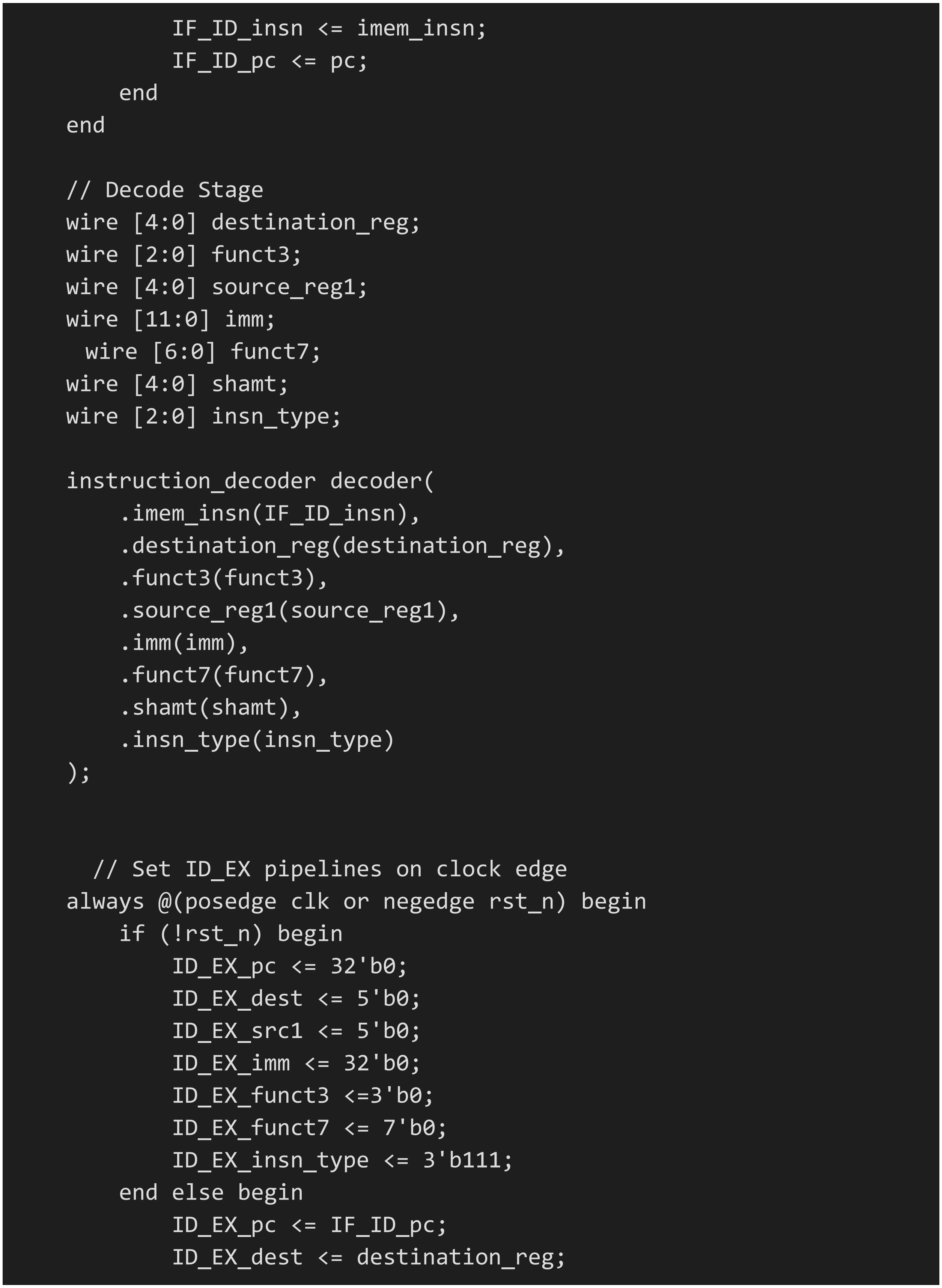
Mihir Phadke (015745301)
Jai Nayyar (015176655)
Ryan Smith (015792504)
Thurman Dao (015708589)
Haydon Behl (016390257)
Anson Lee
Lab 4
Mar 12, 2025

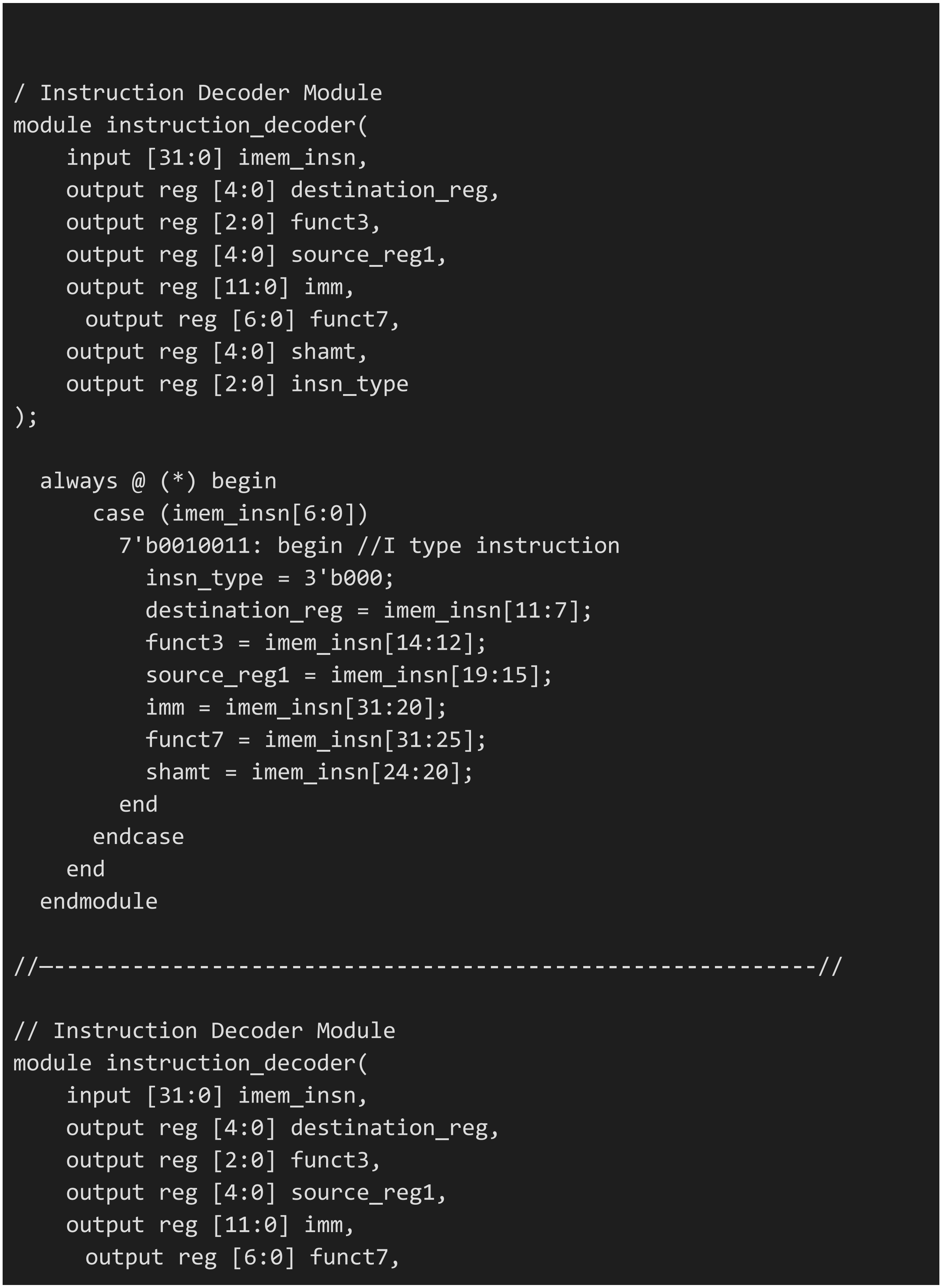
code:

```
`timescale 1ns / 1ps

module cpu(
    input rst_n,
    input clk,
    output reg [31:0] imem_addr,
    input [31:0] imem_insn,
    output reg [31:0] dmem_addr,
    inout [31:0] dmem_data,
    output reg dmem_wen
);
    // stall register
    reg stall;

    // IF/ID pipeline register
    reg [31:0] IF_ID_insn, IF_ID_pc;

    // ID/EX pipeline register
    reg [31:0] ID_EX_pc, ID_EX_imm;
    reg [4:0] ID_EX_dest, ID_EX_src1;
    reg [2:0] ID_EX_insn_type;
    reg [2:0] ID_EX_funct3;
    reg [4:0] ID_EX_shamt;
    reg [6:0] ID_EX_funct7;

    // EX/MEM pipeline register
    reg [31:0] EX_MEM_alu_result;
    reg [4:0] EX_MEM_dest;
```

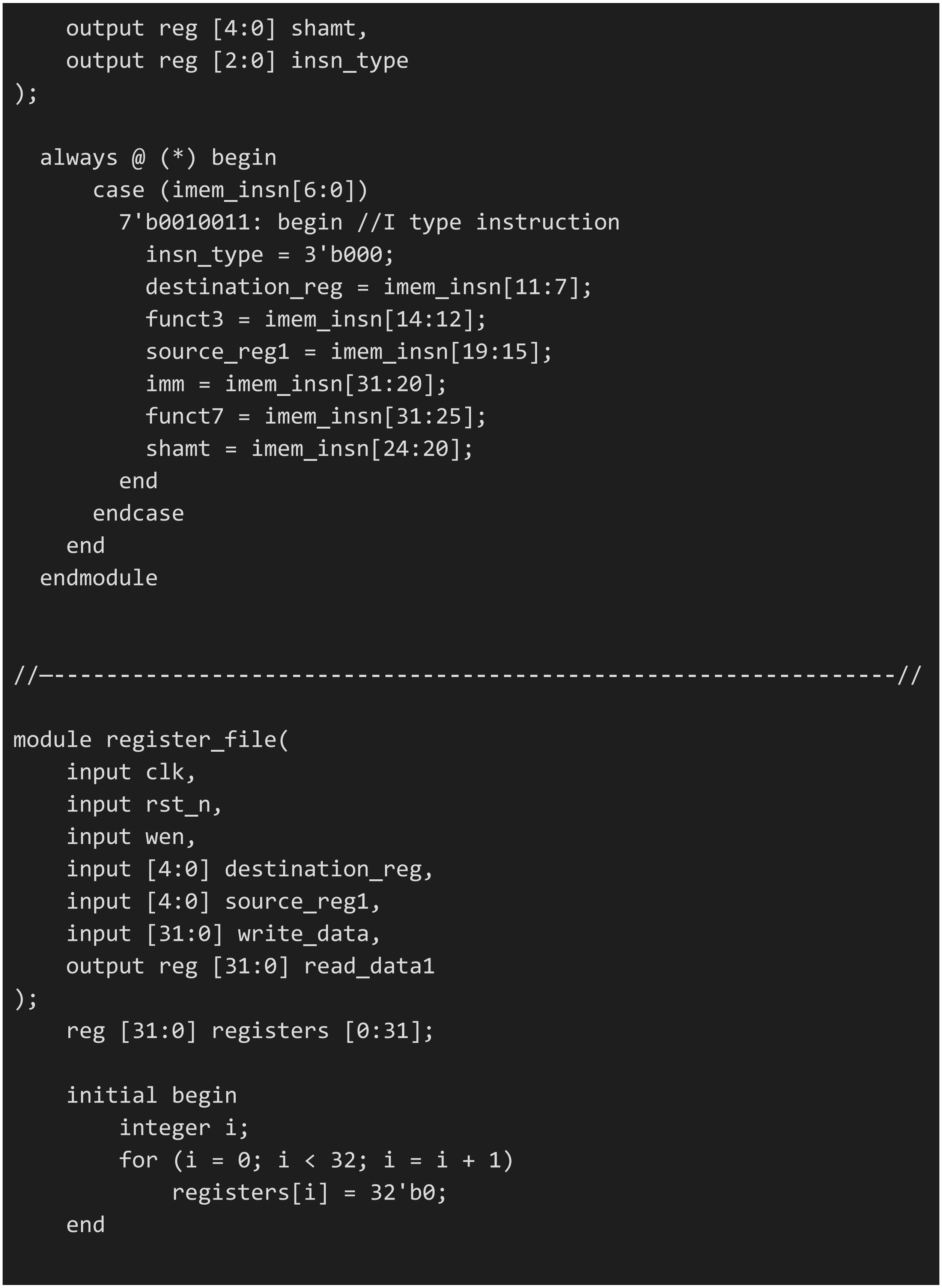
```
// MEM/WB pipeline registers
reg signed [31:0] MEM_WB_result;
reg [4:0] MEM_WB_dest;
reg MEM_WB_wen;

// Clock Cycle Counter
reg [15:0] cycle_counter;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cycle_counter <= 16'b0;
    end
    else begin
        cycle_counter <= cycle_counter + 1;
    end
end

// Program Counter Module
wire [31:0] pc, next_pc;

PC pc_module(
    .rst_n(rst_n),
    .clk(clk),
    .stall(stall),
    .pc(pc)
);

always @(*) begin
    imem_addr = pc;
end


// Instruction Fetch Stage
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        IF_ID_insn <= 32'b0;
        IF_ID_pc <= 32'b0;
    end else begin
```



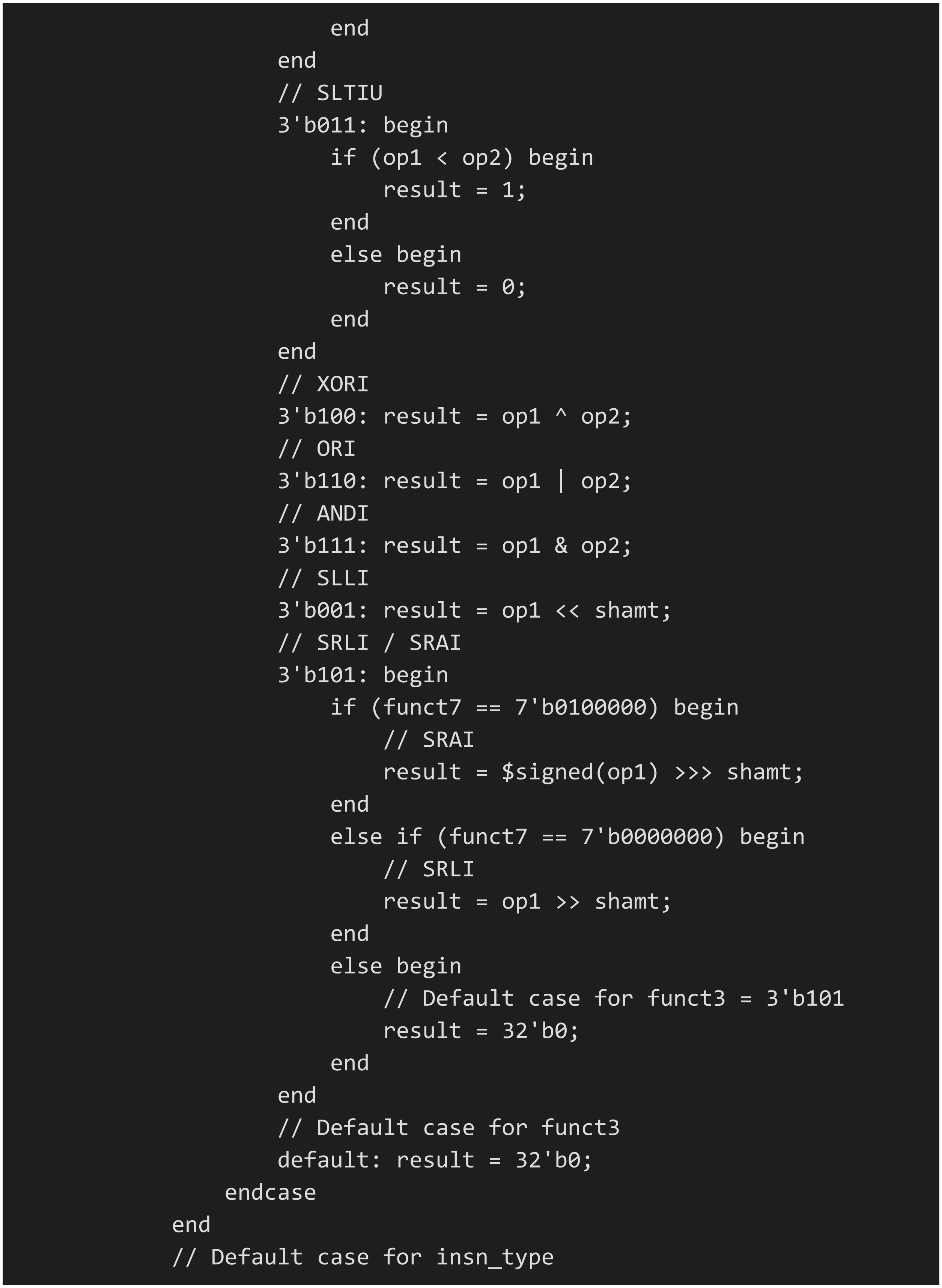
```
            IF_ID_insn <= imem_insn;
            IF_ID_pc <= pc;
        end
    end

    // Decode Stage
    wire [4:0] destination_reg;
    wire [2:0] funct3;
    wire [4:0] source_reg1;
    wire [11:0] imm;
     wire [6:0] funct7;
    wire [4:0] shamt;
    wire [2:0] insn_type;

    instruction_decoder decoder(
        .imem_insn(IF_ID_insn),
        .destination_reg(destination_reg),
        .funct3(funct3),
        .source_reg1(source_reg1),
        .imm(imm),
        .funct7(funct7),
        .shamt(shamt),
        .insn_type(insn_type)
    );


      // Set ID_EX pipelines on clock edge
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            ID_EX_pc <= 32'b0;
            ID_EX_dest <= 5'b0;
            ID_EX_src1 <= 5'b0;
            ID_EX_imm <= 32'b0;
            ID_EX_funct3 <=3'b0;
            ID_EX_funct7 <= 7'b0;
            ID_EX_insn_type <= 3'b111;
        end else begin
            ID_EX_pc <= IF_ID_pc;
            ID_EX_dest <= destination_reg;
            ID_EX_src1 <= source_reg1;
            ID_EX_insn_type <= insn_type;
            ID_EX_funct3 <= funct3;
            ID_EX_funct7 <= funct7;
            ID_EX_shamt <= shamt;
            ID_EX_imm <= {{20{imm[11]}}, imm};
        end
    end

    // Hazard detection
  /*  always @ (*) begin
          if ((ID_EX_dest != 5'b0) && (ID_EX_dest == source_reg1))
begin
              stall = 1'b1;
          end
          else if ((EX_MEM_dest != 5'b0) && (EX_MEM_dest ==
source_reg1)) begin
              stall = 1'b1;
          end
          else if ((MEM_WB_dest != 5'b0) && (MEM_WB_dest ==
source_reg1)) begin
              stall = 1'b1;
          end
          else begin
              stall = 1'b0;
          end
    end*/

    // Forwarding logic
    reg [31:0] forwarded_op1;

    always @(*) begin
        if ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == ID_EX_src1))
begin
            forwarded_op1 = EX_MEM_alu_result;
        end
        else if ((MEM_WB_dest != 5'b0) && (MEM_WB_dest ==
ID_EX_src1)) begin
            forwarded_op1 = MEM_WB_result;
        end
        else begin
            forwarded_op1 = reg_data1;
        end
    end

    // Register File
    wire [31:0] reg_data1;

    register_file reg_file(
        .clk(clk),
        .rst_n(rst_n),
        .wen(MEM_WB_wen),
        .destination_reg(MEM_WB_dest),
        .source_reg1(ID_EX_src1),
        .write_data(MEM_WB_result),
        .read_data1(reg_data1)
    );

    // Execute Stage
    wire [31:0] alu_result;

    ALU alu(
        .op1(forwarded_op1),
        .op2(ID_EX_imm),
        .shamt(ID_EX_shamt),
        .funct3(ID_EX_funct3),
        .funct7(ID_EX_funct7),
        .insn_type(ID_EX_insn_type),
        .result(alu_result)
    );

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            EX_MEM_alu_result <= 32'b0;
            EX_MEM_dest <= 5'b0;
        end else begin
            EX_MEM_alu_result <= alu_result;
            EX_MEM_dest <= ID_EX_dest;
        end
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            MEM_WB_result <= 32'b0;
            MEM_WB_dest <= 5'b0;
            MEM_WB_wen <= 1'b0;
        end else begin
            MEM_WB_result <= EX_MEM_alu_result;
            MEM_WB_dest <= EX_MEM_dest;
            MEM_WB_wen <= (EX_MEM_dest != 5'b0);
        end
    end


    // open files
    integer fd_pc, fd_data;
    initial begin
        fd_pc = $fopen("pc.txt", "w");
        fd_data = $fopen("data.txt", "w");
        if (fd_pc == 0 || fd_data == 0) begin
            $display("Error: Could not open file.");
            $finish;
        end
        $display("File descriptors: fd_pc = %0d, fd_data = %0d",
fd_pc, fd_data);
    end

    // print to trace files
    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin

        end else begin
            $display("IF_ID_pc: %h, MEM_WB_result: %h, MEM_WB_dest:
%h", IF_ID_pc, MEM_WB_result, MEM_WB_dest);
            $fdisplay(fd_pc, "PC: 0x%h", IF_ID_pc);
            $fdisplay(fd_data, "MEM_WB_result: %d", MEM_WB_result);
            $fdisplay(fd_data, "MEM_WB_dest: %d", MEM_WB_dest);
            $fdisplay(fd_data,
"-----------------------------------");
        end
    end

    // close files
    initial begin
        #1000
        $fclose(fd_pc);
        $fclose(fd_data);
        $display("Files closed successfully.");
    end
Endmodule


//–-----------------------------------------------------------//


module PC(
    input rst_n,
    input clk,
    input stall,
    output reg [31:0] pc
);
    always @(posedge clk or negedge rst_n) begin
      if (!rst_n) begin
            pc <= 32'b0;
      end
      else begin
            pc <= pc + 4;
      end
    end
endmodule


//–-----------------------------------------------------------//
```



```
/ Instruction Decoder Module
module instruction_decoder(
    input [31:0] imem_insn,
    output reg [4:0] destination_reg,
    output reg [2:0] funct3,
    output reg [4:0] source_reg1,
    output reg [11:0] imm,
     output reg [6:0] funct7,
    output reg [4:0] shamt,
    output reg [2:0] insn_type
);

  always @ (*) begin
      case (imem_insn[6:0])
        7'b0010011: begin //I type instruction
          insn_type = 3'b000;
          destination_reg = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
          imm = imem_insn[31:20];
          funct7 = imem_insn[31:25];
          shamt = imem_insn[24:20];
        end
      endcase
    end
  endmodule

//—----------------------------------------------------------//

// Instruction Decoder Module
module instruction_decoder(
    input [31:0] imem_insn,
    output reg [4:0] destination_reg,
    output reg [2:0] funct3,
    output reg [4:0] source_reg1,
    output reg [11:0] imm,
     output reg [6:0] funct7,
```



```
    output reg [4:0] shamt,
    output reg [2:0] insn_type
);

  always @ (*) begin
      case (imem_insn[6:0])
        7'b0010011: begin //I type instruction
          insn_type = 3'b000;
          destination_reg = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
          imm = imem_insn[31:20];
          funct7 = imem_insn[31:25];
          shamt = imem_insn[24:20];
        end
      endcase
    end
  endmodule


//-----------------------------------------------------------------//

module register_file(
    input clk,
    input rst_n,
    input wen,
    input [4:0] destination_reg,
    input [4:0] source_reg1,
    input [31:0] write_data,
    output reg [31:0] read_data1
);
    reg [31:0] registers [0:31];

    initial begin
        integer i;
        for (i = 0; i < 32; i = i + 1)
            registers[i] = 32'b0;
    end
    always @(*) begin
        read_data1 = registers[source_reg1];
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            integer i;
            for (i = 0; i < 32; i = i + 1)
                registers[i] <= 32'b0;
        end else if (wen && (destination_reg != 5'b0)) begin
            registers[destination_reg] <= write_data;
        end
    end
Endmodule

//—----------------------------------------------------------------//

module ALU(
    input [31:0] op1,
    input [31:0] op2,
    input [2:0] funct3,
    input [6:0] funct7,
    input [4:0] shamt,
    input [2:0] insn_type,
    output reg [31:0] result
);
    always @ (*) begin
        case (insn_type)
            3'b000: begin
                case (funct3)
                    // ADDI
                    3'b000: result = op1 + op2;
                    // SLTI
                    3'b010: begin
                        if ($signed(op1) < $signed(op2)) begin
                            result = 1;
                        end
                        else begin
                            result = 0;
```

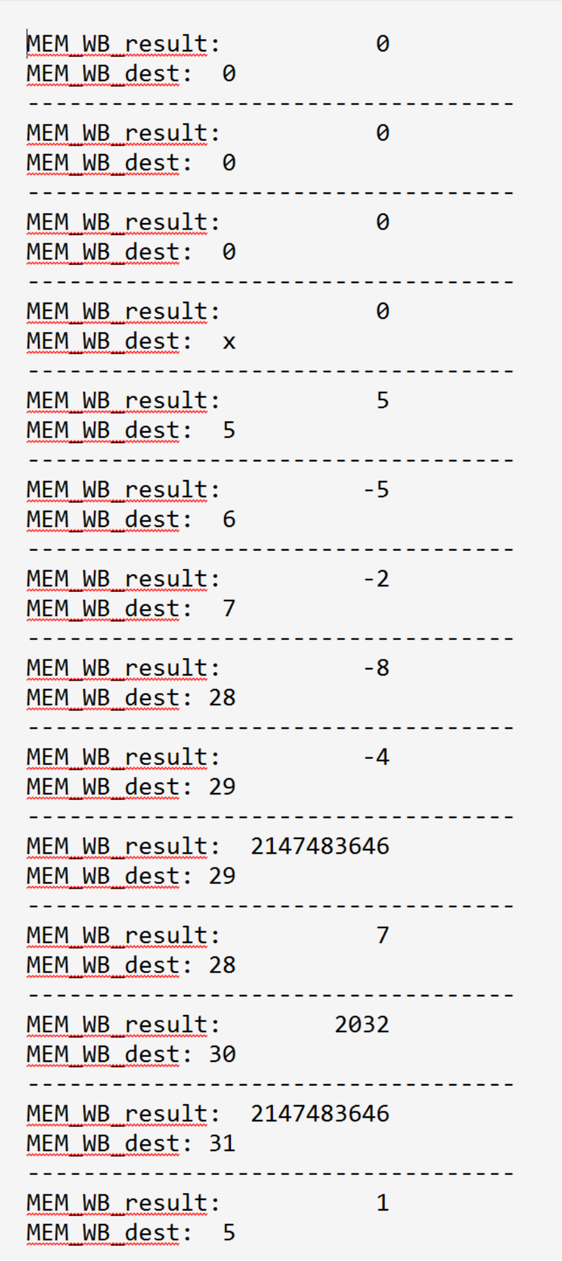


```
                        end
                    end
                    // SLTIU
                    3'b011: begin
                        if (op1 < op2) begin
                            result = 1;
                        end
                        else begin
                            result = 0;
                        end
                    end
                    // XORI
                    3'b100: result = op1 ^ op2;
                    // ORI
                    3'b110: result = op1 | op2;
                    // ANDI
                    3'b111: result = op1 & op2;
                    // SLLI
                    3'b001: result = op1 << shamt;
                    // SRLI / SRAI
                    3'b101: begin
                        if (funct7 == 7'b0100000) begin
                            // SRAI
                            result = $signed(op1) >>> shamt;
                        end
                        else if (funct7 == 7'b0000000) begin
                            // SRLI
                            result = op1 >> shamt;
                        end
                        else begin
                            // Default case for funct3 = 3'b101
                            result = 32'b0;
                        end
                    end
                    // Default case for funct3
                    default: result = 32'b0;
                endcase
            end
            // Default case for insn_type
            default: result = 32'b0;
        endcase
    end
endmodule
```

Code explanation:

The program counter has the task of holding the address of the current instruction and incrementing by 4 on each clock cycle. Should an invalid memory address be accessed, it would default to zero to prevent unpredictable behavior. The instruction decoder extracts fields from a 32-bit instruction, including opcode, registers, and immediate values. If an invalid opcode is detected, it would default to a NOP. The register file is a storage that supports asynchronous reads and synchronous writes. If a register that is invalid is called then a warning is issued, and the operation is ignored. The ALU part of the code would execute arithmetic and logical operations like addition, subtraction, AND, OR, and XOR based on the opcode. The CPU model integrates the PC, instruction decoder, register file, and ALU while handling pipeline execution. If a hazard is detected by the same register being used in the last 2 instructions, then the pipeline stalls or flushes instructions.

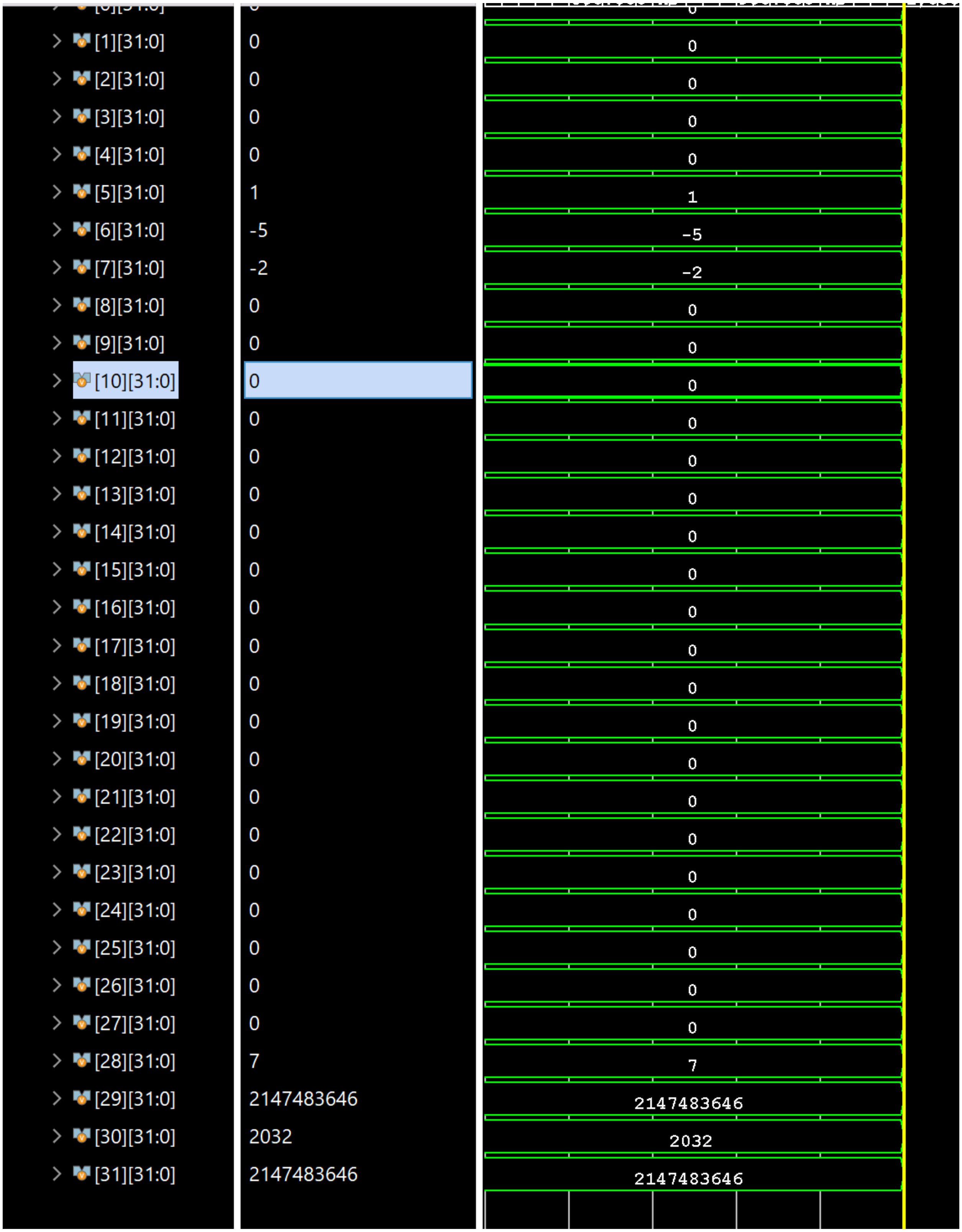
Trace files:



```
MEM_WB_result:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  x
-----------------------------------
MEM_WB_result:           5
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:          -5
MEM_WB_dest:  6
-----------------------------------
MEM_WB_result:          -2
MEM_WB_dest:  7
-----------------------------------
MEM_WB_result:          -8
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:          -4
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:  2147483646
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:           7
MEM_WB_dest: 28
-----------------------------------
MEM_WB_result:        2032
MEM_WB_dest: 30
-----------------------------------
MEM_WB_result:  2147483646
MEM_WB_dest: 31
-----------------------------------
MEM_WB_result:           1
MEM_WB_dest:  5
```

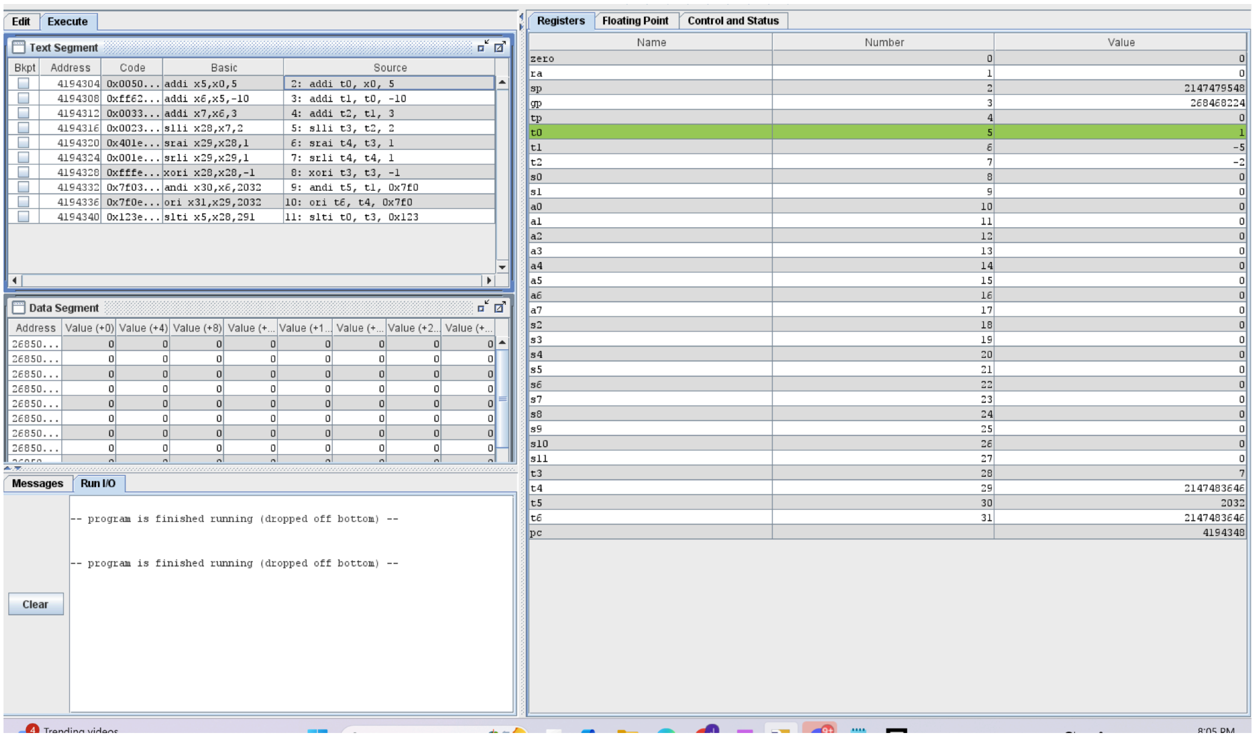
```
PC: 0x00000000
PC: 0x00000000
PC: 0x00000004
PC: 0x00000008
PC: 0x0000000c
PC: 0x00000010
PC: 0x00000014
PC: 0x00000018
PC: 0x0000001c
PC: 0x00000020
PC: 0x00000024
PC: 0x00000028
PC: 0x0000002c
PC: 0x00000030
PC: 0x00000034
PC: 0x00000038
PC: 0x0000003c
PC: 0x00000040
PC: 0x00000044
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000050
PC: 0x00000054
PC: 0x00000058
PC: 0x0000005c
PC: 0x00000060
PC: 0x00000064
PC: 0x00000068
PC: 0x0000006c
PC: 0x00000070
PC: 0x00000074
PC: 0x00000078
PC: 0x0000007c
PC: 0x00000080
PC: 0x00000084
PC: 0x00000088
PC: 0x0000008c
PC: 0x00000090
PC: 0x00000094
PC: 0x00000098
PC: 0x0000009c
PC: 0x000000a0
PC: 0x000000a4
PC: 0x000000a8
PC: 0x000000ac
PC: 0x000000b0
PC: 0x000000b4
PC: 0x000000b8
```

Waveform:



[0][31:0]	0	0
[1][31:0]	0	0
[2][31:0]	0	0
[3][31:0]	0	0
[4][31:0]	0	0
[5][31:0]	1	1
[6][31:0]	-5	-5
[7][31:0]	-2	-2
[8][31:0]	0	0
[9][31:0]	0	0
[10][31:0]	0	0
[11][31:0]	0	0
[12][31:0]	0	0
[13][31:0]	0	0
[14][31:0]	0	0
[15][31:0]	0	0
[16][31:0]	0	0
[17][31:0]	0	0
[18][31:0]	0	0
[19][31:0]	0	0
[20][31:0]	0	0
[21][31:0]	0	0
[22][31:0]	0	0
[23][31:0]	0	0
[24][31:0]	0	0
[25][31:0]	0	0
[26][31:0]	0	0
[27][31:0]	0	0
[28][31:0]	7	7
[29][31:0]	2147483646	2147483646
[30][31:0]	2032	2032
[31][31:0]	2147483646	2147483646

RARS assembly file:



Text Segment

Bkpt	Address	Code	Basic	Source
□	4194304	0x0050...	addi x5,x0,5	2: addi t0, x0, 5
□	4194308	0xff62...	addi x6,x5,-10	3: addi t1, t0, -10
□	4194312	0x0033...	addi x7,x6,3	4: addi t2, t1, 3
□	4194316	0x0023...	slli x28,x7,2	5: slli t3, t2, 2
□	4194320	0x401e...	srai x29,x28,1	6: srai t4, t3, 1
□	4194324	0x001e...	srli x29,x29,1	7: srli t4, t4, 1
□	4194328	0xfffe...	xori x28,x28,-1	8: xori t3, t3, -1
□	4194332	0x7f03...	andi x30,x6,2032	9: andi t5, t1, 0x7f0
□	4194336	0x7f0e...	ori x31,x29,2032	10: ori t6, t4, 0x7f0
□	4194340	0x123e...	slti x5,x28,291	11: slti t0, t3, 0x123

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+...	Value (+1...	Value (+...	Value (+2...	Value (+...
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0
26850...	0	0	0	0	0	0	0	0

```
-- program is finished running (dropped off bottom) --


-- program is finished running (dropped off bottom) --
```

Name	Number	Value
zero	0	0
ra	1	0
sp	2	2147479548
gp	3	268468224
tp	4	0
t0	5	1
t1	6	-5
t2	7	-2
s0	8	0
s1	9	0
a0	10	0
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0
s2	18	0
s3	19	0
s4	20	0
s5	21	0
s6	22	0
s7	23	0
s8	24	0
s9	25	0
s10	26	0
s11	27	0
t3	28	7
t4	29	2147483646
t5	30	2032
t6	31	2147483646
pc		4194348

Lab Learning:

This lab builds on top of the design we implemented in Lab 3, where we add the other immediate instruction. Which shill included planning out dividing the pipeline stages and understanding how the RISC-V assembly code operates which includes the PC and CPU structure via the gateways to the various modules handling the data down the pipeline. Now, we would be adding all of the other immediate institutions to the interpreter. Understanding how the processor deciphers assembly code into binary code and does calculations was instrumental to understanding and scripting this processor. In this particular lab, we implemented the basic immediate instruction execution and coded how the instructions are handled, which included how the binary is formatted in I format. Our implementation was not entirely functional however, as we ran into a problem regarding the stall value updating at each clock edge causing the PC to increment. This led to the issue of !stall always being 0 whenever the value was checked leading to the next instruction to be loaded which turned stall off. We tried to fix this by moving stall into a sequential block instead of a combinational block as well as having stall propagate before PC updates, but this only partially fixed the problem as it stops running after reg 6.