Lab 4

PB22030892 刘铠瑜

任务 1：访存控制单元设计

SLU.v

```
module SLU (
    input                   [31:0]              addr,
    input                   [3:0]               dmem_access,
    input                   [31:0]              rd_in,          // rd_in 读取的是
4 字节对齐的整字
    input                   [31:0]              wd_in,
    output reg              [31:0]              rd_out,
    output reg              [31:0]              wd_out
);

    `define LB                  4'b0000     // Load Byte
    `define LH                  4'b0001     // Load Halfword
    `define LW                  4'b0010     // Load Word
    `define LBU                 4'b0011     // Load Byte Unsigned
    `define LHU                 4'b0100     // Load Halfword Unsigned
    `define SB                  4'b0101     // Store Byte
    `define SH                  4'b0110     // Store Halfword
    `define SW                  4'b0111     // Store Word

    always @(*) begin
        case (dmem_access)
            // LB 和 LH 提取出整字中对应的部分后进行符号扩展
            `LB: begin
                case (addr[1:0])
                    2'b00: rd_out = {{24{rd_in[7]}}, rd_in[7:0]};       // 读取低
字节
                    2'b01: rd_out = {{24{rd_in[15]}}, rd_in[15:8]};     // 读取中
低字节
                    2'b10: rd_out = {{24{rd_in[23]}}, rd_in[23:16]};    // 读取中
高字节
                    2'b11: rd_out = {{24{rd_in[31]}}, rd_in[31:24]};    // 读取高
字节
                endcase
            end

            `LH: begin
                case (addr[1:0])
                    2'b00: rd_out = {{16{rd_in[15]}}, rd_in[15:0]};     // 读取低
半字
                    2'b10: rd_out = {{16{rd_in[31]}}, rd_in[31:16]};    // 读取高
半字
                    default: rd_out = 32'b0;
```

```
            endcase
        end

        // LW 直接读取即可
        `LW: begin
            rd_out = rd_in; // 读取字
        end

        // LBU 和 LHU 进行无符号扩展
        `LBU: begin
            case (addr[1:0])
                2'b00: rd_out = {24'b0, rd_in[7:0]};         // 读取低字节
                2'b01: rd_out = {24'b0, rd_in[15:8]};        // 读取中低字节
                2'b10: rd_out = {24'b0, rd_in[23:16]};       // 读取中高字节
                2'b11: rd_out = {24'b0, rd_in[31:24]};       // 读取高字节
            endcase
        end

        `LHU: begin
            case (addr[1:0])
                2'b00: rd_out = {16'b0, rd_in[15:0]};         // 读取低半字
                2'b10: rd_out = {16'b0, rd_in[31:16]};        // 读取高半字
                default: rd_out = 32'b0;
            endcase
        end

        // SB 和 SH 需要先读取存储器中对应的整字，然后修改对应的字节或半字后，再整字
写入
        `SB: begin
            case (addr[1:0])
                2'b00: wd_out = {rd_in[31:8], wd_in[7:0]};              // 写
入低字节
                2'b01: wd_out = {rd_in[31:16], wd_in[7:0], rd_in[7:0]}; // 写
入中低字节
                2'b10: wd_out = {rd_in[31:24], wd_in[7:0], rd_in[15:0]};// 写
入中高字节
                2'b11: wd_out = {wd_in[7:0], rd_in[23:0]};              // 写
入高字节
            endcase
        end

        `SH: begin
            case (addr[1:0])
                2'b00: wd_out = {rd_in[31:16], wd_in[15:0]};    // 写入低半字
                2'b10: wd_out = {wd_in[15:0], rd_in[15:0]};     // 写入高半字
                default: wd_out = 32'b0;
            endcase
        end

        `SW: begin
            wd_out = wd_in; // 写入字
        end

        default: begin
```

```
                wd_out = 32'b0;
                rd_out = 32'b0;
            end
        endcase
    end
endmodule
```

任务 2：搭建 CPU

BRANCH.v

```
module BRANCH(
    input                   [ 3 : 0]            br_type,

    input                   [31 : 0]            br_src0,
    input                   [31 : 0]            br_src1,

    output      reg         [ 1 : 0]            npc_sel
);

    `define BR_NJP              4'b0000
    `define BR_JALR             4'b0001
    `define BR_JAL              4'b0010
    `define BR_BEQ              4'b0011
    `define BR_BNE              4'b0100
    `define BR_BLT              4'b0101
    `define BR_BGE              4'b0110
    `define BR_BLTU             4'b0111
    `define BR_BGEU             4'b1000

    `define PC_ADD4     2'b00
    `define PC_OFFSET   2'b01
    `define PC_JALR     2'b10

    always @(*) begin
        npc_sel = 2'b00;

        case (br_type)
            `BR_JALR: npc_sel = `PC_JALR;
            `BR_JAL : npc_sel = `PC_OFFSET;
            `BR_BEQ : npc_sel = (br_src0 == br_src1) ?
                                `PC_OFFSET : `PC_ADD4;
            `BR_BNE : npc_sel = (br_src0 != br_src1) ?
                                `PC_OFFSET : `PC_ADD4;
            `BR_BLT : npc_sel = ($signed(br_src0) < $signed(br_src1)) ?
                                `PC_OFFSET : `PC_ADD4;
            `BR_BGE : npc_sel = ($signed(br_src0) >= $signed(br_src1)) ?
                                `PC_OFFSET : `PC_ADD4;
            `BR_BLTU: npc_sel = (br_src0 < br_src1) ?
                                `PC_OFFSET : `PC_ADD4;
            `BR_BGEU: npc_sel = (br_src0 >= br_src1) ?
```

```
                                    `PC_OFFSET : `PC_ADD4;
            default : npc_sel = `PC_ADD4;
        endcase
    end
endmodule
```

DECODER.v

```
// 相对于 Lab3，多了对分支指令和访存指令的处理
module DECODER (
    input                   [31 : 0]            inst,

    output reg              [ 4 : 0]            alu_op,
    output reg              [31 : 0]            imm,

    output reg              [ 4 : 0]            rf_ra0,
    output reg              [ 4 : 0]            rf_ra1,
    output reg              [ 4 : 0]            rf_wa,
    output reg              [ 0 : 0]            rf_we,

    output reg              [ 0 : 0]            alu_src0_sel,
    output reg              [ 0 : 0]            alu_src1_sel,

    output reg              [ 3 : 0]            dmem_access,    // 访存类型
    output reg              [ 0 : 0]            dmem_we,        // 访存写使能
    output reg              [ 1 : 0]            rf_wd_sel,      // rgf 写入数据选择
    output reg              [ 3 : 0]            br_type         // 分支类型
);

    // 指令类型
    `define R_OP            7'b0110011      // R 型运算指令
    `define I_OP            7'b0010011      // I 型运算指令
    `define LUI             7'b0110111      // U 型 LUI 指令
    `define AUIPC           7'b0010111      // U 型 AUIPC 指令
    `define LOAD            7'b0000011      // I 型加载指令
    `define STORE           7'b0100011      // S 型存储指令
    `define BRANCH          7'b1100011      // B 型分支指令
    `define JALR            7'b1100111      // I 型 JALR 指令
    `define JAL             7'b1101111      // J 型 JAL 指令

    // ALU 操作码
    `define ADD                 5'B00000
    `define SUB                 5'B00010
    `define SLT                 5'B00100
    `define SLTU                5'B00101
    `define AND                 5'B01001
    `define OR                  5'B01010
    `define XOR                 5'B01011
    `define SLL                 5'B01110
    `define SRL                 5'B01111
    `define SRA                 5'B10000
    `define SRC0                5'B10001
```

```
    `define SRC1                5'B10010

    // 分支指令类型
    `define BR_NJP              4'b0000
    `define BR_JALR             4'b0001
    `define BR_JAL              4'b0010
    `define BR_BEQ              4'b0011
    `define BR_BNE              4'b0100
    `define BR_BLT              4'b0101
    `define BR_BGE              4'b0110
    `define BR_BLTU             4'b0111
    `define BR_BGEU             4'b1000

    // 访存类型
    `define LB                  4'b0000     // Load Byte
    `define LH                  4'b0001     // Load Halfword
    `define LW                  4'b0010     // Load Word
    `define LBU                 4'b0011     // Load Byte Unsigned
    `define LHU                 4'b0100     // Load Halfword Unsigned
    `define SB                  4'b0101     // Store Byte
    `define SH                  4'b0110     // Store Halfword
    `define SW                  4'b0111     // Store Word

    // RISC-V instruction fields
    wire [6 : 0] opcode = inst[6 : 0];       // Opcode
    wire [4 : 0] rd     = inst[11 : 7];      // Destination register
    wire [2 : 0] funct3 = inst[14 : 12];     // Function 3
    wire [4 : 0] rs1    = inst[19 : 15];     // Source register 1
    wire [4 : 0] rs2    = inst[24 : 20];     // Source register 2
    wire [6 : 0] funct7 = inst[31 : 25];     // Function 7

    // Immediate extraction
    wire [31 : 0] imm_i = {{20{inst[31]}}, inst[31:20]};
    wire [31 : 0] imm_u = {inst[31:12], 12'b0};
    wire [31 : 0] imm_shamt = {26'b0, inst[25:20]};
    wire [31 : 0] imm_b = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8],
1'b0};
    wire [31 : 0] imm_j = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21],
1'b0};
    wire [31 : 0] imm_s = {{20{inst[31]}}, inst[31:25], inst[11:7]};

    always @(*) begin
        // 先对所有变量赋初值，避免锁存器
        alu_op       = 5'b0;
        imm          = 32'b0;
        rf_ra0       = 5'b0;
        rf_ra1       = 5'b0;
        rf_wa        = 5'b0;
        rf_we        = 1'b0;
        alu_src0_sel = 1'b0;
        alu_src1_sel = 1'b0;
        dmem_access  = 4'b0;
        dmem_we      = 1'b0;
        rf_wd_sel    = 2'b0;
```

```
        br_type      = 4'b0;

        case (opcode)
            `R_OP: begin
                rf_ra0       = rs1;
                rf_ra1       = rs2;
                rf_wa        = rd;
                rf_we        = 1'b1;
                alu_src0_sel = 1'b0; // Use rs1
                alu_src1_sel = 1'b0; // Use rs2
                rf_wd_sel    = 2'b01;
                case ({funct7, funct3})
                    10'b0000000000: alu_op = `ADD;  // ADD
                    10'b0100000000: alu_op = `SUB;  // SUB
                    10'b0000000010: alu_op = `SLT;  // SLT
                    10'b0000000011: alu_op = `SLTU; // SLTU
                    10'b0000000111: alu_op = `AND;  // AND
                    10'b0000000110: alu_op = `OR;   // OR
                    10'b0000000100: alu_op = `XOR;  // XOR
                    10'b0000000001: alu_op = `SLL;  // SLL
                    10'b0000000101: alu_op = `SRL;  // SRL
                    10'b0100000101: alu_op = `SRA;  // SRA
                    default:        alu_op = 5'b11111; // Undefined
                endcase
            end

            `I_OP: begin
                rf_ra0       = rs1;
                rf_wa        = rd;
                rf_we        = 1'b1;
                alu_src0_sel = 1'b0; // Use rs1
                alu_src1_sel = 1'b1; // Use immediate
                imm          = imm_i;
                rf_wd_sel    = 2'b01;
                case (funct3)
                    3'b000: alu_op = `ADD;   // ADDI
                    3'b010: alu_op = `SLT;   // SLTI
                    3'b011: alu_op = `SLTU;  // SLTIU
                    3'b111: alu_op = `AND;   // ANDI
                    3'b110: alu_op = `OR;    // ORI
                    3'b100: alu_op = `XOR;   // XORI
                    3'b001: begin            // SLLI
                        if (imm_shamt[5] == 1'b0) begin
                            alu_op = `SLL;
                            imm    = imm_shamt;
                        end else begin
                            alu_op = 5'b11111;
                        end
                    end
                    3'b101: begin
                        if (imm_shamt[5] == 1'b0) begin
                            imm = imm_shamt;
                            if (funct7 == 7'b0000000)
                                alu_op = `SRL;   // SRLI
```

```
                    else if (funct7 == 7'b0100000)
                        alu_op = `SRA;   // SRAI
                    else
                        alu_op = 5'b11111;
                end else begin
                    alu_op = 5'b11111;
                end
            end

            default: alu_op = 5'b11111;
        endcase
    end

    `LUI: begin
        rf_wa        = rd;
        rf_we        = 1'b1;
        alu_src0_sel = 1'b1; // Use PC
        alu_src1_sel = 1'b1; // Use immediate
        imm          = imm_u;
        alu_op       = `SRC1;
        rf_wd_sel    = 2'b01;
    end

    `AUIPC: begin
        rf_wa        = rd;
        rf_we        = 1'b1;
        alu_src0_sel = 1'b1; // Use PC
        alu_src1_sel = 1'b1; // Use immediate
        imm          = imm_u;
        alu_op       = `ADD;
        rf_wd_sel    = 2'b01;
    end

    `LOAD: begin
        rf_ra0       = rs1;
        rf_wa        = rd;
        rf_we        = 1'b1;
        alu_src0_sel = 1'b0; // Use rs1
        alu_src1_sel = 1'b1; // Use immediate
        imm          = imm_i;
        alu_op       = `ADD;
        rf_wd_sel    = 2'b10;
        case (funct3)
            3'b000: dmem_access = `LB;
            3'b001: dmem_access = `LH;
            3'b010: dmem_access = `LW;
            3'b100: dmem_access = `LBU;
            3'b101: dmem_access = `LHU;
            default: dmem_access = 4'b1111;
        endcase
    end

    `STORE: begin
        rf_ra0       = rs1;
```

```
                rf_ra1       = rs2;
                alu_src0_sel = 1'b0; // Use rs1
                alu_src1_sel = 1'b1; // Use immediate
                imm          = imm_s;
                alu_op       = `ADD;
                dmem_we      = 1'b1;
                case (funct3)
                    3'b000: dmem_access = `SB;
                    3'b001: dmem_access = `SH;
                    3'b010: dmem_access = `SW;
                    default: dmem_access = 4'b1111;
                endcase
            end

            `BRANCH: begin
                rf_ra0       = rs1;
                rf_ra1       = rs2;
                alu_src0_sel = 1'b1; // Use PC
                alu_src1_sel = 1'b1; // Use immediate
                imm          = imm_b;
                alu_op       = `ADD;
                case (funct3)
                    3'b000: br_type = `BR_BEQ;
                    3'b001: br_type = `BR_BNE;
                    3'b100: br_type = `BR_BLT;
                    3'b101: br_type = `BR_BGE;
                    3'b110: br_type = `BR_BLTU;
                    3'b111: br_type = `BR_BGEU;
                    default: br_type = 4'b1111;
                endcase
            end

            `JALR: begin
                rf_ra0       = rs1;
                rf_wa        = rd;
                rf_we        = 1'b1;
                alu_src0_sel = 1'b0; // Use rs1
                alu_src1_sel = 1'b1; // Use immediate
                imm          = imm_i;
                alu_op       = `ADD;
                br_type      = `BR_JALR;
                rf_wd_sel    = 2'b00;
            end

            `JAL: begin
                rf_wa        = rd;
                rf_we        = 1'b1;
                alu_src0_sel = 1'b1; // Use PC
                alu_src1_sel = 1'b1; // Use immediate
                imm          = imm_j;
                alu_op       = `ADD;
                br_type      = `BR_JAL;
                rf_wd_sel    = 2'b00;
            end
```

```
            default: begin
                alu_op      = 5'b11111;
            end
        endcase
    end
endmodule
```

CPU.v

```
module CPU (
    input                   [ 0 : 0]            clk,
    input                   [ 0 : 0]            rst,

    input                   [ 0 : 0]            global_en,

/* ----------------------------- Memory (inst) ----------------------------- */
    output                  [31 : 0]            imem_raddr,
    input                   [31 : 0]            imem_rdata,

/* ----------------------------- Memory (data) ----------------------------- */
    input                   [31 : 0]            dmem_rdata,
    output                  [ 0 : 0]            dmem_we,
    output                  [31 : 0]            dmem_addr,
    output                  [31 : 0]            dmem_wdata,

/* ---------------------------------- Debug ---------------------------------- */
    output                  [ 0 : 0]            commit,
    output                  [31 : 0]            commit_pc,
    output                  [31 : 0]            commit_inst,
    output                  [ 0 : 0]            commit_halt,
    output                  [ 0 : 0]            commit_reg_we,
    output                  [ 4 : 0]            commit_reg_wa,
    output                  [31 : 0]            commit_reg_wd,
    output                  [ 0 : 0]            commit_dmem_we,
    output                  [31 : 0]            commit_dmem_wa,
    output                  [31 : 0]            commit_dmem_wd,

    input                   [ 4 : 0]            debug_reg_ra,
    output                  [31 : 0]            debug_reg_rd
);

    `define HALT_INST 32'H00100073

    // Commit
    reg  [ 0 : 0]   commit_reg          ;
    reg  [31 : 0]   commit_pc_reg       ;
    reg  [31 : 0]   commit_inst_reg     ;
    reg  [ 0 : 0]   commit_halt_reg     ;
    reg  [ 0 : 0]   commit_reg_we_reg   ;
    reg  [ 4 : 0]   commit_reg_wa_reg   ;
    reg  [31 : 0]   commit_reg_wd_reg   ;
    reg  [ 0 : 0]   commit_dmem_we_reg  ;
```

```
reg  [31 : 0]   commit_dmem_wa_reg  ;
reg  [31 : 0]   commit_dmem_wd_reg  ;

// Commit
always @(posedge clk) begin
    if (rst) begin
        commit_reg          <= 1'B0;
        commit_pc_reg       <= 32'H0;
        commit_inst_reg     <= 32'H0;
        commit_halt_reg     <= 1'B0;
        commit_reg_we_reg   <= 1'B0;
        commit_reg_wa_reg   <= 5'H0;
        commit_reg_wd_reg   <= 32'H0;
        commit_dmem_we_reg  <= 1'B0;
        commit_dmem_wa_reg  <= 32'H0;
        commit_dmem_wd_reg  <= 32'H0;
    end
    else if (global_en) begin
        commit_reg          <= 1'B1;
        commit_pc_reg       <= pc;
        commit_inst_reg     <= inst;
        commit_halt_reg     <= inst == `HALT_INST;
        commit_reg_we_reg   <= rf_we;
        commit_reg_wa_reg   <= rf_wa;
        commit_reg_wd_reg   <= rf_wd;
        commit_dmem_we_reg  <= dmem_we;
        commit_dmem_wa_reg  <= dmem_addr;
        commit_dmem_wd_reg  <= dmem_wdata;
    end
end

assign commit           = commit_reg;
assign commit_pc        = commit_pc_reg;
assign commit_inst      = commit_inst_reg;
assign commit_halt      = commit_halt_reg;
assign commit_reg_we    = commit_reg_we_reg;
assign commit_reg_wa    = commit_reg_wa_reg;
assign commit_reg_wd    = commit_reg_wd_reg;
assign commit_dmem_we   = commit_dmem_we_reg;
assign commit_dmem_wa   = commit_dmem_wa_reg;
assign commit_dmem_wd   = commit_dmem_wd_reg;


wire [31 : 0] pc;
wire [31 : 0] pc_add4;
wire [31 : 0] pc_offset;
wire [31 : 0] pc_jalr;
wire [31 : 0] npc;
wire [ 1 : 0] npc_sel;
wire [ 3 : 0] br_type;

wire [31 : 0] inst;
wire [31 : 0] rf_rd0;
wire [31 : 0] rf_rd1;
```

```
    wire [31 : 0] rf_wd;
    wire [ 1 : 0] rf_wd_sel;
    wire [ 4 : 0] rf_ra0;
    wire [ 4 : 0] rf_ra1;
    wire [ 4 : 0] rf_wa;
    wire [ 0 : 0] rf_we;

    wire [31 : 0] alu_src0;
    wire [31 : 0] alu_src1;
    wire [ 0 : 0] alu_src0_sel;
    wire [ 0 : 0] alu_src1_sel;
    wire [ 4 : 0] alu_op;
    wire [31 : 0] alu_res;
    wire [31 : 0] imm;

    wire [ 3 : 0] dmem_access;
    wire [31 : 0] dmem_rd_out;
    wire [31 : 0] dmem_rd_in;
    wire [31 : 0] dmem_wd_in;
    wire [31 : 0] dmem_wd_out;


    PC my_pc (
        .clk    (clk),
        .rst    (rst),
        .en     (global_en),
        .npc    (npc),
        .pc     (pc)
    );

    // pc_add4 = pc + 4
    ADD_4 add_pc (
        .in     (pc),
        .out    (pc_add4)
    );

    // pc_jalr = pc_offset & ~1
    CLR_LSB clr_lsb (
        .in     (pc_offset),
        .out    (pc_jalr)
    );

    // npc 选择器
    MUX_3 #(.WIDTH(32)) pc_mux (
        .src0   (pc_add4),
        .src1   (pc_offset),
        .src2   (pc_jalr),
        .sel    (npc_sel),
        .res    (npc)
    );

    BRANCH branch (
        .br_type    (br_type),
        .br_src0    (rf_rd0),
```

```
        .br_src1    (rf_rd1),
        .npc_sel    (npc_sel)
    );

    assign pc_offset = alu_res;

    DECODER decoder (
        .inst           (inst),
        .alu_op         (alu_op),
        .imm            (imm),
        .rf_ra0         (rf_ra0),
        .rf_ra1         (rf_ra1),
        .rf_wa          (rf_wa),
        .rf_we          (rf_we),
        .alu_src0_sel   (alu_src0_sel),
        .alu_src1_sel   (alu_src1_sel),
        .dmem_access    (dmem_access),
        .dmem_we        (dmem_we),
        .rf_wd_sel      (rf_wd_sel),
        .br_type        (br_type)
    );

    // 寄存器堆写回选择器
    MUX_3 #(.WIDTH(32)) rf_wd_mux (
        .src0       (pc_add4),
        .src1       (alu_res),
        .src2       (dmem_rd_out),
        .sel        (rf_wd_sel),
        .res        (rf_wd)
    );

    REGFILE regfile (
        .clk        (clk),
        .rf_ra0     (rf_ra0),
        .rf_ra1     (rf_ra1),
        .dbg_rf_ra  (debug_reg_ra),
        .rf_wa      (rf_wa),
        .rf_we      (rf_we),
        .rf_wd      (rf_wd),
        .rf_rd0     (rf_rd0),
        .rf_rd1     (rf_rd1),
        .dbg_rf_rd  (debug_reg_rd)
    );

    MUX_2 #(.WIDTH(32)) alu_mux0 (
        .src0       (rf_rd0),
        .src1       (pc),
        .sel        (alu_src0_sel),
        .res        (alu_src0)
    );

    MUX_2 #(.WIDTH(32)) alu_mux1 (
        .src0       (rf_rd1),
        .src1       (imm),
```

```
        .sel        (alu_src1_sel),
        .res        (alu_src1)
    );

    ALU alu (
        .alu_op     (alu_op),
        .alu_src0   (alu_src0),
        .alu_src1   (alu_src1),
        .alu_res    (alu_res)
    );

    SLU slu (
        .addr           (alu_res),
        .dmem_access    (dmem_access),
        .rd_in          (dmem_rd_in),
        .wd_in          (dmem_wd_in),
        .rd_out         (dmem_rd_out),
        .wd_out         (dmem_wd_out)
    );

    assign dmem_wd_in = rf_rd1;

    assign imem_raddr = pc;
    assign inst = imem_rdata;

    assign dmem_addr = alu_res;
    assign dmem_wdata = dmem_wd_out;
    assign dmem_rd_in = dmem_rdata;

endmodule
```

任务 4：思考与总结

1. 加载部分读取出来后需要处理对应的部分 存储部分可以直接通过操作掩码来完成正确的存储

```
module SLU (
    input                   [31:0]              addr,
    input                   [ 3:0]              dmem_access,
    input                   [31:0]              rd_in,      // 从存储器读取的数
据
    input                   [31:0]              wd_in,      // 要写入存储器的数
据
    output reg              [ 3:0]              mask,       // 读写掩码
    output reg              [31:0]              rd_out,     // 读出的数据
    output reg              [31:0]              wd_out      // 写入的数据
);

    `define LB                  4'b0000     // Load Byte
    `define LH                  4'b0001     // Load Halfword
    `define LW                  4'b0010     // Load Word
    `define LBU                 4'b0011     // Load Byte Unsigned
```

```
    `define LHU                 4'b0100     // Load Halfword Unsigned
    `define SB                  4'b0101     // Store Byte
    `define SH                  4'b0110     // Store Halfword
    `define SW                  4'b0111     // Store Word

    always @(*) begin
        // 默认值
        mask = 4'b1111; // 默认掩码为全1
        rd_out = rd_in; // 默认读取数据为输入数据
        wd_out = wd_in; // 默认写入数据为输入数据

        // 生成掩码信号
        case (dmem_access)
            // 写操作
            `SB: begin
                case (addr[1:0])
                    2'b00: mask = 4'b0001;
                    2'b01: mask = 4'b0010;
                    2'b10: mask = 4'b0100;
                    2'b11: mask = 4'b1000;
                endcase
            end
            `SH: begin
                case (addr[1:0])
                    2'b00: mask = 4'b0011;
                    2'b10: mask = 4'b1100;
                    default: mask = 4'b0000; // 对齐错误
                endcase
            end
            `SW: mask = 4'b1111;
            // 读操作
            `LB, `LBU: begin
                case (addr[1:0])
                    2'b00: mask = 4'b0001;
                    2'b01: mask = 4'b0010;
                    2'b10: mask = 4'b0100;
                    2'b11: mask = 4'b1000;
                endcase
            end
            `LH, `LHU: begin
                case (addr[1:0])
                    2'b00: mask = 4'b0011;
                    2'b10: mask = 4'b1100;
                    default: mask = 4'b0000; // 对齐错误
                endcase
            end
            `LW: mask = 4'b1111;
            default: mask = 4'b0000;
        endcase

        // 处理读出数据
        if (dmem_access inside {`LB, `LH, `LW, `LBU, `LHU}) begin
            case (dmem_access)
                `LB: begin
```

```
                    case (addr[1:0])
                        2'b00: rd_out = {{24{rd_in[ 7]}}, rd_in[ 7: 0]};    // 读取低字节
                        2'b01: rd_out = {{24{rd_in[15]}}, rd_in[15: 8]};    // 读取中低字节
                        2'b10: rd_out = {{24{rd_in[23]}}, rd_in[23:16]};    // 读取中高字节
                        2'b11: rd_out = {{24{rd_in[31]}}, rd_in[31:24]};    // 读取高字节
                    endcase
                end
                `LH: begin
                    case (addr[1:0])
                        2'b00: rd_out = {{16{rd_in[15]}}, rd_in[15:0]};     // 读取低半字
                        2'b10: rd_out = {{16{rd_in[31]}}, rd_in[31:16]};    // 读取高半字
                    endcase
                end
                `LW: rd_out = rd_in;                                        // 直接输出
                `LBU: begin
                    case (addr[1:0])
                        2'b00: rd_out = {24'b0, rd_in[7:0]};        // 读取低字节
                        2'b01: rd_out = {24'b0, rd_in[15:8]};       // 读取中低字节
                        2'b10: rd_out = {24'b0, rd_in[23:16]};      // 读取中高字节
                        2'b11: rd_out = {24'b0, rd_in[31:24]};      // 读取高字节
                    endcase
                end
                `LHU: begin
                    case (addr[1:0])
                        2'b00: rd_out = {16'b0, rd_in[15:0]};        // 读取低半字
                        2'b10: rd_out = {16'b0, rd_in[31:16]};       // 读取高半字
                        default: rd_out = 32'b0;
                    endcase
                end
                default: rd_out = 32'b0;
            endcase
        end
    end
endmodule
```

2. 关键路径分析

1. 取指阶段 (Instruction Fetch)

- 从 `PC` 模块输出 `pc` 到 `imem_raddr`，然后通过指令存储器获取 `imem_rdata`。
- 关键路径可能是 `PC` 的更新逻辑（`npc` 的计算）到指令存储器的访问延迟。

2. 指令解码阶段 (Instruction Decode)

- 从 `DECODER` 模块中解析指令字段（如 `alu_op`、`rf_ra0`、`rf_ra1` 等）。

- 关键路径可能是 DECODER 中复杂的指令解析逻辑，尤其是涉及多个立即数的生成和控制信号的设置。

3. 寄存器文件访问 (Register File Access)

- 从 REGFILE 模块读取寄存器数据 rf_rd0 和 rf_rd1。
- 关键路径可能是寄存器文件的访问延迟，尤其是当 rf_ra0 和 rf_ra1 依赖于解码阶段的输出时。

4. ALU 运算阶段 (ALU Execution)

- 从 MUX_2 选择 alu_src0 和 alu_src1，然后通过 ALU 计算结果 alu_res。
- 关键路径可能是 MUX_2 的选择延迟加上 ALU 的运算延迟，尤其是复杂运算（如乘法或移位）的实现。

5. 访存阶段 (Memory Access)

- 从 SLU 模块中处理数据存储器的读写操作。
- 关键路径可能是 SLU 的地址计算和数据处理逻辑，尤其是涉及字节对齐或扩展的操作。

6. 写回阶段 (Write Back)

- 从 MUX_3 选择写回数据 rf_wd，然后通过 REGFILE 写入目标寄存器。
- 关键路径可能是 MUX_3 的选择延迟加上寄存器文件的写入延迟。

综合关键路径

1. PC 更新到指令存储器访问：

- PC → ADD_4 → MUX_3 → imem_raddr → imem_rdata

2. 指令解码到 ALU 运算：

- DECODER → REGFILE → MUX_2 → ALU

3. ALU 运算到数据存储器访问：

- ALU → SLU → dmem_rdata

4. 数据存储器访问到寄存器写回：

- SLU → MUX_3 → REGFILE