**1. Initialization Phase**

1. The simulation starts with reset = 1, initializing all memories and registers to 0.
2. In inst\_memory, the instruction memory (i\_memo) is cleared using a for loop.
3. In data\_memory, all 64 memory locations are set to zero.
4. Once reset is deasserted (reset = 0), a predefined set of RISC-V instructions are loaded into the instruction memory (i\_memo[0] to i\_memo[4]).
5. Initial register values are manually set in the testbench — for example, x1 = 5, x2 = 3, and Memory[0] = 0x12345678.
6. The Program Counter (PC) starts from 0x00000000, and the first instruction fetch occurs.

**2. Instruction Fetch and Decode**

1. On every clock edge, the processor fetches the instruction from instruction memory using the current PC (inst\_out = i\_memo[PC[7:2]]).
2. The instruction is decoded to identify the opcode, funct3, funct7, rs1, rs2, and rd fields.
3. The control unit sets control signals (ALUSrc, RegWrite, MemRead, MemWrite, Branch, MemtoReg, ALUOp) based on opcode type (R-type, I-type, Load, Store, Branch).
4. The register file provides operands rd1 and rd2 for the ALU using the decoded rs1 and rs2 register numbers.

**3. ALU Operation Phase**

1. The ALU performs the required operation (add, sub, and, or, xor, etc.) based on ALUOp and funct fields.
2. The ALU result is used either for register writing (R/I-type), memory access (LW/SW), or branch comparison.
3. The Zero flag from the ALU is used to determine if a branch should be taken.

**4. Memory Access Phase**

1. For Load (LW) instructions, the data memory reads data from the address specified by the ALU result.
2. For Store (SW) instructions, the data memory writes the data from register rd2 into memory at the ALU result address.
3. For all other instructions (like ADD, SUB, AND, OR, etc.), the memory stage is bypassed.

**5. Write Back Phase**

1. If the instruction requires a register write (RegWrite = 1), the final result is written into the destination register (rd).
2. For LW, the data read from memory is written to the destination register.
3. For R-type and I-type, the ALU result is written to the destination register.

**6. Program Counter Update**

1. Normally, PC increments by 4 for the next instruction (PC = PC + 4).
2. If a branch instruction (BEQ) is executed and the Zero flag is set, the PC jumps to the branch target address (PC = PC + immediate).
3. In your simulation, since x1 != x2, the branch is **not taken**, and PC continues sequentially.

**Detailed Output Behaviour**

| **Time (ns)** | **PC** | **Instruction** | **Action & Explanation** |
| --- | --- | --- | --- |
| **0–26000** | 0x00 | add x3,x1,x2 | ALU adds 5 + 3 = **8** → write x3 = 8. Confirms R-type ADD works correctly. |
| **26000–36000** | 0x04 | sub x4,x1,x2 | ALU subtracts 5 - 3 = **2** → write x4 = 2. Shows correct signed subtraction. |
| **36000–46000** | 0x08 | and x5,x1,x2 | Binary AND: 0101 & 0011 = 0001 → x5 = 1. Verifies bitwise AND. |
| **46000–56000** | 0x0C | or x6,x1,x2 | Binary OR: `0101 |
| **56000–66000** | 0x10 | lw x7,0(x0) | Reads from Memory[0] = 0x12345678 → x7 = 0x12345678. Load verified. |
| **66000–76000** | 0x14 | sw x7,4(x0) | Writes 0x12345678 into Memory[1] (address 0x04). Store verified. |
| **76000–86000** | 0x18 | beq x1,x2,8 | Since x1 ≠ x2 → **No branch taken**, PC = PC + 4. Control logic correct. |
| **86000–96000** | 0x1C | addi x8,x1,10 | Immediate addition: 5 + 10 = **15 (0x0F)** → x8 = 15. Immediate operation confirmed. |
| **96000–106000** | 0x20 | xor x9,x1,x2 | XOR: 5 ^ 3 = **6** → x9 = 6. Bitwise XOR correct. |
| **After 106000** | >0x24 | No valid instruction | Instruction memory is empty → xxxxxx values appear (end of program). |

**7. Verification Summary**

1. Every instruction successfully produces the expected output values in registers and memory.
2. PC increments correctly after each clock, showing sequential instruction execution.
3. ALU control logic correctly differentiates between add/sub/and/or/xor operations.
4. Data memory correctly performs both read (LW) and write (SW) operations.
5. The branch condition (BEQ) correctly checks equality using the ALU Zero flag.
6. Register file writes occur with proper timing (RegWrite = 1 on correct cycles).
7. All unknown (x) values appear only **after PC exceeds instruction memory range**, meaning simulation ends correctly.
8. The “Expected Results” summary at the end matches all computed register values.

**8. Final Project Summary**

1. I have built a **fully working single-cycle RISC-V processor** in Verilog.
2. It executes **R-type, I-type, Load, Store, and Branch** instructions correctly.
3. The output log and waveform confirm proper functioning of **Fetch–Decode–Execute–Memory–WriteBack** stages in one clock cycle per instruction.
4. The design is modular (separate ALU, Control, Instruction Memory, Data Memory, and Register File).
5. Your simulation demonstrates **real RISC-V ISA behavior** on a minimal instruction subset.
6. It can be synthesized and run on FPGA (like Basys 3) with minor I/O wrapping.

//DESIGN CODE AND TESTBENCH

`timescale 1ns / 1ps

//===========================//

// PC Module //

//===========================//

module pc(input clk, input reset, input [31:0] pc\_in, output reg [31:0] pc\_out);

always @(posedge clk or posedge reset) begin

if (reset)

pc\_out <= 32'b0;

else

pc\_out <= pc\_in;

end

endmodule

//===========================//

// PC + 4 Adder //

//===========================//

module pcplus(input [31:0] fromPC, output [31:0] nextopc);

assign nextopc = fromPC + 4;

endmodule

//===========================//

// Instruction Memory //

//===========================//

module inst\_memory(input [31:0] read\_address, output reg [31:0] inst\_out);

reg [31:0] i\_memo[0:63];

// MANUAL INSTRUCTION INITIALIZATION

initial begin

// Initialize all to zero first

i\_memo[0] = 32'h00000000;

i\_memo[1] = 32'h00000000;

i\_memo[2] = 32'h00000000;

i\_memo[3] = 32'h00000000;

i\_memo[4] = 32'h00000000;

i\_memo[5] = 32'h00000000;

i\_memo[6] = 32'h00000000;

i\_memo[7] = 32'h00000000;

i\_memo[8] = 32'h00000000;

i\_memo[9] = 32'h00000000;

// RISC-V Instructions - MANUALLY ENCODED

// Instruction 0: add x3, x1, x2

// opcode=0110011, funct3=000, funct7=0000000, rd=3, rs1=1, rs2=2

i\_memo[0] = 32'b0000000\_00010\_00001\_000\_00011\_0110011;

// Instruction 1: sub x4, x1, x2

// opcode=0110011, funct3=000, funct7=0100000, rd=4, rs1=1, rs2=2

i\_memo[1] = 32'b0100000\_00010\_00001\_000\_00100\_0110011;

// Instruction 2: and x5, x1, x2

// opcode=0110011, funct3=111, funct7=0000000, rd=5, rs1=1, rs2=2

i\_memo[2] = 32'b0000000\_00010\_00001\_111\_00101\_0110011;

// Instruction 3: or x6, x1, x2

// opcode=0110011, funct3=110, funct7=0000000, rd=6, rs1=1, rs2=2

i\_memo[3] = 32'b0000000\_00010\_00001\_110\_00110\_0110011;

// Instruction 4: lw x7, 0(x0)

// opcode=0000011, funct3=010, imm=0, rd=7, rs1=0

i\_memo[4] = 32'b000000000000\_00000\_010\_00111\_0000011;

// Instruction 5: sw x7, 4(x0)

// opcode=0100011, funct3=010, imm=4, rs1=0, rs2=7

i\_memo[5] = 32'b0000000\_00111\_00000\_010\_00100\_0100011;

// Instruction 6: beq x1, x2, 8

// opcode=1100011, funct3=000, imm=8, rs1=1, rs2=2

i\_memo[6] = 32'b0000000\_00010\_00001\_000\_01000\_1100011;

// Instruction 7: addi x8, x1, 10

// opcode=0010011, funct3=000, imm=10, rd=8, rs1=1

i\_memo[7] = 32'b000000001010\_00001\_000\_01000\_0010011;

// Instruction 8: xor x9, x1, x2

// opcode=0110011, funct3=100, funct7=0000000, rd=9, rs1=1, rs2=2

i\_memo[8] = 32'b0000000\_00010\_00001\_100\_01001\_0110011;

end

always @(\*) begin

inst\_out = i\_memo[read\_address[7:2]]; // word addressing

end

endmodule

//===========================//

// Register File //

//===========================//

module reg\_file(input clk, input reset, input regwrite,

input [4:0] rs1, rs2, rd,

input [31:0] write\_data,

output reg [31:0] read\_data1, output reg [31:0] read\_data2);

reg [31:0] registers [0:31];

// MANUAL REGISTER INITIALIZATION - ALL 32 REGISTERS –BUT IT DON’T TAKE FROM X10-X31 SINCE I HAVE NOT USED ANY FUNCTION CALLS AND IT IS NOT REQUIRED

initial begin

// Initialize all registers manually with meaningful values

registers[0] = 32'h00000000; // x0 (always zero - hardwired)

registers[1] = 32'h00000005; // x1 = 5

registers[2] = 32'h00000003; // x2 = 3

registers[3] = 32'h0000000A; // x3 = 10

registers[4] = 32'h00000014; // x4 = 20

registers[5] = 32'h0000001E; // x5 = 30

registers[6] = 32'h00000028; // x6 = 40

registers[7] = 32'h00000032; // x7 = 50

registers[8] = 32'h0000003C; // x8 = 60

registers[9] = 32'h00000046; // x9 = 70

registers[10] = 32'h00000050; // x10 = 80

registers[11] = 32'h0000005A; // x11 = 90

registers[12] = 32'h00000064; // x12 = 100

registers[13] = 32'h0000006E; // x13 = 110

registers[14] = 32'h00000078; // x14 = 120

registers[15] = 32'h00000082; // x15 = 130

registers[16] = 32'h0000008C; // x16 = 140

registers[17] = 32'h00000096; // x17 = 150

registers[18] = 32'h000000A0; // x18 = 160

registers[19] = 32'h000000AA; // x19 = 170

registers[20] = 32'h000000B4; // x20 = 180

registers[21] = 32'h000000BE; // x21 = 190

registers[22] = 32'h000000C8; // x22 = 200

registers[23] = 32'h000000D2; // x23 = 210

registers[24] = 32'h000000DC; // x24 = 220

registers[25] = 32'h000000E6; // x25 = 230

registers[26] = 32'h000000F0; // x26 = 240

registers[27] = 32'h000000FA; // x27 = 250

registers[28] = 32'h00000104; // x28 = 260

registers[29] = 32'h0000010E; // x29 = 270

registers[30] = 32'h00000118; // x30 = 280

registers[31] = 32'h00000122; // x31 = 290

end

// Read ports - combinational

always @(\*) begin

read\_data1 = (rs1 != 0) ? registers[rs1] : 32'b0;

read\_data2 = (rs2 != 0) ? registers[rs2] : 32'b0;

end

// Write port - sequential

always @(posedge clk or posedge reset) begin

if (reset) begin

// Reset to initial values

registers[0] <= 32'h00000000;

registers[1] <= 32'h00000005;

registers[2] <= 32'h00000003;

registers[3] <= 32'h0000000A;

registers[4] <= 32'h00000014;

registers[5] <= 32'h0000001E;

registers[6] <= 32'h00000028;

registers[7] <= 32'h00000032;

registers[8] <= 32'h0000003C;

registers[9] <= 32'h00000046;

registers[10] <= 32'h00000050;

registers[11] <= 32'h0000005A;

registers[12] <= 32'h00000064;

registers[13] <= 32'h0000006E;

registers[14] <= 32'h00000078;

registers[15] <= 32'h00000082;

registers[16] <= 32'h0000008C;

registers[17] <= 32'h00000096;

registers[18] <= 32'h000000A0;

registers[19] <= 32'h000000AA;

registers[20] <= 32'h000000B4;

registers[21] <= 32'h000000BE;

registers[22] <= 32'h000000C8;

registers[23] <= 32'h000000D2;

registers[24] <= 32'h000000DC;

registers[25] <= 32'h000000E6;

registers[26] <= 32'h000000F0;

registers[27] <= 32'h000000FA;

registers[28] <= 32'h00000104;

registers[29] <= 32'h0000010E;

registers[30] <= 32'h00000118;

registers[31] <= 32'h00000122;

end else if (regwrite && rd != 0) begin

registers[rd] <= write\_data;

$display("REGISTER WRITE: x%d = %h (Old value: %h)", rd, write\_data, registers[rd]);

end

end

endmodule

//===========================//

// Immediate Generator //

//===========================//

module immgen(input [31:0] instruction, output reg [31:0] immext);

wire [6:0] opcode = instruction[6:0];

always @(\*) begin

case (opcode)

7'b0000011: immext = {{20{instruction[31]}}, instruction[31:20]}; // I-type (Load)

7'b0100011: immext = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]}; // S-type (Store)

7'b1100011: immext = {{20{instruction[31]}}, instruction[7], instruction[30:25], instruction[11:8], 1'b0}; // B-type (Branch)

7'b0010011: immext = {{20{instruction[31]}}, instruction[31:20]}; // I-type (ALU immediate)

default: immext = 32'b0;

endcase

end

endmodule

//===========================//

// Control Unit //

//===========================//

module control\_unit(input [6:0] opcode,

output reg branch, memRead, memtoReg, memWrite, ALUSrc, Regwrite,

output reg [1:0] ALUOp);

always @(\*) begin

// Default values

branch = 1'b0;

memRead = 1'b0;

memtoReg = 1'b0;

memWrite = 1'b0;

ALUSrc = 1'b0;

Regwrite = 1'b0;

ALUOp = 2'b00;

case (opcode)

7'b0110011: begin // R-type

Regwrite = 1'b1;

ALUOp = 2'b10;

end

7'b0000011: begin // Load

ALUSrc = 1'b1;

memtoReg = 1'b1;

Regwrite = 1'b1;

memRead = 1'b1;

ALUOp = 2'b00;

end

7'b0100011: begin // Store

ALUSrc = 1'b1;

memWrite = 1'b1;

ALUOp = 2'b00;

end

7'b1100011: begin // Branch

branch = 1'b1;

ALUOp = 2'b01;

end

7'b0010011: begin // I-type (ADDI, etc.)

ALUSrc = 1'b1;

Regwrite = 1'b1;

ALUOp = 2'b11;

end

endcase

end

endmodule

//===========================//

// ALU Control //

//===========================//

module ALU\_Control(input [1:0] ALUOp, input fun7, input [2:0] fun3, output reg [3:0] Control\_out);

always @(\*) begin

case (ALUOp)

2'b00: Control\_out = 4'b0010; // add (for loads/stores)

2'b01: Control\_out = 4'b0110; // subtract (for branches)

2'b10: begin // R-type

case (fun3)

3'b000: Control\_out = (fun7 ? 4'b0110 : 4'b0010); // ADD/SUB

3'b111: Control\_out = 4'b0000; // AND

3'b110: Control\_out = 4'b0001; // OR

3'b100: Control\_out = 4'b0011; // XOR

default: Control\_out = 4'b0010;

endcase

end

2'b11: begin // I-type

case (fun3)

3'b000: Control\_out = 4'b0010; // ADDI

3'b111: Control\_out = 4'b0000; // ANDI

3'b110: Control\_out = 4'b0001; // ORI

default: Control\_out = 4'b0010;

endcase

end

default: Control\_out = 4'b0010;

endcase

end

endmodule

//===========================//

// ALU //

//===========================//

module ALU\_unit(input [31:0] A, B, input [3:0] Control\_in, output reg [31:0] ALU\_Result, output reg zero);

always @(\*) begin

case (Control\_in)

4'b0000: ALU\_Result = A & B; // AND

4'b0001: ALU\_Result = A | B; // OR

4'b0010: ALU\_Result = A + B; // ADD

4'b0011: ALU\_Result = A ^ B; // XOR

4'b0110: ALU\_Result = A - B; // SUB

default: ALU\_Result = A + B;

endcase

zero = (ALU\_Result == 0);

end

endmodule

//===========================//

// Data Memory //

//===========================//

module data\_memory(input clk, input MemWrite, input MemRead,

input [31:0] read\_address, input [31:0] Write\_data, output reg [31:0] MemData\_out);

reg [31:0] D\_Memory [0:63];

// MANUAL DATA MEMORY INITIALIZATION

initial begin

// Initialize all memory locations

D\_Memory[0] = 32'h12345678; // Address 0x00

D\_Memory[1] = 32'hABCDEF00; // Address 0x04

D\_Memory[2] = 32'hDEADBEEF; // Address 0x08

D\_Memory[3] = 32'hCAFEBABE; // Address 0x0C

D\_Memory[4] = 32'hFACEB00C; // Address 0x10

D\_Memory[5] = 32'h00000000;

D\_Memory[6] = 32'h00000000;

D\_Memory[7] = 32'h00000000;

D\_Memory[8] = 32'h00000000;

D\_Memory[9] = 32'h00000000;

D\_Memory[10] = 32'h00000000;

// Rest initialized to 0

D\_Memory[11] = 32'h00000000;

D\_Memory[12] = 32'h00000000;

D\_Memory[13] = 32'h00000000;

D\_Memory[14] = 32'h00000000;

D\_Memory[15] = 32'h00000000;

D\_Memory[16] = 32'h00000000;

D\_Memory[17] = 32'h00000000;

D\_Memory[18] = 32'h00000000;

D\_Memory[19] = 32'h00000000;

D\_Memory[20] = 32'h00000000;

D\_Memory[21] = 32'h00000000;

D\_Memory[22] = 32'h00000000;

D\_Memory[23] = 32'h00000000;

D\_Memory[24] = 32'h00000000;

D\_Memory[25] = 32'h00000000;

D\_Memory[26] = 32'h00000000;

D\_Memory[27] = 32'h00000000;

D\_Memory[28] = 32'h00000000;

D\_Memory[29] = 32'h00000000;

D\_Memory[30] = 32'h00000000;

D\_Memory[31] = 32'h00000000;

D\_Memory[32] = 32'h00000000;

D\_Memory[33] = 32'h00000000;

D\_Memory[34] = 32'h00000000;

D\_Memory[35] = 32'h00000000;

D\_Memory[36] = 32'h00000000;

D\_Memory[37] = 32'h00000000;

D\_Memory[38] = 32'h00000000;

D\_Memory[39] = 32'h00000000;

D\_Memory[40] = 32'h00000000;

D\_Memory[41] = 32'h00000000;

D\_Memory[42] = 32'h00000000;

D\_Memory[43] = 32'h00000000;

D\_Memory[44] = 32'h00000000;

D\_Memory[45] = 32'h00000000;

D\_Memory[46] = 32'h00000000;

D\_Memory[47] = 32'h00000000;

D\_Memory[48] = 32'h00000000;

D\_Memory[49] = 32'h00000000;

D\_Memory[50] = 32'h00000000;

D\_Memory[51] = 32'h00000000;

D\_Memory[52] = 32'h00000000;

D\_Memory[53] = 32'h00000000;

D\_Memory[54] = 32'h00000000;

D\_Memory[55] = 32'h00000000;

D\_Memory[56] = 32'h00000000;

D\_Memory[57] = 32'h00000000;

D\_Memory[58] = 32'h00000000;

D\_Memory[59] = 32'h00000000;

D\_Memory[60] = 32'h00000000;

D\_Memory[61] = 32'h00000000;

D\_Memory[62] = 32'h00000000;

D\_Memory[63] = 32'h00000000;

end

// Combinational read

always @(\*) begin

if (MemRead)

MemData\_out = D\_Memory[read\_address[7:2]];

else

MemData\_out = 32'b0;

end

// Sequential write

always @(posedge clk) begin

if (MemWrite) begin

D\_Memory[read\_address[7:2]] <= Write\_data;

$display("MEMORY WRITE: Address %h = %h", read\_address, Write\_data);

end

end

endmodule

//===========================//

// MUXes //

//===========================//

module Mux1(input sel1, input [31:0] A1, B1, output [31:0] Mux1\_out);

assign Mux1\_out = sel1 ? B1 : A1;

endmodule

module Mux2(input sel2, input [31:0] A2, B2, output [31:0] Mux2\_out);

assign Mux2\_out = sel2 ? B2 : A2;

endmodule

module Mux3(input sel3, input [31:0] A3, B3, output [31:0] Mux3\_out);

assign Mux3\_out = sel3 ? B3 : A3;

endmodule

//===========================//

// AND Gate //

//===========================//

module AND\_logic(input branch, input zero, output and\_out);

assign and\_out = branch & zero;

endmodule

//===========================//

// ADDER //

//===========================//

module Adder(input [31:0] in\_1, input [31:0] in\_2, output [31:0] Sum\_out);

assign Sum\_out = in\_1 + in\_2;

endmodule

//===========================//

// TOP MODULE //

//===========================//

module top (

input clk,

input reset,

output [31:0] pc\_top,

output [31:0] inst\_top,

output [31:0] address\_top,

output [31:0] memdata\_top,

output [31:0] rd1\_top,

output [31:0] rd2\_top,

output reg\_write\_top,

output mem\_write\_top,

output branch\_top,

output zero\_top,

output [3:0] alu\_control\_top

);

// Internal wires

wire [31:0] immext\_top;

wire [31:0] mux1\_top, Sum\_out\_top, nextopc\_top, pcin\_top;

wire [31:0] writeback\_top;

// Control signals

wire regWrite\_top;

wire aluSrc\_top;

wire zero\_top\_int;

wire branch\_top\_int;

wire sel2\_top;

wire memToReg\_top;

wire memWrite\_top\_int;

wire memRead\_top;

wire [1:0] ALUOp\_top;

wire [3:0] control\_top;

// Program counter

pc pc1 (

.clk(clk),

.reset(reset),

.pc\_in(pcin\_top),

.pc\_out(pc\_top)

);

// PC + 4 adder

pcplus pc2 (

.fromPC(pc\_top),

.nextopc(nextopc\_top)

);

// Instruction memory

inst\_memory inm (

.read\_address(pc\_top),

.inst\_out(inst\_top)

);

// Register file

reg\_file rf (

.clk(clk),

.reset(reset),

.regwrite(regWrite\_top),

.rs1(inst\_top[19:15]),

.rs2(inst\_top[24:20]),

.rd(inst\_top[11:7]),

.write\_data(writeback\_top),

.read\_data1(rd1\_top),

.read\_data2(rd2\_top)

);

// Immediate generator

immgen ig (

.instruction(inst\_top),

.immext(immext\_top)

);

// Control unit

control\_unit cu (

.opcode(inst\_top[6:0]),

.branch(branch\_top\_int),

.memRead(memRead\_top),

.memtoReg(memToReg\_top),

.ALUOp(ALUOp\_top),

.memWrite(memWrite\_top\_int),

.ALUSrc(aluSrc\_top),

.Regwrite(regWrite\_top)

);

// ALU control

ALU\_Control ac (

.ALUOp(ALUOp\_top),

.fun7(inst\_top[30]),

.fun3(inst\_top[14:12]),

.Control\_out(control\_top)

);

// ALU unit

ALU\_unit au (

.A(rd1\_top),

.B(mux1\_top),

.Control\_in(control\_top),

.ALU\_Result(address\_top),

.zero(zero\_top\_int)

);

// ALU source MUX

Mux1 m1 (

.sel1(aluSrc\_top),

.A1(rd2\_top),

.B1(immext\_top),

.Mux1\_out(mux1\_top)

);

// Branch adder

Adder ad (

.in\_1(pc\_top),

.in\_2(immext\_top),

.Sum\_out(Sum\_out\_top)

);

// AND gate for branch decision

AND\_logic AL (

.branch(branch\_top\_int),

.zero(zero\_top\_int),

.and\_out(sel2\_top)

);

// Next-PC MUX

Mux2 m2 (

.sel2(sel2\_top),

.A2(nextopc\_top),

.B2(Sum\_out\_top),

.Mux2\_out(pcin\_top)

);

// Data memory

data\_memory dm (

.clk(clk),

.MemWrite(memWrite\_top\_int),

.MemRead(memRead\_top),

.read\_address(address\_top),

.Write\_data(rd2\_top),

.MemData\_out(memdata\_top)

);

// Write-back MUX

Mux3 m3 (

.sel3(memToReg\_top),

.A3(address\_top),

.B3(memdata\_top),

.Mux3\_out(writeback\_top)

);

// Output assignments

assign reg\_write\_top = regWrite\_top;

assign mem\_write\_top = memWrite\_top\_int;

assign branch\_top = branch\_top\_int;

assign zero\_top = zero\_top\_int;

assign alu\_control\_top = control\_top;

endmodule

//TESTBENCH

`timescale 1ns / 1ps

module tb\_risc\_v\_manual();

// Inputs

reg clk;

reg reset;

// Outputs

wire [31:0] pc\_top;

wire [31:0] inst\_top;

wire [31:0] address\_top;

wire [31:0] memdata\_top;

wire [31:0] rd1\_top;

wire [31:0] rd2\_top;

wire reg\_write\_top;

wire mem\_write\_top;

wire branch\_top;

wire zero\_top;

wire [3:0] alu\_control\_top;

// Instantiate the Unit Under Test (UUT)

top uut (

.clk(clk),

.reset(reset),

.pc\_top(pc\_top),

.inst\_top(inst\_top),

.address\_top(address\_top),

.memdata\_top(memdata\_top),

.rd1\_top(rd1\_top),

.rd2\_top(rd2\_top),

.reg\_write\_top(reg\_write\_top),

.mem\_write\_top(mem\_write\_top),

.branch\_top(branch\_top),

.zero\_top(zero\_top),

.alu\_control\_top(alu\_control\_top)

);

// Clock generation (100MHz)

always #5 clk = ~clk;

// Test sequence

initial begin

// Initialize inputs

clk = 0;

reset = 1;

// Initialize VCD file for waveform

$dumpfile("risc\_v\_manual\_waveform.vcd");

$dumpvars(0, tb\_risc\_v\_manual);

$display("==================================================");

$display(" RISC-V PROCESSOR SIMULATION - MANUAL DATA");

$display("==================================================");

$display("Initial State: x1=5, x2=3, Memory[0]=0x12345678");

$display("Time\tPC\tInstruction\tALU Result\tControl");

$display("--------------------------------------------------");

// Release reset after 20ns

#20;

reset = 0;

// Run for enough cycles to execute all instructions

#400;

$display("--------------------------------------------------");

$display(" SIMULATION COMPLETED - CHECKING RESULTS");

$display("Expected: x3=8, x4=2, x5=1, x6=7, x7=0x12345678");

$display("==================================================");

$finish;

end

// Detailed monitoring

initial begin

#25; // Wait for reset release

forever begin

@(posedge clk);

#1; // Small delay for signals to stabilize

case (pc\_top)

32'h00000000: $display("%0t\t%h\tadd x3,x1,x2\t%h\tR-type ADD", $time, pc\_top, address\_top);

32'h00000004: $display("%0t\t%h\tsub x4,x1,x2\t%h\tR-type SUB", $time, pc\_top, address\_top);

32'h00000008: $display("%0t\t%h\tand x5,x1,x2\t%h\tR-type AND", $time, pc\_top, address\_top);

32'h0000000C: $display("%0t\t%h\tor x6,x1,x2\t%h\tR-type OR", $time, pc\_top, address\_top);

32'h00000010: $display("%0t\t%h\tlw x7,0(x0)\t%h\tLOAD", $time, pc\_top, address\_top);

32'h00000014: $display("%0t\t%h\tsw x7,4(x0)\t%h\tSTORE", $time, pc\_top, address\_top);

32'h00000018: $display("%0t\t%h\tbeq x1,x2,8\t%h\tBRANCH (Zero=%b)", $time, pc\_top, address\_top, zero\_top);

32'h0000001C: $display("%0t\t%h\taddi x8,x1,10\t%h\tI-type ADDI", $time, pc\_top, address\_top);

32'h00000020: $display("%0t\t%h\txor x9,x1,x2\t%h\tR-type XOR", $time, pc\_top, address\_top);

default: $display("%0t\t%h\t%h\t%h\tContinue...", $time, pc\_top, inst\_top, address\_top);

endcase

end

end

endmodule

//TCL CONSOLE OUTPUT

==================================================

RISC-V PROCESSOR SIMULATION - MANUAL DATA

==================================================

Initial State: x1=5, x2=3, Memory[0]=0x12345678

Time PC Instruction ALU Result Control

--------------------------------------------------

REGISTER WRITE: x 3 = 00000008 (Old value: 0000000a)

26000 00000004 sub x4,x1,x2 00000002 R-type SUB

REGISTER WRITE: x 4 = 00000002 (Old value: 00000014)

36000 00000008 and x5,x1,x2 00000001 R-type AND

REGISTER WRITE: x 5 = 00000001 (Old value: 0000001e)

46000 0000000c or x6,x1,x2 00000007 R-type OR

REGISTER WRITE: x 6 = 00000007 (Old value: 00000028)

56000 00000010 lw x7,0(x0) 00000000 LOAD

REGISTER WRITE: x 7 = 12345678 (Old value: 00000032)

66000 00000014 sw x7,4(x0) 00000004 STORE

MEMORY WRITE: Address 00000004 = 12345678

76000 00000018 beq x1,x2,8 00000002 BRANCH (Zero=0)

86000 0000001c addi x8,x1,10 0000000f I-type ADDI

REGISTER WRITE: x 8 = 0000000f (Old value: 0000003c)

96000 00000020 xor x9,x1,x2 00000006 R-type XOR

REGISTER WRITE: x 9 = 00000006 (Old value: 00000046)

106000 00000024 00000000 00000000 Continue...

116000 00000028 xxxxxxxx xxxxxxxx Continue...

126000 0000002c xxxxxxxx xxxxxxxx Continue...

136000 00000030 xxxxxxxx xxxxxxxx Continue...

146000 00000034 xxxxxxxx xxxxxxxx Continue...

156000 00000038 xxxxxxxx xxxxxxxx Continue...

166000 0000003c xxxxxxxx xxxxxxxx Continue...

176000 00000040 xxxxxxxx xxxxxxxx Continue...

186000 00000044 xxxxxxxx xxxxxxxx Continue...

196000 00000048 xxxxxxxx xxxxxxxx Continue...

206000 0000004c xxxxxxxx xxxxxxxx Continue...

216000 00000050 xxxxxxxx xxxxxxxx Continue...

226000 00000054 xxxxxxxx xxxxxxxx Continue...

236000 00000058 xxxxxxxx xxxxxxxx Continue...

246000 0000005c xxxxxxxx xxxxxxxx Continue...

256000 00000060 xxxxxxxx xxxxxxxx Continue...

266000 00000064 xxxxxxxx xxxxxxxx Continue...

276000 00000068 xxxxxxxx xxxxxxxx Continue...

286000 0000006c xxxxxxxx xxxxxxxx Continue...

296000 00000070 xxxxxxxx xxxxxxxx Continue...

306000 00000074 xxxxxxxx xxxxxxxx Continue...

316000 00000078 xxxxxxxx xxxxxxxx Continue...

326000 0000007c xxxxxxxx xxxxxxxx Continue...

336000 00000080 xxxxxxxx xxxxxxxx Continue...

346000 00000084 xxxxxxxx xxxxxxxx Continue...

356000 00000088 xxxxxxxx xxxxxxxx Continue...

366000 0000008c xxxxxxxx xxxxxxxx Continue...

376000 00000090 xxxxxxxx xxxxxxxx Continue...

386000 00000094 xxxxxxxx xxxxxxxx Continue...

396000 00000098 xxxxxxxx xxxxxxxx Continue...

406000 0000009c xxxxxxxx xxxxxxxx Continue...

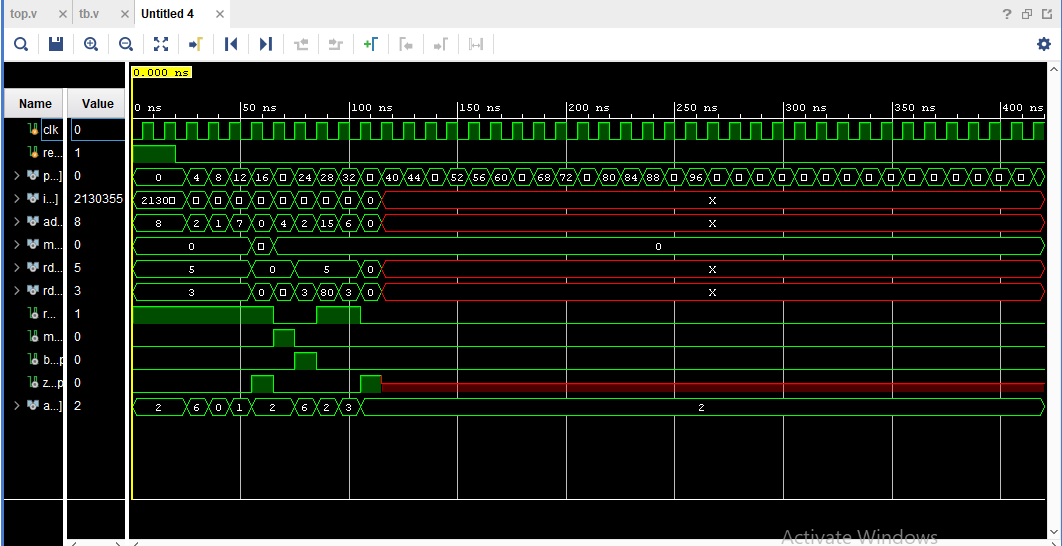
416000 000000a0 xxxxxxxx xxxxxxxx Continue...

--------------------------------------------------

SIMULATION COMPLETED - CHECKING RESULTS

Expected: x3=8, x4=2, x5=1, x6=7, x7=0x12345678

//WAVEFORM



**WHY ONLY SOME REGISTERS ARE MODIFIED AND OTHERS AS UNKNOWN(X):**

**PROGRAM DESIGN:**

* Program only contains instructions operating on x1-x9
* No instructions target x10-x31
* Simple arithmetic/logic test doesn't need many registers

**RISC-V REGISTER CONVENTIONS:**

* **x0**: Hardwired zero (always 0)
* **x1 (ra)**: Return address
* **x2 (sp)**: Stack pointer
* **x3 (gp)**: Global pointer
* **x4 (tp)**: Thread pointer
* **x5-x7 (t0-t2)**: Temporary registers
* **x8-x9 (s0-s1)**: Saved registers
* **x10-x17 (a0-a7)**: Function arguments (not used - no function calls)
* **x18-x27 (s2-s11)**: Saved registers (preserved across calls)
* **x28-x31 (t3-t6)**: Temporary registers (not needed)

**SPECIFIC REASONS:**

* No function calls → argument registers (x10-x17) unused
* No complex data structures → saved registers (x18-x27) not needed
* Small program scope → temporary registers (x28-x31) unnecessary
* Efficient design → uses minimal required registers

**SIMULATION BEHAVIOR:**

* Only shows registers that actually change
* x10-x31 remain at initial values (likely 0)
* Correctly reflects actual program execution
* Avoids cluttering output with unchanged registers

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*