COD LAB4 实验报告

姓名：刘芷辰

学号：PB21111728

日期：2023.5.15

1. 实验题目

单周期CPU

2. 实验目标

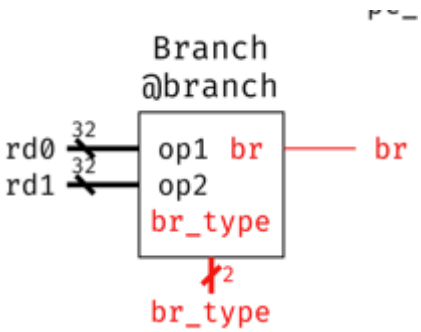
- 理解单周期CPU的结构和工作原理
- 熟练掌握单周期CPU数据通路和控制器的设计和描述方法
- 理解单周期CPU的调试方法

3. 实验内容

3.1 CPU设计与原理

> branch模块

- 框图



- 代码

```
module Branch(
    input [31:0] op1,op2,
```

```
    input [1:0] br_type,
    output reg br
    );
    always @(*) begin
        case (br_type)
            // beq
            2'b01:  br = (op1 == op2) ? 1 : 0;

            //blt
            2'b10:
            if (op1[31] == op2[31]) begin
                if (op1[30:0] < op2[30:0])
                    br = 1;
                else
                    br = 0;
            end
            else if ( op1[31] == 1 ) br = 1;
            else br = 0;

            default: br = 0;
        endcase

    end
endmodule
```

- 设计思路

  brtype 2'b01表示beq，2'b10表示blt

  br=1表示跳转

  br=0表示不需跳转

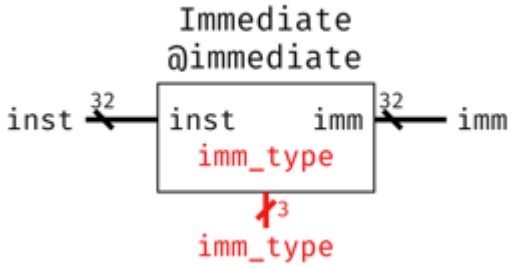
  对于beq，只需要判断op1和op2是否相等，相等则跳转，br=1

  对于blt，首先比较符号位是否相同，相同则比较后31位，op1小于op2则跳转，br=1

  若符号位不相同要满足小于则只需要op1的符号位是1即可，此时需要跳转，br=1，其余情况为0

> imm模块

- 框图



- 代码

```
module Immediate(
    input [31:0] inst,
    output reg [31:0] imm
    );
    always@(*) begin
        case(inst[6:0])
        //I-type
        //addi
        7'b0010011: imm = {{20{inst[31]}},inst[31:20]};
        //lw
        7'b0000011: imm = {{20{inst[31]}},inst[31:20]};
        //jalr
        7'b1100111: imm = {{20{inst[31]}},inst[31:20]};


        //B-type
        //beq/blt
        7'b1100011: imm = {{20{inst[31]}},inst[7],inst[30:25],inst[11:8],1'b0};


        //S-type
        //sw
        7'b0100011: imm = {{20{inst[31]}},inst[31:25],inst[11:7]};


        //U-type
        //lui
        7'b0110111: imm = {inst[31:12],12'b0};
        //auipc
        7'b0010111: imm = {inst[31:12],12'b0};


        //J-type
        //jal
```

```
        7'b1101111: imm = {{12{inst[31]}},inst[19:12],inst[20],inst[30:21],1'b0};

        default: imm = 32'b0;
        endcase
    end
endmodule
```

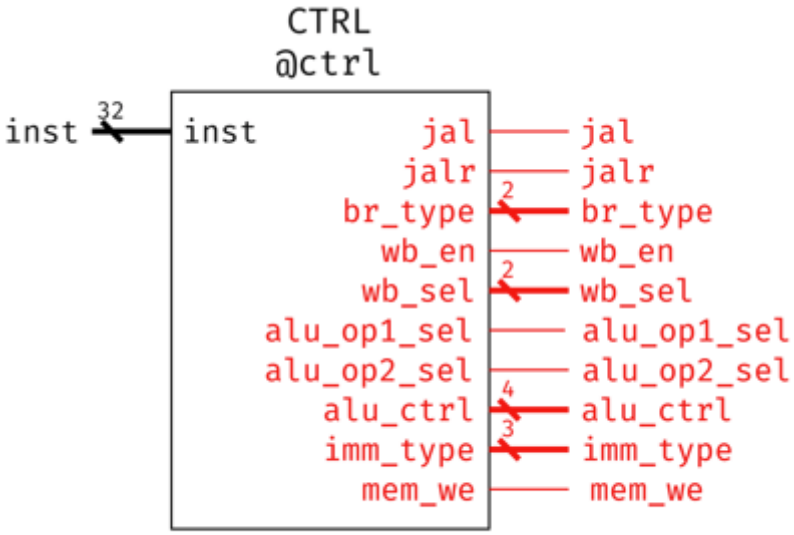
- 设计思路

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

根据不同的指令类型对imm进行扩展

CTRL模块

- 框图



- 代码

```
module CTRL(
    input [31:0] inst,
    output reg jal,
    output reg jalr,
```

```
    output reg [1:0] br_type,
    output reg wb_en,
    output reg [1:0] wb_sel,
    output reg alu_op1_sel,
    output reg alu_op2_sel,
    output reg [3:0] alu_ctrl,
    output reg mem_we
    );
    always@(*) begin
        case(inst[6:0])
        //add
        7'b0110011: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b00;
            alu_op1_sel = 0;
            alu_op2_sel = 0;
            alu_ctrl = 0;
            mem_we = 0;
        end


        //addi
        7'b0010011: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b00;
            alu_op1_sel = 0;
            alu_op2_sel = 1;
            alu_ctrl = 0;
            mem_we = 0;
        end
        //jalr
        7'b1100111: begin
            jal = 0;
            jalr = 1;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b01;
            alu_op1_sel = 0;
            alu_op2_sel = 1;
```

```
            alu_ctrl = 0;
            mem_we = 0;
        end
        //lw
        7'b0000011: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b10;
            alu_op1_sel = 0;
            alu_op2_sel = 1;
            alu_ctrl = 0;
            mem_we = 0;
        end


        //beq,blt
        7'b1100011: begin
            jal = 0;
            jalr = 0;
            if(inst[14:12] == 3'b000)
                br_type = 2'b01;
            else if(inst[14:12] == 3'b100)
                br_type = 2'b10;
            else br_type = 2'b00;
            wb_en = 0;
            wb_sel = 2'b00;
            alu_op1_sel = 1;
            alu_op2_sel = 1;
            alu_ctrl = 0;
            mem_we = 0;
        end


        //sw
        7'b0100011: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 0;
            wb_sel = 2'b00;
            alu_op1_sel = 0;
            alu_op2_sel = 1;
            alu_ctrl = 0;
```

```
            mem_we = 1;
        end


        //lui
        7'b0110111: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b11;
            alu_op1_sel = 0;
            alu_op2_sel = 0;
            alu_ctrl = 4'b1000;
            mem_we = 0;
        end
        //auipc
        7'b0010111: begin
            jal = 0;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b00;
            alu_op1_sel = 1;
            alu_op2_sel = 1;
            alu_ctrl = 0;
            mem_we = 0;
        end


        //jal
        7'b1101111: begin
            jal = 1;
            jalr = 0;
            br_type = 2'b00;
            wb_en = 1;
            wb_sel = 2'b01;
            alu_op1_sel = 1;
            alu_op2_sel = 1;
            alu_ctrl = 0;
            mem_we = 0;
        end


        default: begin
```

```
            jal = 0;
            jalr = 0;
            br_type = 2'b11;
            wb_en = 0;
            wb_sel = 2'b00;
            alu_op1_sel = 0;
            alu_op2_sel = 0;
            alu_ctrl = 10;
            mem_we = 0;
        end
        endcase
    end
endmodule
```

- 设计原理
  - add：不产生jar、jalr、br_type信号，结果来自alu_res所以wb_sel为0，结果写入rd所以wb_en为1，相加所以alu_ctrl选择0，相加的数都来自寄存器所以两个alu_op_sel均为0，不需要存入存储器所以mem_we为0
  - addi：

    ✓ addi rd, rs1, imm # x[rd] = x[rs1] + sext(imm)

    不产生jar、jalr、br_type信号，结果来自alu_res所以wb_sel为0，结果写入rd所以wb_en为1，相加所以alu_ctrl选择0，相加的数都一个来自寄存器一个来自imm所以两个alu_op_sel一个为0，一个为1，不需要存入存储器所以mem_we为0
  - jalr：

    ✓ jalr rd, offset(rs1) # t = pc + 4; pc = (x[rs1] + sext(offset)) & ~1; x[rd]=t

    jalr为1，其余跳转信号为0，选择pc_add4作为写入rd的数据所以wb_sel为1，wb_en为1，跳转的pc是寄存器内容和imm相加，所以alu_ctrl选择0，alu_op_sel一个为0，一个为1，不需要存入存储器所以mem_we为0
  - lw：

    ✓ lw rd, offset(rs1) # x[rd] = M[x[rs1] + sext(offset)]

    不产生jar、jalr、br_type信号，存入rd的数据来自mem_rd,所以wb_sel为2，wb_en为1，地址为寄存器内容加上偏移量，所以alu_ctrl选择0，alu_op_sel一个为0，一个为1，不需要存入存储器所以mem_we为0
  - beq，blt

✓ beq rs1, rs2, offset # if (rs1 == rs2) pc += sext(offset)

✓ blt rs1, rs2, offset # if (rs1 $<_s$ rs2) pc += sext(offset)

br_type信号根据inst[14:12]来确定，其余跳转信号为0，不写入寄存器所以wb_sel、wb_en为0，Branch模块得到br信号， br=1 就会选择 alu_res为下一个pc ， pc+偏移量则alu_op_sel都为1，alu_ctrl选择0，不需要存入存储器所以mem_we为0

- sw

  ✓ sw rs2, offset(rs1) # M[x[rs1] + sext(offset)] = x[rs2]

  不产生jar、jalr、br_type信号，不写入寄存器所以wb_sel、wb_en为0，地址为寄存器内容加上偏移量，所以alu_ctrl选择0,alu_op_sel一个为0，一个为1，需要存入存储器所以mem_we为1

- lui

  ✓ lui rd, imm # x[rd] = imm[31:12] << 12

  不产生jar、jalr、br_type信号，将imm写入rd，则wb_en为1，wb_sel为3，左移12位但不是通过alu实现而是imm模块，因此alu_ctrl都可以，alu_op_sel均为0，不需要存入存储器所以mem_we为0

- auipc

  ✓ auipc rd, imm # x[rd] = pc + imm[31:12] << 12

  不产生jar、jalr、br_type信号，pc+imm存入rd，所以wb_sel为0，wb_en为1，同B-type，alu_op_sel都为1，alu_ctrl选择0，不需要存入存储器所以mem_we为0

- jal

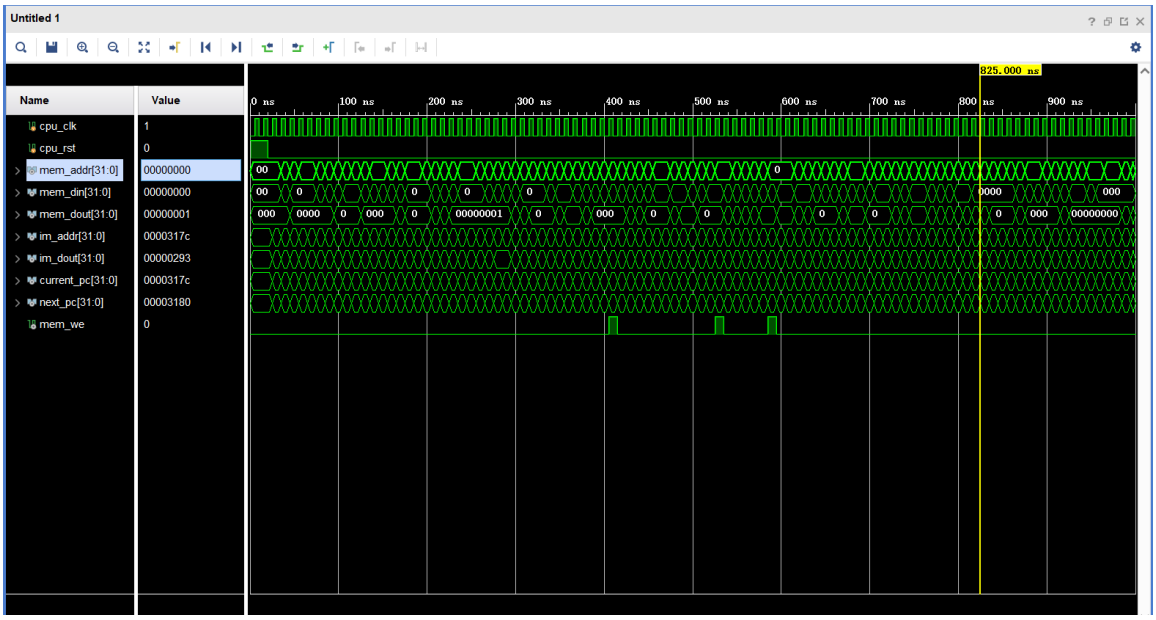
  ✓ jal rd, offset # x[rd] = pc+4, pc += sext(offset)

  jal为1，其余跳转信号为0，pc_add4作为写入rd的数据所以wb_sel为1，wb_en为1，同auipc，alu_op_sel都为1，alu_ctrl选择0，不需要存入存储器所以mem_we为0

3.2 实验结果

> test

- 仿真

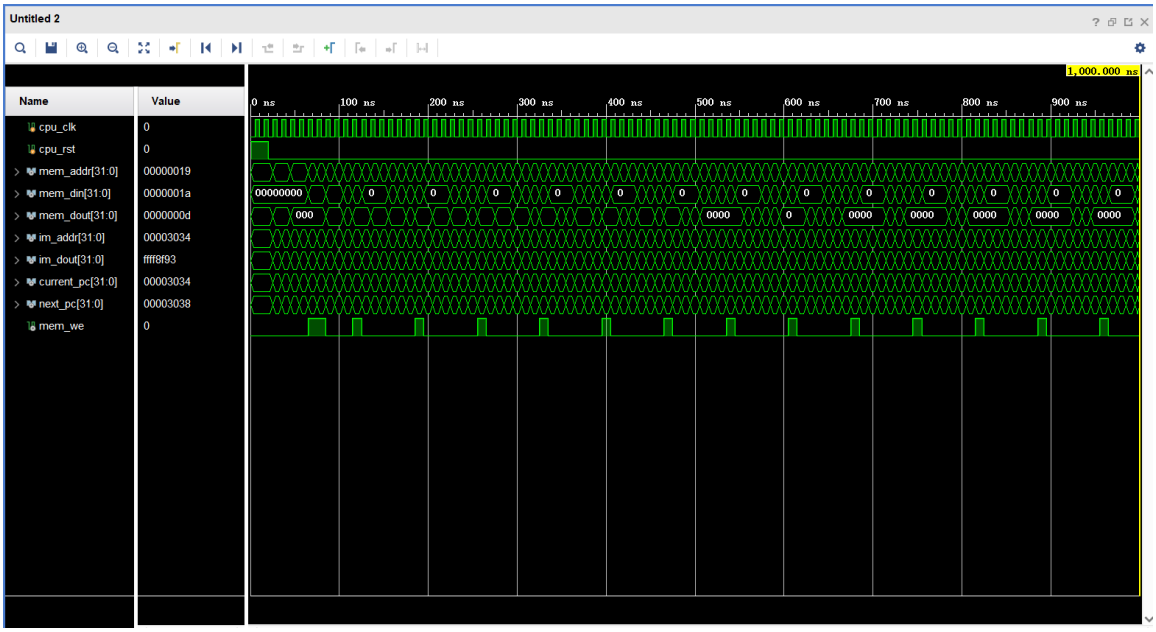


- 烧写

  已在线下检查

FLS

- 仿真



- 烧写

  已在线下检查

4. 分析数据通路差异

在pc_next的处理上有区别，在各类跳转指令比如jal、jalr、beq、blt上体现

5. 总结

本次实验难度适中

通过单周期CPU的设计，更加了解了计算机体系结构，对riscv指令集的架构理解更加深入

对理解较为复杂的verilog代码之间的联系很有帮助