of the memory location
STW  = 0b001101                    # Store the contents of the memory location

# Control Flow Instruction
BZ   = 0b001110                    # Branch
BEQ  = 0b001111                    # Branch if equal
JR   = 0b010000                    # Jump to new PC
HALT = 0b010001                    # Stop executing the program

# Instruction Format
R_TYPE = [ADD,SUB,MUL,OR,AND,XOR]
I_TYPE = [ADDI,SUBI,MULI,ORI,ANDI,XORI]
IJ_TYPE = [LDW,STW,BZ,BEQ,JR,HALT]

# Initialize variables and data structures

trace = []                         # trace of instruction
PC = 0                             # Program Counter
idx = 0                            # Current index in trace
current_instruction = 0            # Current Instruction
fetch_instruction = []             # Fetched Instruction
opcode = 0                         # Opcode of the current instruction
output = 0                         # Output value

reg = {}                           # Register dictionary to store register values
temporary_reg = [0] * 32           # Temporary register file
RS = 0                             # Source Register 1
RD = 0                             # Destination Register
RT = 0                             # Source Register 2
Imm = 0                            # Immediate value
src1 = 0                           # Value of source register 1
src2 = 0                           # Value of source register 2
destination = 0                    # Value of destination register

no_of_arithmetic = 0               # Number of arithmetic instructions executed
no_of_logic = 0                    # Number of logic instructions executed
no_of_memory = 0                   # Number of memory access instructions executed
no_of_control = 0                  # Number of control flow instructions executed
total_instruction = 0              # Total number of instructions

memory_dictionary = {}             # Memory dictionary to store memory contents
```

```
mem_addr = 0                        # Memory Address
temp_addr = 0                       # Temporary memory address
branch_flag = []                    # Branch flags
penalty = 0                         # Penalty cycle
halt = 0                            # Flags to indicate halt instruction

"""
    Function name: trace_reader

    Read the trace file and populate the trace list.

    Parameters:
    - path (str): Path to the directory containing the trace file.
    - file_name (str): Name of the trace file.

    Returns:
    - trace (list): List containing the trace of instructions.
    """

def trace_reader(path,file_name):

    global trace
    trace_file = open(os.path.join(path,file_name),"r")
    for read_line in trace_file:
        read_line.strip()
        read_val = int(read_line,16)
        trace.append(read_val)
    return trace

"""
    Function name: two_complement
    Convert a value to its two's complement representation.
    Parameters:
    - value (int): Value to be converted.
    - bit_size (int): Size of the bit representation.
    Returns:
    - value (int): Two's complement representation of the value.
    """

def two_complement(value,bit_size):
    if((value & (1 << (bit_size - 1))) != 0):
        value = value - (1 << bit_size)
        return value
    else:
        return value

"""
Function name: instruction_fetch
    Fetch the next instruction from the trace.
    Parameters:
    - trace (list): List containing the trace of instructions.
    Returns:
    - current_instruction (int): Current instruction fetched from the trace.
    """
```

```
def instruction_fetch(trace):
    global PC,idx,current_instruction,fetch_instruction,halt
    idx = int(PC/4)
    current_instruction = trace[idx]
    fetch_instruction.append(current_instruction)
    if(halt != 1):
        PC += 4
    else:
        return
    return current_instruction

"""
    Function name: instruction_decode
    Decode the current instruction and extract relevant fields.

    Parameters:
    - current_instruction (int): Current instruction to be decoded.
    Returns:
    - opcode (int): Opcode of the current instruction.
    """

def instruction_decode(current_instruction):
    global opcode,RS,RD,RT,Imm,src1,src2,destination,R_TYPE,I_TYPE
    opcode = (current_instruction >> 26) & 63
    if(opcode in R_TYPE):
        RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        RD = (current_instruction >> 11) & 31
        src1 = temporary_reg[RS]
        src2 = temporary_reg[RT]
    elif(opcode in I_TYPE or opcode in IJ_TYPE):
        RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        Imm = current_instruction & 65535
        src1 = temporary_reg[RS]
        destination = temporary_reg[RT]
    return opcode

"""
    Function name: instruction_execute
    Execute the instruction based on its opcode.
    Parameters:
    - opcode (int): Opcode of the current instruction.
    Returns:
    - output (int): Output value produced by the instruction.
    """

def instruction_execute(opcode):
    global PC,RS,RD,RT,Imm,src1,src2,destination
    global no_of_arithmetic,no_of_logic,no_of_memory,no_of_control
    global temp_addr,mem_addr,output
    global branch_flag,halt,penalty
    if(opcode == ADD):
        no_of_arithmetic += 1
        output = src1 + src2
```

```
elif(opcode == ADDI):
    no_of_arithmetic += 1
    output = src1 + two_complement(Imm,16)
elif(opcode == SUB):
    no_of_arithmetic += 1
    output = src1 - src2
elif(opcode == SUBI):
    no_of_arithmetic += 1
    output = src1 - two_complement(Imm,16)
elif(opcode == MUL):
    no_of_arithmetic += 1
    output = src1 * src2
elif(opcode == MULI):
    no_of_arithmetic += 1
    output = src1 * two_complement(Imm,16)
elif(opcode == OR):
    no_of_logic += 1
    output = src1 | src2
elif(opcode == ORI):
    no_of_logic += 1
    output = src1 | Imm
elif(opcode == AND):
    no_of_logic += 1
    output = src1 & src2
elif(opcode == ANDI):
    no_of_logic += 1
    output = src1 & Imm
elif(opcode == XOR):
    no_of_logic += 1
    output = src1 ^ src2
elif(opcode == XORI):
    no_of_logic += 1
    output = src1 ^ Imm
elif(opcode == LDW or opcode == STW):
    no_of_memory += 1
    temp_addr = src1 + two_complement(Imm,16)
    mem_addr = int(temp_addr/4)
elif(opcode == BZ):
    no_of_control += 1
    if(src1 == 0):
        PC = PC - 4
        PC = (PC + (4 * (two_complement(Imm,16))))
        penalty += 2
        branch_flag.append(opcode)
elif(opcode == BEQ):
    no_of_control += 1
    if(src1 == destination):
        PC = PC - 4
        PC = (PC + (4 * (two_complement(Imm,16))))
        penalty += 2
        branch_flag.append(opcode)
elif(opcode == JR):
    no_of_control += 1
    PC = src1
    penalty += 2
```

```
            branch_flag.append(opcode)
        elif(opcode == HALT):
            no_of_control += 1
            halt = 1
            return
        return output

    """
        Function name: memory_access
        Access memory based on the opcode of the instruction.
        Parameters:
        - trace (list): List containing the trace of instructions.
        """

    def memory_access(trace):
        global opcode,memory_dictionary,temporary_reg,mem_addr,output,RT
        if(opcode == LDW):
            output = two_complement(trace[mem_addr],32)
        elif(opcode == STW):
            trace[mem_addr] = temporary_reg[RT]
            memory_dictionary[4 * mem_addr] = trace[mem_addr]

    """
        Function name: write_back
        Write back the output value to the appropriate register.
        Parameters:
        - output (int): Output value produced by the instruction.
        """

    def write_back(output):
        global opcode,temporary_reg,RD,RT
        if(opcode in R_TYPE):
            temporary_reg[RD] = output
            reg[RD] = temporary_reg[RD]
        elif(opcode in I_TYPE or opcode == LDW):
            temporary_reg[RT] = output
            reg[RT] = temporary_reg[RT]

    """
        Function name: show()
        Show the final simulation results.
        """
    def show():
        global
    total_instruction,no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,PC,reg,memor
    y_dictionary
        print("\n---------------------------------------------------------------------")
        print("MIPS-Lite: Functional Simulator")
        print("\n---------------------------------------------------------------------")
        print(r" Total number of instructions            = ",total_instruction)
        print(r" Number of Arithmetic instructions       = ",no_of_arithmetic)
        print(r" Number of Logical instructions          = ",no_of_logic)
        print(r" Number of Memory Acess instructions     = ",no_of_memory)
        print(r" Number of Control Transfer instructions = ",no_of_control)
        print(r" PC: ",PC)
```

```
    for i,j in reg.items():
        print(r" R",i,": ",j)
    for i,j in memory_dictionary.items():
        print(r" Memory : Address: ",i,", Values: ",j)
    print("-----------------------------------------------------------------\n")

"""
    Function name: main
    Main function to execute the simulator.
    """

def main():
    global current_instruction,opcode,output,halt
    global no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,total_instruction
    read_inputs =
trace_reader(r'C:\Users\saksh\OneDrive\Desktop\Spring_2022\ECE_586\Project','final_pro
j_trace.txt')
    while(halt != 1):
        current_instruction = instruction_fetch(read_inputs)
        opcode = instruction_decode(current_instruction)
        output = instruction_execute(opcode)
        memory_access(read_inputs)
        write_back(output)
    total_instruction = no_of_arithmetic + no_of_logic + no_of_memory + no_of_control
    show()

# Execute the main function
main()
```

b. Functional and Timing Simulator without forwarding:

```
"""
Author: Jahnvi Mehta
Computer Architecture_ECE586_Spring2023
Functional simulator + Timing simulator assuming no pipeline forwarding
MIPS-lite

"""

import os
#Define opcode values as per defined in Project_specs

# Arithmetic Instruction
ADD  = 0b000000
SUB  = 0b000010
MUL  = 0b000100
ADDI = 0b000001
SUBI = 0b000011
MULI = 0b000101

# Logical Instruction
OR   = 0b000110
```

```
AND  = 0b001000
XOR  = 0b001010
ORI  = 0b000111
ANDI = 0b001001
XORI = 0b001011

# Memory Access Instruction
LDW  = 0b001100                    # Load the contents of the memory location
STW  = 0b001101                    # Store the contents of the memory location

# Control Flow Instruction
BZ   = 0b001110                    # Branch
BEQ  = 0b001111                    # Branch if equal
JR   = 0b010000                    # Jump to new PC
HALT = 0b010001                    # Stop executing the program

# Instruction Format
R_TYPE = [ADD,SUB,MUL,OR,AND,XOR]
I_TYPE = [ADDI,SUBI,MULI,ORI,ANDI,XORI]
IJ_TYPE = [LDW,STW,BZ,BEQ,JR,HALT]

# Segregating instructions for hazard detection
REG_REG_INS = [ADD,SUB,MUL,OR,AND,XOR,BEQ]
REG_IMM_INS = [ADDI,SUBI,MULI,ORI,ANDI,XORI,LDW,BZ,JR]

# Initialize variables and data structures

trace = []                         # trace of instruction
PC = 0                             # Program Counter
idx = 0                            # Current index in trace
current_instruction = 0            # Current Instruction
fetch_instruction = []             # Fetched Instruction
opcode = 0                         # Opcode of the current instruction
output = 0                         # Output value

reg = {}                           # Register dictionary to store register values
temporary_reg = [0] * 32           # Temporary register file
RS = 0                             # Source Register 1
RD = 0                             # Destination Register
RT = 0                             # Source Register 2
Imm = 0                            # Immediate value
src1 = 0                           # Value of source register 1
src2 = 0                           # Value of source register 2
destination = 0                    # Value of destination register

no_of_arithmetic = 0               # Number of arithmetic instructions executed
no_of_logic = 0                    # Number of logic instructions executed
no_of_memory = 0                   # Number of memory access instructions executed
no_of_control = 0                  # Number of control flow instructions executed
total_instruction = 0              # Total number of instructions
current_ins_count = 0                          # To keep a track of current
instruction count for detecting hazards

memory_dictionary = {}             # Memory dictionary to store memory contents
mem_addr = 0                       # Memory Address
```

```
temp_addr = 0                       # Temporary memory address
branch_flag = []                    # Branch flags
penalty = 0                         # Penalty cycle
halt = 0                            # Flags to indicate halt instruction
total_stalls = 0                    # Total number of stalls encountered
stalls_list = []                              # List of all the stalls encountered
total_hazards = 0                   # Total number of data hazards

"""
    Function name: trace_reader

    Read the trace file and populate the trace list.

    Parameters:
    - path (str): Path to the directory containing the trace file.
    - file_name (str): Name of the trace file.

    Returns:
    - trace (list): List containing the trace of instructions.
    """

def trace_reader(path,file_name):

    global trace
    trace_file = open(os.path.join(path,file_name),"r")
    for read_line in trace_file:
        read_line.strip()
        read_val = int(read_line,16)
        trace.append(read_val)
    return trace

"""
    Function name: two_complement
    Convert a value to its two's complement representation.
    Parameters:
    - value (int): Value to be converted.
    - bit_size (int): Size of the bit representation.
    Returns:
    - value (int): Two's complement representation of the value.
    """

def two_complement(value,bit_size):
    if((value & (1 << (bit_size - 1))) != 0):
        value = value - (1 << bit_size)
        return value
    else:
        return value

"""
Function name: instruction_fetch
    Fetch the next instruction from the trace.
    Parameters:
    - trace (list): List containing the trace of instructions.
    Returns:
    - current_instruction (int): Current instruction fetched from the trace.
```

```
    """

def instruction_fetch(trace):
    global PC,idx,current_instruction,fetch_instruction,halt
    idx = int(PC/4)
    current_instruction = trace[idx]
    fetch_instruction.append(current_instruction)
    if(halt != 1):
        PC += 4
    else:
        return
    return current_instruction

"""
    Function name: instruction_decode
    Decode the current instruction and extract relevant fields.

    Parameters:
    - current_instruction (int): Current instruction to be decoded.
    Returns:
    - opcode (int): Opcode of the current instruction.
    """

def instruction_decode(current_instruction):
    global opcode,RS,RD,RT,Imm,src1,src2,destination,R_TYPE,I_TYPE
    opcode = (current_instruction >> 26) & 63
    if(opcode in R_TYPE):
        RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        RD = (current_instruction >> 11) & 31
        src1 = temporary_reg[RS]
        src2 = temporary_reg[RT]
    elif(opcode in I_TYPE or opcode in IJ_TYPE):
        RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        Imm = current_instruction & 65535
        src1 = temporary_reg[RS]
        destination = temporary_reg[RT]
    return opcode

"""
    Function name: instruction_execute
    Execute the instruction based on its opcode.
    Parameters:
    - opcode (int): Opcode of the current instruction.
    Returns:
    - output (int): Output value produced by the instruction.
    """

def instruction_execute(opcode):
    global PC,RS,RD,RT,Imm,src1,src2,destination
    global no_of_arithmetic,no_of_logic,no_of_memory,no_of_control
    global temp_addr,mem_addr,output
    global branch_flag,halt,penalty
    if(opcode == ADD):
```

```
        no_of_arithmetic += 1
        output = src1 + src2
    elif(opcode == ADDI):
        no_of_arithmetic += 1
        output = src1 + two_complement(Imm,16)
    elif(opcode == SUB):
        no_of_arithmetic += 1
        output = src1 - src2
    elif(opcode == SUBI):
        no_of_arithmetic += 1
        output = src1 - two_complement(Imm,16)
    elif(opcode == MUL):
        no_of_arithmetic += 1
        output = src1 * src2
    elif(opcode == MULI):
        no_of_arithmetic += 1
        output = src1 * two_complement(Imm,16)
    elif(opcode == OR):
        no_of_logic += 1
        output = src1 | src2
    elif(opcode == ORI):
        no_of_logic += 1
        output = src1 | Imm
    elif(opcode == AND):
        no_of_logic += 1
        output = src1 & src2
    elif(opcode == ANDI):
        no_of_logic += 1
        output = src1 & Imm
    elif(opcode == XOR):
        no_of_logic += 1
        output = src1 ^ src2
    elif(opcode == XORI):
        no_of_logic += 1
        output = src1 ^ Imm
    elif(opcode == LDW or opcode == STW):
        no_of_memory += 1
        temp_addr = src1 + two_complement(Imm,16)
        mem_addr = int(temp_addr/4)
    elif(opcode == BZ):
        no_of_control += 1
        if(src1 == 0):
            PC = PC - 4
            PC = (PC + (4 * (two_complement(Imm,16))))
            penalty += 2
            branch_flag.append(opcode)
    elif(opcode == BEQ):
        no_of_control += 1
        if(src1 == destination):
            PC = PC - 4
            PC = (PC + (4 * (two_complement(Imm,16))))
            penalty += 2
            branch_flag.append(opcode)
    elif(opcode == JR):
        no_of_control += 1
```

```
        PC = src1
        penalty += 2
        branch_flag.append(opcode)
    elif(opcode == HALT):
        no_of_control += 1
        halt = 1
        return
    return output

"""
    Function name: memory_access
    Access memory based on the opcode of the instruction.
    Parameters:
    - trace (list): List containing the trace of instructions.
    """

def memory_access(trace):
    global opcode,memory_dictionary,temporary_reg,mem_addr,output,RT
    if(opcode == LDW):
        output = two_complement(trace[mem_addr],32)
    elif(opcode == STW):
        trace[mem_addr] = temporary_reg[RT]
        memory_dictionary[4 * mem_addr] = trace[mem_addr]

"""
    Function name: write_back
    Write back the output value to the appropriate register.
    Parameters:
    - output (int): Output value produced by the instruction.
    """

def write_back(output):
    global opcode,temporary_reg,RD,RT
    if(opcode in R_TYPE):
        temporary_reg[RD] = output
        reg[RD] = temporary_reg[RD]
    elif(opcode in I_TYPE or opcode == LDW):
        temporary_reg[RT] = output
        reg[RT] = temporary_reg[RT]

def detect_hazards_without_forwarding():
    global current_ins_count,total_stalls,total_hazards,stalls_list,fetch_instruction
    if(current_ins_count == 0):
        return
    elif(current_ins_count == 1):
        if(((fetch_instruction[current_ins_count] >> 26) & 63) in REG_REG_INS):
            if(((fetch_instruction[current_ins_count-1] >> 26) & 63) in R_TYPE):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31)) or
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31))):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
```

```
            elif(((fetch_instruction[current_ins_count-1] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-1] >> 26) & 63) == LDW):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31)) or
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31))):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
        elif(((fetch_instruction[current_ins_count] >> 26) & 63) in REG_IMM_INS):
            if(((fetch_instruction[current_ins_count-1] >> 26) & 63) in R_TYPE):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31)):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            elif(((fetch_instruction[current_ins_count-1] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-1] >> 26) & 63) == LDW):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31)):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
    else:
        if(((fetch_instruction[current_ins_count-1] >> 26) & 63) in branch_flag):
            return
        elif(((fetch_instruction[current_ins_count] >> 26) & 63) in REG_REG_INS):
            if(((fetch_instruction[current_ins_count-1] >> 26) & 63) in R_TYPE):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31)) or
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31))):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            elif(((fetch_instruction[current_ins_count-1] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-1] >> 26) & 63) == LDW):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31)) or
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31))):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            if((current_ins_count-1) in stalls_list):
                return
            elif(((fetch_instruction[current_ins_count-2] >> 26) & 63) in R_TYPE):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-2] >> 11) & 31)) or
```

```
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-2] >> 11) & 31))):
                    total_stalls += 1
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            elif(((fetch_instruction[current_ins_count-2] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-2] >> 26) & 63) == LDW):
                if((((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-2] >> 16) & 31)) or
(((fetch_instruction[current_ins_count] >> 16) & 31) ==
((fetch_instruction[current_ins_count-2] >> 16) & 31))):
                    total_stalls += 1
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
        elif(((fetch_instruction[current_ins_count] >> 26) & 63) in REG_IMM_INS):
            if(((fetch_instruction[current_ins_count-1] >> 26) & 63) in R_TYPE):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 11) & 31)):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            elif(((fetch_instruction[current_ins_count-1] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-1] >> 26) & 63) == LDW):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-1] >> 16) & 31)):
                    total_stalls += 2
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            if(fetch_instruction[current_ins_count-1] in stalls_list):
                return
            elif(((fetch_instruction[current_ins_count-2] >> 26) & 63) in R_TYPE):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-2] >> 11) & 31)):
                    total_stalls += 1
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return
            elif(((fetch_instruction[current_ins_count-2] >> 26) & 63) in I_TYPE or
((fetch_instruction[current_ins_count-2] >> 26) & 63) == LDW):
                if(((fetch_instruction[current_ins_count] >> 21) & 31) ==
((fetch_instruction[current_ins_count-2] >> 16) & 31)):
                    total_stalls += 1
                    total_hazards += 1
                    stalls_list.append(fetch_instruction[current_ins_count])
                    return


"""
    Function name: show()
    Show the final simulation results.
    """
```

```
def show():
    global
total_instruction,no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,PC,reg,memor
y_dictionary
    print("\n------------------------------------------------------------------")
    print("MIPS-Lite: Functional & Timing Simulator without Forwarding")
    print("\n------------------------------------------------------------------")
    print(r" Total number of instructions            = ",total_instruction)
    print(r" Number of Arithmetic instructions       = ",no_of_arithmetic)
    print(r" Number of Logical instructions          = ",no_of_logic)
    print(r" Number of Memory Acess instructions     = ",no_of_memory)
    print(r" Number of Control Transfer instructions = ",no_of_control)
    print(r" PC: ",PC)
    print(r"-> Total number of stalls = ",total_stalls)
    print(r"-> Number of Data Hazards = ",total_hazards)
    print(r"-> Average Stall Penalty  = ",round((total_stalls/total_hazards),4))
    for i,j in reg.items():
        print(r" R",i,": ",j)
    for i,j in memory_dictionary.items():
        print(r" Memory : Address: ",i,", Values: ",j)
    print("------------------------------------------------------------------\n")

"""
    Function name: main
    Main function to execute the simulator.
    """

def main():
    global current_instruction,opcode,output,halt,current_ins_count
    global no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,total_instruction
    read_inputs =
trace_reader(r'C:\Users\saksh\OneDrive\Desktop\Spring_2022\ECE_586\Project','final_pro
j_trace.txt')
    while(halt != 1):
        current_instruction = instruction_fetch(read_inputs)
        opcode = instruction_decode(current_instruction)
        detect_hazards_without_forwarding()
        current_ins_count += 1
        output = instruction_execute(opcode)
        memory_access(read_inputs)
        write_back(output)
    total_instruction = no_of_arithmetic + no_of_logic + no_of_memory + no_of_control
    show()

# Execute the main function
main()
```

c. <u>Functional and Timing Simulator with forwarding:</u>

```
"""
Author: Sakshi Garge, Muskan Pashine
Computer Architecture_ECE586_Spring2023
Functional simulator + Timing simulator with pipeline forwarding
```

```
MIPS-lite

"""
import os

#Define opcode values as per defined in Project_specs

# Arithmetic Instruction
ADD  = 0b000000
SUB  = 0b000010
MUL  = 0b000100
ADDI = 0b000001
SUBI = 0b000011
MULI = 0b000101

# Logical Instruction
OR   = 0b000110
AND  = 0b001000
XOR  = 0b001010
ORI  = 0b000111
ANDI = 0b001001
XORI = 0b001011

# Memory Access Instruction
LDW  = 0b001100                    # Load the contents of the memory location
STW  = 0b001101                    # Store the contents of the memory location

# Control Flow Instruction
BZ   = 0b001110                    # Branch
BEQ  = 0b001111                    # Branch if equal
JR   = 0b010000                    # Jump to new PC
HALT = 0b010001                    # Stop executing the program

# Instruction Format
R_TYPE = [ADD,SUB,MUL,OR,AND,XOR]
I_TYPE = [ADDI,SUBI,MULI,ORI,ANDI,XORI]
IJ_TYPE = [LDW,STW,BZ,BEQ,JR,HALT]

# Segregating instructions for hazard detection
INS_1_SRC = [ADDI,SUBI,MULI,ORI,ANDI,XORI,LDW,BZ,JR]
INS_2_SRC = [ADD,SUB,MUL,OR,AND,XOR,BEQ]

# Initialize variables and data structures

trace = []                         # trace of instruction
PC = 0                             # Program Counter
idx = 0                            # Current index in trace
current_instruction = 0            # Current Instruction
fetch_instruction = []             # Fetched Instruction
instr_count = 0                      #instr count
opcode = 0                         # Opcode of the current instruction
output = 0                         # Output value

reg = {}                           # Register dictionary to store register values
temporary_reg = [0] * 32           # Temporary register file
```

```
RS = 0                                  # Source Register 1
RD = 0                                  # Destination Register
RT = 0                                  # Source Register 2
Imm = 0                                 # Immediate value
src1 = 0                                # Value of source register 1
src2 = 0                                # Value of source register 2
destination = 0                         # Value of destination register

no_of_arithmetic = 0                    # Number of arithmetic instructions executed
no_of_logic = 0                         # Number of logic instructions executed
no_of_memory = 0                        # Number of memory access instructions executed
no_of_control = 0                       # Number of control flow instructions executed
total_instruction = 0                   # Total number of instructions
no_of_stalls = 0
no_of_data_hazards = 0
total_ins = 0
total_clock_cycles = 0

memory_dictionary = {}                  # Memory dictionary to store memory contents
mem_addr = 0                            # Memory Address
temp_addr = 0                           # Temporary memory address
branch_flag = []                        # Branch flags
penalty = 0                             # Penalty cycle
halt = 0                                # Flags to indicate halt instruction
stalls = []

"""
    Function name: trace_reader
    Read the trace file and populate the trace list.

"""

def trace_reader(path,file_name):

    global trace
    trace_file = open(os.path.join(path,file_name),"r")
     for read_line in trace_file:
         read_line.strip()
         read_val = int(read_line,16)
         trace.append(read_val)
    return trace

"""
    Function name: two_complement
    Convert a value to its two's complement representation.

"""

def two_complement(value,bit_size):
    if((value & (1 << (bit_size - 1))) != 0):
         value = value - (1 << bit_size)
         return value
    else:
          return value
```

```
"""
Function name: instruction_fetch
    Fetch the next instruction from the trace.
"""

def instruction_fetch(trace):
    global PC,idx,current_instruction,fetch_instruction,halt
    idx = int(PC/4)
    current_instruction = trace[idx]
    fetch_instruction.append(current_instruction)
    if(halt != 1):
        PC += 4
    else:
        return
    return current_instruction

"""
    Function name: instruction_decode
    Decode the current instruction and extract relevant fields.
"""

def instruction_decode(current_instruction):
    global opcode,RS,RD,RT,Imm,src1,src2,destination,R_TYPE,I_TYPE
    opcode = (current_instruction >> 26) & 63
    if(opcode in R_TYPE):
         RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        RD = (current_instruction >> 11) & 31
        src1 = temporary_reg[RS]
        src2 = temporary_reg[RT]
    elif(opcode in I_TYPE or opcode in IJ_TYPE):
        RS = (current_instruction >> 21) & 31
        RT = (current_instruction >> 16) & 31
        Imm = current_instruction & 65535
        src1 = temporary_reg[RS]
        destination = temporary_reg[RT]
    return opcode

"""
    Function name: instruction_execute
    Execute the instruction based on its opcode.

"""

def instruction_execute(opcode):
    global PC,RS,RD,RT,Imm,src1,src2,destination
    global no_of_arithmetic,no_of_logic,no_of_memory,no_of_control
    global temp_addr,mem_addr,output
    global branch_flag,halt,penalty
    if(opcode == ADD):
        no_of_arithmetic += 1
        output = src1 + src2
     elif(opcode == ADDI):
        no_of_arithmetic += 1
        output = src1 + two_complement(Imm,16)
```

```
elif(opcode == SUB):
    no_of_arithmetic += 1
    output = src1 - src2
elif(opcode == SUBI):
    no_of_arithmetic += 1
    output = src1 - two_complement(Imm,16)
elif(opcode == MUL):
    no_of_arithmetic += 1
    output = src1 * src2
elif(opcode == MULI):
    no_of_arithmetic += 1
    output = src1 * two_complement(Imm,16)
elif(opcode == OR):
    no_of_logic += 1
    output = src1 | src2
elif(opcode == ORI):
    no_of_logic += 1
    output = src1 | Imm
elif(opcode == AND):
    no_of_logic += 1
    output = src1 & src2
elif(opcode == ANDI):
    no_of_logic += 1
    output = src1 & Imm
elif(opcode == XOR):
    no_of_logic += 1
    output = src1 ^ src2
elif(opcode == XORI):
    no_of_logic += 1
    output = src1 ^ Imm
elif(opcode == LDW or opcode == STW):
    no_of_memory += 1
    temp_addr = src1 + two_complement(Imm,16)
    mem_addr = int(temp_addr/4)
elif(opcode == BZ):
    no_of_control += 1
    if(src1 == 0):
        PC = PC - 4
        PC = (PC + (4 * (two_complement(Imm,16))))
        penalty += 2
        branch_flag.append(opcode)
elif(opcode == BEQ):
     no_of_control += 1
    if(src1 == destination):
        PC = PC - 4
        PC = (PC + (4 * (two_complement(Imm,16))))
        penalty += 2
        branch_flag.append(opcode)
elif(opcode == JR):
    no_of_control += 1
    PC = src1
    penalty += 2
    branch_flag.append(opcode)
elif(opcode == HALT):
    no_of_control += 1
```

```
        halt = 1
        return
    return output

"""
     Function name: memory_access
    Access memory based on the opcode of the instruction.

"""

def memory_access(trace):
    global opcode,memory_dictionary,temporary_reg,mem_addr,output,RT
    if(opcode == LDW):
        output = two_complement(trace[mem_addr],32)
    elif(opcode == STW):
        trace[mem_addr] = temporary_reg[RT]
        memory_dictionary[4 * mem_addr] = trace[mem_addr]

"""
    Function name: write_back
    Write back the output value to the appropriate register.

"""

def write_back(output):
    global opcode,temporary_reg,RD,RT
    if(opcode in R_TYPE):
        temporary_reg[RD] = output
        reg[RD] = temporary_reg[RD]
    elif(opcode in I_TYPE or opcode == LDW):
        temporary_reg[RT] = output
        reg[RT] = temporary_reg[RT]


def detect_hazards_forwarding():
    global no_of_stalls, no_of_data_hazards, stalls, fetch_instruction, instr_count

    # Check if there are any instructions
    if instr_count == 0:
        return

    # Check if there is only one instruction
    elif instr_count == 1:
        # Check if the current instruction has two source operands
        if ((fetch_instruction[instr_count] >> 26) & 63) in INS_2_SRC:
            # Check if the previous instruction is a load instruction
            if ((fetch_instruction[instr_count-1] >> 26) & 63) == LDW:
                # Check if the destination register of the current instruction matches
                  # either of the source registers of the previous instruction
                if (((fetch_instruction[instr_count] >> 21) & 31) ==
((fetch_instruction[instr_count-1] >> 16) & 31)) or (((fetch_instruction[instr_count]
>> 16) & 31) == ((fetch_instruction[instr_count-1] >> 16) & 31)):
                    no_of_stalls += 1
                    no_of_data_hazards += 1
                    stalls.append(fetch_instruction[instr_count])
```

```
                    return

        # Check if the current instruction has one source operand
        elif ((fetch_instruction[instr_count] >> 26) & 63) in INS_1_SRC:
            # Check if the previous instruction is a load instruction
            if ((fetch_instruction[instr_count-1] >> 26) & 63) == LDW:
                # Check if the source register of the current instruction matches
                # the destination register of the previous instruction
                if ((fetch_instruction[instr_count] >> 21) & 31) ==
((fetch_instruction[instr_count-1] >> 16) & 31):
                    no_of_stalls += 1
                    no_of_data_hazards += 1
                    stalls.append(fetch_instruction[instr_count])
                    return

    # For instructions other than the first two
    else:
        # Check if the previous instruction is a branch instruction
        if ((fetch_instruction[instr_count-1] >> 26) & 63) in branch_flag:
            return

        # Check if the current instruction has two source operands
        elif ((fetch_instruction[instr_count] >> 26) & 63) in INS_2_SRC:
            # Check if the previous instruction is a load instruction
            if ((fetch_instruction[instr_count-1] >> 26) & 63) == LDW:
                # Check if the destination register of the current instruction matches
                # either of the source registers of the previous instruction
                if (((fetch_instruction[instr_count] >> 21) & 31) ==
((fetch_instruction[instr_count-1] >> 16) & 31)) or (((fetch_instruction[instr_count]
>> 16) & 31) == ((fetch_instruction[instr_count-1] >> 16) & 31)):
                    no_of_stalls += 1
                    no_of_data_hazards += 1
                    stalls.append(fetch_instruction[instr_count])
                    return

        # Check if the current instruction has one source operand
        elif ((fetch_instruction[instr_count] >> 26) & 63) in INS_1_SRC:
            # Check if the previous instruction is a load instruction
            if ((fetch_instruction[instr_count-1] >> 26) & 63) == LDW:
                # Check if the source register of the previous instruction
                if((((fetch_instruction[instr_count] >> 21) & 31) ==
((fetch_instruction[instr_count-1] >> 16) & 31)) or (((fetch_instruction[instr_count]
>> 16) & 31) == ((fetch_instruction[instr_count-1] >> 16) & 31))):
                    no_of_stalls += 1
                    no_of_data_hazards += 1
                    stalls.append(fetch_instruction[instr_count])
                    return

def show():
    global
total_instruction,no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,PC,reg,stall
s,memory_dictionary,total_clock_cycles,halt
    print("MIPS-Lite: Functional & Timing Simulator with Forwarding")
    print("\n-----------------------------------------------------------------")
    print("----------------- Instruction Counts ----------------------------")
```

```
    print(r"-> Total number of instructions  = ",total_instruction)
    print(r"-> Arithmetic instructions       = ",no_of_arithmetic)
    print(r"-> Logical instructions          = ",no_of_logic)
    print(r"-> Memory Acess instructions     = ",no_of_memory)
    print(r"-> Control Transfer instructions = ",no_of_control)
    print("-----------------------------------------------------------------\n")
    print("------------------ Final Register State -----------------------------")
    print(r"-> PC: ",PC)
    for i,j in reg.items():
        print(r"-> R",i,": ",j)
    print(r"-> Total number of stalls = ",stalls)
    print(r"-> Total number of stalls = ",no_of_stalls)
    print(r"-> Number of Data hazards = ",no_of_data_hazards)
    for i,j in memory_dictionary.items():
        print(r"-> Memory => Address: ",i,", Contents: ",j)
    print("-----------------------------------------------------------------\n")
    print("-------------------- Timing Simulator -----------------------------")
    print(r"-> Total number of clock cycles = ",total_clock_cycles)
    if(halt == 1):
        print(r"-> Program Halted!")
    print("-----------------------------------------------------------------\n")

def main():
    global current_instruction,opcode,output,halt,instr_count
    global
no_of_arithmetic,no_of_logic,no_of_memory,no_of_control,total_instruction,total_clock_
cycles
    read_inputs =
trace_reader(r'C:\Users\saksh\OneDrive\Desktop\Spring_2022\ECE_586\Project','final_pro
j_trace.txt')
    while(halt != 1):
        current_instruction = instruction_fetch(read_inputs)
        opcode = instruction_decode(current_instruction)
        detect_hazards_forwarding()
        instr_count += 1
         output = instruction_execute(opcode)
        memory_access(read_inputs)
        write_back(output)
    total_instruction = no_of_arithmetic + no_of_logic + no_of_memory + no_of_control
    total_clock_cycles = total_instruction + penalty + no_of_stalls + 4
    show()

# Execute the main function
main()
```

5.Output Transcripts:

All the output transcripts are for the 'final_proj_trace.txt' trace file that was shared for the demo.

a. Functional Simulator:

```
---------------------------------------------------------------------
MIPS-Lite: Functional Simulator

---------------------------------------------------------------------
 Total number of instructions          =  911
 Number of Arithmetic instructions     =  375
 Number of Logical instructions        =  61
 Number of Memory Acess instructions   =  300
 Number of Control Transfer instructions =  175
 PC:  112
 R 11 :  1044
 R 12 :  1836
 R 13 :  2640
 R 18 :  3440
 R 19 :  -1
 R 20 :  -2
 R 22 :  76
 R 23 :  3
 R 24 :  -1
 R 14 :  25
 R 15 :  -188
 R 17 :  29
 R 16 :  213
 R 21 :  -1
 R 25 :  3
 Memory : Address:  2400 , Values:  2
 Memory : Address:  2404 , Values:  4
 Memory : Address:  2408 , Values:  6
 Memory : Address:  2412 , Values:  8
 Memory : Address:  2416 , Values:  10
 Memory : Address:  2420 , Values:  12
 Memory : Address:  2424 , Values:  14
 Memory : Address:  2428 , Values:  16
 Memory : Address:  2432 , Values:  18
 Memory : Address:  2436 , Values:  29
 Memory : Address:  2440 , Values:  22
 Memory : Address:  2444 , Values:  24
 Memory : Address:  2448 , Values:  26
 Memory : Address:  2452 , Values:  28
 Memory : Address:  2456 , Values:  30
 Memory : Address:  2460 , Values:  32
 Memory : Address:  2464 , Values:  34
 Memory : Address:  2468 , Values:  36
 Memory : Address:  2472 , Values:  38
```

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

---

b. Functional and Timing Simulator without forwarding:

MIPS-Lite: Functional & Timing Simulator without Forwarding

```
---------------------------------------------------------------------
 Total number of instructions          =  911
 Number of Arithmetic instructions     =  375
 Number of Logical instructions        =  61
 Number of Memory Acess instructions   =  300
 Number of Control Transfer instructions =  175
 PC:  112
-> Total number of stalls =  554
-> Number of Data Hazards =  307
-> Average Stall Penalty  =  1.8046
 R 11 :  1044
 R 12 :  1836
 R 13 :  2640
 R 18 :  3440
 R 19 :  -1
 R 20 :  -2
 R 22 :  76
 R 23 :  3
 R 24 :  -1
 R 14 :  25
 R 15 :  -188
 R 17 :  29
 R 16 :  213
 R 21 :  -1
 R 25 :  3
 Memory : Address:  2400 , Values:  2
 Memory : Address:  2404 , Values:  4
 Memory : Address:  2408 , Values:  6
 Memory : Address:  2412 , Values:  8
 Memory : Address:  2416 , Values:  10
 Memory : Address:  2420 , Values:  12
 Memory : Address:  2424 , Values:  14
 Memory : Address:  2428 , Values:  16
 Memory : Address:  2432 , Values:  18
 Memory : Address:  2436 , Values:  29
 Memory : Address:  2440 , Values:  22
 Memory : Address:  2444 , Values:  24
 Memory : Address:  2448 , Values:  26
 Memory : Address:  2452 , Values:  28
 Memory : Address:  2456 , Values:  30
 Memory : Address:  2460 , Values:  32
 Memory : Address:  2464 , Values:  34
 Memory : Address:  2468 , Values:  36
 Memory : Address:  2628 , Values:  16
 Memory : Address:  2632 , Values:  18
 Memory : Address:  2636 , Values:  29
---------------------------------------------------------------------
```

c. Functional and Timing Simulator with forwarding:

MIPS-Lite: Functional & Timing Simulator with Forwarding

------------------ Instruction Counts ------------------------------
-> Total number of instructions = 911
-> Arithmetic instructions = 375
-> Logical instructions = 61
-> Memory Acess instructions = 300
-> Control Transfer instructions = 175

------------------ Final Register State ---------------------------
-> PC: 112
-> R 11 : 1044
-> R 12 : 1836
-> R 13 : 2640
-> R 18 : 3440
-> R 19 : -1
-> R 20 : -2
-> R 22 : 76
-> R 23 : 3
-> R 24 : -1
-> R 14 : 25
-> R 15 : -188
-> R 17 : 29
-> R 16 : 213
-> R 21 : -1
-> R 25 : 3
-> Total number of stalls = [992018416, 992018416]
-> Total number of stalls = 60
-> Number of Data hazards = 60
-> Memory => Address: 2400 , Contents: 2
-> Memory => Address: 2404 , Contents: 4
-> Memory => Address: 2408 , Contents: 6
-> Memory => Address: 2412 , Contents: 8
-> Memory => Address: 2416 , Contents: 10
-> Memory => Address: 2420 , Contents: 12
-> Memory => Address: 2424 , Contents: 14

-> Memory => Address: 2428 , Contents: 16
-> Memory => Address: 2432 , Contents: 18
-> Memory => Address: 2436 , Contents: 29
-> Memory => Address: 2440 , Contents: 22
-> Memory => Address: 2444 , Contents: 24
-> Memory => Address: 2448 , Contents: 26
-> Memory => Address: 2452 , Contents: 28
-> Memory => Address: 2456 , Contents: 30
-> Memory => Address: 2460 , Contents: 32
-> Memory => Address: 2464 , Contents: 34
-> Memory => Address: 2468 , Contents: 36
-> Memory => Address: 2472 , Contents: 38
-> Memory => Address: 2476 , Contents: 59
-> Memory => Address: 2480 , Contents: 42
-> Memory => Address: 2484 , Contents: 44
-> Memory => Address: 2488 , Contents: 46
---------- Timing Simulator ----------
-> Total number of clock cycles = 1213
-> Program Halted!
