Lab 4

PB21081601 张芷苒

实验目的

- 理解单周期CPU的结构和工作原理
- 熟练掌握单周期CPU数据通路和控制器的设计和描述方法
- 理解单周期CPU的调试方法

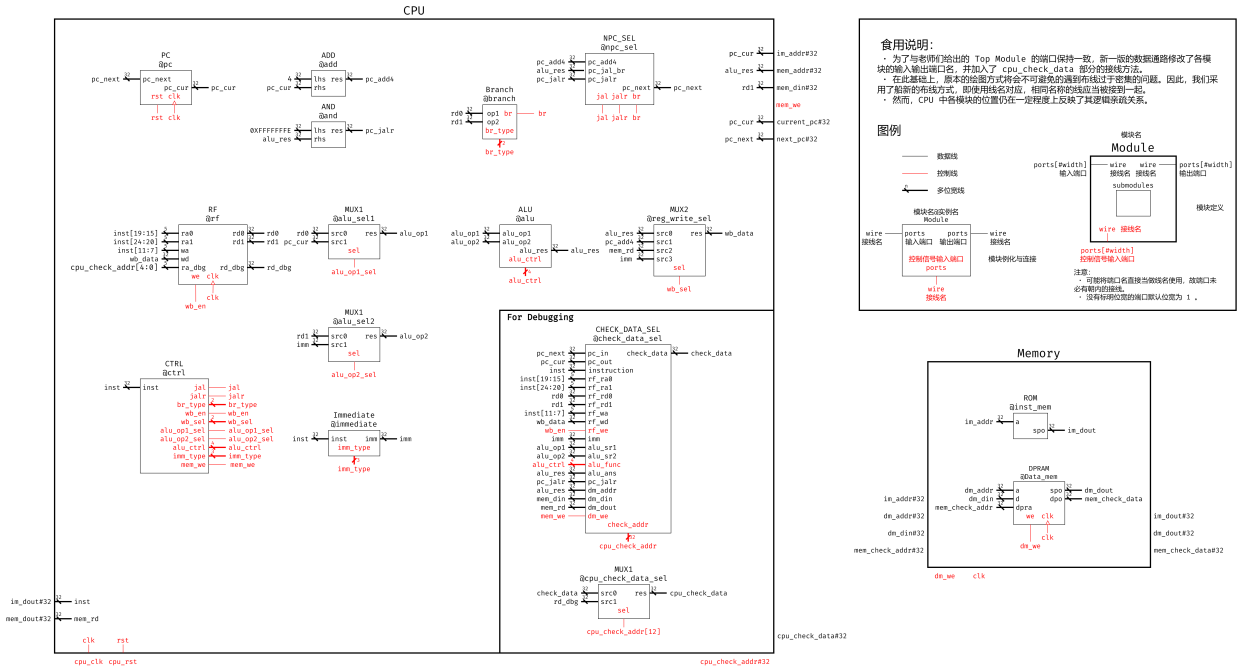
实验要求

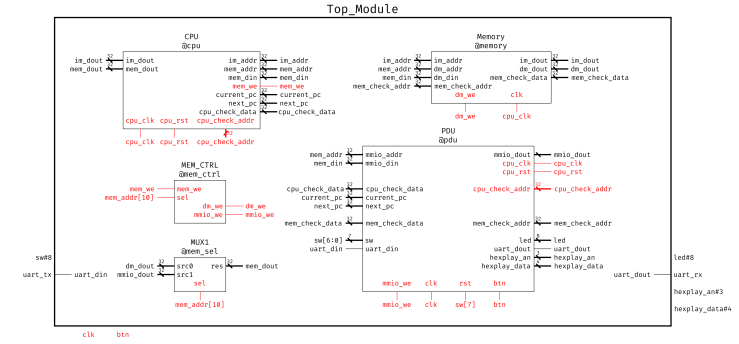
- 设计单周期CPU数据通路并进行功能仿真
- 测试Lab3生成的 FLS.coe
- 测试 test.coe

实验内容

数据通路

本次实验的数据通路:





重要模块设计

PC 模块

该模块实现了一个简单的程序计数器，用于存储和更新指令地址。

在 `always @(posedge clk)` 块中，使用时钟的上升沿作为触发器，在时钟上升沿时执行以下操作：

1. 如果 `rst` 为高电平，表示复位信号激活，模块正在进行复位操作。在这种情况下，将当前指令地址 `pc_cur` 设置为 32'h2ffc，即一个预定义的复位地址。
2. 如果 `rst` 为低电平，表示模块处于正常操作模式。在这种情况下，将当前指令地址 `pc_cur` 更新为下一指令地址 `pc_next`。这意味着在时钟上升沿时，当前指令地址会根据输入的下一指令地址进行更新。

```
module PC(
  input [31:0] pc_next, // 输入下一指令地址
  input clk, rst, // 时钟和复位信号
  output reg [31:0] pc_cur // 当前指令地址
);
  always @(posedge clk) begin
    if (rst) // 异步复位
      pc_cur <= 32'h2ffc; // 复位时将当前指令地址设置为32'h2ffc
    else
      pc_cur <= pc_next; // 在时钟上升沿时更新当前指令地址为下一指令地址
  end

  // initial begin
  //     pc_cur <= 32'h2ffc;
  // end
endmodule
```

PC 选择模块

该模块根据不同的控制信号选择下一个 PC（程序计数器） 地址。

在 `always @(*)` 块中，使用组合逻辑根据输入的控制信号选择下一个 PC 地址。

```
module npc_sel(
  input [31:0] pc_add4, pc_jal_br, pc_jalr, // 输入PC的增加4、JAL/BR跳转的PC、JALR跳转的
PC
  input jal, jalr, br, // 输入JAL、JALR、BR控制信号
  output reg [31:0] pc_next // 输出下一PC地址
);
  always @(*) begin
    if (jal || br) // 如果JAL或BR控制信号为高电平，则选择pc_jal_br作为下一PC地址
      pc_next = pc_jal_br;
    else if (jalr) // 如果JALR控制信号为高电平，则选择pc_jalr作为下一PC地址
      pc_next = pc_jalr;
    else // 否则选择pc_add4作为下一PC地址
      pc_next = pc_add4;
  end
```

```
endmodule
```

立即数模块

该模块根据指令的类型和内容提取立即数（Immediate）。

```
module Immediate(
    input [31:0] inst,
    input [2:0] imm_type,
    output reg [31:0] imm
);
    always @(*) begin
        case (imm_type)
            3'b001: // I-type
                if (inst[31])
                    imm = {20'hFFFFF, inst[31:20]}; // ˇűşĹŔŠŐš
                else
                    imm = {20'b0, inst[31:20]};
            3'b010: // S-type
                if (inst[31])
                    imm = {20'hFFFFF, inst[31:25], inst[11:7]};
                else
                    imm = {20'b0, inst[31:25], inst[11:7]};
            3'b011: // B-type
                if (inst[31])
                    imm = {3'b111, 16'hFFFF, inst[31], inst[7], inst[30:25],
inst[11:8], 1'b0};
                else
                    imm = {19'b0, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
            3'b100: // U-type, xóŇĆ12Îť
                imm = {inst[31:12], 12'b0};
            3'b101: // J-type
                if (inst[31])
                    imm = {11'h7FF, inst[31], inst[19:12], inst[20], inst[30:21],
1'b0};
                else
                    imm = {11'b0, inst[31], inst[19:12], inst[20], inst[30:21],
1'b0};
            default: imm = 32'b0;
        endcase
    end
endmodule
```

Branch 模块

该模块根据操作数和分支类型进行比较，并确定是否应该进行分支。

根据输入的 `br_type`（分支类型码），使用 `case` 语句来判断分支类型，并根据操作数的比较结果设置输出信号 `br`。

```
module branch(
```

```
    input [31:0] op1, op2,
    input [2:0] br_type,
    output reg br //是否分支
);
    always @(*) begin
        case (br_type)
            3'b000: br = 0; // 不分支
            3'b001: // beq
            begin
                if (op1 == op2) br = 1; // 相等时分支
                else br = 0;
            end
            3'b010: // blt // 有符号数比较
            begin
                if (~(op1[31] ^ op2[31])) // 均为正或均为负
                    br = (op1 < op2 ? 1'b1 : 1'b0); // 和无符号数比较相同
                else if (op1[31] && !op2[31])
                    br = 1'b1;
                else
                    br = 1'b0;
            end
            3'b011: // bne 不等时分支
            begin
                if (op1 != op2) br = 1'b1; // 不相等时分支
                else br = 1'b0;
            end
            3'b100: // bgeu 无符号大于等于时分支
            begin
                if (op1 >= op2) br = 1'b1; // 大于等于时分支
                else br = 1'b0;
            end
            3'b101: // bltu 无符号小于时分支
            begin
                if (op1 < op2) br = 1'b1; // 小于时分支
                else br = 1'b0;
            end
            default: br = 1'b0;
        endcase
    end
endmodule
```

控制模块

该模块根据输入的指令字段值，使用组合逻辑来判断并设置相应的控制信号，以控制处理器的各个功能和操作

```
module CTRL(
    input [31:0] inst,
    output reg jal, jalr,
    output reg [2:0] br_type,
    output reg wb_en, // write_back, RF写使能
```

```
    output reg [1:0] wb_sel,
    output reg [1:0] alu_op1_sel,
    output reg alu_op2_sel,
    output reg [3:0] alu_ctrl,
    output reg [2:0] imm_type,
    output reg mem_we
);

always @(*) begin
    case (inst[6:0])
        7'b0010011: begin
            if (inst[14:12] == 3'b000) // addi
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b1; // imm
                alu_ctrl = 4'b0000; // 加法
                imm_type = 3'b001; // I-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b111) // andi
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b1; // imm
                alu_ctrl = 4'b0101; // 与操作
                imm_type = 3'b001; // I-type
                mem_we = 1'b0; // 不写入存储器
            end
            else // 全0
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b0; // 不写回寄存器
                wb_sel = 2'b00; // 随意
                alu_op1_sel = 2'b00; // rs1
                alu_op2_sel = 1'b0; // rs2
                alu_ctrl = 4'b0000; // 加法
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
        end
        7'b0110011: begin
```

```
            if (inst[14:12] == 3'b111) // and
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b000) // add
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                alu_ctrl = 4'b0000; // 加法
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b110) // or
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                alu_ctrl = 4'b0110; // 或操作
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b001) // sll
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                alu_ctrl = 4'b1001; // 左移操作
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b101 && inst[31:25] == 7'b0000000) // srl
```

```
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                alu_ctrl = 4'b1000; // 逻辑右移操作
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else if (inst[14:12] == 3'b101 && inst[31:25] == 7'b0100000) // sra
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b1; // 写回寄存器
                wb_sel = 2'b0; // alu 输出
                alu_op1_sel = 2'b00; // inst[19:15]
                alu_op2_sel = 1'b0; // inst[24:20]
                alu_ctrl = 4'b1000; // 算术右移操作
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
            else // 全0
            begin
                jal = 1'b0;
                jalr = 1'b0;
                br_type = 3'b00; // 不分支
                wb_en = 1'b0; // 不写回寄存器
                wb_sel = 2'b00; // 随意
                alu_op1_sel = 2'b00; // rs1
                alu_op2_sel = 1'b0; // rs2
                alu_ctrl = 4'b0000; // 加法
                imm_type = 3'b000; // R-type
                mem_we = 1'b0; // 不写入存储器
            end
        end
        7'b0110111: begin // lui
            jal = 1'b0;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b1; // 写回寄存器
            wb_sel = 2'b0; // alu 输出
            alu_op1_sel = 2'b10; // 12
            alu_op2_sel = 1'b1; // imm(已经左移12位)
            alu_ctrl = 4'b0000; // 加法
            imm_type = 3'b100; // U-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b0010111: begin // auipc
```

```
            jal = 1'b0;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b1; // 写回寄存器
            wb_sel = 2'b0; // alu 输出
            alu_op1_sel = 2'b01; // pc
            alu_op2_sel = 1'b1; // imm(已经左移12位)
            alu_ctrl = 4'b0000; // 加法
            imm_type = 3'b100; // U-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b1101111: begin // jal
            jal = 1'b1;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b1; // 写回寄存器
            wb_sel = 2'b01; // pc+4
            alu_op1_sel = 2'b01; // inst[19:12], rs1
            alu_op2_sel = 1'b1; // imm
            alu_ctrl = 4'b0000; // 加法, 结果给 NPC_sel
            imm_type = 3'b101; // J-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b1100111: begin // jalr
            jal = 1'b0;
            jalr = 1'b1;
            br_type = 3'b00; // 不分支
            wb_en = 1'b1; // 写回寄存器
            wb_sel = 2'b01; // pc+4
            alu_op1_sel = 2'b00; // rs1
            alu_op2_sel = 1'b1; // imm
            alu_ctrl = 4'b0000; // 加法, 结果给 AND, 再给 NPC_sel
            imm_type = 3'b001; // I-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b1100011: begin
            if (inst[14:12] == 3'b000) // beq
            br_type = 3'b001; // beq
            else if (inst[14:12] == 3'b100) // blt
            br_type = 3'b010; // blt
            else if (inst[14:12] == 3'b001) // bne
            br_type = 3'b011; // bne
            else if (inst[14:12] == 3'b111) // bgeu
            br_type = 3'b100; // bgeu
            else if (inst[14:12] == 3'b110) // bltu
            br_type = 3'b101; // bltu
            else
            br_type = 3'b000; // 不分支
            jal = 1'b0;
            jalr = 1'b0;
            wb_en = 1'b0; // 不写回寄存器
            wb_sel = 2'b11; // 随意
```

```
            alu_op1_sel = 2'b01; // pc
            alu_op2_sel = 1'b1; // imm
            alu_ctrl = 4'b0000; // 加法
            imm_type = 3'b011; // B-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b0000011: begin // lw
            jal = 1'b0;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b1; // 写回寄存器
            wb_sel = 2'b10; // mem_read_data
            alu_op1_sel = 2'b00; // rs1
            alu_op2_sel = 1'b1; // imm
            alu_ctrl = 4'b0000; // 加法，结果作为地址给 mem_addr
            imm_type = 3'b001; // I-type
            mem_we = 1'b0; // 不写入存储器
        end
        7'b0100011: begin // sw
            jal = 1'b0;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b0; // 不写回寄存器
            wb_sel = 2'b10; // 随意
            alu_op1_sel = 2'b00; // rs1
            alu_op2_sel = 1'b1; // imm
            alu_ctrl = 4'b0000; // 加法，结果作为地址给 mem_addr
            imm_type = 3'b010; // S-type
            mem_we = 1'b1; // 写入存储器
        end
        default: begin // 全0
            jal = 1'b0;
            jalr = 1'b0;
            br_type = 3'b00; // 不分支
            wb_en = 1'b0; // 不写回寄存器
            wb_sel = 2'b00; // 随意
            alu_op1_sel = 2'b00; // rs1
            alu_op2_sel = 1'b0; // rs2
            alu_ctrl = 4'b0000; // 加法
            imm_type = 3'b000; // R-type
            mem_we = 1'b0; // 不写入存储器
        end
    endcase
end
endmodule
```

实验结果

数据通路的比较

通过对比课本上的数据通路与本次实验中的数据通路，发现：

1. CTRL 模块仅由 `Inst[6:0]` 控制，使得ALU控制信号必须由额外的 ALU_Control 模块控制
2. 跳转指令中仅支持 `beq` 指令
3. ALU 模块的操作数种类更少，支持的指令更少
4. 回写的数据类型更少

上版展示

vivado并没有给我报错，所有该写的模块都写了，但是上板结果就是不对...

仿真目前也没看出什么问题，不知道能不能在ddl之前解决，先把实验报告和源码和 bit 文件交了，证明我确实花了很多时间做了这个实验，球球助教手下留情。