Mihir Phadke (015745301)
Jai Nayyar (015176655)
Ryan Smith (015792504)
Thurman Dao (015708589)
Haydon Behl (016390257)
Anson Lee
Group 9
Lab 5

March 27, 2025

Code:

```
// ALU Module
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
        3'b001: begin
            case(funct3)
                // add or sub
                3'b000: begin
```

```
                    if(funct7 == 7'b0000000) begin
                        result = op1 + op2;
                    end
                    else if(funct7 == 7'b0100000) begin
                        result = op1 - op2;
                    end
                end
                3'b001:
                    result = op1 << op2[4:0];
                3'b010: begin
                    if ($signed(op1) < $signed(op2)) begin
                        result = 1;
                    end
                    else begin
                        result = 0;
                    end
                end
                3'b011: begin
                    if (op1 < op2) begin
                        result = 1;
                    end
                    else begin
                        result = 0;
                    end
                end
                3'b100:
                    result = op1 ^ op2;
                3'b101: begin
                    if(funct7 == 7'b0100000) begin
                        result = $signed(op1) >>> op2[4:0];
                    end
                    else if(funct7 == 7'b0000000) begin
                        result = op1 >> op2[4:0];
                    end
                end
                3'b110:
                    result = op1 | op2;
                3'b111:
                    result = op1 & op2;
```

```
                default: result = 32'b0;
            endcase
            end
            // Default case for insn_type
            default: result = 32'b0;
        endcase
    end
endmodule

// Program Counter Module
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
`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, ID_EX_src2;
    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;
    reg ID_EX_alu_op_mux;
    reg ID_EX_wen;

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

    // 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 [4:0] source_reg2;
    wire [11:0] imm;
     wire [6:0] funct7;
    wire [4:0] shamt;
    wire [2:0] insn_type;
    wire alu_op_mux;
    wire wen;

    instruction_decoder decoder(
        .imem_insn(IF_ID_insn),
        .destination_reg(destination_reg),
        .funct3(funct3),
        .source_reg1(source_reg1),
```

```
        .source_reg2(source_reg2),
        .imm(imm),
        .funct7(funct7),
        .shamt(shamt),
        .insn_type(insn_type),
        .alu_op_mux(alu_op_mux),
        .wen(wen)
    );


      // 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_src2 <= 5'b0;
            ID_EX_imm <= 32'b0;
            ID_EX_funct3 <=3'b0;
            ID_EX_funct7 <= 7'b0;
            ID_EX_insn_type <= 3'b111;
            ID_EX_wen <= 0;
        end else begin
            ID_EX_pc <= IF_ID_pc;
            ID_EX_dest <= destination_reg;
            ID_EX_src1 <= source_reg1;
            ID_EX_src2 <= source_reg2;
            ID_EX_insn_type <= insn_type;
            ID_EX_funct3 <= funct3;
            ID_EX_funct7 <= funct7;
            ID_EX_shamt <= shamt;
            ID_EX_alu_op_mux <= alu_op_mux;
            ID_EX_imm <= {{20{imm[11]}}, imm};
            ID_EX_wen <= wen;
        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;
    reg [31:0] forwarded_op2;

always @(*) begin
    forwarded_op1 = reg_data1;
    forwarded_op2 = reg_data2;

    if ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == ID_EX_src1) &&
EX_MEM_wen) begin
        forwarded_op1 = EX_MEM_alu_result;
    end
    else if ((MEM_WB_dest != 5'b0) && (MEM_WB_dest == ID_EX_src1) &&
MEM_WB_wen) begin
        forwarded_op1 = MEM_WB_result;
    end

    if ((EX_MEM_dest != 5'b0) && (EX_MEM_dest == ID_EX_src2) &&
EX_MEM_wen) begin
        forwarded_op2 = EX_MEM_alu_result;
    end
    else if ((MEM_WB_dest != 5'b0) && (MEM_WB_dest == ID_EX_src2) &&
```

```
MEM_WB_wen) begin
        forwarded_op2 = MEM_WB_result;
    end

    if (ID_EX_alu_op_mux) begin
        forwarded_op2 = ID_EX_imm;
    end
end



    // Register File
    wire [31:0] reg_data1;
    wire [31:0] reg_data2;

    register_file reg_file(
        .clk(clk),
        .rst_n(rst_n),
        .wen(MEM_WB_wen),
        .destination_reg(MEM_WB_dest),
        .source_reg1(ID_EX_src1),
        .source_reg2(ID_EX_src2),
        .write_data(MEM_WB_result),
        .read_data1(reg_data1),
        .read_data2(reg_data2)
    );

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

    ALU alu(
        .op1(forwarded_op1),
        .op2(forwarded_op2),
        .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;
            EX_MEM_wen <= ID_EX_wen;
        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_wen;
        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
// 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 [4:0] source_reg2,
    output reg [11:0] imm,
     output reg [6:0] funct7,
    output reg [4:0] shamt,
    output reg [2:0] insn_type,
    output reg alu_op_mux,
    output reg wen
);

  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];
          source_reg2 = 0;
          imm = imem_insn[31:20];
          funct7 = imem_insn[31:25];
          shamt = imem_insn[24:20];
          alu_op_mux = 1;
          wen = 1;
        end
        7'b0110011: begin //R type instruction
          insn_type = 3'b001;
          destination_reg = imem_insn[11:7];
          funct3 = imem_insn[14:12];
          source_reg1 = imem_insn[19:15];
          source_reg2 = imem_insn[24:20];
          funct7 = imem_insn[31:25];
          imm = 0;
          shamt = 0;
          alu_op_mux = 0;
          wen = 1;
        end
        default: begin
          insn_type = 3'b111;
          destination_reg = 0;
          funct3 = 0;
          source_reg1 = 0;
          source_reg2 = 0;
          imm = 0;
          funct7 = 0;
          shamt = 0;
          wen = 0;
          alu_op_mux = 0;
        end
      endcase
    end
  endmodule
// Register File Module
```

```
module register_file(
    input clk,
    input rst_n,
    input wen,
    input [4:0] destination_reg,
    input [4:0] source_reg1,
    input [4:0] source_reg2,
    input [31:0] write_data,
    output reg [31:0] read_data1,
    output reg [31:0] read_data2
);
    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];
        read_data2 = registers[source_reg2];
    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
```

Code explanation:

This code builds on the code of lab four with the addition of R-type instructions. These instructions were implemented by extending the alu module. To differentiate between i and r type, the insn_type is checked in a case statement, with 001 denoting r type. Inside this r-type case, the various r-type operations, such as addition, subtraction, and shifts, can be accessed via the instructions func 3 and func 7. The forwarding logic also had to be updated to handle the r-type instructions. To do this, we had to make sure the second operand can also be forwarded, unlike in I-type where only one source needs forwarding. 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:

pc.txt | data.txt

Lab_5.sim > sim_1 > behav > xsim > pc.txt

```
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
```

Program counter trace file

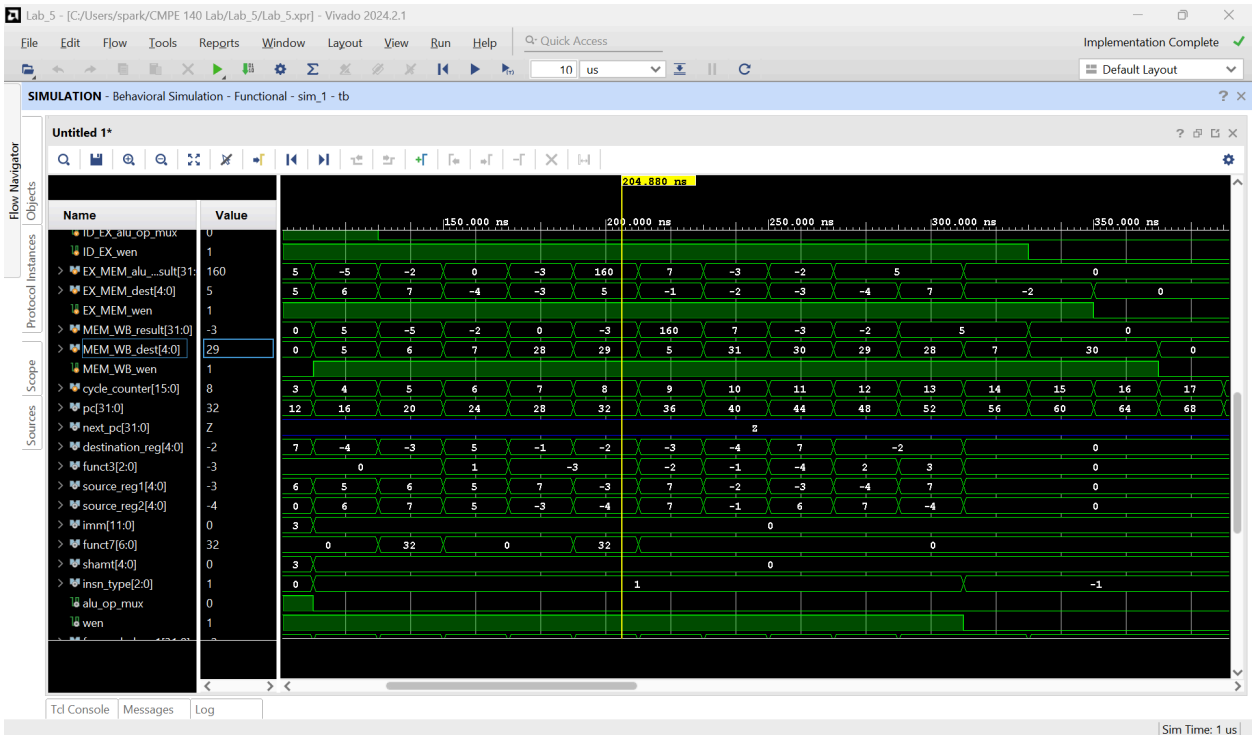
pc.txt | data.txt

Lab_5.sim > sim_1 > behav > xsim > data.txt

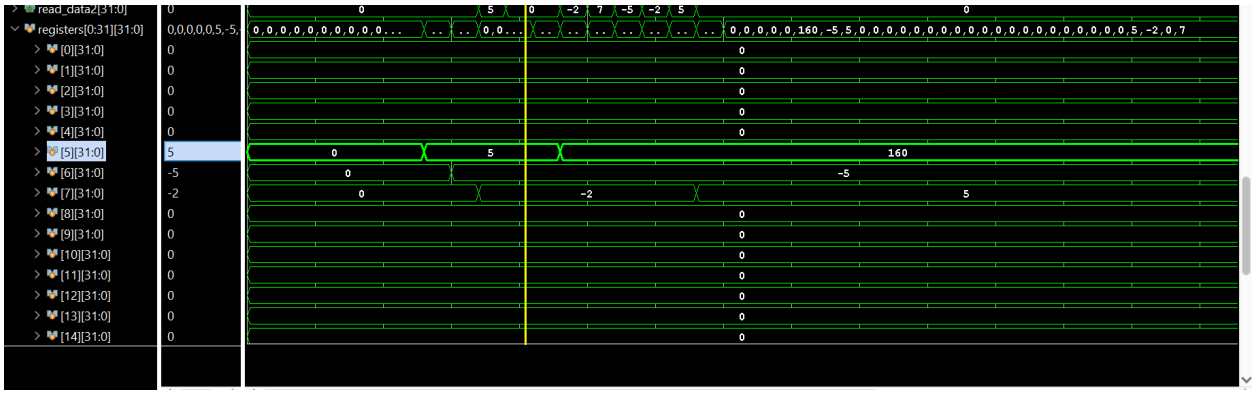
```
MEM_WB_result:           0
MEM_WB_dest:  0
---------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
---------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
---------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
---------------------------------
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:           0
MEM_WB_dest: 28
---------------------------------
MEM_WB_result:          -3
MEM_WB_dest: 29
---------------------------------
MEM_WB_result:         160
MEM_WB_dest:  5
---------------------------------
MEM_WB_result:           7
MEM_WB_dest: 31
---------------------------------
MEM_WB_result:          -3
MEM_WB_dest: 30
---------------------------------
MEM_WB_result:          -2
MEM_WB_dest: 29
---------------------------------
MEM_WB_result:           5
MEM_WB_dest: 28
---------------------------------
MEM_WB_result:           5
MEM_WB_dest:  7
---------------------------------
MEM_WB_result:           0
MEM_WB_dest: 30
---------------------------------
MEM_WB_result:           0
MEM_WB_dest: 30
```

Write back trace file

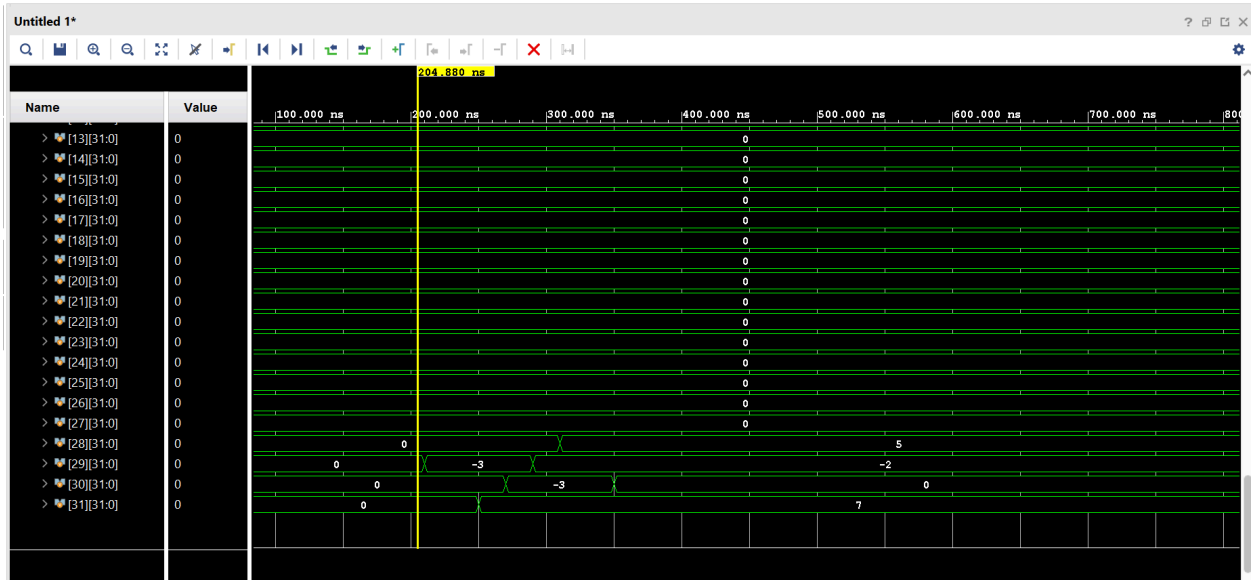
Waveform:



Write back result waveforms

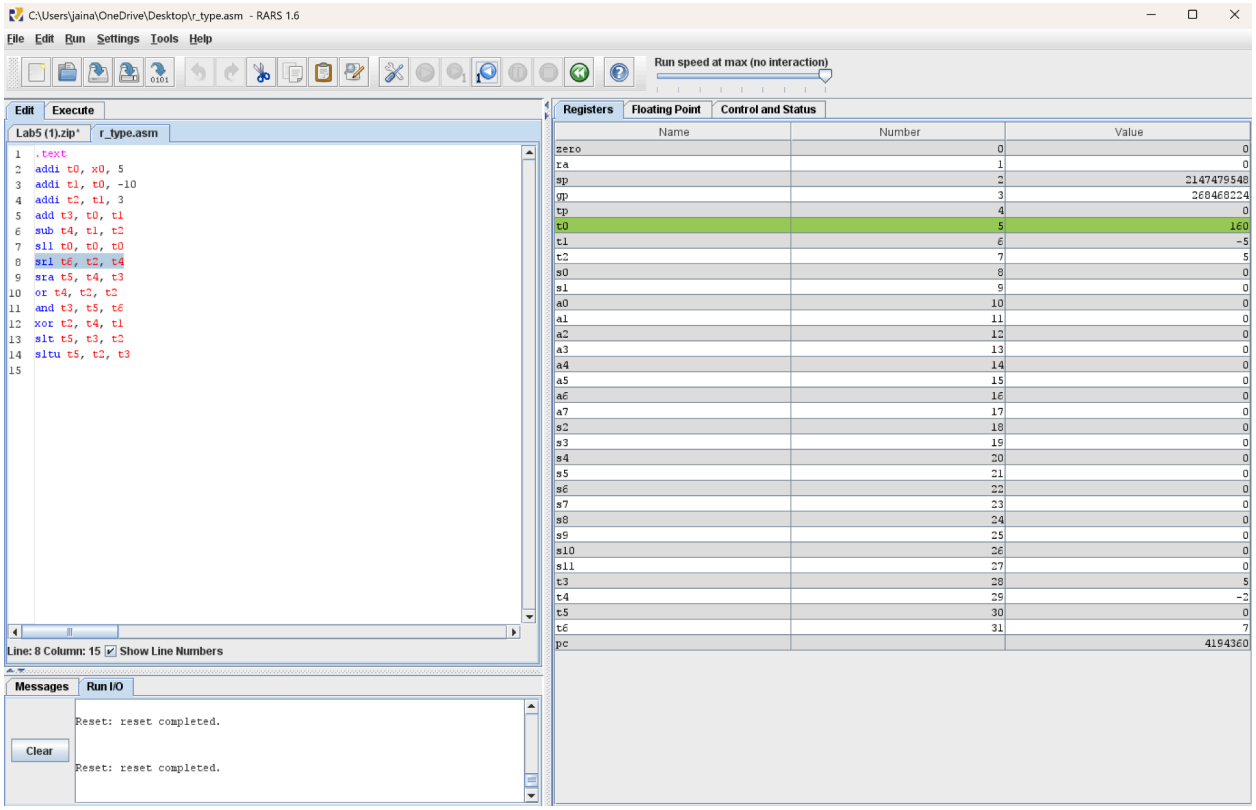


Values written to registers 5, 6, 7



Values written to registers 28, 29, 30, 31

RARS assembly file:



Lab Learnings:

This lab builds on top of the design we implemented in Lab 4, where we add R-type instructions. The biggest difference between the two is that R-type instructions use two source registers when implementing instructions, which differs from the I-type that uses just one source register. With this additional instruction set, the forwarding logic had to be updated as it only accounted for one source register. Adding forwarding for the other register was rather simple, as the code was essentially the same. One of the most important learning opportunities in this lab was reading from the pipeline stages to the output in the waveform. Doing this allowed for an in-depth visual to track any issues during the execution of each instruction at each stage of the pipeline. This proved extremely helpful during debugging, as we could pinpoint exactly which instruction was failing and know what needed to be fixed in our design.