Computer Architecture
Final project report
ECE586

Team members:
Phil Nevins
Daniel Anishchenko
Cody Reid
Kenneth Sutter
Angela Albaka

Project:
This project is about Mips-Lite and 5 pipeline stages.The code simulates the execution of MIPS instructions in a pipeline without hazard detection(Mode 0) and without and with forwarding(Mode 1 and Mode 2), depending on the selected mode.

5 pipeline stages:
Fetch Function: It fetches the instruction from memory.
Decode Function: It decodes the fetched instruction, extracting opcode, registers, and immediate values. It also determines the type of the instruction (R-Type or I-Type) based on the opcode.
Execute Function: It performs arithmetic, logical, and control transfer operations based on the opcode.
Memory Access Function: This stage accesses memory if the instruction requires it, in case of load or store instructions.
Write Back Function: It writes results back to registers if necessary.

High Level Algorithm

1. Prompt user for mode
2. Read in a “memory image”
3. Shift bits to decode all the required values for the different formats below
4. Store these values index by line (PC = line * 4)
5. Use the 6 bit Opcode to determine what this instructions opcode is
    a. If OPCODE = R ⟶ GOTO R-type parsing Function
    b. If OPCODE = I ⟶ GOTO I-type parsing Function
6. Execute pipeline in order
7. Check for hazards and branches, update values accordingly
8. increment PC
9. repeat until EOF

10. (Halt will be used to print final states)

Hazard Check Algorithm

Check what type of function (R or I type)

This will determine which destination registers we need to check against our current registers, as this changes based on:

R to R

R to I

I to R

I to I

Check if the current destination register is updated by the previous instruction (1 back),

Add 2 clock cycles for delay without forwarding, update stall counter

Add no clock cycles for delay with forwarding

Check if the current destination register is updated by the 2nd previous instruction (2 back),

Add 1 clock cycle for delay without forwarding, update stall counter

Add no clock cycles for delay with forwarding

CheckForMispredictedBranch Algorithm

Initialize Variables

Set currentMemImgIndex to the current program counter (PC) divided by 4.

Define branch opcodes for BZ, BEQ, and JR.

Initialize is_branch to 0.

Check for Branch Instruction

Loop through the branch opcodes and set is_branch to 1 if the current instruction opcode matches any branch opcode.

Branch Taken Check

If is_branch is true:

If the branch condition is met (branch_taken_condition returns true):

If mode is 2, exit the function.

Increment ClockCount and StallCount by 2.

Flush the next two instructions by setting them to NOP if they are within the instruction range.

Branch_taken_condition Algorithm

Initialize Variables

Set currentMemImgIndex to the current program counter (PC) divided by 4.

Get the opcode of the current instruction.

Branch Condition Check

Switch based on the opcode:

For BEQ (0xF): Return true if the source and target registers are equal.

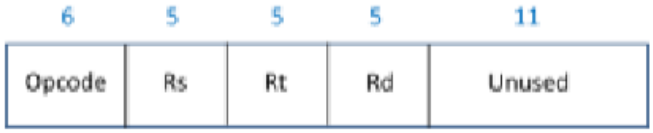
For BZ (0xE): Return true if the source register is 0.

For JR (0x10): Set the PC to the address in the source register (adjusted by -4) and return true.

For other opcodes: Return false (branch not taken).

R-type Parsing Function

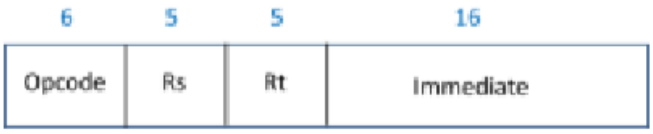
(i) R-type format:



- [10:0] ⟶ Unused
- [15:11] ⟶ Rd (Destination Register)
- [20:16] ⟶ Rt (2nd Source Register)
- [25:21] ⟶Rs (1st Source Register)
- [31:26] ⟶ Opcode

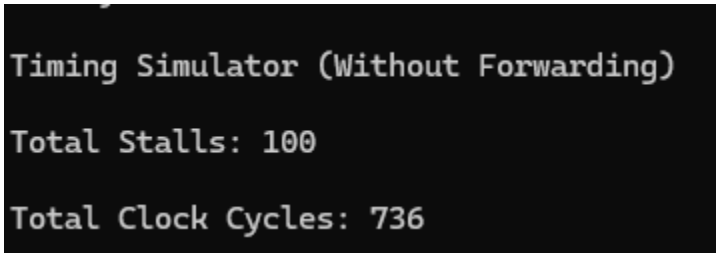
I-type Parsing Function

(ii) I-type format:

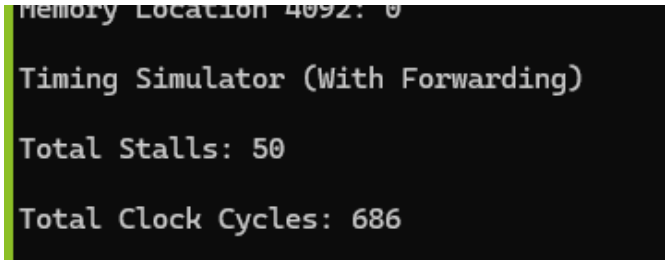


- [15:0] ⟶ Immediate Value
- [20:16] ⟶ Rt (Destination Register)
- [25:21] ⟶ Rs (Source Register)
- [31:26] ⟶ Opcode

Total stalls in Mode 1:



Total stalls in Mode 2:



As we noticed, total stalls in Mode 1 (without forwarding) is 100 and total stall in Mode 2(with forwarding) is 50, so using forwarding we decreased stalls to 50%, which means speedup is increased to double as below:

Speedup=Stalls in Mode 1/Stalls in Mode2=100/50=2

Testing Document and Discussion

(https://docs.google.com/spreadsheets/d/19fFJFA-K0ZAEee5rNZPSYR05Sd7kwwYZCv1gQDp59NI/edit#gid=1149716745) [Link encase its hard to read the screenshots]
Our testing file is designed to facilitate the simulation of various instructions by allowing us to input the opcode, Rt, Rs/Rd, and immediate values, which then compile into hexadecimal code representing the corresponding instructions. This setup enables us to construct instruction sets to validate the functionality of our MIPS-lite CPU simulator.
Our testing covers all logical and arithmetic instructions, including scenarios where we initialize registers with values, store them in memory, retrieve them, and perform subsequent mathematical operations. Specifically, the test cases include:

- Logical and Arithmetic Instructions: Comprehensive coverage of all implemented logical (AND, OR, XOR) and arithmetic (ADD, SUB, MUL) instructions.
- Control Instructions: Branch on Zero (BZ), Branch if Equal (BEQ), and Jump Register (JR) instructions. These are configured to ensure they operate correctly by checking the values in R22 and R23, which should remain zero if the control instructions function as expected.

Our test results indicate that the simulator operates as intended, correctly loading registers with values, performing operations, and manipulating memory contents. The final outputs, including the machine state and execution results, align with the expected outcomes except for discrepancies in the final clock cycle and stall cycle counts.
One issue we encountered involves saving the value 25 at memory address 1400 from the supplied memory image (MemImg). While the addresses 1404 and 1408 correctly store the intended values, address 1400 does not. This anomaly is currently under investigation to determine the cause and rectify it.

Following our presentation, we identified errors in the provided example memory image (MemImg) file. This discovery is likely the reason we had to adjust the immediate value for all I-type instructions to ensure the simulation worked correctly. During our initial simulation, we assumed there was an error in our program, and modified the immediate value (shifting left 2 bits. This was assumed because the MIPS word is 4 bytes) appeared to resolve the issue, as the register outputs matched the expected results.

However, upon further investigation and testing, we found that without shifting the immediate value, all the instructions we were tasked with implementing function correctly. Our test cases demonstrate that the logical and arithmetic operations perform as expected, validating the correctness of our simulator. Detailed testing results, which include various scenarios and edge cases, support this finding and are documented extensively in the provided link and visual aids below.

This insight highlights the importance of verifying the integrity of provided input files and demonstrates our simulator's robustness in handling different instruction sets and scenarios.

Expected Output VS Actual (everything matches, all instructions are used)

	binary value	Register #	R Values Expected
	0	0	0
	1	1	1
	10	2	2
	11	3	3
	100	4	-1
	101	5	-5
	110	6	-4
	111	7	5
	1000	8	10
	1001	9	50
	1010	10	15
	1011	11	15
	1100	12	10
	1101	13	2
	1110	14	5
	1111	15	5
	10000	16	15
	10001	17	1
	10010	18	2
	10011	19	3
	10100	20	-1
	10101	21	-5
	10110	22	0
	10111	23	5
	11000	24	188
	11001	25	-8
	11010	26	-4
	11011	27	0
	11100	28	10
	11101	29	0
	11110	30	0
	11111	31	0
	flag		
	PC		248

RUN OF THIS CODE

```
Instruction Counts
Total Number of Instructions: 67
Arithmetic Instructions: 39
Logical Instructions: 6
Memory Access Instructions: 6
Control Transfer Instructions: 8
Final Register States (Decimal)

Program Counter: 248
R[0]: 0
R[1]: 1
R[2]: 2
R[3]: 3
R[4]: -1
R[5]: -5
R[6]: -4
R[7]: 5
R[8]: 10
R[9]: 50
R[10]: 15
R[11]: 15
R[12]: 10
R[13]: 2
R[14]: 5
R[15]: 5
R[16]: 15
R[17]: 1
R[18]: 2
R[19]: 3
R[20]: -1
R[21]: -5
R[22]: 0
R[23]: 0
R[24]: 188
R[25]: -8
R[26]: -4
R[27]: 0
R[28]: 10
R[29]: 0
R[30]: 0
R[31]: 0
Total Stalls: 46
Memory Location 60: 15
Memory Location 72: 1
Memory Location 76: 2
Memory Location 80: 3
Memory Location 84: -1
Memory Location 88: -5

Mode 0 Selected, no timing information


Process returned 0 (0x0)   execution time : 2.406 s
Press any key to continue.
```

Mem Add Dec	Mem Add Hex	PC (Effective Mem Addr)	al Value at that mem location
	....		
00060	3C	60	15
00064	00000040		
00068	00000044		
00072	00000048	00000072	1
00076	4C	00000076	2
00080	00000050	00000080	3
00084	00000054	00000084	-1
00088	00000058	00000088	-5

I-type Mem Loc Hex	PC Dec Value	what does it do	opcode (6 bits)	rs (5 bits)	rt (5 bits)	Imm (16 bits)	Binary #	Hex (0x#)
00000000	00000000	ADDI, R1 = R0 + 1	000001	00000	00001	0000000000000001	00000100000000010000000000000001	04010001
00000004	00000004	ADDI, R2 = R0 + 2	000001	00000	00010	0000000000000010	00000100000000100000000000000010	04020002
00000008	00000008						00000000000000000000000000000000	
C	00000012	SUBI, R4 = R0 - 1	000011	00000	00100	0000000000000001	00001100000001000000000000000001	0C040001
00000010	00000016	SUBI, R5 = R0 - 5	000011	00000	00101	0000000000000101	00001100000001010000000000000101	0C050005
00000014	00000020						00000000000000000000000000000000	
00000018	00000024	MULI, R7 = R1 * 5	000101	00001	00111	0000000000000101	00010100001001110000000000000101	14270005
1C	00000028	MULI, R8 = R1 * 10	000101	00001	01000	0000000000001010	00010100001010000000000000001010	1428000A
00000020	00000032						00000000000000000000000000000000	
00000024	00000036	ORI, R10 = R8 OR 5	000111	01000	01010	0000000000000101	00011101000010100000000000000101	1D0A0005
00000028	00000040						00000000000000000000000000000000	
2C	00000044	ANDI, R12 = R8 AND 15	001001	01000	01100	0000000000001111	00100101000011000000000000001111	250C000F
00000030	00000048						00000000000000000000000000000000	
00000034	00000052	XORI, R14 = R8 XOR 15	001011	01000	01110	0000000000001111	00101101000011100000000000001111	2D0E000F
00000038	00000056						00000000000000000000000000000000	
3C	00000060	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000040	00000064	STW, R10 -> R0 + 60	001101	00000	01010	0000000000111100	00110100000010100000000000111100	340A003C
00000044	00000068	LDW, R0 + 60 -> R16	001100	00000	10000	0000000000111100	00110000000100000000000000111100	3010003C
00000048	00000072	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
4C	00000076	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000050	00000080	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000054	00000084	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000058	00000088	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
5C	00000092	STW, R1 -> R0 + 72	001101	00000	00001	0000000001001000	00110100000000010000000001001000	34010048
00000060	00000096	STW, R2 -> R0 + 76	001101	00000	00010	0000000001001100	00110100000000100000000001001100	3402004C
00000064	00000100	STW, R3 -> R0 + 80	001101	00000	00011	0000000001010000	00110100000000110000000001010000	34030050
00000068	00000104	STW, R4 -> R0 + 84	001101	00000	00100	0000000001010100	00110100000000100000000001010100	34040054
6C	00000108	STW R5 -> R0 + 88	001101	00000	00101	0000000001011000	00110100000000101000000001011000	34050058
00000070	00000112	LDW, R0 + 72 -> R17	001100	00000	10001	0000000001001000	00110000000100010000000001001000	30110048
00000074	00000116	LDW, R0 + 76 -> R18	001100	00000	10010	0000000001001100	00110000000100100000000001001100	3012004C
00000078	00000120	LDW, R0 + 80 -> R19	001100	00000	10011	0000000001010000	00110000000100110000000001010000	30130050
7C	00000124	LDW, R0 + 84 -> R20	001100	00000	10100	0000000001010100	00110000000101000000000001010100	30140054
00000080	00000128	LDW, R0 + 88 -> R21	001100	00000	10101	0000000001011000	00110000000101010000000001011000	30150058
00000084	00000132	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000088	00000136	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
8C	00000140	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000090	00000144	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000094	00000148	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
00000098	00000152	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
9C	00000156	ADDI, R24 = R0 + 188	000001	00000	11000	0000000010111100	00000100000110000000000010111100	041800BC
A0	00000160	ADDI, R22 = R0 + 5	000001	00000	10110	0000000000000101	00000100000010110000000000000101	04160005
A4	00000164	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
A8	00000168	SUBI, R22 = R22 - 1	000011	10110	10110	0000000000000001	00001110110101100000000000000001	0ED60001
AC	00000172	BZ, R0, R22, imm = -8	001110	10110	00000	1111111111111000	00111010110000001111111111111000	3AC0FFF8
B0	00000176	JR, Rs = R24	010000	11000	00000	0000000000000000	01000011000000000000000000000000	43000000
B4	00000180	ADDI, R23 = R0 + 188	000001	00000	10111	0000000010111100	00000100000010111000000010111100	041700BC
B8	00000184	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
BC	00000188	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
C0	00000192	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
C4	00000196	ADDI, R25 = R0 + (-8)	000001	00000	11001	1111111111111000	00000100000110011111111111111000	0419FFF8
C8	00000200	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
CC	00000204	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
D0	00000208	ADDI, R26 = R0 + (-5)	000001	00000	11010	1111111111111011	00000100000011010111111111111011	041AFFFB
D4	00000212	ADDI, R26 = R26 + 1	000001	11010	11010	0000000000000001	00000111010110100000000000000001	075A0001
D8	00000216	BEQ, R0 =? R26, => +8	001111	00000	11010	0000000000001000	00111100000110100000000000001000	3C1A0008
DC	00000220	ADDI, R27 = R0 + 5	000001	11011	00000	0000000000000101	00000111011000000000000000000101	07600005
E0	00000224	BEQ, R14 =? R15, => +8	001111	01111	1110	0000000000001000	00111101111010110000000000001000	3DEE0008
E4	00000228	ADDI, R28 = R0 + 5	000001	00000	11100	0000000000000101	00000100000111000000000000000101	041C0005
E8	00000232	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
EC	00000236	ADDI, R28 = R0 + 10	000001	00000	11100	0000000000001010	00000100000011100000000000001010	041C000A
F0	00000240	ADD, ALL 0 LINE	000000	00000	00000	0000000000000000	00000000000000000000000000000000	00000000
F4	00000244	HALT	010001	00000	00000	0000000000000000	01000100000000000000000000000000	44000000

I-Type testing

What does it do	opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	unused (11 bits)	Binary #	Hex (0x#)
ADD R3 = R1 + R2	000000	00001	00010	00011	0000000000000000	00000000001000100001100000000000	00221800
						00000000000000000000000000000000	
						00000000000000000000000000000000	
SUB R6 = R5 - R4	000010	00101	00100	00110	0000000000000000	00001000101001000011000000000000	08A43000
						00000000000000000000000000000000	
						00000000000000000000000000000000	
MUL R9 = R7 * R8	000100	00111	01000	01001	0000000000000000	00010000111010000100100000000000	10E84800
						00000000000000000000000000000000	
OR, R11 = R7 OR R8	000110	00111	01000	01011	0000000000000000	00011000111010000101100000000000	18E85800
						00000000000000000000000000000000	
AND, R13 = R8 & R3	001000	1000	00011	01101	0000000000000000	00100001000000110110100000000000	21036800
						00000000000000000000000000000000	00000000
XOR, R15 = R8 XOR R10	001010	01000	01010	01111	0000000000000000	00101001000010100111100000000000	290A7800

R-type testing

Simulation of test file supplied in final project demo :

Registers(Mode 0):

```
Please select a mode from the following list...
Mode 0 - Pipeline Stages without hazard tracking
Mode 1 - Timing Without Forwarding
Mode 2 - Timing With Forwarding
(0, 1, or 2): 0
You have selected mode: 0, Pipeline Stages



Instruction Counts
Total Number of Instructions: 638
Arithmetic Instructions: 333
Logical Instructions: 50
Memory Access Instructions: 103
Control Transfer Instructions: 152
Final Register States (Decimal)

Program Counter: 100
R[0]: 0
R[1]: 1200
R[2]: 1400
R[3]: 100
R[4]: 50
R[5]: 50
R[6]: 0
R[7]: 25
R[8]: 2550
R[9]: 1275
R[10]: 50
R[11]: 50
R[12]: 32
R[13]: 0
R[14]: 0
R[15]: 0
R[16]: 0
R[17]: 0
R[18]: 0
R[19]: 0
R[20]: 0
R[21]: 0
R[22]: 0
R[23]: 0
R[24]: 0
R[25]: 0
R[26]: 0
R[27]: 0
R[28]: 0
R[29]: 0
R[30]: 0
R[31]: 0
Memory Location 0: 67175400
```

Memory locations(Mode 0):

```
Memory Location 0: 67175400
Memory Location 4: 67241136
Memory Location 8: 14336
Memory Location 12: 16384
Memory Location 16: 18432
Memory Location 20: 20480
Memory Location 24: 67829810
Memory Location 28: 67895328
Memory Location 32: 1028325389
Memory Location 36: 807600128
Memory Location 40: 809762816
Memory Location 44: 140781568
Memory Location 48: 614858753
Memory Location 52: 952107010
Memory Location 56: 82247681
Memory Location 60: 16990208
Memory Location 64: 19154944
Memory Location 68: 69271556
Memory Location 72: 71434244
Memory Location 76: 88735745
Memory Location 80: 1098907648
Memory Location 84: 0
Memory Location 88: 872940924
Memory Location 92: 873006464
Memory Location 96: 1140850688
Memory Location 100: 0
```

```
Memory Location 1312: 0
Memory Location 1316: 0
Memory Location 1320: 0
Memory Location 1324: 0
Memory Location 1328: 0
Memory Location 1332: 0
Memory Location 1336: 0
Memory Location 1340: 0
Memory Location 1344: 0
Memory Location 1348: 0
Memory Location 1352: 0
Memory Location 1356: 0
Memory Location 1360: 0
Memory Location 1364: 0
Memory Location 1368: 0
Memory Location 1372: 0
Memory Location 1376: 0
Memory Location 1380: 0
Memory Location 1384: 0
Memory Location 1388: 0
Memory Location 1392: 0
Memory Location 1396: 0
Memory Location 1400: 0
Memory Location 1404: 2550
Memory Location 1408: 1275
Memory Location 1412: 0
Memory Location 1416: 0
Memory Location 1420: 0
Memory Location 1424: 0
Memory Location 1428: 0
Memory Location 1432: 0
Memory Location 1436: 0
```

Registers(Mode 1):

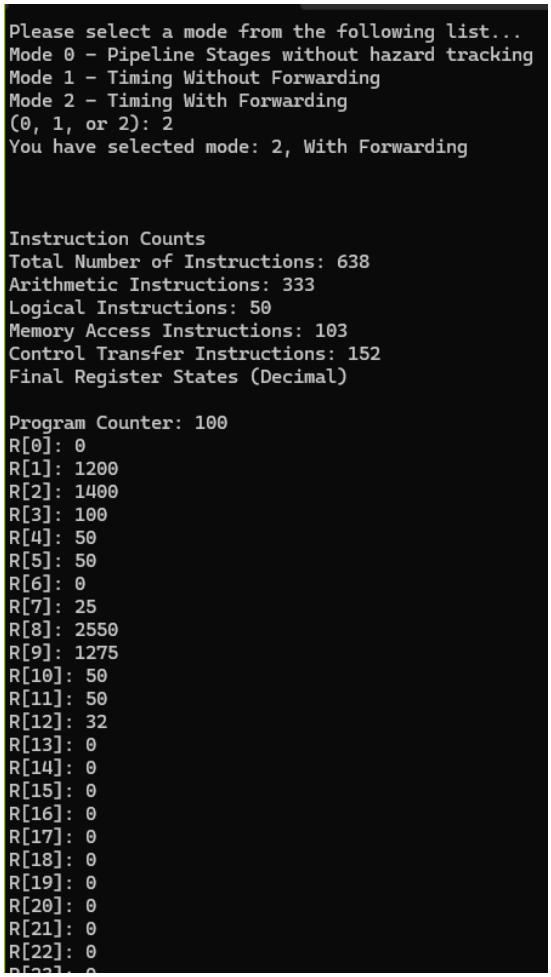
```
Please select a mode from the following list...
Mode 0 - Pipeline Stages without hazard tracking
Mode 1 - Timing Without Forwarding
Mode 2 - Timing With Forwarding
(0, 1, or 2): 1
You have selected mode: 1, Without Forwarding



Instruction Counts
Total Number of Instructions: 638
Arithmetic Instructions: 333
Logical Instructions: 50
Memory Access Instructions: 103
Control Transfer Instructions: 152
Final Register States (Decimal)

Program Counter: 100
R[0]: 0
R[1]: 1200
R[2]: 1400
R[3]: 100
R[4]: 50
R[5]: 50
R[6]: 0
R[7]: 25
R[8]: 2550
R[9]: 1275
R[10]: 50
R[11]: 50
R[12]: 32
R[13]: 0
R[14]: 0
R[15]: 0
R[16]: 0
R[17]: 0
R[18]: 0
R[19]: 0
R[20]: 0
R[21]: 0
R[22]: 0
R[23]: 0
R[24]: 0
R[25]: 0
R[26]: 0
R[27]: 0
R[28]: 0
R[29]: 0
R[30]: 0
R[31]: 0
```

Memory locations(Mode 1):

```
Memory Location 0: 67175400
Memory Location 4: 67241136
Memory Location 8: 14336
Memory Location 12: 16384
Memory Location 16: 18432
Memory Location 20: 20480
Memory Location 24: 67829810
Memory Location 28: 67895328
Memory Location 32: 1028325389
Memory Location 36: 807600128
Memory Location 40: 809762816
Memory Location 44: 140781568
Memory Location 48: 614858753
Memory Location 52: 952107010
Memory Location 56: 82247681
Memory Location 60: 16990208
Memory Location 64: 19154944
Memory Location 68: 69271556
Memory Location 72: 71434244
Memory Location 76: 88735745
Memory Location 80: 1098907648
Memory Location 84: 0
Memory Location 88: 872940924
Memory Location 92: 873006464
Memory Location 96: 1140850688
Memory Location 100: 0
Memory Location 104: 0
```

```
Memory Location 1292: 0
Memory Location 1296: 0
Memory Location 1300: 0
Memory Location 1304: 0
Memory Location 1308: 0
Memory Location 1312: 0
Memory Location 1316: 0
Memory Location 1320: 0
Memory Location 1324: 0
Memory Location 1328: 0
Memory Location 1332: 0
Memory Location 1336: 0
Memory Location 1340: 0
Memory Location 1344: 0
Memory Location 1348: 0
Memory Location 1352: 0
Memory Location 1356: 0
Memory Location 1360: 0
Memory Location 1364: 0
Memory Location 1368: 0
Memory Location 1372: 0
Memory Location 1376: 0
Memory Location 1380: 0
Memory Location 1384: 0
Memory Location 1388: 0
Memory Location 1392: 0
Memory Location 1396: 0
Memory Location 1400: 0
Memory Location 1404: 2550
Memory Location 1408: 1275
Memory Location 1412: 0
Memory Location 1416: 0
Memory Location 1420: 0
Memory Location 1424: 0
Memory Location 1428: 0
Memory Location 1432: 0
Memory Location 1436: 0
Memory Location 1440: 0
Memory Location 1444: 0
Memory Location 1448: 0
Memory Location 1452: 0
Memory Location 1456: 0
```

Registers(Mode 2):



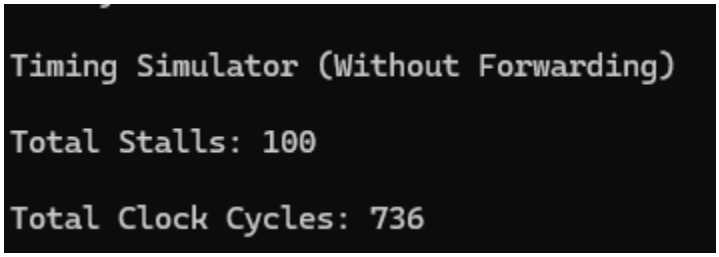
```
Please select a mode from the following list...
Mode 0 - Pipeline Stages without hazard tracking
Mode 1 - Timing Without Forwarding
Mode 2 - Timing With Forwarding
(0, 1, or 2): 2
You have selected mode: 2, With Forwarding



Instruction Counts
Total Number of Instructions: 638
Arithmetic Instructions: 333
Logical Instructions: 50
Memory Access Instructions: 103
Control Transfer Instructions: 152
Final Register States (Decimal)

Program Counter: 100
R[0]: 0
R[1]: 1200
R[2]: 1400
R[3]: 100
R[4]: 50
R[5]: 50
R[6]: 0
R[7]: 25
R[8]: 2550
R[9]: 1275
R[10]: 50
R[11]: 50
R[12]: 32
R[13]: 0
R[14]: 0
R[15]: 0
R[16]: 0
R[17]: 0
R[18]: 0
R[19]: 0
R[20]: 0
R[21]: 0
R[22]: 0
```

Memory locations(Mode 2):

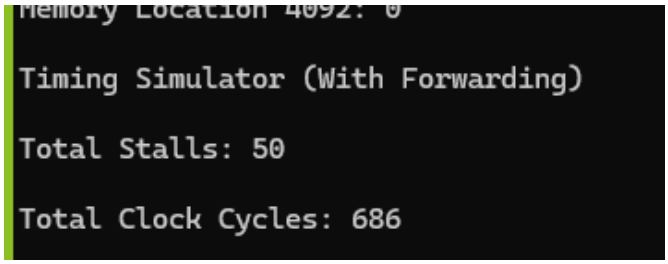
```
R[31]: 0
Memory Location 0: 67175400
Memory Location 4: 67241136
Memory Location 8: 14336
Memory Location 12: 16384
Memory Location 16: 18432
Memory Location 20: 20480
Memory Location 24: 67829810
Memory Location 28: 67895328
Memory Location 32: 1028325389
Memory Location 36: 807600128
Memory Location 40: 809762816
Memory Location 44: 140781568
Memory Location 48: 614858753
Memory Location 52: 952107010
Memory Location 56: 82247681
Memory Location 60: 16990208
Memory Location 64: 19154944
Memory Location 68: 69271556
Memory Location 72: 71434244
Memory Location 76: 88735745
Memory Location 80: 1098907648
Memory Location 84: 0
Memory Location 88: 872940924
Memory Location 92: 873006464
Memory Location 96: 1140850688
```

```
Memory Location 1304: 0
Memory Location 1308: 0
Memory Location 1312: 0
Memory Location 1316: 0
Memory Location 1320: 0
Memory Location 1324: 0
Memory Location 1328: 0
Memory Location 1332: 0
Memory Location 1336: 0
Memory Location 1340: 0
Memory Location 1344: 0
Memory Location 1348: 0
Memory Location 1352: 0
Memory Location 1356: 0
Memory Location 1360: 0
Memory Location 1364: 0
Memory Location 1368: 0
Memory Location 1372: 0
Memory Location 1376: 0
Memory Location 1380: 0
Memory Location 1384: 0
Memory Location 1388: 0
Memory Location 1392: 0
Memory Location 1396: 0
Memory Location 1400: 0
Memory Location 1404: 2550
Memory Location 1408: 1275
Memory Location 1412: 0
Memory Location 1416: 0
Memory Location 1420: 0
Memory Location 1424: 0
Memory Location 1428: 0
Memory Location 1432: 0
Memory Location 1436: 0
Memory Location 1440: 0
Memory Location 1444: 0
Memory Location 1448: 0
Memory Location 1452: 0
Memory Location 1456: 0
Memory Location 1460: 0
Memory Location 1464: 0
Memory Location 1468: 0
Memory Location 1472: 0
Memory Location 1476: 0
Memory Location 1480: 0
Memory Location 1484: 0
Memory Location 1488: 0
Memory Location 1492: 0
Memory Location 1496: 0
Memory Location 1500: 0
Memory Location 1504: 0
```

Total stalls in Mode 1:



Total stalls in Mode 2:



As we noticed, total stalls in Mode 1 (without forwarding) is 100 and total stall in Mode 2(with forwarding) is 50, so using forwarding we decreased stalls to 50%, which means speedup is increased to double as below:

Speedup=Stalls in Mode 1/Stalls in Mode2=100/50=2

C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>
#define NUM_INSTRUCTIONS 5000

// Function declarations
void ClockSignal();
void Fetch();
void Decode();
void Execute();
void MemAccess();
void WriteBack();
void CheckForHazard();
void PrintRegisters();
void CheckForMispredictedBranch();
void getMode(int *mode);
int branch_taken_condition();

// Define the structure for storing instruction information
struct MemoryRegUnit {
   int Instruction; // Fetched Instruction Variable
   int opcode;      // 6 bits
   int Reg_S;       // 5 bits
   int Reg_T;       // 5 bits
   int Reg_D;       // 5 bits
   int Imm;     // 16 bits
   char Type;       // R or I
   int VisitedFlag; // Did we update this memory address? True - 1, False - 0
};

// Array to hold the instruction memory
struct MemoryRegUnit InstructionMemory[NUM_INSTRUCTIONS];
int MemoryImage[NUM_INSTRUCTIONS]; // Dummy variable to store hex values from file

int R[32] = {0}; // 32 MIPS registers
```

```
int PC = 0;                 // Program Counter
int ClockCount = 0;         // Clock Cycle Counter
int InstructionCount = 1;   // Instruction Counter
int StallCount = 0;
int A;                      // Temp variable for address of Reg + Imm
char line[1024];
int LineOfFile = 0;         // Counter for the line number
int mode;
char modeName[50];

int ArithmeticCount = 0;
int LogicalCount = 0;
int MemAccessCount = 1;
int ControlTransferCount = 0;

int main() {
   FILE *file;
   char filename[] = "memimg.txt";  // Name of the file containing hex values

   getMode(&mode);
   printf("You have selected mode: %d, %s\n", mode, modeName);

   // Open the file for reading
   file = fopen(filename, "r");
   if (file == NULL) {
      perror("Failed to open file");
      return EXIT_FAILURE;
   }

   while (fgets(line, sizeof(line), file)) {
      // Convert the hexadecimal string to signed int
      if (sscanf(line, "%x", &MemoryImage[LineOfFile]) == 1) {
         LineOfFile++;
      } else {
         // Failed to convert string to number
         fprintf(stderr, "Failed to parse line as hexadecimal: %s", line);
      }
```

```
    }
    // Close the file
    fclose(file);

    // Execute the instructions
    while (PC / 4 < LineOfFile) { // Changed from <= to < to avoid out-of-bounds
        ClockSignal();
    }

    return EXIT_SUCCESS;
}

void ClockSignal() {
    // Run the pipeline stages
    R[0] = 0;
    Fetch();
    Decode();
    Execute();
    MemAccess();
    WriteBack();

    CheckForHazard();
    CheckForMispredictedBranch();
    PC += 4;
    ClockCount++;
}

void Fetch() {
    int currentMemImgIndex = PC / 4;
    if (currentMemImgIndex >= 0 && currentMemImgIndex < NUM_INSTRUCTIONS) {
        InstructionMemory[currentMemImgIndex].Instruction =
MemoryImage[currentMemImgIndex]; // Store the instruction
        InstructionCount++;
    }
}

void Decode() {
    int currentMemImgIndex = PC / 4;
```

```
   if (currentMemImgIndex >= 0 && currentMemImgIndex < NUM_INSTRUCTIONS) {
      InstructionMemory[currentMemImgIndex].opcode =
InstructionMemory[currentMemImgIndex].Instruction >> 26; // Isolate the opcode

      // Adjust the type based on specific opcode logic
      int type = (InstructionMemory[currentMemImgIndex].opcode == 0x0 ||
            InstructionMemory[currentMemImgIndex].opcode == 0x2 ||
            InstructionMemory[currentMemImgIndex].opcode == 0x4 ||
            InstructionMemory[currentMemImgIndex].opcode == 0x6 ||
            InstructionMemory[currentMemImgIndex].opcode == 0x8 ||
            InstructionMemory[currentMemImgIndex].opcode == 0xA) ? 0 : 1; // 0 for R-type, 1
for I-type

      if (type == 0) { // R-Type
         InstructionMemory[currentMemImgIndex].Reg_S =
(InstructionMemory[currentMemImgIndex].Instruction >> 21) & 0x1F;
         InstructionMemory[currentMemImgIndex].Reg_T =
(InstructionMemory[currentMemImgIndex].Instruction >> 16) & 0x1F;
         InstructionMemory[currentMemImgIndex].Reg_D =
(InstructionMemory[currentMemImgIndex].Instruction >> 11) & 0x1F;
         InstructionMemory[currentMemImgIndex].Type = 'R';
      } else { // I-Type
         InstructionMemory[currentMemImgIndex].Reg_S =
(InstructionMemory[currentMemImgIndex].Instruction >> 21) & 0x1F;
         InstructionMemory[currentMemImgIndex].Reg_T =
(InstructionMemory[currentMemImgIndex].Instruction >> 16) & 0x1F;
         InstructionMemory[currentMemImgIndex].Imm =
InstructionMemory[currentMemImgIndex].Instruction & 0xFFFF;
         InstructionMemory[currentMemImgIndex].Type = 'I';
      }
            // Sign-extend the 16-bit immediate value to a 32-bit signed integer
      if (InstructionMemory[currentMemImgIndex].Imm & 0x8000) { // Check if the sign bit is 1
          InstructionMemory[currentMemImgIndex].Imm |= 0xFFFF0000; // Extend the sign to
the upper 16 bits
      }
   }
}
```

```
void Execute() {
   int currentMemImgIndex = PC / 4;

   if (currentMemImgIndex >= 0 && currentMemImgIndex < NUM_INSTRUCTIONS) {
      int opcode = InstructionMemory[currentMemImgIndex].opcode;

      switch (opcode) {
         case 0: // ADD (R-type)
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] +
R[InstructionMemory[currentMemImgIndex].Reg_T];
            ArithmeticCount++;
            break;

         case 1: // ADDI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] +
InstructionMemory[currentMemImgIndex].Imm;
            ArithmeticCount++;
            break;

         case 2: // SUB (R-type)
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] -
R[InstructionMemory[currentMemImgIndex].Reg_T];
            ArithmeticCount++;
            break;

         case 3: // SUBI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] -
InstructionMemory[currentMemImgIndex].Imm;
            ArithmeticCount++;
            break;

         case 4: // MUL (R-type)
```

```
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] *
R[InstructionMemory[currentMemImgIndex].Reg_T];
            ArithmeticCount++;
            break;

        case 5: // MULI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] *
InstructionMemory[currentMemImgIndex].Imm;
            ArithmeticCount++;
            break;

        case 6: // OR (R-type)
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] |
R[InstructionMemory[currentMemImgIndex].Reg_T];
            LogicalCount++;
            break;

        case 7: // ORI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] |
InstructionMemory[currentMemImgIndex].Imm;
            LogicalCount++;
            break;

        case 8: // AND (R-type)
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] &
R[InstructionMemory[currentMemImgIndex].Reg_T];
            LogicalCount++;
            break;

        case 9: // ANDI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] &
InstructionMemory[currentMemImgIndex].Imm;
```

```
            LogicalCount++;
            break;

        case 10: // XOR (R-type)
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_S] ^
R[InstructionMemory[currentMemImgIndex].Reg_T];
            LogicalCount++;
            break;

        case 11: // XORI (I-type)
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_S] ^
InstructionMemory[currentMemImgIndex].Imm;
            LogicalCount++;
            break;

        case 12: // LDW (I-type)
            A = R[InstructionMemory[currentMemImgIndex].Reg_S] +
InstructionMemory[currentMemImgIndex].Imm;
            break;

        case 13: // STW (I-type)
            A = R[InstructionMemory[currentMemImgIndex].Reg_S] +
InstructionMemory[currentMemImgIndex].Imm;
            break;

        case 14: // BZ (I-type)
            if (R[InstructionMemory[currentMemImgIndex].Reg_S] != 0) {
                PC = PC + (InstructionMemory[currentMemImgIndex].Imm << 2) - 4; // Imm is
shifted left by 2 for MIPS word = 4 bytes, indexing issue somewhere needs - 4
                ClockCount += 2; // Add stall cycles for the branch
                StallCount += 2;
            }
            ControlTransferCount++;
            break;

        case 15: // BEQ (I-type)
```

```
            if (R[InstructionMemory[currentMemImgIndex].Reg_S] ==
R[InstructionMemory[currentMemImgIndex].Reg_T]) {
                PC += (InstructionMemory[currentMemImgIndex].Imm << 2); // Imm is shifted left
by 2 for MIPS word = 4 bytes
            }
            ControlTransferCount++;
            break;

        case 16: // JR
            PC = R[InstructionMemory[currentMemImgIndex].Reg_S] - 4; // Subtract 4 to
compensate for the PC += 4 in ClockSignal
            ControlTransferCount++;
            break;

        case 17: // Halt
            ControlTransferCount++;
            printf("\n\n\n");
            PrintRegisters();
            exit(0);
            break;

        default:
            printf("Unsupported opcode: %x\n", opcode);
            break;
      }
   }
}

void MemAccess() {
   int currentMemImgIndex = PC / 4;
   int targetMemImgIndex = A / 4;
   int opcode = InstructionMemory[currentMemImgIndex].opcode;

   switch (opcode) {
      // LDW
      case 12:
         R[InstructionMemory[currentMemImgIndex].Reg_T] =
MemoryImage[targetMemImgIndex];
```

```
            MemAccessCount++;
            break;

        // STW
        case 13:
            InstructionMemory[targetMemImgIndex].Instruction =
R[InstructionMemory[currentMemImgIndex].Reg_T];
            InstructionMemory[targetMemImgIndex].VisitedFlag = 1;
            MemAccessCount++;
            break;
    }
}

void WriteBack() {
    // Writing results back to the register file
    int currentMemImgIndex = PC / 4;
    int opcode = InstructionMemory[currentMemImgIndex].opcode;

    switch (opcode) {
        case 0: // ADD
        case 2: // SUB
        case 4: // MUL
        case 6: // OR
        case 8: // AND
        case 10: // XOR
            // R-type instructions write to Reg_D
            R[InstructionMemory[currentMemImgIndex].Reg_D] =
R[InstructionMemory[currentMemImgIndex].Reg_D];
            break;

        case 1: // ADDI
        case 3: // SUBI
        case 5: // MULI
        case 7: // ORI
        case 9: // ANDI
        case 11: // XORI
        case 12: // LDW
            // I-type instructions write to Reg_T
```

```
            R[InstructionMemory[currentMemImgIndex].Reg_T] =
R[InstructionMemory[currentMemImgIndex].Reg_T];
            break;

        default:
            break;
    }
}


void PrintRegisters() {
    printf("Instruction Counts\n");

    printf("Total Number of Instructions: %d\n", InstructionCount);
    printf("Arithmetic Instructions: %d\n", ArithmeticCount);
    printf("Logical Instructions: %d\n", LogicalCount);
    printf("Memory Access Instructions: %d\n", MemAccessCount);
    printf("Control Transfer Instructions: %d\n", ControlTransferCount);

    printf("Final Register States (Decimal)\n\n");
    printf("Program Counter: %d\n", PC + 4);

    for (int i = 0; i < 32; i++) {
        printf("R[%d]: %d\n", i, R[i]);
    }

    for (int i = 0; i < 1024; i++) {
        //if (InstructionMemory[i].VisitedFlag == 1)
            {
            printf("Memory Location %d: %d\n", i * 4, InstructionMemory[i].Instruction);
        }
    }

    if (mode == 1) {
        printf("\nTiming Simulator (Without Forwarding)\n\n");
        printf("Total Stalls: %d\n", StallCount);
        printf("\nTotal Clock Cycles: %d\n", ClockCount);
    } else if (mode == 2) {
```

```
        printf("\nTiming Simulator (With Forwarding)\n\n");
        printf("Total Stalls: %d\n", StallCount);
        printf("\nTotal Clock Cycles: %d\n", ClockCount);
    } else if (mode == 0) {
        printf("\n");
    }
}

void CheckForHazard() {
    int currentMemImgIndex = PC / 4;

    // Avoid hazard checking for initial instructions
    if (PC == 0 || PC == 4) {
        return;
    }

    // Skip hazard checking for R0 registers
    if (InstructionMemory[currentMemImgIndex].Reg_S == 0 ||
InstructionMemory[currentMemImgIndex].Reg_T == 0) {
        return;
    }

    switch (InstructionMemory[currentMemImgIndex].Type) {
        case 'R':
            // If R type
            if ((InstructionMemory[currentMemImgIndex].Reg_S ==
InstructionMemory[currentMemImgIndex - 4].Reg_T) ||
                (InstructionMemory[currentMemImgIndex].Reg_T ==
InstructionMemory[currentMemImgIndex - 4].Reg_T)) {
                if (mode == 2) {
                    break;
                }
                ClockCount += 2;
                StallCount += 2;
                break;
            } else if ((InstructionMemory[currentMemImgIndex].Reg_S ==
InstructionMemory[currentMemImgIndex - 8].Reg_T) ||
```

```
                (InstructionMemory[currentMemImgIndex].Reg_T ==
InstructionMemory[currentMemImgIndex - 8].Reg_T)) {
            if (mode == 2) {
               break;
            }
            ClockCount++;
            StallCount++;
            break;
         }
         break;

      case 'I':
         // If I type
         if (InstructionMemory[currentMemImgIndex].Reg_S ==
InstructionMemory[currentMemImgIndex - 4].Reg_T) {
            if (mode == 2) {
               break;
            }
            ClockCount += 2;
            StallCount += 2;
            break;

         } else if (InstructionMemory[currentMemImgIndex].Reg_S ==
InstructionMemory[currentMemImgIndex - 8].Reg_D) {
            if (mode == 2) {
               break;
            }
            ClockCount++;
            StallCount++;
            break;
         }
         break;

      default:
         printf("\n\n\n++++++++++++Unknown Instruction Type++++++++++++\n");
         printf("Line: %d", currentMemImgIndex);
         break;
   }
```

```
}

void CheckForMispredictedBranch() {
   int currentMemImgIndex = PC / 4;
   // Assume that branches and jumps are denoted by a specific set of opcodes

   // Opcodes for BZ, BEQ, and JR defined in hexadecimal
   int branch_opcodes[] = {0xE, 0xF, 0x10};

   int is_branch = 0; // Initialize is_branch to 0

   for (int i = 0; i < sizeof(branch_opcodes) / sizeof(branch_opcodes[0]); i++) {
      if (InstructionMemory[currentMemImgIndex].opcode == branch_opcodes[i]) {
         is_branch = 1;
         break;
      }
   }

   if (is_branch) {
      // Check if the branch is taken
      if (branch_taken_condition()) {
        if (mode == 2) {
            return;
        }
         ClockCount += 2; // Add stall cycles for flushing
         StallCount += 2;
         //Flush the next two instructions
         if ((PC + 4) / 4 < NUM_INSTRUCTIONS) {
            InstructionMemory[(PC + 4) / 4].Instruction = 0; // NOP
         }
         if ((PC + 8) / 4 < NUM_INSTRUCTIONS) {
            InstructionMemory[(PC + 8) / 4].Instruction = 0; // NOP
         }

      }
   }
}
```

```
int branch_taken_condition() {
   int currentMemImgIndex = PC / 4;

   // Get the opcode of the current instruction
   int opcode = InstructionMemory[currentMemImgIndex].opcode;

   // Check if the branch is taken based on the opcode
   switch (opcode) {
      case 0xF: // BEQ opcode
         // BEQ: Branch if Rs == Rt
         return R[InstructionMemory[currentMemImgIndex].Reg_S] ==
R[InstructionMemory[currentMemImgIndex].Reg_T];

      case 0xE: // BZ opcode
         // BZ: Branch if Rs == 0
         return R[InstructionMemory[currentMemImgIndex].Reg_S] == 0;

      case 0x10: // JR opcode
         // JR: Jump to address in Rs
         PC = R[InstructionMemory[currentMemImgIndex].Reg_S] - 4; // Adjust PC for next
instruction fetch
         return 1;

      default:
         // For other opcodes, branch is not taken
         return 0;
   }
}

void getMode(int *mode) {
   int input;
   while (1) {
      printf("Please select a mode from the following list...\n");
      printf("Mode 0 - Pipeline Stages without hazard tracking\n");
      printf("Mode 1 - Timing Without Forwarding\n");
      printf("Mode 2 - Timing With Forwarding\n");
      printf("(0, 1, or 2): ");
      int result = scanf("%d", &input);
```

```
        // Check if the input is valid and within the range
        if (result == 1 && (input == 0 || input == 1 || input == 2)) {
            *mode = input;
            break;
        } else {
            printf("Invalid input. Please enter 0, 1, or 2.\n");
            // Clear the input buffer in case of invalid input
            while (getchar() != '\n');
        }
    }

    // Assign the mode name based on the mode
    switch (*mode) {
        case 0:
            strcpy(modeName, "Pipeline Stages");
            break;
        case 1:
            strcpy(modeName, "Without Forwarding");
            break;
        case 2:
            strcpy(modeName, "With Forwarding");
            break;
        default:
            strcpy(modeName, "Unknown Mode");
            break;
    }
}
```