Lab 3

PB22030892 刘铠瑜

任务 1：译码器设计

DECODER.v

```
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
);

    // 指令类型
    `define R_TYPE            7'b0110011
    `define I_TYPE            7'b0010011
    `define LUI               7'b0110111
    `define AUIPC             7'b0010111

    // ALU 操作类型
    `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

    // 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]};

    always @(*) begin
        // 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;

        case (opcode)
            `R_TYPE: 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
                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_TYPE: 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;
                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; // Undefined
                        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; // Undefined
                        end else begin
                            alu_op = 5'b11111; // Undefined
                        end
                    end
                    default: alu_op = 5'b11111; // Undefined
                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;
            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;
            end
            default: begin
                alu_op       = 5'b11111;
            end
        endcase
    end

endmodule
```

任务 2：搭建 CPU

PC.v

```
module PC (
    input                   [ 0 : 0]            clk,
    input                   [ 0 : 0]            rst,
    input                   [ 0 : 0]            en,
    input                   [31 : 0]            npc,

    output      reg         [31 : 0]            pc
);

    always @(posedge clk) begin
        if (rst) begin
            pc <= 32'H0040_0000;    // 指令段的起始地址
        end else if (en) begin
            pc <= npc;
        end
    end

endmodule
```

REGFILE.v

```
// 在之前实验的基础上添加了 debug 信号
module REGFILE (
    input                   [ 0 : 0]        clk,

    input                   [ 4 : 0]        rf_ra0,     // 读寄存器地址
    input                   [ 4 : 0]        rf_ra1,
    input                   [ 4 : 0]        dbg_rf_ra,  // debug读寄存器地址
    input                   [ 4 : 0]        rf_wa,      // 写寄存器地址
    input                   [ 0 : 0]        rf_we,      // 写使能信号
    input                   [31 : 0]        rf_wd,      // 写数据

    output                  [31 : 0]        rf_rd0,     // 读数据
    output                  [31 : 0]        rf_rd1,
    output                  [31 : 0]        dbg_rf_rd   // debug读数据
);

    reg [31 : 0] reg_file [0 : 31];

    // 用于初始化寄存器
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            reg_file[i] = 0;
    end

    always @(posedge clk) begin
        if (rf_we && (rf_wa != 0)) begin
            reg_file[rf_wa] <= rf_wd;
        end
    end
```

```
    assign rf_rd0 = reg_file[rf_ra0];
    assign rf_rd1 = reg_file[rf_ra1];
    assign dbg_rf_rd = reg_file[dbg_rf_ra];

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,   // TODO
    output                  [31 : 0]            debug_reg_rd    // TODO
);

    `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  <= 0;
        commit_dmem_wa_reg  <= 0;
        commit_dmem_wd_reg  <= 0;
    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] npc;
wire [31 : 0] inst;
wire [31 : 0] rf_rd0;
wire [31 : 0] rf_rd1;
wire [31 : 0] rf_wd;
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;


    PC my_pc (
        .clk    (clk),
        .rst    (rst),
        .en     (global_en),    // 当 global_en 为高电平时，PC 才会更新，CPU 才会执行
指令。
        .npc    (npc),
        .pc     (pc)
    );

    // npc = pc + 4
    ADD4 add4 (
        .in     (pc),
        .out    (npc)
    );

    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)
    );

    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)
    );

    // alu_src0 选择 rf_rd0 或 pc
    MUX #(.WIDTH(32)) mux0 (
        .src0       (rf_rd0),
        .src1       (pc),
        .sel        (alu_src0_sel),
        .res        (alu_src0)
```

```
    );

    // alu_src1 选择 rf_rd1 或 imm
    MUX #(.WIDTH(32)) 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)
    );

    assign rf_wd = alu_res;     // 寄存器的写入数据 rf_wd 就是 alu_res
    assign imem_raddr = pc;
    assign inst = imem_rdata;

endmodule
```

任务 3：思考与总结

1. CPU, PC, REGFILE 这三个模块
2. alu_src0 选择 pc: lui rd, imm alu_src0 选择 rf_rd0: add rd, rs1, rs2 alu_src1 选择 rf_rd1: add rd, rs1, rs2 alu_src1 选择 imm: addi rd, rs1, immediate
3. 关键路径分析
    - 取指阶段 (Instruction Fetch)：
        - 从 PC 模块输出 pc，通过 imem_raddr 访问指令存储器，获取指令 imem_rdata。
        - 关键路径：PC 模块的输出到指令存储器的读取延迟。
    - 指令解码阶段 (Instruction Decode)：
        - 从 DECODER 模块中解析指令 inst，生成控制信号（如 alu_op、rf_ra0、rf_ra1 等）。
        - 关键路径：inst 到 DECODER 模块的输出信号生成延迟。
    - 寄存器文件访问 (Register File Access)：
        - 从 REGFILE 模块中读取寄存器值 rf_rd0 和 rf_rd1。
        - 关键路径：rf_ra0 和 rf_ra1 到寄存器文件输出 rf_rd0 和 rf_rd1 的延迟。
    - 执行阶段 (Execution)：
        - ALU 操作：alu_src0 和 alu_src1 通过 MUX 选择后，送入 ALU，计算结果 alu_res。
        - 关键路径：MUX 的选择信号生成延迟 + ALU 的计算延迟。
    - 写回阶段 (Write Back)：

        - 将 alu_res 写回寄存器文件。
        - 关键路径：alu_res 到寄存器文件写入的延迟。
    - 可能的关键路径 综合来看，从 PC 输出到 ALU 计算完成可能是整个 CPU 的关键路径，具体包括：
        - PC 输出到指令存储器读取指令。
        - 指令存储器输出到 DECODER 模块解析。
        - DECODER 输出控制信号到寄存器文件读取。
        - 寄存器文件输出到 MUX 和 ALU。
        - ALU 完成计算。

如果关键路径的延迟大于一个时钟周期，可能会带来以下影响：

- 时钟频率受限： 时钟周期必须大于关键路径的延迟，导致 CPU 的时钟频率降低，整体性能下降。
- 数据冒险 (Data Hazard)：

  如果指令依赖于前一条指令的结果，而结果尚未写回寄存器文件，可能导致数据冒险问题。
- 流水线设计困难：

  如果延迟过长，可能需要引入流水线寄存器，将关键路径分割成多个阶段，但这会增加设计复杂性。
- 功能错误：

  如果时钟周期不足以覆盖关键路径的延迟，可能导致信号未稳定时被采样，进而引发功能错误。