Mihir Phadke
Haydon Behl
Ryan Smith
Jai Nayyar
Thurman Dao
Anson Lee
Apr 16, 2025

```
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)
            // I-type arithmetic instructions
            3'b000: begin
                case (funct3)
                    3'b000: result = op1 + op2;          // ADDI
                    3'b010: result = ($signed(op1) < $signed(op2)) ? 1 : 0;
// SLTI
                    3'b011: result = (op1 < op2) ? 1 : 0;  // SLTIU
                    3'b100: result = op1 ^ op2;            // XORI
                    3'b110: result = op1 | op2;            // ORI
                    3'b111: result = op1 & op2;            // ANDI
                    3'b001: result = op1 << shamt;         // SLLI
                    3'b101: begin
                        if (funct7 == 7'b0100000)
                            result = $signed(op1) >>> shamt; // SRAI
                        else if (funct7 == 7'b0000000)
                            result = op1 >> shamt;           // SRLI
                        else
                            result = 32'b0;
                    end
                    default: result = 32'b0;
                endcase
            end
            // R-type arithmetic instructions
            3'b001: begin
```

```
                case(funct3)
                    3'b000: begin
                        if(funct7 == 7'b0000000)
                            result = op1 + op2;  // ADD
                        else if(funct7 == 7'b0100000)
                            result = op1 - op2;  // SUB
                    end
                    3'b001: result = op1 << op2[4:0];         // SLL
                    3'b010: result = ($signed(op1) < $signed(op2)) ? 1 : 0;
// SLT
                    3'b011: result = (op1 < op2) ? 1 : 0;       // SLTU
                    3'b100: result = op1 ^ op2;                 // XOR
                    3'b101: begin
                        if(funct7 == 7'b0100000)
                            result = $signed(op1) >>> op2[4:0]; // SRA
                        else if(funct7 == 7'b0000000)
                            result = op1 >> op2[4:0];           // SRL
                    end
                    3'b110: result = op1 | op2;                 // OR
                    3'b111: result = op1 & op2;                 // AND
                    default: result = 32'b0;
                endcase
            end
            // S-type (store) and Load: effective address = base + offset.
            3'b010, 3'b011: begin
                result = op1 + op2;
            end
            // U-type
            3'b100: begin
                result = op1 + op2;
            end
            // J-type (jump): target address = base + offset.
            3'b101: begin
                result = op1 + op2;
            end
            // B-type (branch): target address = PC + offset.
            3'b110: begin
                result = op1 + op2;
            end
            // Default: result = 0.
            default: result = 32'b0;
        endcase
    end
```

```
endmodule
```

```
module PC(
    input         rst_n,
    input         clk,
    input         stall,
    input         branch_jump,   // Asserted when a branch or jump is taken
    input  [31:0] new_pc,        // Target address when branch/jump is
active
    output reg [31:0] pc
);
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            pc <= 32'b0;
        else if (stall)
            pc <= pc;            // Hold PC during a stall
        else if (branch_jump)
            pc <= new_pc;        // Load new target address for branch/jump
        else
            pc <= pc + 4;        // Otherwise, proceed sequentially
    end
endmodule
```

```
module cpu(
    input         rst_n,
    input         clk,
    output [31:0] imem_addr,
    input  [31:0] imem_insn,
    output reg [31:0] dmem_addr,
    inout  [31:0] dmem_data,
    output reg        dmem_wen,
    output      [3:0] byte_en
);
    // Stall signal (for hazards)
    reg stall;

    //-------------------------------------------------------
    // Pipeline Registers
    //-------------------------------------------------------
    // IF/ID pipeline registers
    reg [31:0] IF_ID_insn, IF_ID_pc;
```

```
    reg ID_wen;
    reg ID_dmem_wen;
    // A flush register to clear IF/ID on branch/jump.
    reg flush_pipeline;

    // ID/EX pipeline registers
    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;
    reg        ID_EX_dmem_wen;
    reg [31:0] ID_EX_store_data;

    // EX/MEM pipeline registers
    reg [31:0] EX_MEM_alu_result;
    reg [4:0]  EX_MEM_dest;
    reg        EX_MEM_wen;
    reg        EX_MEM_dmem_wen;
    reg [2:0]  EX_MEM_insn_type;  // (e.g. 010: store, 011: load)
    reg [2:0]  EX_MEM_funct3;
    reg [31:0] EX_MEM_store_data;

    // MEM/WB pipeline registers
    reg signed [31:0] MEM_WB_result;
    reg [4:0]         MEM_WB_dest;
    reg               MEM_WB_wen;
    reg [2:0]         MEM_WB_insn_type;
    reg [2:0]         MEM_WB_funct3;
    reg [31:0]        MEM_WB_alu_result;
    reg [31:0]        MEM_WB_ram_result;

    //------------------------------------------------------
    // dmem_data Tri-state Control & Byte Enable Signals
    //------------------------------------------------------
    reg [31:0] dmem_data_out;
    assign dmem_data = (dmem_wen) ? dmem_data_out : 32'hz;

    reg [3:0] byte_en_dmem_reg;
    assign byte_en = byte_en_dmem_reg;
```

```
    reg [3:0] byte_en_reg;  // For register file (if needed)

    //------------------------------------------------------
    // Cycle Counter (for debugging/tracing)
    //------------------------------------------------------
    reg [15:0] cycle_counter;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            cycle_counter <= 16'b0;
        else
            cycle_counter <= cycle_counter + 1;
    end

    //------------------------------------------------------
    // Program Counter (PC) Module Instantiation
    // The PC now accepts branch/jump override signals.
    //------------------------------------------------------
    wire [31:0] pc;
    wire branch_jump;
    wire [31:0] new_pc;

    PC pc_module(
        .rst_n(rst_n),
        .clk(clk),
        .stall(stall),
        .branch_jump(branch_jump),
        .new_pc(branch_target),
        .pc(pc)
    );

    // The instruction memory address is driven by the PC.
    assign imem_addr = pc;

    //------------------------------------------------------
    // Flush Pipeline Register
    // We register the branch/jump decision (from EX stage) so that the
IF/ID
    // registers are flushed in the next clock cycle.
    //------------------------------------------------------
    always @(posedge clk or negedge rst_n) begin
        if ((!rst_n) | flush_pipeline)
            flush_pipeline <= 1'b0;
        else
```

```
            flush_pipeline <= branch_jump;
    end

    //---------------------------------------------------------
    // Instruction Fetch (IF) Stage
    //---------------------------------------------------------
    always @(posedge clk or negedge rst_n) begin
        if ((!rst_n) | branch_taken) begin // Do not grab next insn if we
are about to jump!
            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: Instruction Decoder Instantiation
    //---------------------------------------------------------
    // Our updated decoder outputs a 32-bit immediate.
    wire [4:0] destination_reg;
    wire [2:0] funct3;
    wire [4:0] source_reg1;
    wire [4:0] source_reg2;
    wire [31:0] imm;
    wire [6:0] funct7;
    wire [4:0] shamt;
    wire [2:0] insn_type;
    // insn_type encoding:
    // 000: I-type, 001: R-type, 010: S-type, 011: Load,
    // 100: U-type, 101: J-type, 110: Branch, 111: Default (NOP)
    wire alu_op_mux;

    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(ID_wen),
        .dmem_wen(ID_dmem_wen)
    );

    //-----------------------------------------------------
    // ID/EX Pipeline Register Update
    //-----------------------------------------------------
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n | branch_jump) 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;  // Default to NOP
            ID_EX_shamt      <= 5'b0;
            ID_EX_alu_op_mux <= 1'b0;
            ID_EX_wen        <= 1'b0;
            ID_EX_dmem_wen   <= 1'b0;
            ID_EX_store_data <= 32'b0;
        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        <= imm;
            ID_EX_wen        <= ID_wen;
            ID_EX_dmem_wen   <= ID_dmem_wen;
        end
    end

    //-----------------------------------------------------
    // Register File Instance
    //-----------------------------------------------------
```

```
    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),
        .byte_en(byte_en_reg),
        .read_data1(reg_data1),
        .read_data2(reg_data2)
    );

    //---------------------------------------------------------
    // Forwarding Logic (for ALU operands)
    //---------------------------------------------------------
    wire [31:0] forwarded_op1;
    wire [31:0] forwarded_op2;

    // Register "skips" for when the requested data is not yet written to
reg.
    assign forwarded_op1 = ((EX_MEM_dest != 5'b0) && (EX_MEM_dest ==
ID_EX_src1) && EX_MEM_wen) ? EX_MEM_alu_result :
                           ((MEM_WB_dest != 5'b0) && (MEM_WB_dest ==
ID_EX_src1) && MEM_WB_wen) ? MEM_WB_result :
                           reg_data1;
    assign forwarded_op2 = (ID_EX_alu_op_mux) ? ID_EX_imm :
                           ((EX_MEM_dest != 5'b0) && (EX_MEM_dest ==
ID_EX_src2) && EX_MEM_wen) ? EX_MEM_alu_result :
                           ((MEM_WB_dest != 5'b0) && (MEM_WB_dest ==
ID_EX_src2) && MEM_WB_wen) ? MEM_WB_result :
                           reg_data2;

    //---------------------------------------------------------
    // ALU Operand for Jump Instructions
    // For JAL, the base should be the PC. For JALR, use register.
    //---------------------------------------------------------
    wire [31:0] alu_op1;
    assign alu_op1 = ((ID_EX_insn_type == 3'b101) && (ID_EX_src1 == 5'b0))
? ID_EX_pc : forwarded_op1;
```

```
    //------------------------------------------------------
    // Execute (EX) Stage: ALU Instance
    //------------------------------------------------------
    wire [31:0] alu_result;
    ALU alu (
        .op1(alu_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)
    );

    //------------------------------------------------------
    // Branch Decision & Target Computation (EX Stage)
    //------------------------------------------------------
    wire branch_taken;
    wire [31:0] branch_target;

    assign branch_taken = (flush_pipeline) ? 1'b0 : // cant jump during a
flush
        (ID_EX_insn_type == 3'b101) ? 1'b1 : // JAL
        (ID_EX_insn_type == 3'b110) ?
        ((ID_EX_funct3 == 3'b000) ? (forwarded_op1 == forwarded_op2) :  //
BEQ
        (ID_EX_funct3 == 3'b001) ? (forwarded_op1 != forwarded_op2) :  //
BNE
        (ID_EX_funct3 == 3'b100) ? ($signed(forwarded_op1) <
$signed(forwarded_op2)) :  // BLT
        (ID_EX_funct3 == 3'b101) ? ($signed(forwarded_op1) >=
$signed(forwarded_op2)) : // BGE
        (ID_EX_funct3 == 3'b110) ? (forwarded_op1 < forwarded_op2) :   //
BLTU
        (ID_EX_funct3 == 3'b111) ? (forwarded_op1 >= forwarded_op2) :  //
BGEU
        1'b0)
        : 1'b0;

    assign branch_target = ID_EX_pc + ID_EX_imm;

    //------------------------------------------------------
    // Branch/Jump Control Signals for PC Update
```

```
    // For jumps (insn_type 3'b101) the branch is unconditional.
    // For branches (insn_type 3'b110) use branch_taken.
    //----------------------------------------------------------
    assign branch_jump = (((ID_EX_insn_type == 3'b101) || ((ID_EX_insn_type
== 3'b110) && branch_taken)) && (!flush_pipeline));

    //----------------------------------------------------------
    // EX/MEM Pipeline Register Update
    //----------------------------------------------------------
    always @(posedge clk or negedge rst_n) begin
        if ((!rst_n) | branch_jump) begin
            EX_MEM_alu_result <= 32'b0;
            EX_MEM_dest       <= 5'b0;
            EX_MEM_wen        <= 1'b0;
            EX_MEM_dmem_wen   <= 1'b0;
            EX_MEM_funct3     <= 3'b0;
            EX_MEM_insn_type  <= 3'b0;
            EX_MEM_store_data <= 32'b0;
        end else begin
            EX_MEM_alu_result <= alu_result;
            EX_MEM_dest       <= ID_EX_dest;
            EX_MEM_wen        <= ID_EX_wen;
            EX_MEM_dmem_wen   <= ID_EX_dmem_wen;
            EX_MEM_funct3     <= ID_EX_funct3;
            EX_MEM_insn_type  <= ID_EX_insn_type;
            EX_MEM_store_data <= ID_EX_store_data;
        end
    end

    //----------------------------------------------------------
    // Memory (MEM) Stage
    //----------------------------------------------------------
    // Fix X values on dmem_data.
    wire [31:0] fixed_data;
    genvar i;
    generate
        for(i = 0; i < 32; i = i + 1) begin : fix_x_bits
            assign fixed_data[i] = (dmem_data[i] === 1'bx) ? 1'b1 :
dmem_data[i];
        end
    endgenerate

    // Load extraction logic (active for load instructions, insn_type
```

```
3'b011)
    wire [31:0] load_result;
    assign load_result = (MEM_WB_insn_type == 3'b011) ? (
        (MEM_WB_funct3 == 3'b000) ? { {24{fixed_data[7]}},
fixed_data[7:0]} : // LB
        (MEM_WB_funct3 == 3'b001) ? { {16{fixed_data[15]}},
fixed_data[15:0]} : // LH
        (MEM_WB_funct3 == 3'b010) ? fixed_data :
// LW
        (MEM_WB_funct3 == 3'b100) ? {24'b0, fixed_data[7:0]} :
// LBU
        (MEM_WB_funct3 == 3'b101) ? {16'b0, fixed_data[15:0]} :
// LHU
        fixed_data
    ) : 32'b0;

    // Write-back selection: use load_result for loads; otherwise use the
ALU result.
    assign MEM_WB_result = (MEM_WB_insn_type == 3'b011) ? load_result :
MEM_WB_alu_result;

    //-----------------------------------------------------
    // MEM/WB Pipeline Register Update
    //-----------------------------------------------------
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            MEM_WB_alu_result <= 32'b0;
            MEM_WB_dest     <= 5'b0;
            MEM_WB_wen      <= 1'b0;
            MEM_WB_insn_type<= 3'b0;
            MEM_WB_funct3   <= 3'b0;
            dmem_wen        <= 1'b0;
            dmem_addr       <= 32'b0;
            byte_en_dmem_reg<= 4'b0;
            byte_en_reg     <= 4'b0;
        end else begin
            dmem_data_out = EX_MEM_store_data;
            if (EX_MEM_insn_type == 3'b011) begin
                dmem_addr <= EX_MEM_alu_result; // Effective address for
load
                byte_en_dmem_reg <= 4'b0000;
                byte_en_reg <= 4'b1111;
            end else if (EX_MEM_insn_type == 3'b010) begin
```

```
                dmem_addr <= EX_MEM_alu_result; // Effective address for
store
                case (EX_MEM_funct3)
                    3'b000: byte_en_dmem_reg <= 4'b0001; // SB
                    3'b001: byte_en_dmem_reg <= 4'b0011; // SH
                    3'b010: byte_en_dmem_reg <= 4'b1111; // SW
                    default: byte_en_dmem_reg <= 4'b0000;
                endcase
                byte_en_reg <= 4'b0000;
            end else begin
                dmem_addr <= 32'b0;
                byte_en_dmem_reg <= 4'b0000;
                byte_en_reg <= 4'b1111;
            end

            MEM_WB_funct3    <= EX_MEM_funct3;
            MEM_WB_alu_result<= EX_MEM_alu_result;
            MEM_WB_dest      <= EX_MEM_dest;
            MEM_WB_wen       <= EX_MEM_wen;
            MEM_WB_insn_type <= EX_MEM_insn_type;
            dmem_wen         <= EX_MEM_dmem_wen;
        end
    end

    //------------------------------------------------------------
    // File Trace Output (for debugging/tracing)
    //------------------------------------------------------------
    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
    end

    always @(posedge clk or negedge rst_n) begin
        if (rst_n) 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

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

endmodule
```

```
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 [31:0] imm,          // 32-bit full immediate (after
shift/sign-extension where needed)
    output reg [6:0]  funct7,
    output reg [4:0]  shamt,
    output reg [2:0]  insn_type,    // 000: I-type, 001: R-type, 010:
S-type, 011: Load,
                                    // 100: U-type, 101: J-type, 110:
Branch, 111: Default/NOP
    output reg        alu_op_mux,
    output reg        wen,
    output reg        dmem_wen
);

  always @ (*) begin
      case (imem_insn[6:0])
        // I-type instructions (e.g. ADDI, SLTI, ...)
        7'b0010011: begin
          insn_type       = 3'b000;
          destination_reg = imem_insn[11:7];
          funct3          = imem_insn[14:12];
          source_reg1     = imem_insn[19:15];
          source_reg2     = 5'b0;
```

```
          imm             = {{20{imem_insn[31]}}, imem_insn[31:20]};
          funct7          = imem_insn[31:25];
          shamt           = imem_insn[24:20];
          alu_op_mux      = 1'b1;
          wen             = 1'b1;
          dmem_wen        = 1'b0;
        end

        // R-type instructions (e.g. ADD, SUB, ...)
        7'b0110011: begin
          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             = 32'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b0;
          wen             = 1'b1;
          dmem_wen        = 1'b0;
        end

        // S-type instructions (Store instructions: e.g. SW)
        7'b0100011: begin
          insn_type       = 3'b010;
          destination_reg = 5'b0;
          funct3          = imem_insn[14:12];
          source_reg1     = imem_insn[19:15];
          source_reg2     = imem_insn[24:20];
          // Construct immediate from imm[11:5] and imm[4:0].
          imm             = {{20{imem_insn[31]}}, imem_insn[31:25],
imem_insn[11:7]};
          funct7          = 7'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b1;
          wen             = 1'b0;
          dmem_wen        = 1'b1;
        end

        // Load instructions (e.g. LW)
        7'b0000011: begin
          insn_type       = 3'b011;
```

```
      destination_reg = imem_insn[11:7];
      funct3          = imem_insn[14:12];
      source_reg1     = imem_insn[19:15];
      source_reg2     = 5'b0;
      imm             = {{20{imem_insn[31]}}, imem_insn[31:20]};
      funct7          = 7'b0;
      shamt           = 5'b0;
      alu_op_mux      = 1'b1;
      wen             = 1'b1;
      dmem_wen        = 1'b0;
    end

    // U-type instructions: LUI
    7'b0110111: begin
      insn_type       = 3'b100;
      destination_reg = imem_insn[11:7];
      funct3          = 3'b0; // not used
      source_reg1     = 5'b0;
      source_reg2     = 5'b0;
      // Immediate is bits [31:12] shifted left by 12.
      imm             = {imem_insn[31:12], 12'b0};
      funct7          = 7'b0;
      shamt           = 5'b0;
      alu_op_mux      = 1'b1;
      wen             = 1'b1;
      dmem_wen        = 1'b0;
    end

    // U-type instructions: AUIPC
    7'b0010111: begin
      insn_type       = 3'b100;
      destination_reg = imem_insn[11:7];
      funct3          = 3'b0; // not used
      source_reg1     = 5'b0;
      source_reg2     = 5'b0;
      imm             = {imem_insn[31:12], 12'b0};
      funct7          = 7'b0;
      shamt           = 5'b0;
      alu_op_mux      = 1'b1;
      wen             = 1'b1;
      dmem_wen        = 1'b0;
    end
```

```
        // J-type instruction: JAL
        7'b1101111: begin
          insn_type       = 3'b101;
          destination_reg = imem_insn[11:7];
          funct3          = 3'b0; // not used
          source_reg1     = 5'b0;
          source_reg2     = 5'b0;
          // Assemble immediate: imm[20|10:1|11|19:12] with a 0 as LSB.
          imm             = {{11{imem_insn[31]}}, imem_insn[31],
                             imem_insn[19:12], imem_insn[20],
                             imem_insn[30:21], 1'b0};
          funct7          = 7'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b0;
          wen             = 1'b1;  // Write return address (PC+4) into rd.
          dmem_wen        = 1'b0;
        end

        // J-type instruction: JALR (I-type format)
        7'b1100111: begin
          insn_type       = 3'b101;
          destination_reg = imem_insn[11:7];
          funct3          = imem_insn[14:12]; // should be 000
          source_reg1     = imem_insn[19:15];
          source_reg2     = 5'b0;
          imm             = {{20{imem_insn[31]}}, imem_insn[31:20]};
          funct7          = 7'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b0;
          wen             = 1'b1;
          dmem_wen        = 1'b0;
        end

        // B-type instructions: Branches (BEQ, BNE, BLT, BGE, BLTU, BGEU)
        7'b1100011: begin
          insn_type       = 3'b110;
          destination_reg = 5'b0;
          funct3          = imem_insn[14:12];
          source_reg1     = imem_insn[19:15];
          source_reg2     = imem_insn[24:20];
          // Assemble branch immediate: imm[12|10:5|4:1|11] with 0 as LSB.
          imm             = {{19{imem_insn[31]}}, imem_insn[31],
                             imem_insn[7], imem_insn[30:25],
```

```
                              imem_insn[11:8], 1'b0};
          funct7          = 7'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b0;
          wen             = 1'b0;
          dmem_wen        = 1'b0;
        end

        // Default case: Unknown instruction -> NOP.
        default: begin
          insn_type       = 3'b111;
          destination_reg = 5'b0;
          funct3          = 3'b0;
          source_reg1     = 5'b0;
          source_reg2     = 5'b0;
          imm             = 32'b0;
          funct7          = 7'b0;
          shamt           = 5'b0;
          alu_op_mux      = 1'b0;
          wen             = 1'b1;
          dmem_wen        = 1'b0;
        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 [4:0] source_reg2,
    input [31:0] write_data,
    input [3:0] byte_en,
    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;
            if (byte_en[3])
                    registers[destination_reg][31:24] <= #0.1
write_data[31:24];
            if (byte_en[2])
                    registers[destination_reg][23:16] <= #0.1
write_data[23:16];
            if (byte_en[1])
                    registers[destination_reg][15:8] <= #0.1
write_data[15:8];
            if (byte_en[0])
                    registers[destination_reg][7:0] <= #0.1
write_data[7:0];
        end
    end
endmodule
```

Code Explanation:

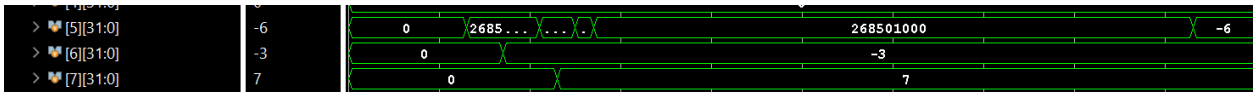
In this lab, we are going to be adding branch instructions into our project. These branch instructions are processed by being identified during the decode stage, where both operands and immediate offsets are extracted. The branch is based on operand and the funct3 is made in the execute stage only if the condition is met. Then, the program counter is updated to the branch target address. Finally, the pipeline flushes the next instruction when fetched and stalls when new instructions are loaded.

The way our flush logic works is that we trigger a flush flag as soon as the EX stage result confirms we will be jumping, at which point we spend the next clock cycle stalling to load the instruction at the jump address. After this point, the CPU continues as normal. One important part of the flush that we had to account for is the event in which two jump instructions

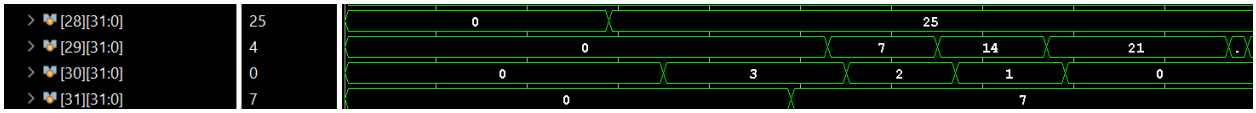
enter the pipeline. It's important that we ignore the jump signal from the latter instruction preemptively, as if it reaches our execution logic then it will override the first jump. To do so, we have combinational wires in place to ensure that we only jump to a new address under valid conditions, for example when our pipeline is not actively being flushed.

Aside from flush logic, the top-level of our CPU is mostly the same, albeit cleaned and organized from top to bottom since lab 6. The major module changes are in the PC, Decoder, and ALU which all have to accommodate for the new instructions. We also made some changes to our write logic in the regfile, not out of necessity for the lab but for ease of debugging, such that the write-enable flag is no longer just a flag, but a bus indicating which bits of the register must be overwritten.

Waveforms:



Values in registers 5, 6, 7.



Values in register 28, 29, 30, 31

Trace Files:

PC Counter Trace File

```
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: 0x00000000
PC: 0x00000038
PC: 0x0000003c
PC: 0x00000000
PC: 0x00000044
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000050
PC: 0x00000054
PC: 0x00000058
PC: 0x00000000
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000050
PC: 0x00000054
PC: 0x00000058
PC: 0x00000000
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000050
PC: 0x00000054
PC: 0x00000058
PC: 0x00000000
PC: 0x00000048
PC: 0x0000004c
PC: 0x00000000
PC: 0x00000058
PC: 0x0000005c
PC: 0x00000060
PC: 0x00000064
PC: 0x00000068
PC: 0x0000006c
PC: 0x00000070
```

MEM WB result and destination trace file

```
12  ------------------------------------
13  MEM_WB_result:   268500992
14  MEM_WB_dest:  5
15  ------------------------------------
16  MEM_WB_result:   268500992
17  MEM_WB_dest:  5
18  ------------------------------------
19  MEM_WB_result:          -3
20  MEM_WB_dest:  6
21  ------------------------------------
22  MEM_WB_result:   268500992
23  MEM_WB_dest:  5
24  ------------------------------------
25  MEM_WB_result:   268500996
26  MEM_WB_dest:  5
27  ------------------------------------
28  MEM_WB_result:           7
29  MEM_WB_dest:  7
30  ------------------------------------
31  MEM_WB_result:   268500992
32  MEM_WB_dest:  5
33  ------------------------------------
34  MEM_WB_result:   268501000
35  MEM_WB_dest:  5
36  ------------------------------------
37  MEM_WB_result:          25
38  MEM_WB_dest: 28
39  ------------------------------------
40  MEM_WB_result:           0
41  MEM_WB_dest: 29
42  ------------------------------------
43  MEM_WB_result:          -3
44  MEM_WB_dest:  0
45  ------------------------------------
46  MEM_WB_result:           3
47  MEM_WB_dest: 30
48  ------------------------------------
49  MEM_WB_result:           0
50  MEM_WB_dest:  0
51  ------------------------------------
52  MEM_WB_result:           0
53  MEM_WB_dest:  0
54  ------------------------------------
55  MEM_WB_result:           0
56  MEM_WB_dest:  0
57  ------------------------------------
58  MEM_WB_result:           0
59  MEM_WB_dest:  0
60  ------------------------------------
61  MEM_WB_result:           0
62  MEM_WB_dest:  0
63  ------------------------------------
64  MEM_WB_result:           0
65  MEM_WB_dest:  0
66  ------------------------------------
67  MEM_WB_result:           7
68  MEM_WB_dest: 31
69  ------------------------------------
70  MEM_WB_result:           3
71  MEM_WB_dest:  0
```

```
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           7
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:           2
MEM_WB_dest: 30
-----------------------------------
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:           2
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:          14
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:           1
MEM_WB_dest: 30
-----------------------------------
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:           1
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:          21
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest: 30
-----------------------------------
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:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:           0
MEM_WB_dest:  0
```

```
-----------------------------------
MEM_WB_result:          -6
MEM_WB_dest:  5
-----------------------------------
MEM_WB_result:          -6
MEM_WB_dest:  0
-----------------------------------
MEM_WB_result:         -21
MEM_WB_dest: 29
-----------------------------------
MEM_WB_result:           4
MEM_WB_dest: 29
-----------------------------------
```

Lab Learnings:
This lab builds on top of the stuff from Labs 4, 5, and 6. This lab implements the jump and branch instructions. The branch is already known at the decode stage and the next instruction has been fetched so the pipeline has to flush the fetched instruction and then stall until the branch is resolved. Then the branch is resolved at the end of the execution stage, then the stall ends and the next instruction is fetched from the branch target.