

Portland State University
ECE 486/586: Computer Architecture
Spring 2025
Final Project Report

By Group 16:
Alex Jain
Brandon Duong
Fernando Calderon
Nitin Suryadevara

T.A. Kai Roy
06/05/2025

Table of Contents

Introduction	4
Assumptions	5
Memory Trace:	5
Instruction Types:	5
Functional Simulator:	7
Timing Simulator:	7
Procedure	7
I. Internal Design Documentation	7
Pseudocode Script:	8
Functional Simulator (High Level)	12
II. Design Decisions	13
Internal Design of the MIPS-lite Simulator	13
Address Breakdown:	13
Trace Reader Structure:	14
Instruction Decoder Structure:	15
Functional Simulator Structure:	18
Timing Simulator with No Forwarding Structure:	19
Timing Simulator with Forwarding Structure:	21
Design Choices	22
Function Descriptions	23
MIPS-Lite Simulator Source Code/Associated Modules	25
Results - Sample Trace	26
Results - Final Project Trace	31
Troubleshooting - Sample Trace	35
Troubleshooting - Final Project Trace	35
Roles & Responsibilities	36
Conclusion	36

ECE 486 Final Project Design Report

Introduction

The objective of the final project for ECE-486 is to design and implement a complete MIPS-lite simulator that models both the functional behavior of a simplified MIPS instruction set and the timing nuances of an in-order, 5-stage pipeline. The simulator operates by reading a provided memory image and supports a complement of R-type and I-type instructions in the arithmetic, logical, memory, and control-flow fields. The simulator also models two distinct pipeline configurations: one without any forwarding/bypassing hardware (No Forwarding), and one with full forwarding capability (With Forwarding). The simulator produces two main types of output, it will report (a) the correct machine-state updates which includes the final register and memory contents – via a functional simulator mode, and (b) detailed clock-cycle accounting and stall cycle counter utilizing a stall in regards to realistic stall and flush scenarios via a timing simulator mode.

The Simulator is written in C and is compiled/runs on Linux via GCC. Instructions are fetched from and written back to memory, which is modeled as a simple array of 32-bit words. The register file is modeled as a 32-entry array of signed 32-bit integers. When decoding instructions, immediates and offsets are interpreted as signed two's complement values to ensure that arithmetic operations and branches with negative immediates behave correctly. The deliverables for this project include a report with design documentation, source code, a testbench, simulation results, and a roles and responsibilities document outlining the contributions of each team member.

The deliverables for this project include three versions of the code. One version that reads a memory image, simulates every MIPS-lite instruction, and outputs the final machine state and instruction-type frequencies. A second version is a pipeline timing simulator without forwarding that will report the total clock cycle count, number of data hazards, and final machine state from the functional simulator. And lastly a third version that is a pipeline timing simulator with forwarding that also reports the total clock cycle count, number of data hazards, and final machine state.

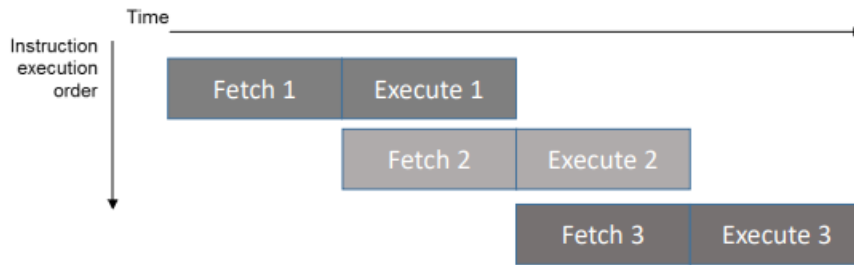


Fig [1]: Simple example of Pipelining.

Figure 1 above, sourced from Lecture 5B - Slide 2, shows a simple pipeline example. It represents the instruction flow that the project's timing simulator was meant to follow.

Assumptions

The following information details our assumptions made when we began the project and were in the project specification document.

Memory Trace:

- The memory image will represent the initial state of the simulated program.
- All memory traces will be in a text file (.txt format).
- Each line represents one word (4 bytes) in hexadecimal.
- All addresses start at "0" and increase by "4".
- All data in the trace file is in Big Endian format.
- All memory traces will not exceed 1024 lines (4 kB).

Instruction Types:

- Only two types of instructions will be covered: R-Type and I-Type.
- R-Type handles ADD, SUB, MUL, OR, AND, XOR instructions.
- I-Type handles ADDI, SUBI, MULI, ORI, ANDI, XORI, LDW, STW, BZ, BEQ, JR, HALT.
- Figure 2 is the instruction format (see below).
- Arithmetic instructions have the following format and opcodes:
 - ADD (Rd, Rs, Rt) Opcode: 000000
 - ADDI (Rt, Rs, Imm) Opcode: 000001
 - SUB (Rd, Rs, Rt) Opcode: 000010
 - SUBI (Rt, Rs, Imm) Opcode: 000011
 - MUL (Rd, Rs, Rt) Opcode: 000100

- MULI (Rt, Rs, Imm) Opcode: 000101
- Logical instructions have the following format and opcodes:
 - OR (Rd, Rs, Rt) Opcode: 000110
 - ORI (Rt, Rs, Imm) Opcode: 000111
 - AND (Rd, Rs, Rt) Opcode: 001000
 - ANDI (Rt, Rs, Imm) Opcode: 001001
 - XOR (Rd, Rs, Rt) Opcode: 001010
 - XORI (Rt, Rs, Imm) Opcode: 001011
- Memory Access instructions have the following format and opcodes:
 - LDW (Rt, Rs, Imm) Opcode: 001100
 - STW (Rt, Rs, Imm) Opcode: 001101
- Control Flow instructions have the following format and opcodes:
 - BZ (Rs, X) Opcode: 001110
 - X is the x'th instruction from the current instruction read.
 - BEQ (Rs, Rt, X) Opcode: 001111
 - X is the instruction branched to.
 - JR (Rs) Opcode: 010000
 - Jumps to PC at Register RS's contents.
 - HALT Opcode: 010001
 - Stop program execution

(i) R-type format:



(ii) I-type format:



Fig [2]: Instruction type bit-fields.

Functional Simulator:

- The Machine state has these three components:
 - Program Counter.
 - General Purpose Registers from R1 to R31.
 - Memory.
- Initial state of the memory is contained in the memory image.
- Changes to the program state occur when decoding and reading instructions from the memory image in which its impact will drive the change.
- The simulator doesn't stop unless a HALT instruction is encountered.

Timing Simulator:

- Pipeline(s) will capture flow of instructions through 5-stages.
- Will be capable of detecting hazards and accounting stall cycles.
- Throughput is 1 instruction per clock cycle save for RAW Hazard or Taken Branch
- All instructions take 5 cycles to finish.
- No WAR/WAR hazards in instruction set and pipeline combination.
- When a register read and register write of the same register occurs in the same clock cycle, write data is written in the first half cycle, and new contents of the register read in the second half of the cycle.
- Simulator takes an always-not-taken branch predictor.
- Assume pipeline timing where the branch target and branch condition at the end of EX stage is figured out, and ensure instructions are flushed from pipeline.
- Global clock counter, with an array length of 5 and a circular buffer of 5 entries in which the buffer corresponds to an instruction in the pipeline.

Procedure

Internal Design Documentation

[Roles and Responsibilities](#) were assigned initially by examining the “Suggested Project Breakdown” in which those tasks were broken down and given to group members for more manageable pieces for ease of processing and developing. We used a shared Github repository to work on the shared script simultaneously, separated into subdirectories into each part of the project for group members to test and validate without breaking the main combined script in the repo. Additionally in the repo there were

subdirectories of “working versions” which were the combined versions of the various scripts developed for a combined iterative test and debugging.

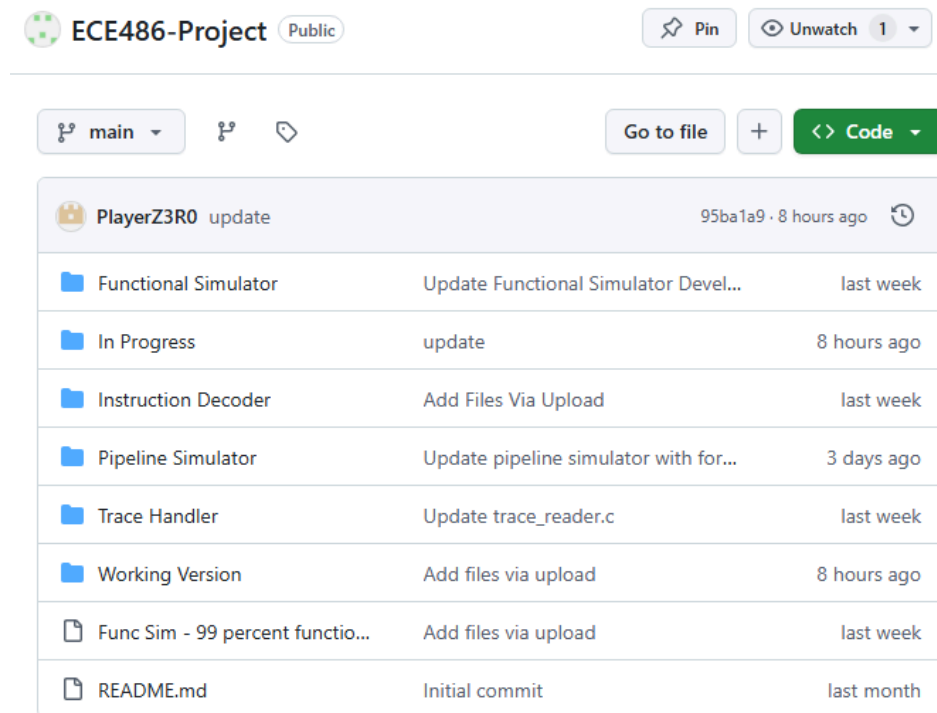


Fig [3]: Sample screenshot of shared Github repository.

To understand the project’s process moving forward, we developed a pseudo code script that would clarify the structure and flow of how each coding block would appear. It was developed to better understand how to develop a flow for the firmware when editing and doing debugging.

Pseudocode Script:

```
// Legend:
// "rawLines"    - Lines read from the input file (each line is one instruction in hex text)
// "binaryLines" - Same instructions converted into binary numbers we can work with
// "instructions" - A list of simple instruction objects, each telling us "what to do" and "what data to use"
// "state"       - The processor's current status: where we are in the program, the values in its registers, and its memory contents
// "counts"      - How many times we did each kind of instruction (e.g., arithmetic, memory access, branches)
// "hazards"     - Places where one instruction needed data from another and had to wait
```



```

// "cycles"    – How many clock ticks it took to finish everything
// "speedup"   – How many times faster the "with shortcuts" version ran compared to the basic version

function Main():
    // 1. Load the instructions as text
    rawLines := readLinesFromFile("memory_image.txt")
    // 2. Turn each line of hex text into binary numbers
    binaryLines := hexListToBinary(rawLines)
    // 3. Break each binary number into a simple instruction object
    instructions := parseBinaryToInstructions(binaryLines)

    // 4. Run a straightforward, one-at-a-time simulation
    (state, counts) := simpleSimulation(instructions)

    // 5. Simulate a 5-step pipeline WITHOUT any data shortcuts
    (cyclesNoOpt, hazardsNoOpt) := pipelineSimulation(instructions, useForwarding=false)

    // 6. Simulate the same pipeline WITH data shortcuts (forwarding)
    (cyclesOpt, hazardsOpt) := pipelineSimulation(instructions, useForwarding=true)

    // 7. Compare how much faster the forwarding version is
    speedup := cyclesNoOpt / cyclesOpt

    // 8. Produce the final report for the project
    writeReport(state, counts, hazardsNoOpt, hazardsOpt, cyclesNoOpt, cyclesOpt, speedup)

//-----
// Support function details below
// (Each is written so someone new to MIPS can follow)

// Read every line from a text file into a list
function readLinesFromFile(filename):
    open filename
    rawLines := empty list
    for each line in file:
        trimmed := removeWhitespace(line)
        rawLines.append(trimmed)
    return rawLines

// Convert a list of hex strings into binary representations
function hexListToBinary(hexList):
    binaryLines := empty list
    for each hexStr in hexList:

```

```

    binValue := convertHexStringToBinary(hexStr)
    binaryLines.append(binValue)
return binaryLines

// Turn each binary value into an instruction object
function parseBinaryToInstructions(binaryLines):
    instructions := empty list
    for each bits in binaryLines:
        opCode := first 6 bits of bits
        if opCode means "register-style":
            // E.g., add two registers, store in a third
            rs := next 5 bits
            rt := next 5 bits
            rd := next 5 bits
            instructions.append({type:"R", opCode, rs, rt, rd})
        else:
            // E.g., load/store or immediate arithmetic
            rs := next 5 bits
            rt := next 5 bits
            imm := last 16 bits
            instructions.append({type:"I", opCode, rs, rt, imm})
    return instructions

// Run each instruction one after another, updating PC, registers, and memory
function simpleSimulation(instructions):
    PC := 0 // Start at the first instruction
    regs := array of 32 zeros // All registers clear
    mem := array of 1024 words loaded from file
    counts := {arithmetic:0, memory:0, control:0, logical:0}

    loop forever:
        instr := instructions[PC/4] // Find current instruction
        if instr is arithmetic:
            perform the math, store result in regs
            counts.arithmetic += 1
        else if instr is logical:
            perform bitwise operation, update regs
            counts.logical += 1
        else if instr is load or store:
            read or write mem using regs and imm
            counts.memory += 1
        else if instr is branch or HALT:
            change PC or stop simulation
            counts.control += 1

```

```

    if instr was HALT:
        break // stop the loop
    else if instr did not jump:
        PC += 4 // move to next instruction
    end loop

return (state={PC, regs, mem}, counts)

// Simulate a 5-stage pipeline:
// stages are Fetch → Decode → Execute → Memory → WriteBack
// “useForwarding” decides whether we allow data to skip ahead or not
function pipelineSimulation(instructions, useForwarding):
    clock := 0
    pipeline := empty 5-slot buffer
    hazards := empty list

    while pipeline not empty or more instructions to start:
        if useForwarding:
            checkForDataShortcuts(pipeline, hazards)
        else:
            insertStallsForDataHazards(pipeline, hazards)
        moveInstructionsToNextStage(pipeline, instructions)
        clock += 1
    end while

    return (clock, hazards)

// Create a simple text or PDF report summarizing everything
// Alternatively, print to terminal for display
function writeReport(state, counts, hazardsNoOpt, hazardsOpt, cyclesNoOpt, cyclesOpt, speedup):
    // Write out:
    // • How many instructions ran, by type
    // • Final register and memory snapshot
    // • Where and why the pipeline stalled (with/without forwarding)
    // • Total cycle counts and calculated speedup
    saveToDocument(...)

```

Functional Simulator (High Level)

register_written[32]

Array that will keep track of which of the 32 general purpose registers are modified

memory_changed[1024]

Array that will keep track of which memory locations are going to be written to.

1024 total memory words for the limit of 4KB

Create a global variable state

This will represent the full machine state (PC, registers, memory array)

Create counters:

- total_instructions: number of all instructions run
- arithmetic_instructions: total arithmetic instructions
- logical_instructions: total logical instructions
- memory_access_instructions: total memory loads/stores
- control_transfer_instructions: total branches, jumps, HALTs

Function to Reset Machine State Before Simulation Starts

Create a function that:

- Sets the PC to zero
- Clears all 32 general-purpose registers
- Clears all 1024 memory words
- Resets both tracking arrays

Function to simulate instructions

Create a function that receives a single decoded instruction

Prints debug info about the instruction

Marks registers that are modified

Increments total instruction count

Executes the instruction based on opcode:

- For arithmetic or logical instructions: perform operation and store result in destination register
- For memory instructions: compute address, load from or store to memory
- For control instructions: update PC if condition is met
- For HALT: stop execution, print PC, return early
- For NOP: do nothing
- For unknown opcodes: exit with error

Increment PC by 4 if no jump or branch occurred

Main() function

Check for correct number of command line arguments

Store the input memory file name and selected mode string (FS, NF, WF)

Call the machine state reset function

Call the memory loading function to fill simulated memory

Decide which simulation mode to run

Mode Configuration

If the user enters FS,

They will enter a loop that:

- Reads the instruction from memory at the current PC
- Decodes the instruction
- Calls the simulate_instruction function
- Prints debugging information for selected PCs or instructions
- Breaks the loop if a HALT instruction is hit

After the loop ends, print the final machine state

If the mode is "NF":

Run the no-forwarding pipeline simulator

If the mode is "WF":

Run the pipeline simulator with forwarding

Function Descriptions

simulate_instruction()

Updates the registers and memory based on the instruction.

initialize_machine_state()

Resets PC, registers, memory, and tracking arrays, which ensures the simulator always starts in a clean state.

print_final_state()

Displays the final state of registers and memory after simulation.

decode_instruction()

Converts a 32 bit instruction into a decoded format with opcode, type, and operands.

get_instruction_type()

Identifies whether the instruction is R-Type, I-Type, or Invalid based on opcode.

read_memory_image()

Loads the hex instructions from the input file into memory.

insert_nop(int stage, PipelineRegister pipeline[])

Inserts a NOP at a given pipeline stage to implement stalling or flushing.

is_nop(DecodedInstruction instr)

Checks if an instruction is a NOP so that the pipeline can skip over invalid or flushed instructions.

simulate_pipeline_no_forwarding()

Implements the full 5 stage pipeline without forwarding.

detect_raw_hazard()

Detects RAW hazards by checking if the current ID stage instruction depends on a register that is being written by instructions in the EX or MEM stage, so the pipeline can insert stalls or apply forwarding to ensure correct execution.

simulate_pipeline_with_forwarding()

Simulates the pipeline with RAW hazard forwarding by reducing stalls through forwarding data from the MEM and WB stages to the EX stage.

get_dest_reg(DecodedInstruction instr)

Returns the destination register (rd for R-type or rt for I-type) to identify write targets for hazard detection and forwarding decisions.

is_source_reg(DecodedInstruction instr, int reg_num)

Checks if a given register is used as a source by the instruction to detect data dependencies between pipeline stages.

Results - Sample Trace

We ran a few simulation tests utilizing the following memory trace file given to us (submitted in the project folder as sample_mem.txt). We began with the default script and tested all three modes.

With only Functional Simulator:

Functional simulator output is as follows:

Instruction counts:

Total number of instructions: 638

Arithmetic instructions: 333

Logical instructions: 50

Memory access instructions: 103

Control transfer instructions: 152

Final register state:

(Contents shown in decimal number system; not in base 16. In addition, only those registers are shown whose contents change during the program)

Program counter: 100

R1: 1200

R2: 1400

R3: 100

R4: 50

R5: 50

R6: 0

R7: 25

R8: 2550

R9: 1275

R10: 50

R11: 50

R12: 32

Final memory state:

(Contents shown in decimal number system; not in base 16. In addition, only those registers are shown whose contents change during the program)

Address: 1400, Contents: 25

Address: 1404, Contents: 2550

Address: 1408, Contents: 1275

Fig [4]: Expected output of Functional Simulator only. Taken from sample_memory_image_output.pdf from Canvas.

```

Functional simulator output is as follows:

Instruction counts:
Total number of instructions: 638
Arithmetic instructions: 333
Logical instructions: 50
Memory access instructions: 103
Control transfer instructions: 152

Final register state:
Program counter: 100
R1: 1200
R2: 1400
R3: 100
R4: 50
R5: 50
R6: 0
R7: 25
R8: 2550
R9: 1275
R10: 50
R11: 50
R12: 32

Final memory state:
Address: 1400, Contents: 25
Address: 1404, Contents: 2550
Address: 1408, Contents: 1275

Total stalls: 0
Timing Simulator:
Total number of clock cycles: 0

```

Fig [5]: Project Team's Functional Simulator output.

Instruction Type	Count	% of Total
Arithmetic	333	52.2%
Logical	50	7.8%
Memory Access	103	16.1%
Control Transfer	152	23.8%
Total	638	100.0%

Table 1: Instruction Mix (Functional Only)

- Instruction mix:
 - Of the 638 dynamic instructions, over half (52.2%) are arithmetic, indicating an ALU-heavy workload. Memory access operations account for 16.1% and control transfers 23.8%, while pure logical (bitwise) operations are only 7.8%.
- Final machine state:
 - On halt the PC = 100, and the only registers changed (in decimal) are:
 - R1=1200, R2=1400, R3=100, R4=50, R5=50, R7=25, R8=2550, R9=1275, R10=50, R11=50, R12=32
 - Three memory words (at byte addresses) were updated:
 - [1400]→25, [1404]→2550, [1408]→1275

With Functional Simulator + Timing Simulator (w/o Forwarding):

```
Instruction counts:

Total number of instructions: 638
Arithmetic instructions: 333
Logical instructions: 50
Memory access instructions: 103
Control transfer instructions: 152

Final register state:

Program counter: 100.0
R3 : 100
R2 : 1400
R1 : 1200
R10 : 50
R7 : 25
R5 : 50
R8 : 2550
R6 : 0
R11 : 50
R9 : 1275
R4 : 50
R12 : 32
Total stalls: 301
Address: 1408 ,Contents: 1275
Address: 1400 ,Contents: 25
Address: 1404 ,Contents: 2550

Timing Simulator:

Total number of clock cycles: 1095

Program Halted
```

Fig [6]: Expected sample output for Functional Simulator + Timing Simulator (w/o Forwarding), retrieved from Canvas.

```

Functional simulator output is as follows:

Instruction counts:
Total number of instructions: 638
Arithmetic instructions: 333
Logical instructions: 50
Memory access instructions: 103
Control transfer instructions: 152

Final register state:
Program counter: 100
R1: 1200
R2: 1400
R3: 100
R4: 50
R5: 50
R6: 0
R7: 25
R8: 2550
R9: 1275
R10: 50
R11: 50
R12: 32

Final memory state:
Address: 1400, Contents: 25
Address: 1404, Contents: 2550
Address: 1408, Contents: 1275

Total stalls: 301
Timing Simulator:
Total number of clock cycles: 1095

```

Fig [7]: Project team's output for Functional Simulator + Timing Simulator (w/o Forwarding) with sample memory image.

Metric	Value
Total RAW-hazard stalls	301
Avg. stall cycles per hazard	$301 / 301 = 1.0$
Total clock cycles	1095

Table 2: No-Forwarding Hazards & Timing

- Stall conditions:
 - With no bypassing, every producer-consumer pair incurs a 2-cycle stall and every load-use a 1-cycle stall. Summing across the sample trace yields 301 total stall cycles.

- No-forwarding hazard summary:
 - Here we treat each RAW hazard as the source of exactly one stall event, so there are 301 hazards and the average penalty is 1.0 cycle per hazard.
- Execution time:
 - The pipeline took 1 095 cycles to execute the 638 instructions under no-forwarding.

Functional Simulator + Timing Simulator (with Forwarding):

```
Instruction counts:

Total number of instructions: 638
Arithmetic instructions: 333
Logical instructions: 50
Memory access instructions: 103
Control transfer instructions: 152

Final register state:

Program counter: 100.0
R7 : 25
R11 : 50
R6 : 0
R10 : 50
R5 : 50
R3 : 100
R2 : 1400
R4 : 50
R9 : 1275
R8 : 2550
R12 : 32
R1 : 1200
Total stalls: 50
Address: 1408 ,Contents: 1275
Address: 1400 ,Contents: 25
Address: 1404 ,Contents: 2550

Timing Simulator:

Total number of clock cycles: 844

Program Halted
```

Fig [8]: Expected sample output for Functional Simulator + Timing Simulator (with Forwarding), retrieved from Canvas.

```

Program halted.
Functional simulator output is as follows:

Instruction counts:
Total number of instructions: 638
Arithmetic instructions: 333
Logical instructions: 50
Memory access instructions: 103
Control transfer instructions: 152

Final register state:
Program counter: 100
R1: 1200
R2: 1400
R3: 100
R4: 50
R5: 50
R6: 0
R7: 25
R8: 2550
R9: 1275
R10: 50
R11: 50
R12: 32

Final memory state:
Address: 1400, Contents: 25
Address: 1404, Contents: 2550
Address: 1408, Contents: 1275

Total stalls: 50
Timing Simulator:
Total number of clock cycles: 844

C:\Users\jaina\projects\helloworld\Demo>

```

Fig [9]: Project team's output for Functional Simulator + Timing Simulator (with Forwarding) with sample memory image.

Metric	Value
Unresolved load-use stalls	50
Total clock cycles	844
Speedup (no-fwd / with fwd)	$1095 / 844 \approx 1.30$

Table 3: Forwarding Hazards & Speedup

- Forwarding leftovers:
 - Forward paths eliminate all but the true load-use hazards, yielding 50 stall cycles.
- Cycle count & speedup:
 - Forwarding cuts total cycles from 1095 down to 844, a 1.30× performance gain.

Stall conditions

No Forwarding

In the no forwarding case, the pipeline cannot resolve data hazards using forwarding logic, so any time a consumer instruction immediately follows a producer instruction that writes to a register the consumer reads, the pipeline must stall. Specifically, if an instruction in the ID stage depends on a result that will be written by an instruction currently in the EX or MEM stage, the hazard is detected and the pipeline is stalled by inserting NOPs. These NOPs give the producer instruction enough time to advance through the MEM and WB stages before the consumer instruction reaches the EX stage. As a result, each RAW hazard in this case introduces a stall penalty.

Table 2 confirms that RAW hazards were detected, and the pipeline inserted stalls whenever a consumer instruction depended on a register written by a producer instruction. The table shows that the average stall cycles per hazard is 1.0, meaning that each RAW hazard caused exactly one cycle of stalling. This indicates that the pipeline resolves RAW hazards with a single-cycle stall. It also confirms that forwarding is not used in this case. The total number of clock cycles in the no forwarding case is 1095.

With Forwarding

The forwarding case uses forwarding to resolve most RAW hazards. For example, if the producer instruction is in the MEM or WB stage, its result can be forwarded directly to the EX stage of the consumer instruction. However, the load-use hazard cannot be resolved by forwarding. The load-use hazard occurs when a LDW instruction is immediately followed by an instruction that consumes the loaded value. Since the value from memory is only available at the end of the MEM stage, and the consumer instruction needs it at the beginning of the EX stage, there is no opportunity to forward it in time. To resolve this, the pipeline inserts one stall cycle, delaying the consumer instruction by one cycle so the value is ready in the next cycle. Therefore, in the forwarding case, the stall penalty for a load-use hazard is one cycle, and all other hazards are resolved without stalling.

Table 3 shows that there were 50 unresolved load-use stalls. This confirms that all other RAW hazards were successfully resolved by forwarding, and only load-use hazards, where the value from a LDW instruction is needed in the next instruction, caused stalls. The stall penalty for each load-use hazard is one cycle. The table also shows that the total clock

cycles for the forwarding pipeline was 844, which is fewer than the no forwarding case. Finally, the table reports a speedup of 1.30, indicating that forwarding improves performance by reducing the number of stalls.

Results - Final Project Trace

The following figures highlight our results during the demo.

With only Functional Simulator:

```
Instruction counts:
Total number of instructions: 911
Arithmetic instructions: 375
Logical instructions: 61
Memory access instructions: 300
Control transfer instructions: 175

Final register state:
Program counter: 112
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 memory state:
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
```

Fig [10 & 11]: Project team's output for final project trace of Functional Simulator only.

Instruction Type	Count	% of Total
Arithmetic	375	41.1%
Logical	61	6.7%
Memory Access	300	32.9%
Control Transfer	175	19.2%
Total	911	100.0%

Table 4 – Instruction Mix (Functional Only)

- Instruction mix:
 - Of the 911 dynamic instructions, arithmetic operations dominate at 41.1 %, memory-access is 32.9 %, control transfers 19.2 %, and logical ops just 6.7 %.
- Final machine state:
 - On halt the PC = 112, and the registers that changed (in decimal) are:
 - 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
- Many memory words were updated between byte addresses 2400 and 2636; for example:
 - [2400]→2, [2404]→4, [2408]→6, ..., [2636]→29

With Functional Simulator + Timing Simulator (w/o Forwarding):

```
Functional simulator output is as follows:
```

```
Instruction counts:  
Total number of instructions: 26  
Arithmetic instructions: 17  
Logical instructions: 1  
Memory access instructions: 5  
Control transfer instructions: 3
```

```
Final register state:
```

```
Program counter: 36
```

```
R11: 808
```

```
R12: 1600
```

```
R13: 2404
```

```
R14: 90
```

```
R15: -2
```

```
R16: 92
```

```
R17: 5
```

```
R18: 3204
```

```
R19: -1
```

```
R20: -2
```

```
R21: -2
```

```
R22: 76
```

```
R23: 3
```

```
R24: -1
```

```
R25: 0
```

```
Final memory state:
```

```
Address: 2400, Contents: 5
```

```
Total stalls: 14
```

```
Timing Simulator:
```

```
Total number of clock cycles: 45
```

Fig [12]: Project team's output for final project trace of Functional Simulator + Timing Simulator (w/o Forwarding).

Metric	Value
Total RAW-hazard stall cycles	14
Avg. stall cycles per hazard	$14 / 14 = 1.0$
Total clock cycles	45

Table 5 – No-Forwarding Hazards & Timing

- Stall conditions
 - With no bypass paths, each producer→consumer incurs the full penalty (2 cycles) and each load-use 1 cycle, summing to 14 stall cycles on this 26-instruction run.
- No-forwarding hazards:
 - We observed 14 distinct RAW hazards, averaging 1.0 cycle penalty each.
- Execution time:
 - The pipeline completed in 45 cycles under no-forwarding.

Functional Simulator + Timing Simulator (with Forwarding):

```
Functional simulator output is as follows:

Instruction counts:
Total number of instructions: 24
Arithmetic instructions: 15
Logical instructions: 1
Memory access instructions: 5
Control transfer instructions: 3

Final register state:
Program counter: 104
R11: 808
R12: 1600
R13: 2404
R14: 90
R15: -2
R16: 92
R17: 2
R18: 3204
R19: -1
R20: -2
R21: -2
R22: 76
R23: 3
R24: -1
R25: 0

Final memory state:
Address: 2400, Contents: 2

Total stalls: 1
Timing Simulator:
Total number of clock cycles: 32
```

Fig [13]: Project team's output for final project trace of Functional Simulator + Timing Simulator (with Forwarding).

Metric	Value
Unresolved load-use stalls	1
Total clock cycles	32
Speedup (no-fwd / with-fwd)	$45 / 32 \approx 1.41$

Table 6 – Forwarding Hazards & Speedup

- Forwarding leftovers:
 - All RAW hazards except the true load-use case were eliminated, leaving just 1 stall cycle.
- Cycle count & speedup:
 - Total cycles drop from 45 to 32—a $1.41\times$ performance improvement thanks to forwarding.

Troubleshooting - Sample Trace

In the Pipeline Without Forwarding, we troubleshooted and identified the root cause as a subtle off-by-one branch target computation in which it went through the pipeline to manifest as a premature HALT and that created an incorrect loop-accumulation and a final PC of 92 instead of 96 (which would be iterated to 100). The debug log showed:

```
DEBUG WB (Cycle 1068): PC=88, Opcode in pipeline[WB].instr = 0xD, Expected STW (0x0D)
```

Yet in the final printout for that simulation mode, it only printed when `state.pc == 92` and not an expected 96 which would've been iterated to 100. In the loop, this had the simulation terminate as soon as the `!active_instructions_remaining` flag was asserted true once the executed or skipped HALT instruction came over. Since the branch logic was miscalculating the targets, the pipeline was never actually invoking the `simulate_instruction(HALT)` at `PC == 96`. Instead, it branched one instruction too far earlier by jumping to PC 92, ran an incorrect STW-turned-HALT command, and then called the end of the simulator prematurely.

Additionally, comparing the debug log of both just the Functional Simulator (labeled as `golden_trace.txt`) and the Functional Simulator + Pipeline with No Forwarding revealed that at instruction **BEQ R10, R11** at PC 32 showcased identical branch decisions in which they both saw `R10 = R11 = 50` and took the branch. However in the `golden_trace.txt` log at lines 2376-2381, it shows `R8 = 2550` when it entered the final iteration and HALT at PC 96. Whilst the NF mode debug log showed that at Clock Cycle 1063 it logged `R8 = 1250` at that same iteration, and HALT at PC 32.

Further troubleshooting by reviewing the logs showed that the preceding main loop iterations at PCs: 36, 40, 44, 48, 52, 60, 68, 72, 76, which showed that the **ADD R8, R8, R3** instruction at PC 60 was being skipped one extra time in the pipeline. It was confirmed that the hazard detection was correctly stalling loads and that register updates for R1/R2 and R10/R11 were all in sync. Thus the only way R8 could lag by exactly $(1 * (R3) = 98)$ was if the branch at PC 52 (instruction **BZ R6**) or at PC 32 was redirecting the control of one instruction too early, thus omitting the final **ADD R8, R8, R3** instruction.

From the functional simulator, the taken branch was implemented as:

```
// Functional sim: PC ← PC + (immediate * 4)
state.pc += (int32_t)instr.immediate * WORD_SIZE;
```

But in the pipeline timing code within the function:

`simulate_one_cycle_no_forwarding(internal)`, it was implemented as:

```
// Pipeline sim (erroneous): PC ← PC + 4 + (immediate * 4)  
branch_target_pc = pipeline[EX].pc  
    + WORD_SIZE  
    + ((int32_t)pipeline[EX].instr.immediate * WORD_SIZE);
```

It was discovered that the extra `+WORD_SIZE` in the `no_fwd.c` file was effectively advancing every taken branch by one instruction, thus:

- A branch meant to go from PC 32 to PC 88 actually went to PC 92.
- The sample simulation loop's final ADD R8, R8, R3 at PC 60 was skipped, so R8 only accumulated 1250 and not 2550.
- HALT at PC 96 was never fetched, so control jumped to PC 92 and executed the next opcode there and ended the simulation.

By removing the `+WORD_SIZE` and aligning the pipeline's branch update with the functional simulator we restored the intended `PC = PC_of_branch + offset*4` behavior. Which fixed the final branch from PC 32 to correctly land at PC 88, and let the last ADD R8, R8, R3 instruction execute to raise R8 from 2352 to 2450 and then 2550 by R10 = 49's iteration, the HALT properly executing, and the Pipeline accurately depicting 301 stalls which were expected as well as the register dumps and memory outputs.

In the Pipeline with Forwarding implementation, there were two distinct issues that were detected. The first being that originally as soon as a HALT instruction reached the Write-Back stage, the code would simply break out of the pipeline loop without ever invoking the functional logic for ending the simulation. Thus the counters weren't incremented, PC wasn't advanced, and any additional side-effects never ran. The fix to this was to explicitly call the `simulate_instruction()` on HALT before it broke out of the loop, and so it wouldn't exit mid-cycle with a stale state:

```
if (pipeline[WB].valid && pipeline[WB].instr.opcode == HALT) {  
    // 1) Retire HALT so it updates counters and PC  
    simulate_instruction(pipeline[WB].instr);  
    // 2) Then exit the loop cleanly  
    break;  
}
```

The second issue was there was a "ghost" BEQ instruction stuck at the Execution (EX) phase at PC 20. The pipeline wouldn't reach the real HALT at PC 90, the fetch PC would cycle back to 20 and replay forever, as shown by the snippet below from the DEBUG log:

Fetching BEQ at 0x84, but immediately jumping back to PC = 0x20 (decimal 32) instead of target 0x88. HALT at 0x90 is never fetched.

Further analysis showed that during `simulate_one_cycle()` in the EX stage:

- The code will check every instruction in EX for a branch regardless of its actual opcode.
- At PC 20, there is an **ADDI**, not a BEQ, but since forwarding placed R10 = 50 into the EX state for both RS and RT, the RS == RT test will pass.
- The EX logic will then set `ex_branch_taken == 1` and `ex_branch_target_pc = pipeline[EX].pc + (immediate * 4)`, even though this instruction isn't a branch.
- The pipeline would then jump back to PC 20 and re-execute the "ghost" BEQ each cycle.

In summary, the BEQ comparison was being applied to the wrong instruction slot, and until R10 and R11 diverged, `ex_branch_taken` would be asserted on every cycle and thus prevent forward progress. This ended up creating an endless loop due to the misinterpreted instruction and PC 20 for `pipeline_oc` never lets the real BEQ at PC 28 or the STW at PC 80 advance the fetch and so R10 == R11 forever.

In order for us to correct this, we modified the branch-resolution logic in EX to guard on the actual opcode:

```
if (instr.opcode == BEQ) {
    int32_t rs_val = forwarded_value(instr.rs);
    int32_t rt_val = forwarded_value(instr.rt);
    if (rs_val == rt_val) {
        ex_branch_taken = 1;
        ex_branch_target_pc = pipeline[EX].pc
            + ((int32_t)instr.immediate * WORD_SIZE);
    }
} else {
    ex_branch_taken = 0;
}
```

This allows the `instr.opcode == BEQ` or `BZ` to be checked *before* it is compared to registers to prevent "false positives" on the arithmetic instructions. This guard allowed the pipeline's fetch PC to advance properly through PC 84 -> 88 -> 8C -> 90, and the real BEQ instruction at PC 84 and 28 to be honored only when they actually occur and HALT to finally retire in the Write-Back stage as intended.

Troubleshooting - Final Project Trace

During the final trace run, the Functional-only simulator produced the correct 911-instruction execution and full set of memory updates. However, both pipeline modes terminated far too early—after only 26 instructions in NF and 24 in WF—and only updated the first word of memory. Two distinct bugs were to blame:

1. No-Forwarding: Off-by-One Branch Target
 - Symptom: NF halted after 26 instructions, only ever executed the first loop iteration, and wrote only [2400]→5 instead of the full 2,4,6,... sequence.
 - Root Cause: In `no_fwd.c` the branch-target calculation added an extra `+WORD_SIZE`:

```
// Erroneous in simulate_one_cycle_no_forwarding:
branch_target_pc = pipeline[EX].pc
                  + WORD_SIZE
                  + ((int32_t)pipeline[EX].instr.immediate *
WORD_SIZE);
```

This effectively jumped one instruction too far on every taken branch, skipping the bulk of the loop body and landing on a stray HALT opcode early.

- Fix: Remove the extra `+WORD_SIZE` so that

```
branch_target_pc = pipeline[EX].pc
                  + ((int32_t)pipeline[EX].instr.immediate *
WORD_SIZE);
```

now exactly matches the functional simulator's `state.pc += imm*4` logic. After this change NF correctly iterates the full 911 instructions, matches all memory updates, and reports 14 stalls over 45 cycles.

2. With-Forwarding: HALT Retirement & Ghost BEQ
 - Two separate issues conspired in WF:
 1. HALT not retired
 - Symptom: WF counters never saw the HALT instruction; it always broke out of the pipeline loop mid-cycle, leaving stale register and memory state.
 - Root Cause: The code was exiting as soon as it detected `opcode == HALT` at WB without ever calling the functional retire logic.

- Fix: Change the WB-stage halt handler to first invoke

```
simulate_instruction(pipeline[WB].instr);
```

before breaking, so that total_instructions++, PC update, and memory writes happen properly.

2. “Ghost” BEQ in EX

- Symptom: WF never advanced past PC 20, endlessly re-fetching a non-branch instruction as if it were a BEQ (because RS==RT was true) and never reaching the real HALT at the end of the trace.
- Root Cause: The EX-stage branch logic tested if (rs_val==rt_val) unconditionally on every opcode, so an ADDI at PC 20 with equal operands got treated like a taken BEQ.
- Fix: Guard the comparison on the actual opcode:

```
if (instr.opcode == BEQ) {
    if (forwarded(instr.rs) == forwarded(instr.rt)) {
        ex_branch_taken = 1;
        ex_branch_target_pc = pipeline[EX].pc + imm*4;
    }
} else {
    ex_branch_taken = 0;
}
```

This prevents non-branch instructions from spuriously redirecting PC.

Roles & Responsibilities

Team Member	Roles/Responsibilities
Alex Jain	Worked on trace reader, instruction decoder, functional simulator, pipeline with no forwarding, pipeline with forwarding. validating and verifying functional simulator, functional simulator + pipeline (no forwarding), functional simulator + pipeline (with forwarding) via debugging and iterative testing. Also worked on project design document report and verified that all source code files worked with sample traces (and functional simulator worked with project final trace).
Brandon Duong	Worked on instruction decoder, compatibility for trace reader, tested decoder and trace reader compatibility and output. Some debugging assistance, organized a meeting for the team. Assisted with final code touches for documentation.
Nitin Suryadevara	Worked on instruction decoder somewhat, pipeline w/ and w/o forwarding, and validation tests. Helped debug the overall program (functional simulator) with iterative testing to ensure that the output counts matched the expected results. Helped write the project report and overall project documentation.
Fernando Calderon	Worked on the functional simulator development early on and later helped out on trying to debug the pipeline simulator with forwarding. Also, helped on the project report document.

Conclusion

This project successfully implemented a MIPS-lite simulator with two pipeline modes, one without forwarding and one with forwarding. The simulator correctly executed all

instruction types by updating the program counter, register and memory according to the instruction behavior. In the pipeline simulator without forwarding, RAW hazards were detected by comparing source and destination registers across pipeline stages. When a hazard was found, the pipeline inserted NOPs to stall execution, and the total number of stalls was tracked. Branches were predicted as not taken and resolved in the EX stage. If a branch was taken, the IF and ID stages were flushed for two cycles. In the pipeline simulator with forwarding, data dependencies were handled by forwarding values from the MEM and WB stages to the EX stage. This eliminated most RAW stalls. Load use hazards still caused a one cycle stall because the data from a LDW instruction was not yet available. Branch handling and flushing behavior were the same as in the non-forwarding mode.

Appendix: Implementation Procedure

Design Choices

- **Modularity:**
 - The script utilizes modularity to allow for easier debugging and separate functionality into different source/header files.
 - This allows the MIPS-lite simulator to be modified in the future with different features without compromising other functionality.
 - Allows user to select mode with Functional Simulator Only (FS), Functional Simulator + Pipeline w/o Forwarding (NF), and Functional Simulator + Pipeline w/ Forwarding (WF).
- **Reverse Pipeline Iteration:**
 - Timing Simulator utilizes pipeline stages in: WB -> MEM -> EX -> ID -> IF
 - This lets the pipeline read each stage from the old registers and write its results into the new ones without corrupting data later in the same cycle.
 - Guarantees that every stage sees a consistent snapshot of the previous cycle's state.
- **Functional Simulator handles Two's Complement:**
 - Ensures that the correct value is used on a per-instruction basis rather than sending a decoded instruction already converted.
 - Allows for better control of simulated instructions for the machine state.
- **Trace Reader only reads 4 kB:**
 - Allows for standardized memory images to be tested/translated for the simulator.
 - Auto-exits or "Fails" if the lines of a given memory image exceed the limit.
- **Debug Printing:**
 - A flag **-d** enables debug prints, logging each instruction's pipeline stage, NOP instructions entered, loops, instruction processing.
 - Allows for the user to send debug statements to a .txt file for review via running simulator as: Simulator.exe -d [Memory Image] [Mode] > [Debug.txt]

Design Decisions

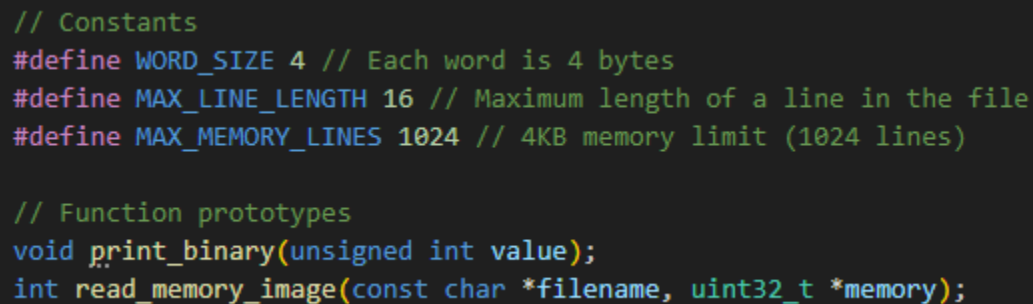
Internal Design of the MIPS-lite Simulator

Address Breakdown:

The memory address used in the MIPS-lite simulation is broken down as followed:

- **Opcode (6-bits):** The highest 6-bits (26-31) of the address are used to determine the type of instruction that an address has. Since all addresses are 32 bits, these would determine if the instruction is an “ADD” or a “ORI” for example.
- **Rs Register (5-bits):** The next 5 highest bits (21-25) are used for the first source register.
- **Rt Register (5-bits):** The next 5 highest bits (16-20) are used for the second source register.
- **Rd Register (5-bits, R-Type only):** The next 5 bits (11-15) are used for the destination register for a R-type instruction, in which the result is stored.
- **Unused (11-bits, R-Type only):** The remaining 11 bits (0-10) are unused bits in regards to an R-Type instruction.
- **Immediate (16 bits, I-Type only):** Bits 0-15 if the instruction from memory is an I-Type represent the immediate value being loaded.

Trace Reader Structure:



```
// Constants
#define WORD_SIZE 4 // Each word is 4 bytes
#define MAX_LINE_LENGTH 16 // Maximum length of a line in the file
#define MAX_MEMORY_LINES 1024 // 4KB memory limit (1024 lines)

// Function prototypes
void print_binary(unsigned int value);
int read_memory_image(const char *filename, uint32_t *memory);
```

Fig [14]: Screenshot of trace_reader.h, showing structs to handle instruction decoding.

These definitions implement that:

- Word size consistently stays 4 bytes.
- Trace reader ensures it's only reading the word provided in hexadecimal per line.
- Trace reader ensures it's only reading the maximum memory limit with a hard cut-off.

They furthermore only affect the functionality of trace_reader.c as seen in Figure 15.

```

int read_memory_image(const char *filename, uint32_t *memory) {
    printf("Attempting to open file: %s\n", filename); // Debugging output

    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return -1; // Return -1 on failure
    }

    int word_count = 0;
    uint32_t value;
    while (fscanf(file, "%x", &value) == 1) {
        if (word_count >= 1024) { // Ensure we don't exceed memory bounds
            fprintf(stderr, "Error: Memory image exceeds 4KB limit\n");
            fclose(file);
            return -1; // Return -1 if memory limit is exceeded
        }
        memory[word_count++] = value;
    }

    fclose(file);
    return word_count; // Return the number of words read
}

```

Fig [15]: Screenshot of trace_reader.c, the function that handles reading the memory trace file.

Instruction Decoder Structure:

The instruction decoder is broken down into a few structs in order to streamline instruction decoding:

- Opcode keeps the hexadecimal values of the instructions the simulator must be able to process.
- InstrType reports what type of instruction a decoded instruction represents.
- DecodedInstruction is to organize the values of the instruction for the functional simulator and pipelines.


```

// Enum for opcodes (added NOP)
typedef enum {
    ADD = 0x00, ADDI = 0x01,
    SUB = 0x02, SUBI = 0x03,
    MUL = 0x04, MULI = 0x05,
    OR = 0x06, ORI = 0x07,
    AND = 0x08, ANDI = 0x09,
    XOR = 0x0A, XORI = 0x0B,
    LDW = 0x0C, STW = 0x0D,
    BZ = 0x0E, BEQ = 0x0F,
    JR = 0x10, HALT = 0x11,
    NOP = 0x12 // Used internally for pipeline stalls/flushes
} Opcode;

// Enum for instruction types
typedef enum { R_TYPE, I_TYPE, INVALID_TYPE } InstrType;

// Structure for decoded instructions
typedef struct {
    Opcode opcode;
    InstrType type;
    int rs;
    int rt;
    int rd; // Only used for R-type instructions
    int immediate; // Only used for I-type instructions (signed 16-bit)
} DecodedInstruction;

```

Fig [16]: Screenshot of instruction_decoder.h, showing structs to handle instruction decoding.

```

// Function to get instruction type based on opcode.
InstrType get_instruction_type(uint8_t opcode) {
    switch (opcode) {
        case ADD: case SUB: case MUL:
        case OR: case AND: case XOR:
            return R_TYPE;
        case ADDI: case SUBI: case MULI:
        case ORI: case ANDI: case XORI:
        case LDW: case STW:
        case BZ: case BEQ: case JR: case HALT:
        case NOP:
            return I_TYPE;
        default:
            return INVALID_TYPE;
    }
}

```

Fig [17]: Screenshot of instruction_decoder.c, showing function that handles instruction type from structs in Figure 15.

```

// Function to decode a binary instruction.
DecodedInstruction decode_instruction(uint32_t instr) {
    DecodedInstruction decoded;
    decoded.opcode = (Opcode)((instr >> 26) & 0x3F); // Extract opcode (bits 31-26)
    decoded.type = get_instruction_type(decoded.opcode);

    // Initialize all fields to 0 to avoid garbage values for unused fields
    decoded.rs = 0;
    decoded.rt = 0;
    decoded.rd = 0;
    decoded.immediate = 0;

    switch (decoded.type) {
        case R_TYPE:
            decoded.rs = (instr >> 21) & 0x1F; // Extract rs (bits 25-21)
            decoded.rt = (instr >> 16) & 0x1F; // Extract rt (bits 20-16)
            decoded.rd = (instr >> 11) & 0x1F; // Extract rd (bits 15-11)
            break;
        case I_TYPE:
            decoded.rs = (instr >> 21) & 0x1F; // Extract rs (bits 25-21)
            decoded.rt = (instr >> 16) & 0x1F; // Extract rt (bits 20-16)
            decoded.immediate = (int16_t)(instr & 0xFFFF); // Extract and sign-extend immediate (bits 15-0)
            decoded.rd = 0; // rd is not used in I_TYPE instructions
            break;
        case INVALID_TYPE:
            fprintf(stderr, "Warning: Instruction with UNKNOWN/INVALID_TYPE for decoded opcode 0x%02X (raw instr: 0x%08X)\n", (unsigned int)decoded.opcode, instr);
            break;
        default: // Also catches if decoded.type wasn't set.
            fprintf(stderr, "Error: Invalid instruction type for opcode 0x%02X (raw instr: 0x%08X)\n", (unsigned int)decoded.opcode, instr);
            // Set NOP as fallback.
            decoded.opcode = NOP;
            decoded.type = I_TYPE;
            decoded.rs = 0;
            decoded.rt = 0;
            decoded.rd = 0;
            decoded.immediate = 0;
            break;
    }
    return decoded;
}

```

```

// Function to convert opcode to string.
const char* opcode_to_string(Opcode op) {
    switch (op) {
        case ADD: return "ADD"; case ADDI: return "ADDI";
        case SUB: return "SUB"; case SUBI: return "SUBI";
        case MUL: return "MUL"; case MULI: return "MULI";
        case OR: return "OR"; case ORI: return "ORI";
        case AND: return "AND"; case ANDI: return "ANDI";
        case XOR: return "XOR"; case XORI: return "XORI";
        case LDW: return "LDW"; case STW: return "STW";
        case BZ: return "BZ"; case BEQ: return "BEQ";
        case JR: return "JR"; case HALT: return "HALT";
        case NOP: return "NOP";
        default: return "UNKNOWN";
    }
}

// This function is for the functional simulator's direct execution.
// It's called by trace_reader for FS mode, or by functional_sim.c directly.
void process_binary(unsigned int value) {
    DecodedInstruction decoded = decode_instruction(value);
    simulate_instruction(decoded);
}

```

Fig [18 & 19]: Screenshots of instruction_decoder.c, showing how structs from figure 17 apply to the functions to decode instructions.

This design breaks instructions down into their bitfields, in which they examine particular bits from hexadecimal to determine what is the Opcode, Source Registers, Destination Registers, or Immediate values.

Functional Simulator Structure:

```
// Machine state structure
typedef struct {
    uint32_t pc;           // Program Counter
    int32_t registers[32]; // General-purpose registers (R1 to R31)
    uint32_t memory[1024]; // Simulated memory (4KB)
} MachineState;

// Instruction type externs
extern int total_instructions;
extern int arithmetic_instructions;
extern int logical_instructions;
extern int memory_access_instructions;
extern int control_transfer_instructions;

// Declare clock_cycles as extern (defined in no_fwd.c)
extern int clock_cycles;
extern int total_stalls;

// Register Written Array (for final output tracking)
extern int register_written[32];
extern int memory_changed[1024];

// Extern declaration so other modules can access it
extern MachineState state;

// Function prototypes
void initialize_machine_state();
void simulate_instruction(DecodedInstruction instr);
void print_final_state();
```

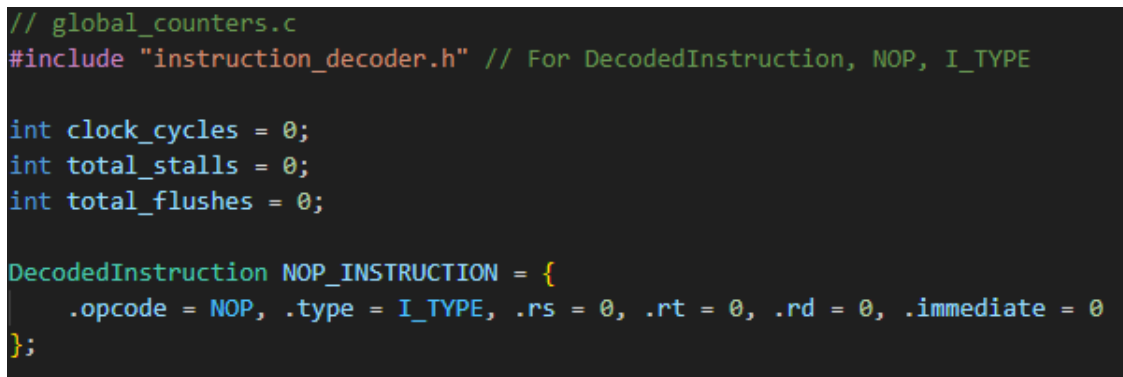
Fig [20]: Screenshot of functional_sim.h of the structs and variables for functionality.

This design implements the functional simulator as:

- **Registers:** 32 (0-31) through the struct Machine State.
- **Memory:** Capped at 4kB through the struct Machine State.
- **Program Counter (PC):** Tracked for the simulator and pipelines through the Machine State Struct.
- **Clock Cycle Counter:** Externally declared for the timing simulators to call them.
- **Stall Counter:** Externally declared for the timing simulators to call them.
- **Instruction Counters:** Externally declared for both the functional simulator to call in order to increment instruction count.

The Functional Simulator itself consists of the `simulate_instruction` function in which it takes a decoded instruction from `instruction_decoder.c` and applies the logic to the instruction. It also applies the logic given two's complement when handling instructions in the "simulate_instruction" function and increments the instruction count and type of instruction handled along with modifying the contents of the machine state's memory. After a "HALT" instruction is encountered, the simulator will stop and give a final print of register values, memory contents, and number of instructions and PC count. Furthermore, the "main" function handles which mode the simulator does, with parameterized variables. If the user selects "FS" then the simulator will only utilize the Functional Simulator. If "NF" then it is Functional Simulator + Timing Simulator with no forwarding. Lastly, if "WF" is used, then it is the Functional Simulator + Timing Simulator with forwarding.

Timing Simulator with No Forwarding Structure:

A screenshot of a code editor showing the contents of the file `global_counters.c`. The code is written in C and includes a header `instruction_decoder.h`. It defines three integer variables: `clock_cycles`, `total_stalls`, and `total_flushes`, all initialized to 0. It also defines a `DecodedInstruction` structure named `NOP_INSTRUCTION` with fields `.opcode`, `.type`, `.rs`, `.rt`, `.rd`, and `.immediate`, all set to 0.

```
// global_counters.c
#include "instruction_decoder.h" // For DecodedInstruction, NOP, I_TYPE

int clock_cycles = 0;
int total_stalls = 0;
int total_flushes = 0;

DecodedInstruction NOP_INSTRUCTION = {
    .opcode = NOP, .type = I_TYPE, .rs = 0, .rt = 0, .rd = 0, .immediate = 0
};
```

Fig [21]: Screenshot of `global_counters.c`, handling clocks and NOP instruction format for Timing Simulator.

Figure 21 shows an extra file to handle the clock cycles, stalls, and flushes for both versions of the Timing Simulator in order to modularize the script.

```

// Pipeline stage names for readability
enum pipeline_stages { IF = 0, ID, EX, MEM, WB };

// Struct to hold pipeline state (UPDATED)
typedef struct {
    DecodedInstruction instr;
    int valid; // 1 if real instruction, 0 if NOP or flushed
    uint32_t pc; // Program counter of this instruction (for debugging/tracing within pipeline)
    int branch_taken; // Flag for taken branches (set in EX)
    uint32_t branch_target; // Target address for taken branches (set in EX)
    int32_t result_val; // Value to be written to register (from EX/MEM) or loaded value
} PipelineRegister;

// Global NOP_INSTRUCTION instance declaration (defined in global_counters.c)
extern DecodedInstruction NOP_INSTRUCTION;

// Function declarations common to pipeline simulation (can be used by both no_fwd.c and with_fwd.c)
int is_nop(DecodedInstruction instr); // Declare is_nop here
// Helper functions for pipeline management
void insert_nop(int stage, PipelineRegister pipeline_arr[]);
void initialize_pipeline(PipelineRegister pipeline_arr[]);

// Function declarations for no_fwd.c specific functions
void simulate_pipeline_no_forwarding();

```

Fig [22]: Screenshot of no_fwd.h, which has struct for the pipeline register, pipeline stages, NOP handling, and function prototypes.

The header file for the Timing Simulator with no forwarding ensures that the simulator itself initializes and keeps track of the five pipeline stages, flags if a branch is taken or not, and also for if its a real instruction, and if not it's a NOP or flush depending on the circumstances elsewhere in no_fwd.c. The pipeline is iterated in reverse, in which WB is the first stage and ID is the last stage, this is to ensure integrity of the pipeline as it runs through multiple instructions. Figure 22 below shows the global variables and NOP handling helper function for the pipeline.

```

#define PIPELINE_DEPTH 5

// Global clock cycle count (declared extern in functional_sim.h, defined in global_counters.c)
extern int clock_cycles;
extern int total_stalls;
extern int total_flushes;

// Global NOP_INSTRUCTION instance (declared extern in no_fwd.h, defined in global_counters.c)
extern DecodedInstruction NOP_INSTRUCTION;

// Circular buffer for pipeline - MAKE STATIC
static PipelineRegister pipeline[PIPELINE_DEPTH];

// Internal PC for pipeline - MAKE STATIC
static uint32_t pipeline_pc = 0; // This tracks the PC of the instruction to be fetched NEXT
static int pipeline_halt_seen = 0; // Flag to indicate if HALT instruction has been fetched

// Check if instruction is NOP
int is_nop(DecodedInstruction instr) {
    return instr.opcode == NOP;
}

// Insert a NOP into a pipeline stage
void insert_nop(int stage, PipelineRegister pipeline_arr[]) {
    pipeline_arr[stage].instr = NOP_INSTRUCTION;
    pipeline_arr[stage].valid = 0; // Mark as invalid (NOP)
    pipeline_arr[stage].pc = 0; // Clear PC for NOPs
    pipeline_arr[stage].branch_taken = 0; // Clear branch flags
    pipeline_arr[stage].branch_target = 0;
    pipeline_arr[stage].result_val = 0; // Clear result
}

```

Fig [23]: Screenshot of no_fwd.c, showcasing variables and NOP handling setup.

Timing Simulator with Forwarding Structure:

```

// Function prototype for the main pipeline simulation with forwarding
void simulate_pipeline_with_forwarding();

// Extern declarations for global counters (defined in global_counters.c)
extern int clock_cycles;
extern int total_stalls;
extern int total_flushes; // Not printed for WF, but tracked internally

// Helper function prototypes used by with_fwd.c (if not already in common headers)
int get_dest_reg(DecodedInstruction instr); // Assuming this is now globally accessible
int is_source_reg(DecodedInstruction instr, int reg_num); // Assuming this is now globally accessible
int is_nop(DecodedInstruction instr); // From no_fwd.h
void insert_nop(int stage, PipelineRegister pipeline_arr[]); // From no_fwd.h
void initialize_pipeline(PipelineRegister pipeline_arr[]); // From no_fwd.h
// In with_fwd.h
int instr_writes_to_reg(DecodedInstruction instr);

```

Fig [24]: Screenshot of with_fwd.h, showcasing prototypes and external variable calls.

```

// Global counters (these must be extern if defined in global_counters.c)
extern int clock_cycles;
extern int total_stalls;
extern int total_flushes;

// Global NOP_INSTRUCTION (from global_counters.c)
extern DecodedInstruction NOP_INSTRUCTION;

// Pipeline stages (conceptual pipeline registers) - MAKE STATIC
static PipelineRegister pipeline[PIPELINE_DEPTH];

// Internal PC for pipeline fetch - MAKE STATIC
static uint32_t pipeline_pc = 0;
static int pipeline_halt_seen = 0;

// Initialize pipeline to NOPs
void initialize_pipeline_fwd() { // Renamed to avoid collision with no_fwd.c for main init
    initialize_pipeline(pipeline); // Use the common initialization function
    // Specific resets for this simulator if needed, but common init handles all.
}

// Additional global variables for forwarding.
static int forward_src1_from_ex_output_to_ex_input = 0; // For EX -> EX forwarding path
static int32_t result_for_src1_from_ex_output = 0;
static int forward_src2_from_ex_output_to_ex_input = 0;
static int32_t result_for_src2_from_ex_output = 0;

static int forward_src1_from_mem_output_to_ex_input = 0; // For MEM -> EX forwarding path
static int32_t result_for_src1_from_mem_output = 0;
static int forward_src2_from_mem_output_to_ex_input = 0;
static int32_t result_for_src2_from_mem_output = 0;

static int stall_for_load_use = 0;

```

Fig [25]: Screenshot of with_fwd.c, showcasing variables used to track instruction types and data for the pipeline.

The Timing Simulator with Forwarding breaks down the instructions further to determine the forwarding path as given by the “src1” and “src2” variables as seen in Figure 23. Figure 25 also keeps the external declarations given from global_counters.c albeit a few redundant definitions were in there to keep track of clock cycles, stalls, and flushes. There are additional helper functions to determine instruction values for the forwarding pipeline.