DESIGN AND SIMULATION OF A SINGLE-CYCLE / PIPELINED RISC-V PROCESSOR

RANIA OSSAMA HASSAN 231001334
ADHAM AHMED MOHAMED 221001783
YOMNA MEDHAT SAAD 231000762
MOHAMMED HESHAM 231000359
JOMANA AMIN 231001427

AGENDA

- Overview of the RISC-V processor project
- ALU & Arithmetic Logic Design
- Control Unit (Main + ALU Control)
- Register File & Immediate Generator
- Program Counter, Branch Logic & PC Update
- Instruction Memory, Data Memory & Top-Level Integration
- Simulation Results – Testbenches, PC updates, instruction execution, waveform verification

OVERVIEW

- Design and implement a RISC-V processor in Verilog HDL.
- Supports R-Type, I-Type, Load/Store, and Branch instructions.
- Simulate both single-cycle and pipelined versions.
- Understand the internal CPU structure: instruction fetch, decode, execute, memory, and write-back.
- Verify module functionality using simulation waveforms.
- Gain hands-on experience in CPU design, control signals, and datapath integration.

ALU & ARITHMETIC LOGIC DESIGN

- Located in the Execute stage of the 5-stage pipe
- Performs all RISC-V operations
- Uses signed comparison for slt (-5 < 10 result = 1
- Outputs a Zero flag using beq to detect equality
- Purely combinational (no clock or reset needed)
- Controlled by 4-bit ALUCtl from the ALU Control U
- Verified with full testbench (all operations testec

add / addi	0010
sub	0110
slt / slti	0100
and / andi	0000
or / ori	0001
xor / xori	0111
sll / slli	0011
srl / srli	1000
sra / srai	1010

ALU & ARITHMETIC LOGIC DESIGN

ALU Control Codes (ALUCtl)

add / addi	0010
sub	0110
slt / slti	0100
and / andi	0000
or / ori	0001
xor / xori	0111
sll / slli	0011
srl / srli	1000
sra / srai	1010

CONTROL UNIT

- Located in the Decode stage of the 5-stage pipeline
- Generates control signals based on opcode and finds what type of instruction the ALU will do (R-type, I-type ... etc)
- Purely combinational (no clock or reset needed)
- Verified with full testbench

Instruction Type	[0]	[1]	[2]	[3]	[4]	[5]	[6]
R-Type	1	1	0	0	1	1	0
I-type	1	1	0	0	1	0	0
Load	1	1	0	0	0	0	0
Store	1	1	0	0	0	1	0
Branch	1	1	0	0	0	1	1

Instruction Type	RegWrite	MemRead	MemWrite	Branch	ALUSrc	MemtoReg	ALUOp
R-Type	1	0	0	0	0	0	10
I-Type	1	0	0	0	1	0	11
Load	1	1	0	0	1	1	00
Store	0	0	1	0	1	X	00
Branch	0	0	0	1	0	X	01

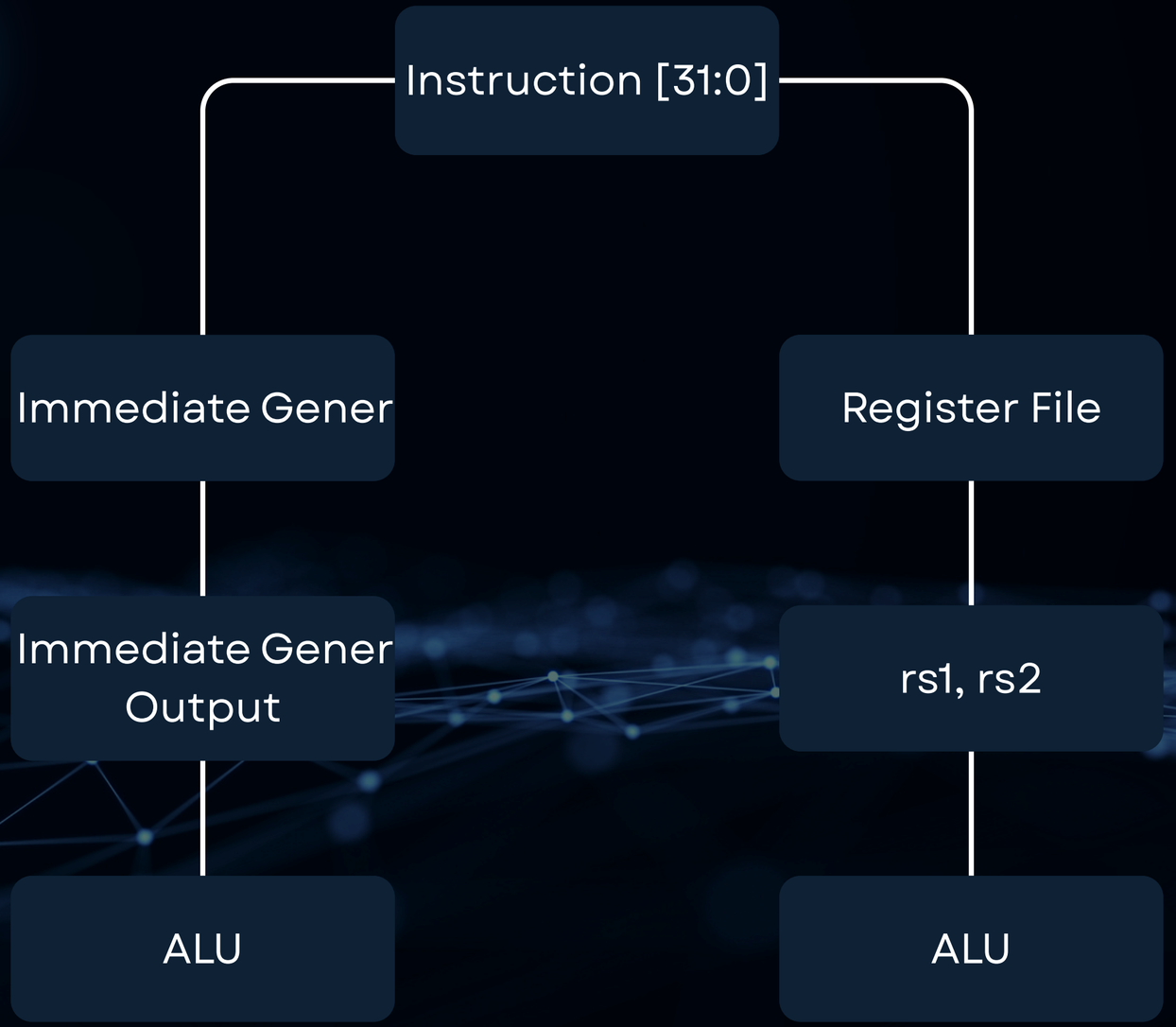
IMMEDIATE GENERATOR

- Extracts and sign-extends immediates from instructions
- I-type, S-type and B-type instructions
- Handles shift-immediates
- Combinational logic

TYPE	INSTRUCTION
I - type	**addi, andi, ld**
S - type	sd
B - type	beq
Shift	slli, srli

REGISTER FILE & IMMEDIATE GENERATOR

- 32 Registers
- x0 always 0
- Two read ports
- One write port
- Synchronous write
- Asynchronous read



PROGRAM COUNTER & BRANCH LOGIC

- Designed the Program Counter (PC) unit for instruction sequencing.
- PC handles two cases:
- Normal instruction: PC increments by 4.
- Branch taken: PC updated to branch target address.
- Implemented asynchronous reset to initialize PC to zero.
- Added control signals:
- PCWrite → allows stalling of PC.
- BranchTaken → indicates if a branch is taken.

```
module PC_Unit (
    input  wire         clk,
    input  wire         reset,

    input  wire         PCWrite,
    input  wire         BranchTaken,
    input  wire [63:0] BranchTarget,

    output reg  [63:0] PC
);

    wire [63:0] PC_plus4;
    wire [63:0] PC_next;

    assign PC_plus4 = PC + 64'd4;

    assign PC_next = (BranchTaken) ? BranchTarget : PC_plus4;

    always @(posedge clk or posedge reset) begin
        if (reset)
            PC <= 64'd0;
        else if (PCWrite)
            PC <= PC_next;
    end

endmodule
```

PROGRAM COUNTER & BRANCH LOGIC

```
`timescale 1ns / 1ps

module PC_Unit_tb;


    reg clk;
    reg reset;
    reg PCWrite;
    reg BranchTaken;
    reg [63:0] BranchTarget;


    wire [63:0] PC;


    PC_Unit dut (
        .clk(clk),
        .reset(reset),
        .PCWrite(PCWrite),
        .BranchTaken(BranchTaken),
        .BranchTarget(BranchTarget),
        .PC(PC)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 1;
        PCWrite = 1;
        BranchTaken = 0;
        BranchTarget = 64'd0;


        #10;
        reset = 0;
```

- Tested functionality using simulation:
- Normal PC increment
- Branch taken
- Branch not taken

```
        #10;
        #10;
        #10;

        PCWrite = 0;
        #20;


        PCWrite = 1;
        #10;

        BranchTaken = 1;
        BranchTarget = 64'd100;
        #10;
        BranchTaken = 0;
        #10;    // PC = 104


        #10;
        $stop;
    end

endmodule
```

RISC-V PIPELINED PROCESSOR

Pipeline Stage Operations & Decisions

- IF → ID: Instruction is fetched using the PC; PC is updated based on sequential execution or a branch decision.
- ID → EX: Instruction is decoded, control signals are generated, source registers are read, and immediates are prepared.
- EX → MEM: ALU performs arithmetic/logic operations, computes memory addresses, and evaluates branch conditions.
- MEM → WB: Data memory is accessed for load/store instructions, and the correct result is selected.
- WB: Final result is written back to the register file if required by the instruction.

Hazard Handling

Data Hazards: Forwarding unit enables ALU-to-ALU and MEM-to-ALU bypassing, ensuring zero stalls for register dependencies.

Control Hazards:

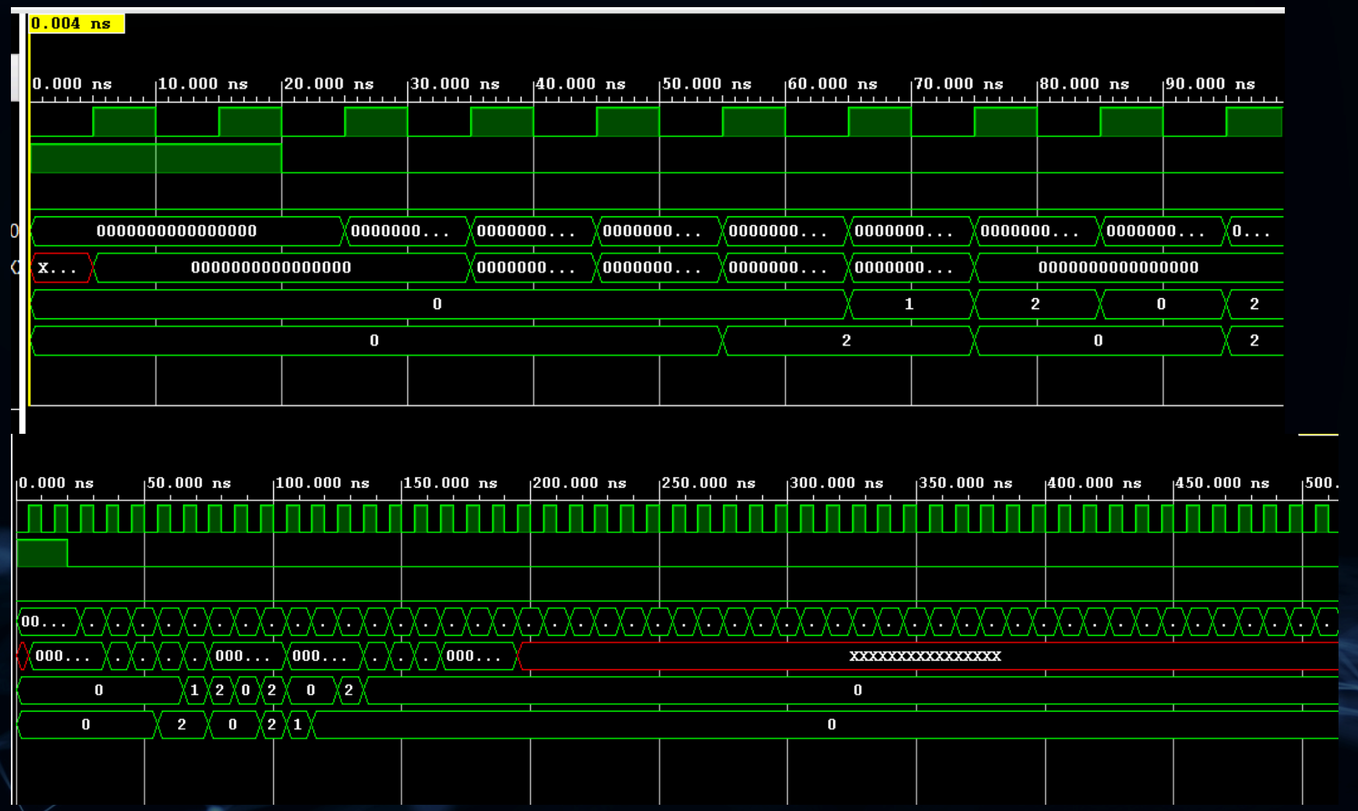
- Simple prediction: branch not taken

```
module ForwardingUnit(
    input  wire [4:0] EX_rs1, EX_rs2,   // source registers of
    input  wire [4:0] MEM_rd, WB_rd,    // destination register
    input  wire MEM_RegWrite, WB_RegWrite,
    output reg [1:0] forwardA, forwardB
);
    always @(*) begin
        forwardA = 2'b00;
        forwardB = 2'b00;

        // EX hazard (forward from EX/MEM)
        if (MEM_RegWrite && MEM_rd != 0 && MEM_rd == EX_rs1)
            forwardA = 2'b10;
        if (MEM_RegWrite && MEM_rd != 0 && MEM_rd == EX_rs2)
            forwardB = 2'b10;

        // MEM/WB hazard (forward from WB)
        if (WB_RegWrite && WB_rd != 0 && WB_rd == EX_rs1)
            forwardA = 2'b01;
        if (WB_RegWrite && WB_rd != 0 && WB_rd == EX_rs2)
            forwardB = 2'b01;
    end
endmodule
```

IMPLEMENTATION, VERIFICATION & RESULTS

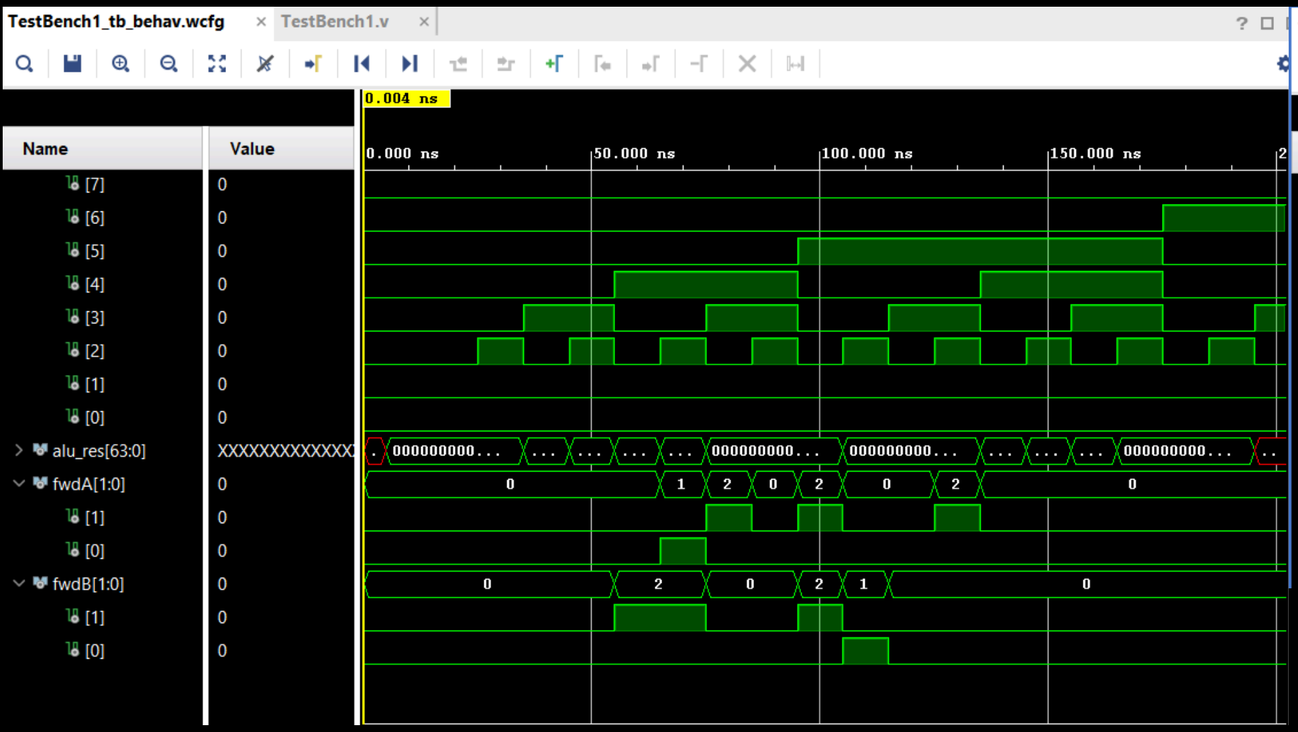
Verification

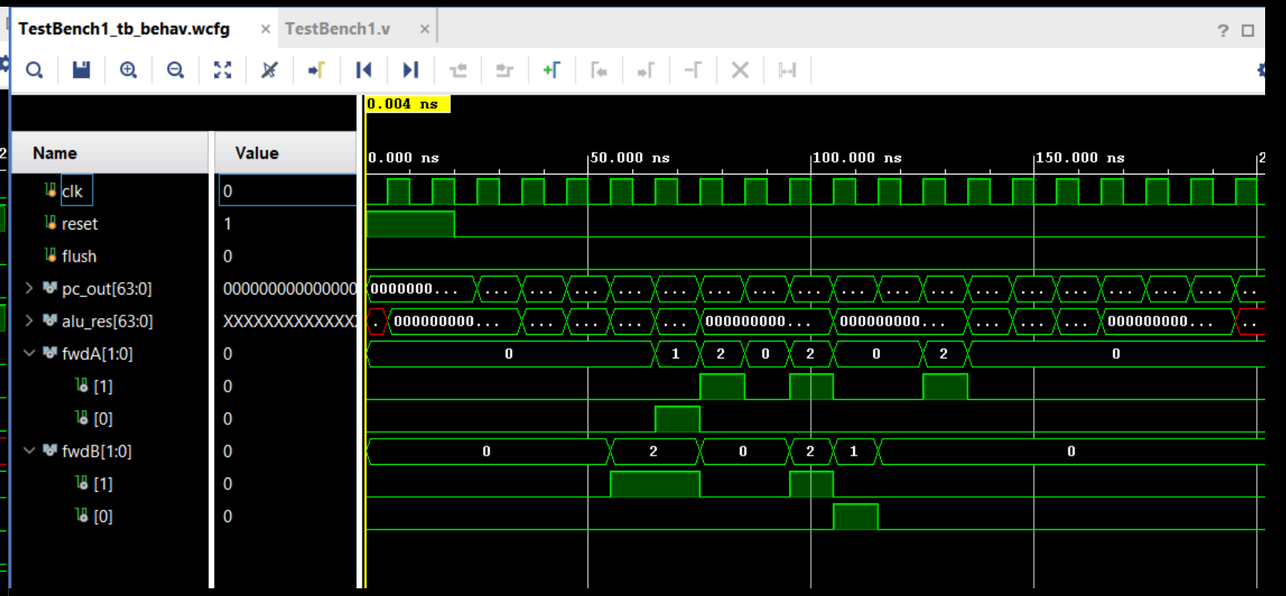
- Implemented and simulated in Vivado, verifying:
  - Correct instruction execution
  - Proper register write-back
  - Correct memory load/store behavior
  - Accurate branch handling and pipeline flushing

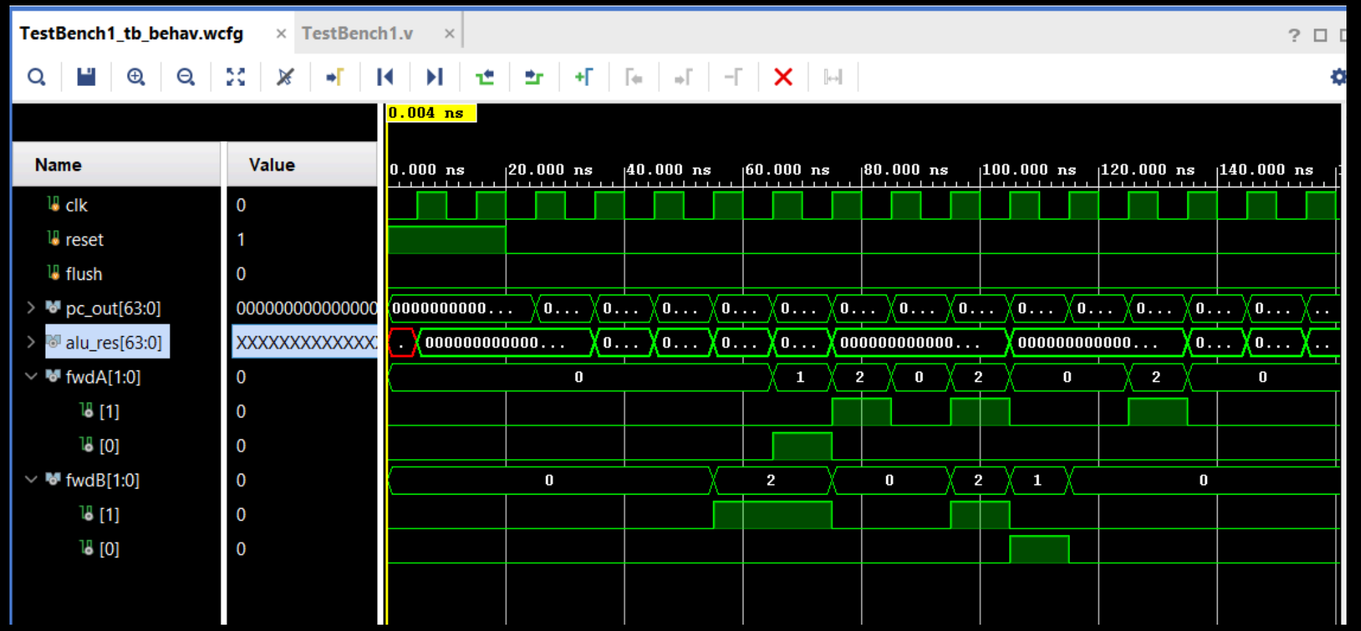


Results

- Expected outputs observed in simulation
- Processor executes RISC-V programs correctly under pipelining
- No functional stalls for ALU-to-ALU dependencies


TestBench1_tb_behav.wcfg
TestBench1.v
0.004 ns
0.000 ns
50.000 ns
100.000 ns
150.000 ns
Name
Value
[7] 0
[6] 0
[5] 0
[4] 0
[3] 0
[2] 0
[1] 0
[0] 0
alu_res[63:0] XXXXXXXXXXXXX
fwdA[1:0] 0
[1] 0
[0] 0
fwdB[1:0] 0
[1] 0
[0] 0


TestBench1_tb_behav.wcfg
TestBench1.v
0.004 ns
0.000 ns
50.000 ns
100.000 ns
150.000 ns
Name
Value
clk 0
reset 1
flush 0
pc_out[63:0] 0000000000000000
alu_res[63:0] XXXXXXXXXXXXX
fwdA[1:0] 0
[1] 0
[0] 0
fwdB[1:0] 0
[1] 0
[0] 0


TestBench1_tb_behav.wcfg
TestBench1.v
0.004 ns
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
Name
Value
clk 0
reset 1
flush 0
pc_out[63:0] 0000000000000000
alu_res[63:0] XXXXXXXXXXXXX
fwdA[1:0] 0
[1] 0
[0] 0
fwdB[1:0] 0
[1] 0
[0] 0

DISCUSSION OF VERIFICATION

The most significant proof of correct operation is the value of x17. For the processor to reach the value 23, the following must have occurred correctly:

1. The ALU correctly compared x24 and x7 in the beq instruction.
2. The Branch Logic detected the equality and calculated the target PC.
3. The IF/ID Flush successfully cleared the instructions that entered the pipe before the branch was resolved.
4. The Write-Back stage successfully committed the value 23 to the register file on the negative edge of the clock.

Signal / Register	Logical Purpose	Expected Value	Simulated Result	Status
x17	Final result after Branch (L1)	23	**23**	**Pass**
x7	Result of logical shift (SRL)	0	**0**	**Pass**
x24	Result of Load from Memory	0	**0**	**Pass**
Memory [48]	Target of Store instruction	0	**0**	**Pass**
Forwarding	ALU-to-ALU Dependency	No Stalls	**Active (FwdA/B)**	**Pass**

THANK YOU

GitHub Repository: https://github.com/JomanaAmin/Computer-Architecture-Project