lab4 report

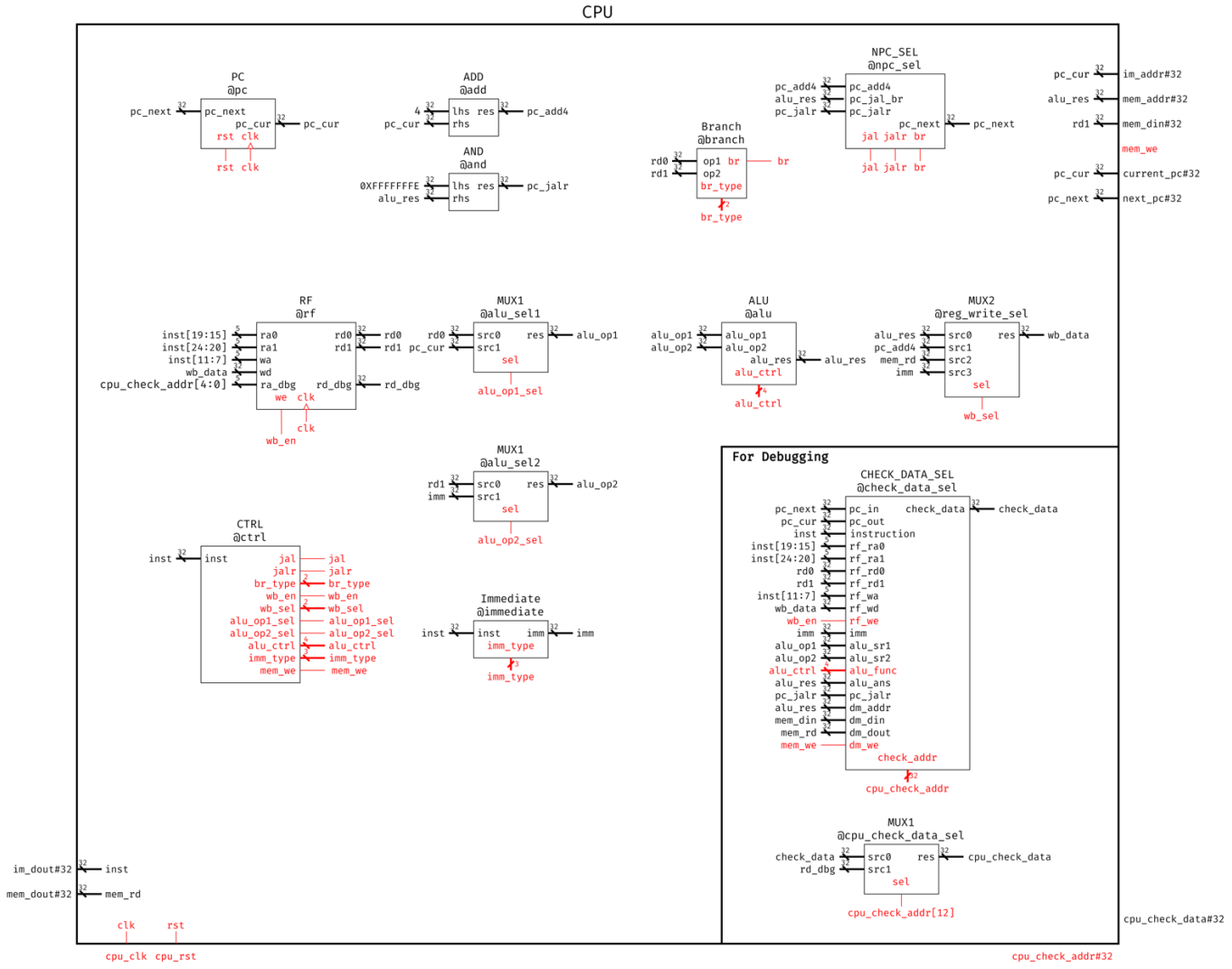
实验内容

- 设计单周期 CPU
- 实现 `add、addi、lui、auipc、beq、blt、jal、jalr、lw、sw` 十条基本指令
- 实现 `sll, srl, sra`, `sub, and, or`, `bne, bge, bltu` 九条指令（**选做**）
- 通过仿真 debug，上板测试

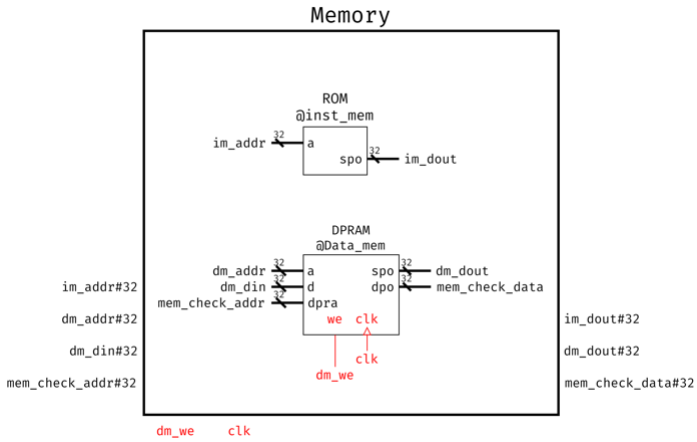
实验原理

数据通路

CPU



MEM



源代码

Control

> 控制模块，CPU 最核心的模块

```
module Control(
    input [31:0] inst, // 指令
    output jal, jalr, // jal or jalr
    output [2:0] br_type, // 跳转类型
    output reg wb_en, // 寄存器写回使能
    output reg [1:0] wb_sel, // 寄存器写回选择
    output reg mem_we, // 内存写使能
    output reg alu_op1_sel, alu_op2_sel, // alu 操作数选择
    output reg [3:0] alu_ctrl, // alu 功能选择
    output reg [2:0] imm_type // 立即数类型
);
assign jal = (inst[6:0] == 7'b1101111);
assign jalr = (inst[6:0] == 7'b1100111);
// 跳转类型，010 代表不跳转
assign br_type = (inst[6:0] == 7'b1100011) ? inst[14:12] : 3'b010;
always @(*) begin
    case (inst[6:0])
        // addi
        7'b0010011: begin
            wb_en = 1'b1;
            // 0:alu结果，1:pc自增，2:内存，3:立即数
            wb_sel = 2'b00;
            mem_we = 1'b0;
```

```
            alu_op1_sel = 1'b0; // 0:寄存器, 1:PC
            alu_op2_sel = 1'b1; // 0:寄存器, 1:立即数
            // 1111 代表不使用 ALU
            alu_ctrl = (inst[14:12] == 3'b000 ? 4'b0000 : 4'b1111);
            imm_type = 3'b000; // 立即数类型
        end
        // add, and, or, sub, sll, srl, sra
        7'b0110011: begin
            wb_en = 1'b1;
            wb_sel = 2'b00;
            mem_we = 1'b0;
            alu_op1_sel = 1'b0;
            alu_op2_sel = 1'b0;
            case (inst[14:12])
                3'b000: alu_ctrl = (inst[30] ? 4'b0001 : 4'b0000); // add, sub
                3'b110: alu_ctrl = 4'b0110; // or
                3'b111: alu_ctrl = 4'b0101; // and
                3'b001: alu_ctrl = 4'b1001; // sll
                3'b101: alu_ctrl = (inst[30] ? 4'b1010 : 4'b1000); // sra, srl
                default: alu_ctrl = 4'b1111;
            endcase
            imm_type = 3'b111; // 111 代表不需要立即数
        end
        // 此处省略。。。
        default: begin
            wb_en = 1'b0;
            wb_sel = 2'b00;
            mem_we = 1'b0;
            alu_op1_sel = 1'b0;
            alu_op2_sel = 1'b0;
            alu_ctrl = 4'b1111;
            imm_type = 3'b111;
        end
    endcase
end
endmodule
```

`jal` 和 `jalr` 可以直接判断，`br_type` 可以直接赋值，其他的根据指令条件赋值。

例如：`addi` 指令需要写回寄存器，写回来源选择 alu 输出，alu 操作数分别为寄存器和立即数，不需要写内存，alu 功能选择加法，立即数类型为 imm[11:0] （000）

Imm

> “

> 立即数生成模块，根据指令生成立即数

```
module IMM(
    input [31:0] inst, // 指令
    input [2:0] imm_type, // 立即数类型
    output reg [31:0] imm // 生成结果
);
// 根据立即数类型生成立即数
always @(*) begin
    case (imm_type)
        3'b000: imm = { {20{inst[31]}}, inst[31:20] }; // addi, jalr, lw
        3'b001: imm = { inst[31:12], 12'b0 }; // lui, auipc
        3'b010: imm = { {11{inst[31]}}, inst[31], inst[19:12], inst[20], inst[30:21], 1'b0 }; // jal
        3'b011: imm = { {19{inst[31]}}, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0 }; // beq, blt
        3'b100: imm = { {20{inst[31]}}, inst[31:25], inst[11:7] }; // sw
        default: imm = 32'b0;
    endcase
end
endmodule
```

在要实现的指令体系下只有如上 5 种立即数类型，根据指令格式分类生成即可。

Branch

> 分支模块，单独处理分支指令

```
module Branch(
    input [31:0] op1, // 操作数 1（寄存器）
    input [31:0] op2, // 操作数 2（寄存器）
    input [2:0] br_type, // 跳转类型
    output reg br // 跳转使能
);
always @(*) begin
    case (br_type)
        3'b000: br = (op1 == op2); // beq
        3'b001: br = (op1 != op2); // bne
        3'b100: br = (op1[31] == op2[31] ? op1 < op2 : op1[31]); // blt
        3'b101: br = (op1[31] == op2[31] ? op1 >= op2 : ~op1[31]); // bge
        3'b110: br = (op1 < op2); // bltu
        default: br = 0;
    endcase
end
```

```
endmodule
```

在要实现的指令体系下只有如上 5 种分支类型，根据指令分类处理。

ALU

> alu 模块，复用 lab1

略

RF

> 寄存器堆，复用 lab2

略

CPU

> CPU 顶层模块，按数据通路接线即可

```
module CPU(
    input clk,
    input rst,
    // MEM And MMIO Data BUS
    output [31:0] im_addr,
    input [31:0] im_dout,
    output [31:0] mem_addr,
    output  mem_we,
    output [31:0] mem_din,
    input [31:0] mem_dout,
    // Debug BUS with PDU
    output [31:0] current_pc,// current_pc
    output [31:0] next_pc,
    input [31:0] cpu_check_addr,// Check current datapath state (code)
    output [31:0] cpu_check_data// Current datapath state data
);
wire [31:0] pc_cur;
wire [31:0] pc_next;
wire [31:0] inst;
assign inst = im_dout;
wire [31:0] mem_rd;
```

```
assign mem_rd = mem_dout;
PC pc(
    .clk(clk),
    .rst(rst),
    .pc_next(pc_next),
    .pc_cur(pc_cur)
);
wire [31:0] pc_add4;
ADD add0(
    .a(pc_cur),
    .b(32'h4),
    .sum(pc_add4)
);
wire [31:0] pc_jalr;
wire [31:0] alu_res;
AND and0(
    .a(alu_res),
    .b(32'hfffffffe),
    .out(pc_jalr)
);
wire [31:0] rd0;
wire [31:0] rd1;
wire [2:0] br_type;
wire br;
Branch branch0(
    .op1(rd0),
    .op2(rd1),
    .br_type(br_type),
    .br(br)
);
wire jal, jalr;
PC_MUX pc_mux0(
    .pc_add4(pc_add4),
    .alu_res(alu_res),
    .pc_jalr(pc_jalr),
    .jal(jal),
    .jalr(jalr),
    .br(br),
    .pc_next(pc_next)
);
wire [31:0] wb_data;
wire wb_en;
wire [31:0] rd_dbg;
RF rf(
    .clk(clk),
```

```
    .we(wb_en),
    .wa(inst[11:7]),
    .wd(wb_data),
    .ra0(inst[19:15]),
    .ra1(inst[24:20]),
    .rd0(rd0),
    .rd1(rd1),
    .ra_dbg(cpu_check_addr[4:0]),
    .rd_dbg(rd_dbg)
);
wire alu_op1_sel;
wire [31:0] alu_op1;
MUX1 alu_sel1(
    .sr0(rd0),
    .sr1(pc_cur),
    .alu_sel(alu_op1_sel),
    .sr_out(alu_op1)
);
wire [31:0] imm;
wire alu_op2_sel;
wire [31:0] alu_op2;
MUX1 alu_sel2(
    .sr0(rd1),
    .sr1(imm),
    .alu_sel(alu_op2_sel),
    .sr_out(alu_op2)
);
wire [3:0] alu_ctrl;
alu #(.WIDTH(32)) alu0(
    .a(alu_op1),
    .b(alu_op2),
    .func(alu_ctrl),
    .y(alu_res)
);
wire [1:0] wb_sel;
MUX2 wb_mux(
    .sr0(alu_res),
    .sr1(pc_add4),
    .sr2(mem_rd),
    .sr3(imm),
    .wb_sel(wb_sel),
    .sr_out(wb_data)
);
wire [2:0] imm_type;
Control control(
```

```
    .inst(inst),
    .jal(jal),
    .jalr(jalr),
    .br_type(br_type),
    .wb_en(wb_en),
    .wb_sel(wb_sel),
    .mem_we(mem_we),
    .alu_op1_sel(alu_op1_sel),
    .alu_op2_sel(alu_op2_sel),
    .alu_ctrl(alu_ctrl),
    .imm_type(imm_type)
);
IMM imm0(
    .inst(inst),
    .imm_type(imm_type),
    .imm(imm)
);
wire [31:0] check_data;
CHECK check(
    .pc_in(pc_next),
    .pc_out(pc_cur),
    .instruction(inst),
    .rf_ra0(inst[19:15]),
    .rf_ra1(inst[24:20]),
    .rf_rd0(rd0),
    .rf_rd1(rd1),
    .rf_wa(inst[11:7]),
    .rf_wd(wb_data),
    .rf_we(wb_en),
    .imm(imm),
    .alu_sr1(alu_op1),
    .alu_sr2(alu_op2),
    .alu_func(alu_ctrl),
    .alu_ans(alu_res),
    .pc_jalr(pc_jalr),
    .dm_addr(alu_res),
    .dm_din(mem_din),
    .dm_dout(mem_rd),
    .dm_we(mem_we),
    .check_addr(cpu_check_addr),
    .check_data(check_data)
);
MUX1 check_mux(
    .sr0(check_data),
    .sr1(rd_dbg),
```

```
157     .alu_sel(cpu_check_addr[12]),
158     .sr_out(cpu_check_data)
159 );
160 assign current_pc = pc_cur;
161 assign next_pc = pc_next;
162 assign im_addr = pc_cur;
163 assign mem_addr = alu_res;
164 assign mem_din = rd1;
165 endmodule
```

实验分析

数据通路的差异

教材上的数据通路：

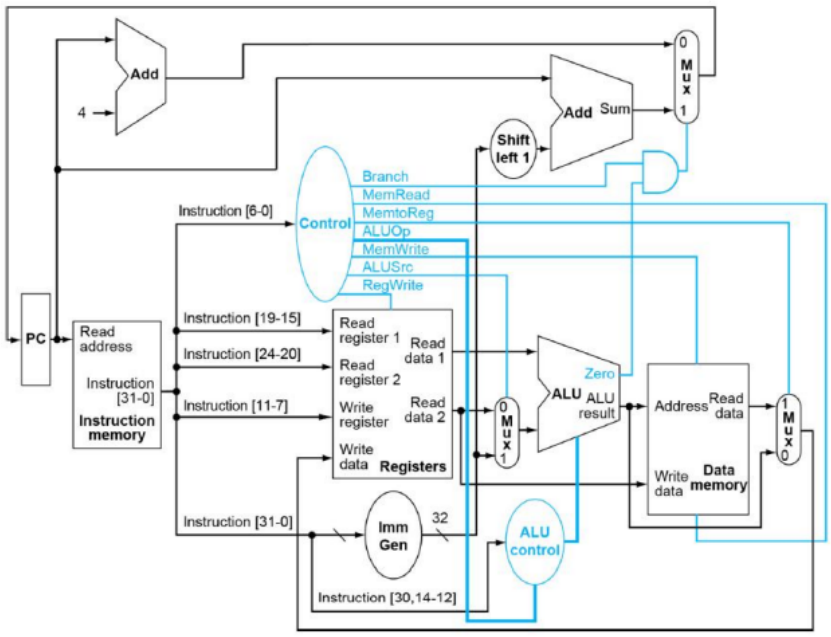


图2 单周期数据通路

本实验数据通路与上图的主要差异：

- 将内存模块独立于 CPU 之外，几乎无影响
- 将分支判断独立于 alu 之外，影响 beq，bne 等指令（条件判断不再需要 alu）
- Control 模块接收整个指令，而不只是 opcode，影响所有指令，能控制的东西更多了
- 寄存器堆写回选择器变为四选一，影响要写寄存器的 addi，add，lui 等指令（例如，lui 指令不再需要 alu）
- pc 写入选择器变为三选一，影响 jal 和 jalr，(jalr 写回数据需要额外把第一位清零)
- .....

实验总结

错误总结

- 仿真时间不足，程序跑不完整(设置里改)
- 看漏了一句话：MEM 需要对传入的地址除以 4，也就是右移两位之后再接入存储器。(没认真看手册)
- Control 模块刚开始是按输出分类，容易遗漏和出错，且不直观(码风问题)
- 编写的测试代码出错：在 test.asm 里添加了选做指令的测试，没注意后面有 auipc 指令(好坑，感受到了代码协作的难)

体验

助教的实验手册 YYDS！