CPU Project 报告

班级：01-6

小组成员：张博钧、刘轩铭

提交时间：2024.6.2

目录

一、开发者说明：

1．小组成员：张博钧 12211615、刘轩铭 12112929

2．负责工作：

（1）张博钧：CPU、部分汇编代码

（2）刘轩铭：部分汇编代码

3．贡献比：

（1）张博钧：55%

（2）刘轩铭：45%

二、开发计划日程安排和实施情况

1．日程安排

（1）CPU 设计：

①Ana_Instruction：5.3 前完成

②ALU、Controller、Decoder、Build_imm：5.11 前完成

③DMem、IOread、MemOrIO：5.13 前完成

④Leds、DigitalOut、CPU：5.16 前完成

（2）汇编代码：

①测试场景一：5.16 前完成

②测试场景二：6.21 前完成

2．实施情况

（1）判断 led 灯和数码管发光的条件（5.8）

（2）当前代码能否运行 addi，能否使用 verilog 中的 zero（5.8）

（3）完善 register 的读取写入（使用 clk 没有问题）（5.8）

（4）完善 pc 指令，否则程序无法正常运行（5.11）

（5）完善 beq 的跳转（5.11）

（6）实现 blt， bge， bltu， bgeu（5.11）

（7）beq，j 这类 B 和 J 指令的当周期跳转，不进行下一周期的修改（5.11）

（8）sw 和 lw 的执行过程（5.11）

（9）lw 读取输入？（lw 直接放地址，在执行中 imm32 会转换成 fffffc10）（5.11）

（10）sw 设备展示？（5.11）

（11）Controller 中的 MemOrIOtoReg，IORead，IOWrite 部分（5.11）

（12）MemOrIO 的控制信号确定（5.12）

（13）alu 对 pc 信号需求（5.12）

（14）分析 mem 数据的读入和写出是否有序（DMem 运行中是否发生错乱）（5.13）

（15）分析 mem 向 register 写回数据时是否发生时序逻辑上的错误（可能 decoder 里需要改动）（5.13）

（16）分析 memorio 里面向 mem 输入是否会有问题（5.13）

（17）想办法实现在数码管间断闪烁十六进制功能（5.15）

（18）外面一个大循环，调节 switch，按下按钮后进入用例，结束后持续显示结果，再按下按钮后跳出用例循环，继续进入外面大循环（5.15）

（19）对应 led 灯信号，应该在 LEDCtrl == 1'b1 时，拨动开关会亮（5.15）

（20）添加从 register 到 led 输出的数据（5.16）

（21）需要改灯，led 灯，只有左 8 是输入，右 8 控制模式（5.16）

（22）coe 文件的操作模式（5.16）

（23）coe 文件，汇编+代码修改（5.20）

（24）测试场景 2（5.22）

（25）场景一的 coe 文件和一些细节上的灯光的问题（5.22）

（26）测试场景二的一些代码和 coe 文件（5.23）

三、CPU 架构说明

1．CPU 特性

（1）ISA

我们参考了 RISCV 指令集并在 CPU 中实现了支持以下 RISCV 指令：add, sub, and, or, lb, lbu, lw, sw, addi, beq, bne, blt, bge, bltu, bgeu, jal, jalr, slli, srli。 add: 编码为 0b0110011, 使用方式为 add rd, rs1, rs2

sub: 编码为 0b0100000（加在 funct7 部分）, 使用方式为 sub rd, rs1, rs2

lw: 编码为 0b0000011, 使用方式为 lw rd, offset(rs1)

sw: 编码为 0b0100011, 使用方式为 sw rs2, offset(rs1)

beq: 编码为 0b1100011, 使用方式为 beq rs1, rs2, offset

bge: 编码为 0b1100011, 使用方式为 bge rs1, rs2, offset

bgeu: 编码为 0b1100011, 使用方式为 bgeu rs1, rs2, offset

blt: 编码为 0b1100011, 使用方式为 blt rs1, rs2, offset

bltu: 编码为 0b1100011, 使用方式为 bltu rs1, rs2, offset

bne: 编码为 0b1100011, 使用方式为 bne rs1, rs2, offset

jal: 编码为 0b1101111, 使用方式为 jal rd, offset

jalr: 编码为 0b1100111, 使用方式为 jalr rd, offset(rs1)

and: 编码为 0b0110011, 使用方式为 and rd, rs1, rs2

or: 编码为 0b0110011, 使用方式为 or rd, rs1, rs2

slli: 编码为 0b0010011, 使用方式为 slli rd, rs1, shamt

srli: 编码为 0b0010011, 使用方式为 srli rd, rs1, shamt

addi: 编码为 0b0010011, 使用方式为 addi rd, rs1, imm

没有专门的异常处理机制。

（2）CPU 时钟

我们设计的是单周期 CPU，时钟频率为 23MHz,不支持 pipiline。

（3）寻址空间设计

本 CPU 采用哈佛结构，指令存储和数据存储分开处理。寻址单位为 32 位（4 字节）。指令空间：PC 寄存器的位宽为 32 位，指令空间的大小为 2^32 字节。数据空间：数据存储器的地址为 32 位，数据空间大小也是 2^32 字节。栈空间的基地址为 0xFFFFFC70。

（4）对外设 IO 的支持

本 CPU 使用 MMIO，相关外设对应的地址如下：

0xFFFFFC70：识别 IO 输入 register（基地址）

0xFFFFFC74：按钮 a 地址

0xFFFFFC78：按钮 b 地址

0xFFFFFC88：模式按钮地址

0xFFFFFC7C：led 高位地址

0xFFFFFC80：led 低位地址

0xFFFFFC84：led 全地址

采用轮询的方式访问 IO。

2．CPU 接口

时钟：clk (P17)

复位：reset

数码管：[7:0] an (G2-G6)

LED：[15:0] ledF6K2 (F6-K2, K1-K3)

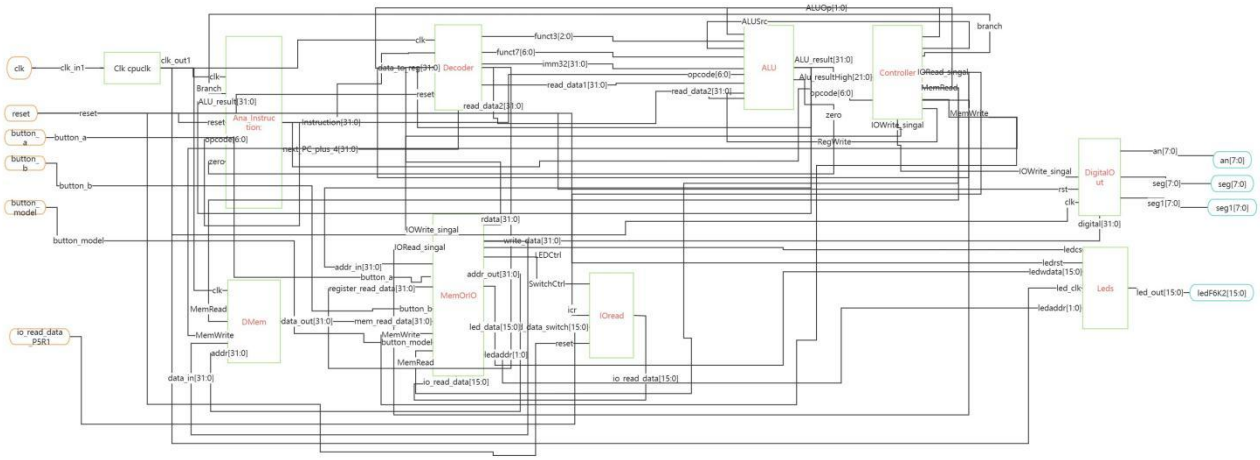
按键：button_a,button_b (分别对应 V1、R11)

模式按键：button_model (R15)

拨码开关：[15:0] io_read_dataP5R1 (P5-R1, U3-T5)

3．CPU 内部结构

CPU 内部各子模块的接口连接关系图如下：



CPU 内部子模块的设计说明（子模块端口规格及功能说明）：

（1）**CPU**：

（CPU 的主模块）

①输入信号：

1）clk

2）reset

3）[15:0] io_read_dataP5R1：16 位开关读入信号

4）button_a：控制数据信号输入

5）button_b：控制数据信号输入

6）button_model：控制模式信号输入

②输出信号：

1）[15:0]ledF6K2：16 位 led 灯控制信号

2）[7:0]seg：左四数码管控制信号

3） [7:0]seg1：右四数码管控制信号

4）[7:0]an：八个数码管显示控制信号

③内部信号：

1）时钟信号

a．wire cpu_clk;

b．wire cpu_uart_clk;

2）Leds 模块信号

a．wire[1:0] ledaddr;

b．wire[15:0] led_input;

3）Digital 模块信号

a．wire[31:0] dagital;

4）IOread 模块信号

a．wire[15:0] io_read_data;

5）Ana_Instruction 模块信号

a．wire[31:0] Instruction;

b．wire[31:0] next_PC_plus_4;

6）Controller 模块信号

a．wire[6:0] opcode;

b．wire[1:0] ALUOp;

c．wire Branch;

d．wire MemRead;

e．wire MemtoReg;

f．wire MemWrite;

g．wire ALUSrc;

h．wire RegWrite;

i．wire MemOrIOtoReg;

j．wire IORead_singal;

k．wire IOWrite_singal;

7）Decoder 模块信号

a．wire[31:0] data_to_reg;//mem 或 io 设备向 register 中写数据（lw）

b．wire[31:0] imm32;

c．wire[2:0] funct3;

d．wire[6:0] funct7;

e．wire[31:0] read_data1;

f．wire[31:0] read_data2;

8）ALU 模块信号

a．wire zero;

b．wire[31:0] ALU_result;

9）MemOrIO 模块信号

a．wire[31:0] mem_addr;//mem_address

b．wire[31:0] mem_data_out;//data mem 输出的数据

c．wire[31:0] rdata;

d．wire[31:0] write_data;

e．wire LEDCtrl;//用于控制 LED 灯是否会亮 == 1'b1 拨动开关 led 灯才会亮

f．wire SwitchCtrl;//用于控制开关输入信息是否有效 == 1'b1 才能有效拨动开关

g．wire DigitalCtrl;//y 用于控制数码管是否会亮

h．wire[15:0] led_data;

10）DMem 模块信号

a．wire[31:0] mem_data_in;//向 data mem 写入的数据

④实例化模块

```
cpuclk Clk(.clk_in1(clk),.clk_out1(cpu_clk),.clk_out2(cpu_uart_clk));
```

传入开发板时钟信号，传出 cpu 所需 23MHz 时钟信号

```
  Leds leds(.ledrst(reset),.led_clk(cpu_clk),
          .ledcs(LEDCtrl),.ledaddr(ledaddr),
          .ledwdata(led_input),.ledout(ledF6K2));
```

传入时钟信号和 reset 信号，控制 led 灯闪烁

```
IOread io_read(.reset(reset),.ior(IORead_singal),
             .switchctrl(SwitchCtrl),.ioread_data_switch(io_read_dataP5R1),
             .ioread_data(io_read_data));
```

传入 IO 开关信号和 cpu 时钟信号，控制 IO 信号的向 CPU 内部输入

```
Ana_Instruction instruction(.clk(cpu_clk),.reset(reset),.Branch(Branch),
                    .zero(zero),.opcode(opcode),.ALU_result(ALU_result),
                    .Instruction(Instruction),.next_PC_plus_4(next_PC_plus_4));
```

传入 cpu 时钟信号 Branch、zero、opcode、ALU_result 信号，控制 PC 跳转，输出正确地址指令

```
Controller con(.Alu_resultHigh({ALU_result[31:10]}),.opcode(opcode),
               .Branch(Branch),.MemRead(MemRead),
               .MemtoReg(MemtoReg),.ALUOp(ALUOp),
               .MemWrite(MemWrite),.ALUSrc(ALUSrc),
               .RegWrite(RegWrite),.MemOrIOtoReg(MemOrIOtoReg),
               .IORead_singal(IORead_singal),.IOWrite_singal(IOWrite_singal));
```

传入 Alu_resultHigh、opcode 信号，控制 Branch，MemRead，MemtoReg，ALUOp，MemWrite，ALUSrc，RegWrite，MemOrIOtoReg，IORead_singal，IOWrite_singal 信号的输出

```
Decoder decoder(.clk(cpu_clk),.reset(reset),.Instruction(Instruction),
                .RegWrite(RegWrite),.data_to_reg(data_to_reg),.next_PC_plus_4(next_PC_plus_4),
                .imm32(imm32),.funct3(funct3),.funct7(funct7),
                .read_data1(read_data1),.read_data2(read_data2));
```

、

传入 cpu 时钟信号、Instruction、RegWrite、data_to_reg、next_PC_plus_4 信号，传出 imm32，funct3，funct7，read_data1，read_data2 信号

```
ALU alu(.read_data1(read_data1),.read_data2(read_data2),
        .imm32(imm32),.ALUOp(ALUOp),.funct3(funct3),
        .funct7(funct7), .ALUSrc(ALUSrc),.opcode(opcode),
        .ALU_result(ALU_result),.zero(zero));
```

传入 read_data1、read_data2、imm32、ALUOp、funct3、funct7、ALUSrc、opcode 信号，传出 ALU_result、zero、check 信号

```
MemOrIO mem_or_io(.button_a(button_a),.button_b(button_b),.button_model(button_model),.MemRead(MemRead),.MemWrite(MemWrite),
                  .IORead_singal(IORead_singal),.IOWrite_singal(IOWrite_singal),
                  .addr_in(ALU_result),.addr_out(mem_addr),
                  .mem_read_data(mem_data_out),.io_read_data(io_read_data),
                  .rdata(rdata),.register_read_data(read_data2),//sw显示读的是rs2
                  .write_data(write_data),.LEDCtrl(LEDCtrl),
                  .SwitchCtrl(SwitchCtrl),.DigitalCtrl(DigitalCtrl),.led_data(led_data),.ledaddr(ledaddr));
```

传入 button_a，button_b，button_model，MemRead，MemWrite，IORead_singal，IOWrite_singal，addr_in，mem_read_data，io_read_data，register_read_data 信号，传出 addr_out，rdata，write_data，LEDCtrl，SwitchCtrl，DigitalCtrl，led_data，ledaddr 信号

```
DMem data_mem(.clk(cpu_clk),.MemRead(MemRead),.MemWrite(MemWrite),
              .addr(mem_addr),.data_in(mem_data_in),.data_out(mem_data_out));
```

传入 cpu 时钟信号，MemRead，MemWrite，mem_addr，mem_data_in 信号，传出 mem_data_out 信号

（2）**Controller**:

（控制模块，输出控制信号）

①输入信号：

1）[21:0] Alu_resultHigh：Alu_Result[31:10]，用于识别 IO 信号

2）[6:0] opcode：Instruction[6:0]，用于输出子模块控制信号

②输出信号：

1）Branch：跳转信号

2）MemRead：向 data memory 读数据信号

3）MemtoReg：data memory 向 register 写入信号

4）[1:0] ALUOp：传入 ALU 模块信号

5）MemWrite：向 data memory 写入信号

6）ALUSrc：使用 Alu_result 计算的控制信号

7）RegWrite：向 register 写入的信号

8）MemOrIOtoReg：data memory 或 IO 向 register 写入的控制信号

9）IORead_singal：从 IO 设备向 CPU 内部输入数据信号

10）IOWrite_singal：从 CPU 向 IO 设备输出数据信号

③内部逻辑：

```
assign IORead_singal = (opcode == LW_OP && Alu_resultHigh == HIGH_ADDR) ? 1'b1 : 1'b0;//lw
assign IOWrite_singal = (opcode == SW_OP && Alu_resultHigh == HIGH_ADDR) ? 1'b1 : 1'b0;//sw时判断输入地址，向设备输入，显示数码管
assign MemOrIOtoReg = IORead_singal || IOWrite_singal;
assign Branch = (opcode == BR_OP || opcode == JAL_OP || opcode == JALR_OP) ? 1'b1 : 1'b0;//最后一个jal
assign MemWrite = (opcode == SW_OP && IOWrite_singal == 1'b0) ? 1'b1 : 1'b0;//sw时为1，可以向mem存
assign MemRead = (opcode == LW_OP && IORead_singal == 1'b0) ? 1'b1 : 1'b0;
assign MemtoReg = (opcode == LW_OP) ? 1'b1 : 1'b0;
assign RegWrite = (opcode == R_TYPE_OP || opcode == LW_OP || opcode == I_TYPE_OP) ? 1'b1 : 1'b0;//最后一个时I指令
assign ALUOp = (opcode == R_TYPE_OP || opcode == I_TYPE_OP) ? 2'b10 :  //add、sub、and、or
               (opcode == LW_OP || opcode == SW_OP || opcode == JALR_OP) ? 2'b00 :  //ld、sw
               (opcode == BR_OP) ? 2'b01 : 2'b11;//11无意义  01beq
assign ALUSrc = (opcode == LW_OP || opcode == I_TYPE_OP || opcode == SW_OP || opcode == JAL_OP || opcode == JALR_OP) ? 1'b1 : 1'b0;
```

opcode == LW_OP && Alu_resultHigh == HIGH_ADDR: IORead_singal = 1’b1;
opcode == SW_OP && Alu_resultHigh == HIGH_ADDR: IOWrite_singal = 1’b1;
IORead_singal || IOWrite_singal: MemOrIOtoReg = 1’b1;
opcode == BR_OP || opcode == JAL_OP || opcode == JALR_OP: Branch = 1’b1;
opcode == SW_OP && IOWrite_singal == 1'b0: MemWrite = 1’b1;
opcode == LW_OP && IORead_singal == 1'b0: 1’b1;
opcode == LW_OP: MemtoReg = 1’b1;
opcode == R_TYPE_OP || opcode == LW_OP || opcode == I_TYPE_OP: RegWrite = 1’b1;
opcode == R_TYPE_OP || opcode == I_TYPE_OP: ALUOp = 2’b10;
opcode == LW_OP || opcode == SW_OP || opcode == JALR_OP: ALUOp = 2’b00;
opcode == BR_OP: ALUOp = 2’b01 : ALUOp = 2’b11;
opcode == LW_OP || opcode == I_TYPE_OP || opcode == SW_OP || opcode == JAL_OP || opcode == JALR_OP: ALUSrc = 1’b1;

（3）**Decoder**:

（register 核心处理模块，该模块用于对指令的解码。）

①输入信号：

1）clk

2）reset：register 重置信号

3）[31:0] Instruction

4）RegWrite：1’b1 时允许向 register 写入

5）[31:0] data_to_reg：从 IO 或 data memory 写入 register 的数据

6）[31:0]next_PC_plus_4：jal 跳转指令地址的 ra 地址

②输出信号：

1）[31:0] imm32

2）[2:0] funct3

3）[6:0] funct7

4）[31:0] read_data1：register 向外输出数据信号

5）[31:0] read_data2：register 向外输出数据信号

③内部逻辑：

1）内部连接信号

```
wire [6:0] opcode;
wire [4:0] register_address1;
wire [4:0] register_address2;
wire [4:0] return_address;
wire [31:0] load_data;
reg [31:0] register[0:31];

assign opcode = Instruction[6:0];
assign return_address = Instruction[11:7];
assign register_address1 = Instruction[19:15];
assign register_address2 = Instruction[24:20];
assign funct3 = Instruction[14:12];
assign funct7 = Instruction[31:25];
```

2）输出 imm32

```
Build_imm imm(.instruction(Instruction), .imm32(imm32));
```

3）读取 register 数据或处理输入 register 数据

```
assign read_data1 = register[register_address1];
assign read_data2 = register[register_address2];
assign load_data = (opcode == 7'b000_0011 && funct3 == 3'b100) ? {24'h00_0000,data_to_reg[7:0]} :  //lbu 读取前8位，用0拓展
                   (opcode == 7'b000_0011 && funct3 == 3'b000) ? {{24{data_to_reg[7]}},data_to_reg[7:0]} ://lb 读取前8位，符号拓展
                   (opcode == JAL_OP) ? next_PC_plus_4 : data_to_reg;//lw和R、I-type
```

opcode == 7'b000_0011 && funct3 == 3'b100：load_data = {24'h00_0000,data_to_reg[7:0]};
lbu 读取前 8 位，用 0 拓展
opcode == 7'b000_0011 && funct3 == 3'b000：load_data={{24{data_to_reg[7]}},data_to_reg[7:0]}
lb 读取前 8 位，符号拓展
opcode == JAL_OP: load_data = next_PC_plus_4;
其余情况：load_data = data_to_reg;

4）register 写入数据

```
initial begin
    for (i=0;i<32;i=i+1)register[i] <= 0;
end
always@(posedge clk, negedge reset)begin
    if(reset == 1'b0) begin                    // 初始化寄存器组
        for(i=0;i<32;i=i+1) register[i] <= 32'h0000_0000;
    end else if(RegWrite == 1'b1 && return_address != 5'b00000) begin
        register[return_address] <= load_data;
    end else if(opcode == JAL_OP && return_address == 5'b00001)begin
        register[return_address] <= load_data;
    end
end
```

当 reset = 1’b0 时，将 register 重置为 0
当 RegWrite == 1'b1 && return_address != 5'b00000，register 写入
当 opcode == JAL_OP && return_address == 5'b00001，register 写入

（4）**ALU**：（该模块为算术逻辑单元）

①输入信号：

1）read_data1, read_data2：两个 32 位操作数

2）imm32: 32 位立即数

3）ALUOp: 2 位 ALU 操作控制信号

4）funct3: 3 位函数码

5）funct7: 7 位函数码

6）ALUSrc: 选择第二个操作数的源

7）opcode: 7 位操作码

②输出信号：

1）ALU_result: 32 位 ALU 运算结果

2）zero: 1 位结果为 0 的标志位

3）check: 1 位分支条件检查标志位

③内部逻辑：

1）根据 ALUSrc 选择第二个操作数的源(read_data2 或 imm32)。

2）根据 ALUOp 和 funct3/funct7 生成 4 位的 ALUControl 控制信号。

3）根据 ALUControl 执行相应的算术逻辑运算,如加、减、与、或、左移、右移等。

4）对于分支指令(beq, bne, blt, bge, bltu, bgeu)进行相应的条件检查,并输出 zero 和 check 标志位。

5）对于跳转指令(jal, jalr)直接使用 imm32 作为结果。

（5）Ana_Instruction：

（该模块处理 RISC-V 处理器的指令获取和程序计数器(PC)更新逻辑）

①IP 核介绍：IP 核类型为 Single Port ROM, Port A Width 为 32, Port A Depth 为 16384, Operating Mode 为 Write First。

②输入信号：

1）clk: 时钟信号

2）clk_j: jalr(跳转并链接寄存器)指令的时钟信号

3）reset: 复位信号

4）Branch: 分支条件信号

5）zero: 来自 ALU 的零标志信号

6）opcode: 来自指令的 7 位操作码

7）ALU_result: 来自 ALU 的 32 位结果

③输出信号：

1）Instruction: 从程序存储器中获取的 32 位指令

2）next_PC_plus_4: 程序计数器(PC)的下一个值加 4

④内部逻辑：

1）内部信号

a．PC: 32 位程序计数器

b．next_PC_4: 程序计数器的下一个值加 4

2）next_PC_plus_4 信号被赋值为 next_PC_4 的值。

3）PC 寄存器在时钟信号的下降沿和复位信号上更新:

4）如果复位信号为低(0),PC 被设置为 0。

5）如果操作码是 110_0111(jalr 指令),PC 根据 Branch 和 zero 条件更新:

6）如果 Branch 为真且 zero 为真,PC 被设置为 ALU 结果。否则,PC 被设置为 0。

7）对于其他指令,PC 根据 Branch 和 zero 条件更新:

8）如果 Branch 为真且 zero 为真,PC 被设置为当前 PC 加上 ALU 结果。否则,PC 被设置为当前 PC 加 4。

9）next_PC_4 寄存器在时钟信号的上升沿和复位信号上更新:

10）如果复位信号为低(0),next_PC_4 被设置为 0。否则,next_PC_4 被设置为当前 PC 加 4。

11）Instruction 输出被赋值为从程序存储器(prgrom)中使用当前 PC 作为地址获取的指令值。

（6）DigitalOut

（该模块为七段数码管显示。）

①输入信号：

1）SwitchCtrl: 用于控制是显示来自 io_read_dataP5R1 还是 digital 的数据的输入信号。

2）IOWrite_singal: 指示是否写入数字显示的输入信号。

3）clk: 系统时钟输入。

4）rst: 复位输入。

5）io_read_dataP5R1: 16 位输入数据,将被显示。

6）digital: 32 位输入数据,将被显示。

②输出信号：

1）seg: 8 位输出,用于驱动七段显示器的段。

2）seg1: 8 位输出,用于驱动七段显示器的段。

3）an: 8 位输出,用于选择活动的七段显示器。

③内部逻辑：

1）内部寄存器：

a．divclk_cnt: 用于生成分频时钟信号的计数器。

b．divclk: 分频后的时钟信号。

c．digital_tube: 32 位寄存器,用于存储要显示的数据。

d．digital_mem: 32 位寄存器,用于存储输入数据。

e．disp_dat: 4 位寄存器,用于存储当前七段显示器要显示的数据。

f．disp_bit: 3 位寄存器,用于跟踪当前更新的七段显示器。

2）时钟分频器：

该模块使用时钟分频器从系统时钟 (clk) 生成较慢的时钟信号 (divclk)。

时钟分频器由 divclk_cnt 计数器控制,当计数器达到 maxcnt 参数值时重置为 0。

3）逻辑处理：

a．digital_tube 寄存器根据 SwitchCtrl 和 IOWrite_singal 输入进行更新。

b．disp_bit 寄存器用于循环遍历 8 个七段显示器,并将适当的数据存储在 disp_dat 寄存器中。

c．seg 和 seg1 输出根据 disp_dat 的值更新,以驱动七段显示器。

d．an 输出根据 disp_bit 的值选择活动的七段显示器。

（7）DMem

（该模块为数据存储器。）

①IP 核介绍：IP 核类型为 Single Port RAM, Write Width 为 32，Read Width 为 32, Write Depth 为 16384, Read Depth 为 16384, Operating Mode 为 Write First。

②输入信号：

1）clk: 输入的时钟信号。

2）MemRead: 输入的内存读使能信号。

3）MemWrite: 输入的内存写使能信号。

4）addr: 输入的 32 位内存地址。

5）data_in: 输入的 32 位写入数据。

③输出信号：

1）data_out：输出的 32 位的读出数据。

④内部逻辑：内部实现了一个 RAM 模块 udram，该模块的功能如下:

该模块使用反相的时钟信号 clock = !clk 作为读写使能 wea 信号由 MemWrite 信号驱动,用于控制是否写入内存 addra 信号由 addr[13:0] 驱动,即使用低 14 位作为 RAM 地址 dina 信号由 data_in 驱动,即写入的数据 douta 信号即读出的数据,连接到输出端 data_out。

（8）IOread

（该模块用于读取外设 IO 的数据。）

①输入信号：

1）reset：复位信号(高电平有效)。

2）ior：从控制器来的 I/O 读信号。

3）switchctrl：从 memorio 经过地址高端线获得的拨码开关模块片选信号。

4）ioread_data_switch：从外设(拨码开关)来的 16 位读数据。

②输出信号：

1）ioread_data：将外设数据送给 memorio 的 16 位输出。

③内部逻辑：

1）内部寄存器：ioread_data_design：用于存储要传递给 memorio 的 16 位数据。

2）逻辑处理：

a．如果 reset 为低电平(0)，则 ioread_data_design 被设置为 16 位全 0。

b．如果 ior 为高电平(1)，则进行以下操作:

c．如果 switchctrl 为高电平(1)，则 ioread_data_design 被赋值为

ioread_data_switch(来自拨码开关的数据)。

d．如果 switchctrl 为低电平(0)，则 ioread_data_design 保持不变。

e．最后,ioread_data 输出被赋值为 ioread_data_design。

（9）Leds

①输入信号：

1）ledrst：复位信号(高电平有效)。

2）led_clk：LED 的时钟信号。

3）ledcs：从 memorio 来的 LED 片选信号,当为高电平(1)时表示 LED 被选中。

4）ledaddr：2 位地址信号,用于选择更新 LED 的低 8 位或高 8 位。

5）ledwdata：要写入 LED 的 16 位数据。

②输出信号：

1）ledout：最终输出到 LED 的 16 位数据。

③内部逻辑：

1）内部寄存器：ledout_design：用于存储要输出到 LED 的 16 位数据。

2）逻辑处理：

a．如果 ledrst 为低电平(0),则 ledout_design 被设置为全 0。

b．如果 ledcs 为高电平(1),则根据 ledaddr 的值进行以下操作:

c．当 ledaddr 为 2'b10 时,ledout_design 的高 8 位被赋值为 ledwdata 的低 8 位,低 8 位保持不变。

d．当 ledaddr 为 2'b01 时,ledout_design 的低 8 位被赋值为 ledwdata 的低 8 位,高 8 位保持不变。

e．当 ledaddr 为 2'b11 时,ledout_design 被赋值为 ledwdata。

f．当 ledaddr 为其他值时,ledout_design 保持不变。

g．如果 ledcs 为低电平(0),则 ledout_design 保持不变。

h．最后,ledout 输出被赋值为 ledout_design。

（10）MemOrIO

（该模块用于确定数据的来源和目的地，负责在内存和 I/O 设备之间进行数据交互,并对相关外设进行控制。）

①输入信号：

1）button_a, button_b, button_model: 这三个信号来自于外部的按钮,用于读取按钮的状态。

2）MemRead, MemWrite: 这两个信号来自于控制器,用于指示是进行内存读操作还是内存写操作。

3）IORead_singal, IOWrite_singal: 这两个信号也来自于控制器,用于指示是进行 I/O 设备的读操作还是写操作。

4）addr_in: 这个 32 位地址信号来自于 ALU 的运算结果,用于指定访问内存或 I/O 设备的地址。

5）mem_read_data: 这个 32 位信号来自于数据存储器,表示从内存读取的数据。

6）io_read_data: 这个 16 位信号来自于 I/O 设备,表示从 I/O 设备读取的数据。

7）register_read_data: 这个 32 位信号来自于寄存器堆,表示从寄存器中读取的数据。

②输出信号：

1）addr_out: 这个 32 位地址信号直接等于输入的 addr_in,表示发送到数据存储器的地址。

2）rdata: 这个 32 位信号是根据输入信号确定的数据源,即来自于内存或 I/O 设备的读取数据。

3）write_data: 这个 32 位信号是根据输入信号确定的数据源,即来自于寄存器堆的写入数据。

4）led_data: 这个 16 位信号表示待写入 LED 的数据。

5）ledaddr: 这个 2 位信号用于指定 LED 的地址。

6）LEDCtrl, SwitchCtrl, DigitalCtrl: 这三个 1 位信号用于控制 LED、开关和数码管的开关状态。

③内部逻辑：

1）如果 IORead_singal 为 1 且地址不是 0xffff_fc74、0xffff_fc78、0xffff_fc88,则 rdata 将是 io_read_data 左移 16 位补充 0;如果 IORead_singal 为 1 且地址是 0xffff_fc74、0xffff_fc78、0xffff_fc88,则 rdata 将分别是 button_a、button_b、button_model 的符号位扩展;否则, rdata 将是 mem_read_data。

2）如果 MemWrite 或 IOWrite_singal 为 1, 则 write_data 将是 register_read_data;否则, write_data 将是 32'h0000_0000。

3）当 IOWrite_singal 为 1 时,将 WD[15:0] 赋给 led_data,并将 LEDCtrl 和 DigitalCtrl 置为 1;当 IORead_singal 为 1 时,将 SwitchCtrl 置为 1;ledaddr 根据地址的不同而取不同的值。

4）addr_out 直接等于 addr_in。

（11）build_imm

（该模块实现了 RISC-V 指令集中不同类型的立即数提取功能。它根据指令的类型,从指令编码中提取出相应的 32 位立即数。）

④输入信号：

1）输入端口 instruction 是一个 32 位的指令码。

⑤输出信号

1）输出端口 imm32 是一个 32 位的立即数。

⑥内部逻辑：

1）该模块定义了 5 种类型的立即数:

imm12_I, imm12_S, imm12_SB: 12 位立即数。

imm20_U, imm20_UJ: 20 位立即数。

imm_I, imm_S, imm_SB, imm_U, imm_UJ: 32 位立即数。

2）再利用位选的方式从 32 位指令中提取出相应的 12 位或 20 位立即数:

imm12_I 从指令的 [31:20] 位提取。

imm12_S 从指令的 [31:25] 和 [11:7] 位提取。

imm12_SB 从指令的 [31]、[7]、[30:25] 和 [11:8] 位提取。

imm20_U 从指令的 [31:12] 位提取。

imm20_UJ 从指令的 [31]、[19:12]、[20] 和 [30:21] 位提取。

3）将提取的 12 位或 20 位立即数符号扩展成 32 位:

imm_I 符号扩展 imm12_I。

imm_S 符号扩展 imm12_S。

imm_SB 符号扩展 imm12_SB 并左移 1 位。

imm_U 左移 12 位。

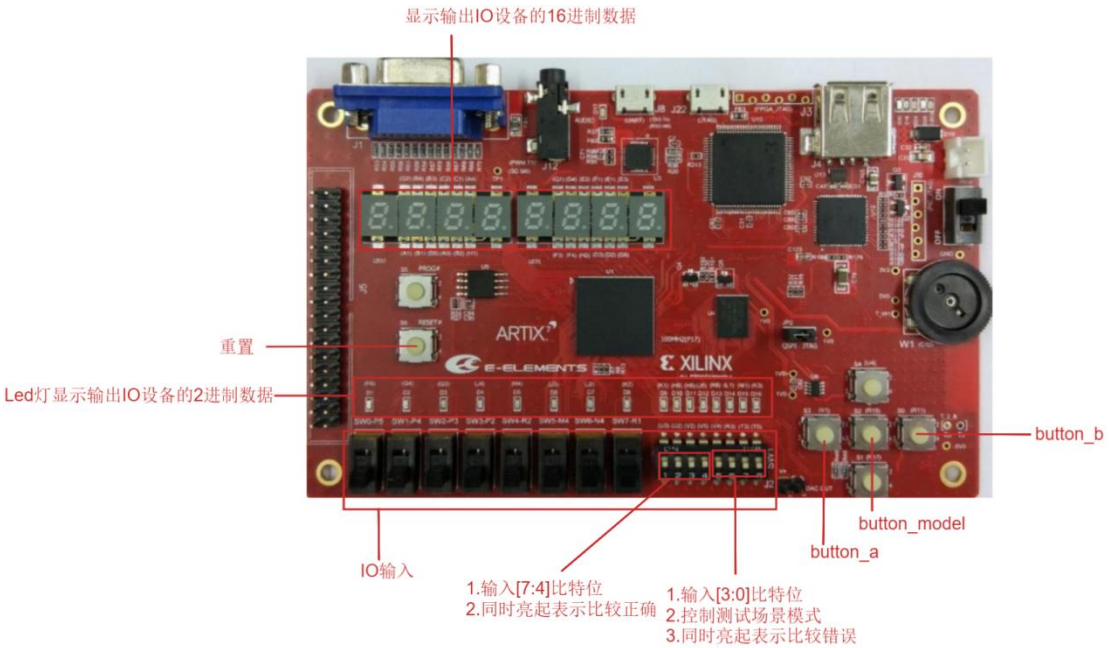
imm_UJ 符号扩展 imm20_UJ 并左移 1 位。

4）利用一个大的 assign 语句,根据指令的操作码选择对应的 32 位立即数赋值

给 imm32 输出端口:对于 load word (lw)和 immediate 算术逻辑指令 (addi, andi, etc.),使用 imm_I。对于 store word (sw),使用 imm_S。对于分支指令 (beq, bne, etc.),使用 imm_SB。对于 upper-immediate 指令 (lui),使用 imm_U。对于 jump and link (jal),使用 imm_UJ。其他情况下输出 32'h0000_0000。

四、系统上板说明

IO 说明：



1. 测试场景一：

如图，开关低 4bit 用来控制输入场景用例编号：

4’b0001 -> 3’b000

4’b0010 -> 3’b001

.....

4’b1000 -> 3’b111

输入用例编号后，按下 button_model 按钮，IO 设备显示输入要测试的用例编号。

按下 button_b 按钮，正式进入测试单元。

3’b000:

低 8bit 位输入 a，按下 button_a，IO 设备 led 在高位 8 显示输入 a 数据，按下 button_b，再低 8bit 位输入 b，按下 button_a，IO 设备 led 同时在高 8 位显示 a 数据，在低 8 位显示 b 数据。

按下 button_b 返回选择用例界面。

3’b001:

全开关可输入 a，按下 button_a，IO 设备 led 灯和数码管同时显示以 Ib 方式存储寄存器后，该寄存器内部 a 的二进制和十六进制的值。

按下 button_b 返回选择用例界面。

3’b010:

全开关可输入 b，按下 button_a，IO 设备 led 灯和数码管同时显示以 Ibu 方式存储寄存器后，该寄存器内部 b 的二进制和十六进制的值。

按下 button_b 返回选择用例界面。

3’b011:

按下 button_a,led 灯表示[3:0]比特位同时亮起，代表比较错误，[7:4]比特位同时亮起，代表比较正确。

3’b100:同 3’b011

3’b101:同 3’b011

3’b110:同 3’b011

3’b111:同 3’b011

按下 button_b 返回选择用例界面。

2．测试场景二：

如图，开关低 4bit 用来控制输入场景用例编号：

4’b0001 -> 3’b000

4’b0010 -> 3’b001

……

4’b1000 -> 3’b111

输入用例编号后，按下 button_model 按钮，IO 设备显示输入要测试的用例编号。

按下 button_b 按钮，正式进入测试单元。

3’b000:

开关低 8 位输入 8bit 数，按下 button_a 按钮，IO 设备 led 灯和数码管分别显示输入的 8bit 数的二进制和十六进制，按下 button_b 按钮，led 灯和数码管显示前导零的个数。

按下 button_a 返回选择用例界面。

3’b001:

全开关输入 16bit 位宽的 IEEE754 编码的半字浮点数，按下 button_a 后，IO 设备 led 灯和数码管分别显示输入的 16bit 浮点数的二进制和十六进制，按下 button_b，数码管显示向上取整后的 16 进制结果。

按下 button_a 返回选择用例界面。

3’b010:

全开关输入 16bit 位宽的 IEEE754 编码的半字浮点数，按下 button_a 后，IO 设备 led 灯和数码管分别显示输入的 16bit 浮点数的二进制和十六进制，按下

button_b，数码管显示向下取整后的 16 进制结果。

按下 button_a 返回选择用例界面。

3’b011:

全开关输入 16bit 位宽的 IEEE754 编码的半字浮点数，按下 button_a 后，IO 设备 led 灯和数码管分别显示输入的 16bit 浮点数的二进制和十六进制，按下 button_b，数码管显示十六进制四舍五入取整后结果。

按下 button_a 返回选择用例界面。

3’b100:

低 8 位开关输入 a，按下 button_a，IO 设备 led 灯和数码管显示输入 a 数据，按下 button_b，显示数据清除，低 8 位开关输入 b，按下 button_a，IO 设备 led 灯和数码管分别显示输入 b 数据的二进制和十六进制，按下 button_b，数码管显示“对 a，b 做加法运算，如果相加和超过 8bit，将高位取出，累加到相加和中，对相加和取反后输出”的结果。

按下 button_a 返回选择用例界面。

3’b101:

低 12 位开关输入数据，按下 button_a，以小端形式显示在数码管上，再按下 button_b，以大端形式显示再数码管上。

按下 button_a 返回选择用例界面。

3’b110:

全开关输入期望的数字，按下 button_a，期望数字显示在 IO 设备的数码管和 led 灯上。按下 button_b，数码管以十六进制方式显示入栈和出栈次数之和。

按下 button_a 返回选择用例界面。

3’b111:

全开关输入期望的数字，按下 button_a，期望数字显示在 IO 设备的数码管和 led 灯上。按下 button_b，数码管以十六进制方式显示每一次入栈的参数，并停留 2-3 秒。

按下 button_a 返回选择用例界面。

五、自测试说明

	单元			集成		
	测试用例	测试结果	测试结论	测试用例	测试结果	测试结论

仿真	**DMem**
	.data
	initial_value: .word 0xffffffff
	repeat_size: .word 0x0057bcf0
	init: .word 0x00000000
		不通过	ip 核 RAM 中 Other Option 设置错误	**Ana_Instruction**
				Controller
				Decoder
				main:
				addi x3, zero, 1
				addi x4, zero, 2
				addi x5, zero, 3
				addi x6, zero, 4
				addi x7, zero, 5
				addi x8, zero, 6
				addi x9, zero, 7
				addi x10, zero, 8
				lw x31, 0
				addi x0, zero, 0
				addi x1, zero, 0#ra
				lw x2, 13#sp
				addi x11, zero, 0#button_a
				addi x12, zero, 0#button_b
				addi x13, zero,
					通过	寄存器存入正确，模块间数据传输正确

				0#button_model addi x14, zero, 0#Compare Integer		

	DMem
	.data
	initial_value: .word
	0xffffffff
	repeat_size: .word
	0x0057bcf0
	init: .word
	0x00000000
		通过	ip 核重置正确，DMem 无误，其于模块无误	**全部模块**
				Top_Loop:
				lw x13,
				0xFFFFFC88
				beq x13, x14,
				Top_Loop
				lw x15,
				0xFFFFFC70
				addi x13, zero, 0
				Show_Choice:
				sw x15,
				0xFFFFFC84
				lw x12,
				0xFFFFFC78
				beq x12, x14,
				Show_Choice
				addi x12, zero, 0
				beq x15, x3, Test1
				beq x15, x4, Test2
				beq x15, x5, Test3
				beq x15, x6, Test4
				beq x15, x7, Test5
				beq x15, x8, Test6
				beq x15, x9, Test7
				beq x15, x10, Test8
				sw x31,
				0xFFFFFC84
				j Top_Loop
					通过	读取指令正常，跳转指令正常工作，led 和数码管输出端口输出数据正常

	Ana_Instruction
	addi x3, zero, 1
	addi x4, zero, 2
	addi x5, zero, 3
	addi x6, zero, 4
	addi x7, zero, 5
	addi x8, zero, 6
	addi x9, zero, 7
	addi x10, zero, 8
		通过	指令传出模块无误			

	测试用例	测试结	测试结	测试用例	测试	测试结

		果	论		结果	论

上板	上板无单元测试			main:
				addi x3, zero, 1
				addi x4, zero, 2
				addi x5, zero, 3
				addi x6, zero, 4
				addi x7, zero, 5
				addi x8, zero, 6
				addi x9, zero, 7
				addi x10, zero, 8
				lw x31, 0
				addi x0, zero, 0
				addi x1, zero, 0#ra
				lw x2, 13#sp
				addi x11, zero, 0#button_a
				addi x12, zero, 0#button_b
				addi x13, zero, 0#button_model
				addi x14, zero, 0#Compare Integer
				Top_Loop:
				lw x13, 0xFFFFFC88
				beq x13, x14, Top_Loop
				lw x15, 0xFFFFFC70
				addi x13, zero, 0
				Show_Choice:
				sw x15, 0xFFFFFC84
				lw x12, 0xFFFFFC78
				beq x12, x14, Show_Choice
				addi x12, zero, 0
				beq x15, x3, Test1
				beq x15, x4, Test2
				beq x15, x5, Test3
				beq x15, x6, Test4
				beq x15, x7, Test5
					通过	读取指令正常，跳转指令正常工作，led和数码管灯光正常亮起

		beq x15, x8, Test6 beq x15, x9, Test7 beq x15, x10, Test8 sw x31, 0xFFFFFC84 j Top_Loop		
		en1.asm(太大)	**通过**	IO 设备显示正常，IO 设备反馈正常，已实现汇编指令运行正常，通过测试一
		en2.asm(太大)	**未通过**	汇编代码未通过，斐波那契测试用例测出有死循环
		en2.asm(太大)	**通过**	IO 设备显示正常，IO 设备反馈正常，通过测试二

六、问题及总结

1．问题

开始的 CPU 制作，需要设计 CPU 架构，对于初学者而言较为困难。设计时，需要控制指令分析和数据的传入传出在同一周期内完成，需要对各个模块的时钟周期和组合逻辑做进一步的协调分析，对于任务分配和 CPU 整体的理解上具有一定挑战性，还有如何设计一个高效的控制单元以正确解码和执行指令等。CPU 设计完毕后，对于如何编写汇编代码，设计利用汇编代码通过 IO 设备正确运行 CPU 的底层逻辑，也

是一个有意思的问题。

2．思考

制作 CPU 一定需要队友之间的密切配合，CPU 各个部分，各个模块的联系非常紧密，如果想要避免细微 bug 的产生，最好抽时间一起来做，一个人负责一个大模块，比如 decode，alu 和 controller。或者一个人写整个 CPU，另一个人写汇编，一个人写 bonus，这样分配。对于一些基础配置上的问题，一定要仔细，这次 pro 的 ip 核设置错误，导致 de 很多天没 de 出来。最后，lab 课课件真的很重要，一定要认真看！!!

3．总结

总的来说，制作 CPU 是一个非常有趣的 project，通过实际动手设计，我们将 lab 课和大课学到的理论知识得以充分实践，丰富了我对机组这门课的理解，同时让我认识到团队合作和有效沟通在复杂项目中的重要性。我十分期待未来还能有这样富有挑战性和趣味性的 project！