# 综合实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 叶福伦 | 学号 | 21051331 |
| 专业 | 计算机专业 | 班级 | 21052313 |
| 指导教师 | 高志刚 | 课程名称 | 计算机组成原理课程设计（甲） |
| 实验序号 | 10 | 实验名称 | 实现JB型指令的CPU设计实验 |
| 实验时间 | 05.13 | 实验地点 | 一教223 |

# 成绩表

|  |  |  |
| --- | --- | --- |
| **考核项目** | **满分** | **得分** |
| **一、实验方案设计** | **30分** |  |
| **二、FPGA程序设计** | **20分** |  |
| **三、测试程序** | **10分** |  |
| **四、实验结果分析** | **20分** |  |
| **五、思考与探索** | **10分** |  |
| **格式** | **10分** |  |
| **合计** | **100分** |  |

# 一、实验方案设计

1. 实验目的与要求（5分）

掌握RISC-V 转移指令的数据通路设计；

掌握掌握指令流和数据流的控制方法；

学习依据新增指令，修改多周期CPU系统结构的方法

具备连接各模块构建整机的能力

实现转移类指令beq、jal、jalr的功能

要求

编写用于测试的汇编程序RIUSJB\_test.s，可以自行设计，也可以采用前述范例程序；将其翻译成机器指令代码，写入RIUSJB\_test.coe文件备用。

将实验9的工程另存为一个新工程，用RIUSJB\_test.coe重新生成指令存储器模块，用测试数据初始化数据存储器。

设置PC多路选择器，新建PC0寄存器，添加相对转移地址加法器，改造W\_Data的多路数据选择器为四选一数据选择器；修改二级指令译码模块ID2和控制单元CU；将它们与其他模块进行正确的连接，构建RIUSJB\_CPU模块

对RIUSJB\_CPU模块进行仿真，在rst\_n之后，每来一个clk，观察指令执行的每一步骤（状态）是否符合预期，确保RIUSJB\_CPU能正确执行目标指令集上的测试程序。

针对使用的实验板卡，设计RIUSJB\_CPU的板级验证实验方案，编写顶层测试模块，要求至少能观察PC、IR、W\_Data、MDR、标志位。可参照实验9方案设计。

复位后，按clk按键，会按步骤执行指令，验证每条指令需要几个周期执行完，执行结果是否正确，将实验结果记录下来。

撰写实验报告，重点内容包括：

1）对仿真结果进行分析；描述你设计的板级验证实验方案、模块结构与连接；说明你的板级操作过程；记录板级实验结果。

2）针对以下有待验证的实验问题，给出证据与分析，得到有效结论：

①beq指令能正确做出数据比较，并进行分支；

②jal指令能完成子程序的调用，也能完成无条件的跳转；

③jalr指令能完成子程序返回；

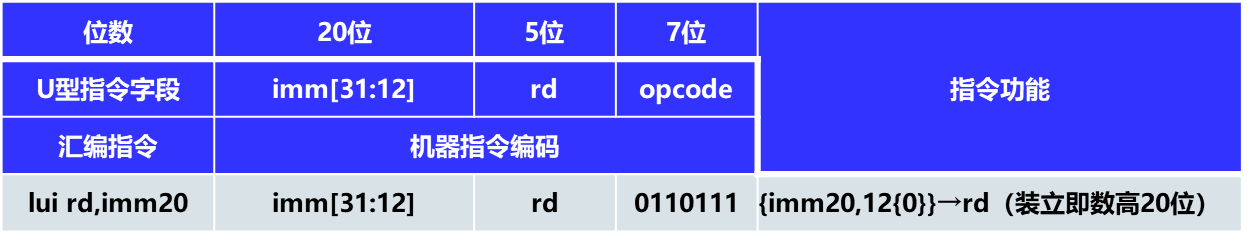
④测试程序执行过程中，循环执行次数及PC变化符合预期。

3）请力所能及回答或实践本实验的“思考与探索”部分。

1. CPU的目标指令集（5分）

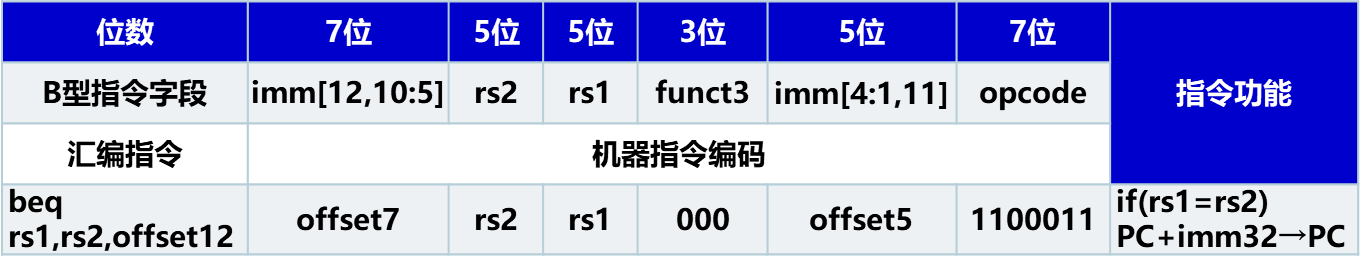










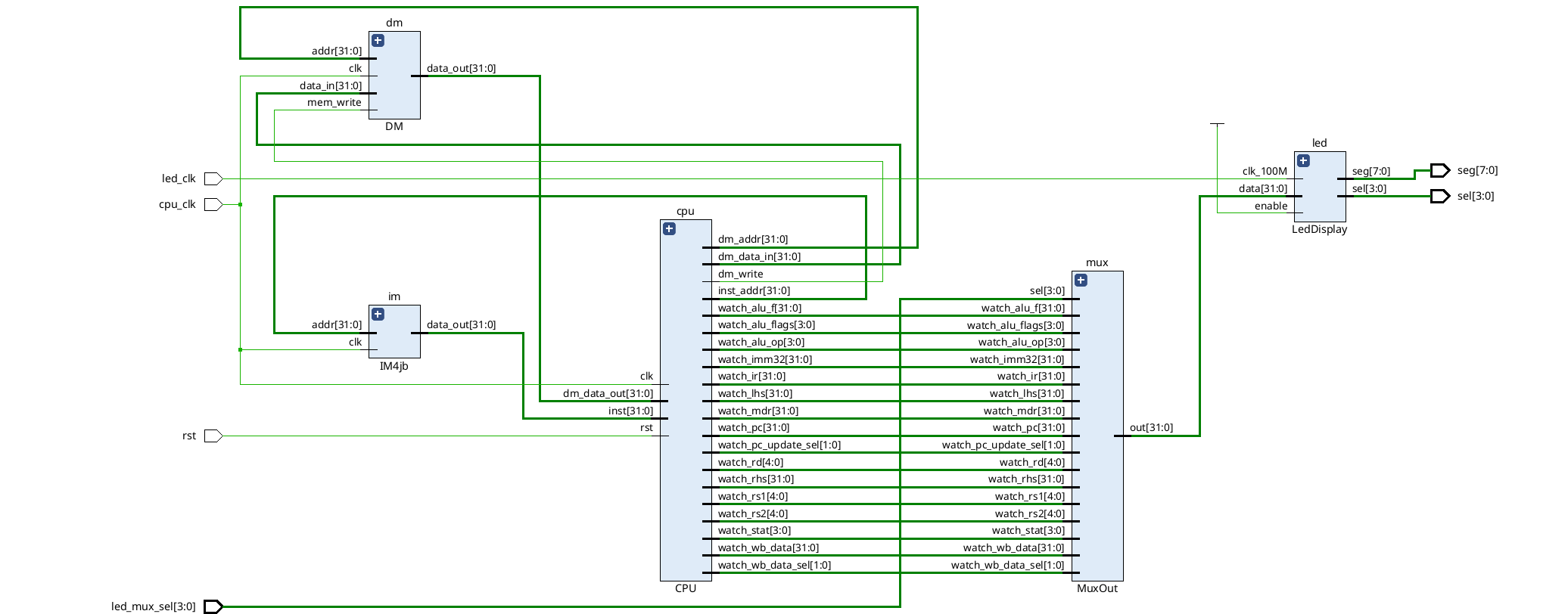






1. CPU的模块构成与连接（15分）

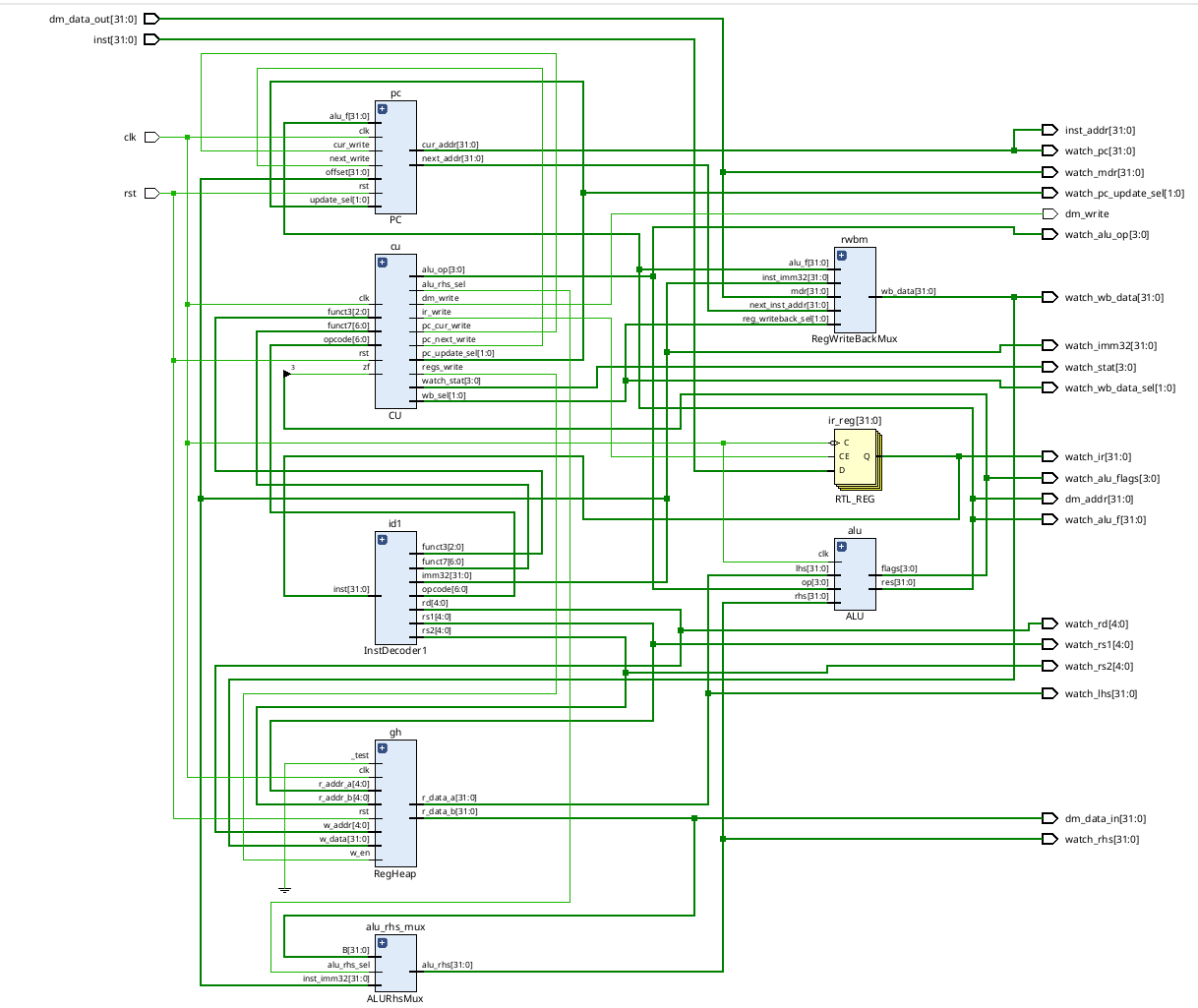
（请附图，并说明各模块的功能，以及模块间的连接关系。也可附开发工具中显示的电路模块连接图，加以说明。）



如图，本次模型机采用哈佛架构。

所以指令存储器im和数据存储器dm分开。cpu\_clk是cpu, im, dm共同的工作时钟。im是指令存储器。存放运行所要的机器指令。dm是数据存储器用于存放数据。im使用inst\_addr端口和inst端口和cpu模块通信。inst\_addr连接地址线。inst连接数据线。dm使用dm\_addr端口、dm\_data\_in端口、dm\_data\_out端口、dm\_write端口和cpu模块通信。dm\_addr连接地址线。数据线分为两条。一条是写数据线。一条是读数据线。dm\_data\_in连接写数据线。dm\_data\_out连接读数据线。dm\_write是写允许信号。cpu的watch\_\*的端口输出cpu内部状态以便用开发板观察。因为开发板上没有很多输出设备所以。经过复用器后输出为复用输出信号。其中sel是复用选择信号。复用输出信号输入到led模块。led驱动数码管显示数据。

下面解析cpu的内部结构

如图cu是控制单元。pc是pc寄存器。id1是初级译码器。gh是寄存器堆。ir\_reg是指令寄存器。

alu是运算单元。cu输入经过初级译码器解析出的opcode从而确定这一个指令周期要执行的状态转移过程。cu通过funct3，funct7，opcode确定alu\_op。rst用于把cu的内部状态回归到初态。zf连接alu的0标志寄存器用于分支指令。watch\_stat输出cu内部状态。\*\_write是对其余几个模块的写控制信号。\*\_sel是如果其余几个模块有多个数据输入源，对应的数据源选择信号。

pc寄存器输入时钟和复位信号。next\_write控制对下一条要执行的指令的地址的修改。cur\_write控制对要执行的指令的修改。pc有4个更新模式。分别使要执行的指令地址+4(顺序执行), 相对跳转: 使寄存器加上offset输入的值(jal, beq)，绝对跳转: 使寄存器变成为alu\_f端口输入的值(jalr)和不变。具体更新寄存器的值用哪种模式由cu通过update\_sel端口决定。输出cur\_addr和next\_addr分别指向本条和下一条要执行的指令地址。

ir寄存器存储本个指令周期所要执行的指令。ir\_write控制该寄存器的写入。

id1寄存器根据ir存储的指令计算出rs1,rs2, rd, funct3， funct7，opcode和imm32.

funct3, funct7, opcode交给cu做进一步的处理。rs1，rs2作为寄存器堆的两个读地址。

rd作为寄存器堆的写地址。imm32交给复用输入器alu\_rhs\_mux。

gh是寄存器堆。clk是时钟信号，rst是复位信号。w\_en是写允许信号，控制信号写回。gh根据两个读地址读出两个32位数据通过r\_data\_a和r\_data\_b端口输出。写回置写允许信号w\_en为1时输入立即数inst\_imm32、alu计算结果alu\_f、数据存储器的数据寄存器输出数据mdr和下一条执行指令地址next\_inst\_addr。并输入写选择信号reg\_writeback\_sel。通过写回数据复用器确定写回的数据。通过w\_data端口输入从而完成了数据的写入。

alu是算数逻辑单元。clk是时钟信号。lhs左操作数端口与寄存器的r\_data\_a端口相连，作为运算的左操作数。rhs右操作数前端接一个输入复用器。根据cu输出的右操作数选择信号选择立刻数或者是r\_data\_b端口数据作为右操作数。通过cu指定的op执行相应运算。输出res作为运算结果。flags是标志寄存器。

1. 板级验证方案（5分）

（说明顶层测试模块的输入/输出信号，及对应使用的I/O设备；可附图说明）

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 端口 | 使用io设备 | 说明 |
| input | led\_clk | 全局时钟信号 | 驱动数码管工作 |
| cpu\_clk | 按键 | 模拟cpu时钟周期 |
| rst | 按键 | 复位信号 |
| led\_mux\_sel[3:0] | 一组逻辑开关 | 确定输出在数码管的是什么信号 |
| output | sel[3:0] | 数码管 | 驱动数码管工作 |
| seg[7:0] |

# 二、FPGA程序设计

（20分）

1. 实验程序源代码

（实验各个模块的代码，包含功能注释）

//alu\_op.v

//alu\_op编码

`define ALU\_ADD\_OP 4'b0000

`define ALU\_SLL\_OP 4'b0001

`define ALU\_SLT\_OP 4'b0010

`define ALU\_SLTU\_OP 4'b0011

`define ALU\_XOR\_OP 4'b0100

`define ALU\_SRL\_OP 4'b0101

`define ALU\_OR\_OP 4'b0110

`define ALU\_AND\_OP 4'b0111

`define ALU\_SUB\_OP 4'b1000

`define ALU\_SRA\_OP 4'b1101

`define ALU\_ADDU\_OP 4'b1010 //扩展不是标准实现,与add区别在标志寄存器的溢出和进位判断上

`define ALU\_SUBU\_OP 4'b1011 //扩展不是标准实现,与sub区别在标志寄存器的溢出和进位判断上

`define ALU\_UNVALID\_OP 4'b1111

//flag.v

`define FLAG\_IS\_ZERO 2'b11

`define FLAG\_IS\_POSITIVE 2'b10

`define FLAG\_CARRY 2'b01

`define FLAG\_OVERFLOW 2'b00

//opcode.v

//opcode编码

`define INST\_R 7'b0110011 //R-type

`define INST\_I 7'b0010011 //I-type

`define INST\_L 7'b0000011 //L-type

`define INST\_S 7'b0100011 //S-type

`define INST\_B 7'b1100011 //B-type(beq)

`define INST\_LUI 7'b0110111 //lui

`define INST\_AUIPC 7'b0010111 //auipc 不使用

`define INST\_JAL 7'b1101111 //jal

`define INST\_JALR 7'b1100111 //jalr

//pc\_update.v

//pc更新类型

`define PC\_STEP 2'b00 //自增一个指令字

`define PC\_JP\_R 2'b01 //jal beq

`define PC\_JP\_F 2'b10 //jalr

`define PC\_HOLD 2'b11 //不变

//rhs\_from.v

//rhs来自imm32还是B寄存器

`define RHS\_FROM\_B\_REG 1'b0

`define RHS\_FROM\_IMM32 1'b1

//wb\_from.v

//回写数据来源

`define WB\_FROM\_IMM32 2'b00 //imm32立即数

`define WB\_FROM\_F\_REG 2'b01 //alu运算结果F寄存器

`define WB\_FROM\_M\_REG 2'b10 //MDR寄存器

`define WB\_FROM\_P\_REG 2'b11 //PC寄存器

//ALU.v

`include "../define/alu\_op.v"

module ALU (

input [31:0] lhs,

input [31:0] rhs,

input clk,

input [3:0] op,

output reg [31:0] res,

output reg [3:0] flags // [is\_zero, is\_positive, carry, overflow]

);

reg \_c;

always @(negedge clk) begin

case(op)

`ALU\_ADD\_OP :begin { \_c, res } = lhs + rhs; flags[0] = \_c ^ res[31] ^ lhs[31] ^ rhs[31]; flags[1] = 0;end

`ALU\_SLL\_OP :begin res = lhs << rhs; flags[1:0] = 0; end

`ALU\_SLT\_OP :begin res = $signed(lhs) < $signed(rhs); flags[1:0] = 0;end

`ALU\_SLTU\_OP:begin res = lhs < rhs; flags[1:0] = 0;end

`ALU\_XOR\_OP :begin res = lhs ^ rhs; flags[1:0] = 0;end

`ALU\_SRL\_OP :begin res = lhs >> rhs; flags[1:0] = 0;end

`ALU\_OR\_OP :begin res = lhs | rhs; flags[1:0] = 0;end

`ALU\_AND\_OP :begin res = lhs & rhs; flags[1:0] = 0;end

`ALU\_SUB\_OP :begin { \_c, res } = lhs - rhs; flags[0] = \_c ^ res[31] ^ lhs[31] ^ rhs[31]; flags[1:0] = 0;end

`ALU\_SRA\_OP :begin res = $signed(lhs) >>> rhs; flags[1:0] = 0;end

`ALU\_ADDU\_OP:begin { flags[1], res } = lhs + rhs; flags[0] = 0;end

`ALU\_SUBU\_OP:begin { flags[1], res } = lhs - rhs; flags[0] = 0;end

default: begin res = 0; flags[1:0] = 0;end

endcase

flags[3] <= res == 32'h0000\_0000 ? 1 : 0;

flags[2] <= res[31];

end

endmodule

//RegHeap.v

module RegHeap(

input clk,

input w\_en,

input rst,

input \_test,

input [4:0] r\_addr\_a,

output reg [31:0] r\_data\_a,

input [4:0] r\_addr\_b,

output reg [31:0] r\_data\_b,

input [4:0] w\_addr,

input [31:0] w\_data

);

reg [31:0]regs[31:0];

integer i;

always@(negedge clk or posedge rst) begin

if(rst) begin

if(\_test) begin

regs[0] <= 0;

for(i=1;i<32;i=i+1) regs[i] <= 1 << i;

end else begin

for(i=0;i<32;i=i+1) regs[i] <= 0;

end

end else begin

if(w\_en == 1 && w\_addr != 0) begin

regs[w\_addr] <= w\_data;

end

end

end

always@(negedge clk)begin

r\_data\_a <= regs[r\_addr\_a];

r\_data\_b <= regs[r\_addr\_b];

end

endmodule

//InstDecoder1.v

`include "../define/opcode.v"

module InstDecoder1(

input [31:0] inst,

output reg [4:0] rs1, rs2, rd,

output reg [31:0] imm32,

output reg [6:0] opcode,

output reg [2:0] funct3,

output reg [6:0] funct7

);

always @(\*) begin

opcode = inst[6:0];

rs1 = inst[19:15];

rs2 = inst[24:20];

rd = inst[11:7];

funct3 = inst[14:12];

funct7 = inst[31:25];

case(opcode)

`INST\_R: begin imm32 = 0; end

`INST\_I: begin

imm32 = (inst[14:12]==3'b101 && inst[30]==1'b1) ?

{27'b0, inst[24:20]}: //对应srai指令

{{20{inst[31]}}, inst[31:20]};

end

`INST\_L: begin imm32 = {{20{inst[31]}}, inst[31:20]} ;end

`INST\_S: begin imm32 = {{20{inst[31]}}, inst[31:25], inst[11:7]} ;end

`INST\_B: begin imm32 = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0} ;end

`INST\_LUI: begin imm32 = {inst[31:12], 12'b0} ;end

`INST\_AUIPC: begin imm32 = {inst[31:12], 12'b0} ;end

`INST\_JAL: begin imm32 ={{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};end

`INST\_JALR: begin imm32 = {{20{inst[31]}}, inst[31:20]} ;end

default: begin imm32 = 0;end

endcase

end

endmodule

//PC.v

`include "../define/pc\_update.v"

//"../07-inst-fetch-decode/PC.v"的升级版，使得能够支持跳转指令

module PC (

input clk,

input rst,

input next\_write,

input cur\_write,

input [1:0] update\_sel,

input [31:0] alu\_f,

input [31:0] offset,

output reg [31:0] next\_addr,

output reg [31:0] cur\_addr

);

always@(posedge rst or negedge clk) begin

if(rst)begin

cur\_addr <= 0;

next\_addr <= 0;

end else begin

if(cur\_write)begin

case(update\_sel)

`PC\_STEP: cur\_addr <= next\_addr;

`PC\_JP\_R: cur\_addr <= cur\_addr + offset;

`PC\_JP\_F: cur\_addr <= alu\_f;

`PC\_HOLD: cur\_addr <= cur\_addr;

endcase

end

if(next\_write)begin

next\_addr <= cur\_addr + 4;

end

end

end

endmodule

//InstDecoder2.v

`include "../define/opcode.v"

`include "../define/alu\_op.v"

//次级译码模块，主要的功能是从opcode, func3, func7中译出alu\_op

module InstDecoder2 (

input [6:0] opcode,

input [2:0] funct3,

input [6:0] funct7,

output reg [3:0] alu\_op

);

always @(\*) begin

case(opcode)

`INST\_R:begin alu\_op <= {funct7[5], funct3}; end

`INST\_I:begin

alu\_op <= (funct3==3'b101)?

{funct7[5], funct3}: //算术和逻辑移位

{1'b0, funct3}; //其他

end

`INST\_L:begin alu\_op <= `ALU\_ADD\_OP; end

`INST\_S:begin alu\_op <= `ALU\_ADD\_OP; end

`INST\_B:begin alu\_op <= `ALU\_XOR\_OP; end

`INST\_JALR:begin alu\_op <= `ALU\_ADD\_OP; end

default:begin alu\_op <= `ALU\_UNVALID\_OP; end

endcase

end

endmodule

//CU.v

module CU (

input rst, //复位信号

input clk,

input [6:0] opcode,

input [2:0] funct3,

input [6:0] funct7,

input zf, //beq命令有用，alu做异或操作，为0则zf=1

output reg pc\_cur\_write,

output reg pc\_next\_write,

output reg [1:0] pc\_update\_sel, //控制pc更新类型

output reg ir\_write, //控制ir寄存器写

output reg regs\_write, //控制寄存器堆写操作

output reg dm\_write, //控制数据存储器写

output [3:0] alu\_op, //alu操作码

output reg alu\_rhs\_sel, //alu右操作数来源

output reg [1:0] wb\_sel, //回写到rd寄存器的数据来源

output [3:0] watch\_stat //当前状态

);

///IF取指令

///PI pc inc

///ID初级译码器译码

///IFID的前一个CPU周期下降沿更新pc\_cur

///IFID: 上升沿根据pc寄存器,从指令寄存器中读出本次指令周期所要执行的指令，下降沿把指令打入IR寄存器，同时pc把下一条要执行的指令地址写入pc\_next

///RR读寄存器 到 暂存器 A, B

///EX(I,B) 执行计算指明右操作数是立刻数还是寄存器B

///M(R,W)访存 从F寄存器读取访问地址，从w\_data写入存储器/读出数据写入MDR寄存器

///WB(I,F,M,P) 从立刻数/ALU运算结果/数据存储器数据寄存器/PC寄存器写入寄存器堆

///JP(F,R) F写入PC寄存器/相对跳转

///BEQ 如果相等则（相对）跳转

///WBP\_JPR 从PC寄存器写入rd，然后相对跳转

///WBP\_JPF 从PC寄存器写入rd，然后把F寄存器的数据写入PC寄存器

parameter ENABLE = 1'b1;

parameter DISABLE = 1'b0;

InstDecoder2 id2(opcode, funct3, funct7, alu\_op);

//对每一步的操作进行编码并把状态保存在stat和next\_stat寄存器

parameter IDLE = 4'b0000;

parameter IFID = 4'b0001;

parameter RR = 4'b0010;

parameter EXI = 4'b0011;

parameter EXB = 4'b0100;

parameter MR = 4'b0101;

parameter MW = 4'b0110;

parameter WBI = 4'b0111;

parameter WBF = 4'b1000;

parameter WBM = 4'b1001;

parameter WBP\_JPF = 4'b1010;

parameter WBP\_JPR = 4'b1011;

parameter BEQ = 4'b1100;

reg [3:0] stat, next\_stat;

assign watch\_stat = stat;

//状态转移

always @(posedge rst or posedge clk) begin

if(rst) stat <= IDLE;

else stat <= next\_stat;

end

///外部状态 opcode

///内部状态 stat

///R-type: IFID -> RR -> EXB -> WBF -> IFID

///I-type: IFID -> RR -> EXI -> WBF -> IFID

///U-type(lui): IFID -> WBI -> IFID

///L-type(lw): IFID -> RR -> EXI -> MR -> WBM -> IFID

///S-type(sw): IFID -> RR -> EXI -> MW -> IFID

///B-type(beq): IFID -> RR -> EXB -> BEQ -> IFID

///J-type(jal): IFID -> WBP\_JPR -> IFID

///J-type(jalr): IFID -> RR -> EXI -> WBP\_JPF -> IFID

//确定下一个状态

always @(\*) begin

case(stat)

IDLE: next\_stat = IFID;

IFID: begin

case(opcode)

`INST\_R: next\_stat = RR;

`INST\_I: next\_stat = RR;

`INST\_LUI: next\_stat = WBI;

`INST\_L: next\_stat = RR;

`INST\_S: next\_stat = RR;

`INST\_B: next\_stat = RR;

`INST\_JAL: next\_stat = WBP\_JPR;

`INST\_JALR: next\_stat = RR;

default: next\_stat = IDLE;

endcase

end

RR: begin

case(opcode)

`INST\_R: next\_stat = EXB;

`INST\_I: next\_stat = EXI;

`INST\_L: next\_stat = EXI;

`INST\_S: next\_stat = EXI;

`INST\_B: next\_stat = EXB;

`INST\_JALR: next\_stat = EXI;

default: next\_stat = IDLE;

endcase

end

EXI: begin

case(opcode)

`INST\_I: next\_stat = WBF;

`INST\_L: next\_stat = MR;

`INST\_S: next\_stat = MW;

`INST\_JALR: next\_stat = WBP\_JPF;

default: next\_stat = IDLE;

endcase

end

EXB: begin

case(opcode)

`INST\_R: next\_stat = WBF;

`INST\_B: next\_stat = BEQ;

default: next\_stat = IDLE;

endcase

end

MR: next\_stat = WBM;

MW: next\_stat = IFID;

WBI: next\_stat = IFID;

WBF: next\_stat = IFID;

WBM: next\_stat = IFID;

WBP\_JPF: next\_stat = IFID;

WBP\_JPR: next\_stat = IFID;

BEQ: next\_stat = IFID;

default: next\_stat = IDLE;

endcase

end

assign pc\_rst = rst;

//根据状态执行操作

always @(posedge rst or posedge clk) begin

if(rst) begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end else begin

case(next\_stat)

IDLE: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

IFID: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= ENABLE;

ir\_write <= ENABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

RR: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= DISABLE;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

EXI: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= DISABLE;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

alu\_rhs\_sel <= `RHS\_FROM\_IMM32;

dm\_write <= (opcode == `INST\_S) ? ENABLE : DISABLE; //数据寄存器要在上升沿写入，这里提前准备

end

EXB: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= DISABLE;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

alu\_rhs\_sel <= `RHS\_FROM\_B\_REG;

dm\_write <= DISABLE;

end

MR: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= DISABLE;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

MW: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

WBI: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= ENABLE;

wb\_sel <= `WB\_FROM\_IMM32;

dm\_write <= DISABLE;

end

WBF: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= ENABLE;

wb\_sel <= `WB\_FROM\_F\_REG;

dm\_write <= DISABLE;

end

WBM: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= ENABLE;

wb\_sel <= `WB\_FROM\_M\_REG;

dm\_write <= DISABLE;

end

WBP\_JPF: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_JP\_F;

ir\_write <= DISABLE;

regs\_write <= ENABLE;

wb\_sel <= `WB\_FROM\_P\_REG;

dm\_write <= DISABLE;

end

WBP\_JPR: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_JP\_R;

ir\_write <= DISABLE;

regs\_write <= ENABLE;

wb\_sel <= `WB\_FROM\_P\_REG;

dm\_write <= DISABLE;

end

BEQ: begin

pc\_cur\_write <= ENABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= ( zf == 1 ) ? `PC\_JP\_R : `PC\_STEP; //zf为1时rhs==lhs，要相对跳转，否则顺序执行

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

default: begin

pc\_cur\_write <= DISABLE;

pc\_next\_write <= DISABLE;

pc\_update\_sel <= `PC\_STEP;

ir\_write <= DISABLE;

regs\_write <= DISABLE;

dm\_write <= DISABLE;

end

endcase

end

end

endmodule

//CPU.v

`include "../define/flag.v"

`include "../define/wb\_from.v"

`include "./CU.v"

`include "../07-inst-fetch-decode/InstDecoder1.v"

`include "./PC.v"

`include "../04-reg-heap/RegHeap.v"

`include "../03-ALU/ALU.v"

module ALURhsMux(

input [31:0] B,

input [31:0] inst\_imm32,

input alu\_rhs\_sel,

output reg [31:0] alu\_rhs

);

always@(\*)begin

case(alu\_rhs\_sel)

`RHS\_FROM\_B\_REG: alu\_rhs <= B;

`RHS\_FROM\_IMM32: alu\_rhs <= inst\_imm32;

endcase

end

endmodule

module RegWriteBackMux(

input [31:0] inst\_imm32,

input [31:0] alu\_f,

input [31:0] mdr,

input [31:0] next\_inst\_addr,

input [1:0] reg\_writeback\_sel,

output reg [31:0] wb\_data

);

always @(\*) begin

case(reg\_writeback\_sel)

`WB\_FROM\_IMM32: wb\_data <= inst\_imm32;

`WB\_FROM\_F\_REG: wb\_data <= alu\_f;

`WB\_FROM\_M\_REG: wb\_data <= mdr;

`WB\_FROM\_P\_REG: wb\_data <= next\_inst\_addr;

endcase

end

endmodule

//CPU.v

module CPU(

input clk,

input rst,

//与指令存储器相连

output [31:0] inst\_addr,

input [31:0] inst, //IFID阶段从指令存储器中读出的指令

//与数据存储器相连

output dm\_write,

output [31:0] dm\_addr,

output [31:0] dm\_data\_in,

input [31:0] dm\_data\_out,

//侵入式接口暴露CPU内部信息

output [3:0] watch\_stat,

output [31:0] watch\_pc,

output [31:0] watch\_ir,

output [4:0] watch\_rs1,

output [4:0] watch\_rs2,

output [4:0] watch\_rd,

output [31:0] watch\_wb\_data,

output [31:0] watch\_imm32,

output [31:0] watch\_lhs,

output [31:0] watch\_rhs,

output [3:0] watch\_alu\_op,

output [31:0] watch\_alu\_f,

output [3:0] watch\_alu\_flags,

output [31:0] watch\_mdr,

output [1:0] watch\_wb\_data\_sel,

output [1:0] watch\_pc\_update\_sel

);

//控制信号

wire pc\_cur\_write;

wire pc\_next\_write;

wire ir\_write;

wire regs\_write;

//wire dm\_write;

//选择信号

wire [1:0] pc\_update\_sel;

wire alu\_rhs\_sel;

wire [1:0] reg\_writeback\_sel;

wire [31:0] pc\_cur;

assign inst\_addr = pc\_cur;

wire [31:0] pc\_next;

reg [31:0] ir;

wire [4:0] reg\_rs1, reg\_rs2, reg\_rd;

wire [31:0] reg\_wb\_data;

wire [6:0] inst\_opcode;

wire [2:0] inst\_func3;

wire [6:0] inst\_func7;

wire [31:0] inst\_imm32;

wire [31:0] A, B;

wire [31:0] alu\_lhs;

assign alu\_lhs = A;

wire [31:0] alu\_rhs;

wire [3:0] alu\_op;

wire [31:0] alu\_f;

wire [3:0] alu\_flags;

assign dm\_addr = alu\_f;

assign dm\_data\_in = B;

assign watch\_pc = pc\_cur;

assign watch\_ir = ir;

assign watch\_rs1 = reg\_rs1;

assign watch\_rs2 = reg\_rs2;

assign watch\_rd = reg\_rd;

assign watch\_wb\_data = reg\_wb\_data;

assign watch\_lhs = alu\_lhs;

assign watch\_rhs = alu\_rhs;

assign watch\_imm32 = inst\_imm32;

assign watch\_alu\_op = alu\_op;

assign watch\_alu\_f = alu\_f;

assign watch\_alu\_flags = alu\_flags;

assign watch\_mdr = dm\_data\_out;

assign watch\_wb\_data\_sel = reg\_writeback\_sel;

assign watch\_pc\_update\_sel = pc\_update\_sel;

CU cu(

rst,

clk,

inst\_opcode,

inst\_func3,

inst\_func7,

alu\_flags[`FLAG\_IS\_ZERO],

pc\_cur\_write,

pc\_next\_write,

pc\_update\_sel,

ir\_write,

regs\_write,

dm\_write,

alu\_op,

alu\_rhs\_sel,

reg\_writeback\_sel,

watch\_stat

);

ALURhsMux alu\_rhs\_mux(

B,

inst\_imm32,

alu\_rhs\_sel,

alu\_rhs

);

RegWriteBackMux rwbm(

inst\_imm32,

alu\_f,

dm\_data\_out,

pc\_next,

reg\_writeback\_sel,

reg\_wb\_data

);

PC pc(

clk,

rst,

pc\_next\_write,

pc\_cur\_write,

pc\_update\_sel,

alu\_f,

inst\_imm32,

pc\_next,

pc\_cur

);

always @(negedge clk) begin

if(ir\_write)begin

ir <= inst;

end

end

InstDecoder1 id1(

ir,

reg\_rs1,

reg\_rs2,

reg\_rd,

inst\_imm32,

inst\_opcode,

inst\_func3,

inst\_func7

);

RegHeap gh(

clk,

regs\_write,

rst,

1'b0, //no test

reg\_rs1,

A,

reg\_rs2,

B,

reg\_rd,

reg\_wb\_data

);

ALU alu(

alu\_lhs,

alu\_rhs,

clk,

alu\_op,

alu\_f,

alu\_flags

);

endmodule

//dm.v

module DM(

input clk,

input mem\_write,

input [31:0] addr, //按字节访问,必须要4字节对齐，否则结果未定义

input [31:0] data\_in, //写入数据

output [31:0] data\_out //读出数据

);

ip\_dm mem (

clk,

mem\_write,

addr[7:2],

data\_in,

data\_out

);

endmodule

//im\_jb.v

module IM4jb (

input clk,

input [31:0] addr,

output [31:0] data\_out

);

ip\_inst\_jb jb (

clk,

addr[7:2],

data\_out

);

endmodule

//top4jb.v

`include "./jb/im\_jb.v"

`include "../include/LedDisplay.v"

module MuxOut(

input [3:0] watch\_stat,

input [31:0] watch\_pc,

input [31:0] watch\_ir,

input [4:0] watch\_rs1,

input [4:0] watch\_rs2,

input [4:0] watch\_rd,

input [31:0] watch\_wb\_data,

input [31:0] watch\_imm32,

input [31:0] watch\_lhs,

input [31:0] watch\_rhs,

input [3:0] watch\_alu\_op,

input [31:0] watch\_alu\_f,

input [3:0] watch\_alu\_flags,

input [31:0] watch\_mdr,

input [1:0] watch\_wb\_data\_sel,

input [1:0] watch\_pc\_update\_sel,

input [3:0] sel,

output reg [31:0] out

);

parameter SHOW\_STAT = 4'b0000;

parameter SHOW\_PC = 4'b0001;

parameter SHOW\_IR = 4'b0010;

parameter SHOW\_RS1 = 4'b0011;

parameter SHOW\_RS2 = 4'b0100;

parameter SHOW\_RD = 4'b0101;

parameter SHOW\_WB\_DATA = 4'b0110;

parameter SHOW\_IMM32 = 4'b0111;

parameter SHOW\_LHS = 4'b1000;

parameter SHOW\_RHS = 4'b1001;

parameter SHOW\_ALU\_OP = 4'b1010;

parameter SHOW\_ALU\_F = 4'b1011;

parameter SHOW\_ALU\_FLAGS = 4'b1100;

parameter SHOW\_MDR = 4'b1101;

parameter SHOW\_WB\_DATA\_SEL = 4'b1110;

parameter SHOW\_PC\_UPDATE\_SEL= 4'b1111;

always @(\*)begin

case(sel)

SHOW\_STAT: out = watch\_stat;

SHOW\_PC: out = watch\_pc;

SHOW\_IR: out = watch\_ir;

SHOW\_RS1: out = watch\_rs1;

SHOW\_RS2: out = watch\_rs2;

SHOW\_RD: out = watch\_rd;

SHOW\_WB\_DATA: out = watch\_wb\_data;

SHOW\_IMM32: out = watch\_imm32;

SHOW\_LHS: out = watch\_lhs;

SHOW\_RHS: out = watch\_rhs;

SHOW\_ALU\_OP: out = watch\_alu\_op;

SHOW\_ALU\_F: out = watch\_alu\_f;

SHOW\_ALU\_FLAGS: out = watch\_alu\_flags;

SHOW\_MDR: out = watch\_mdr;

SHOW\_WB\_DATA\_SEL: out = watch\_wb\_data\_sel;

SHOW\_PC\_UPDATE\_SEL: out = watch\_pc\_update\_sel;

endcase

end

endmodule

module top4riu(

input led\_clk,

input cpu\_clk,

input rst,

input [3:0] led\_mux\_sel,

output [3:0] sel,

output [7:0] seg

);

wire [3:0] watch\_stat;

wire [31:0] watch\_pc;

wire [31:0] watch\_ir;

wire [4:0] watch\_rs1;

wire [4:0] watch\_rs2;

wire [4:0] watch\_rd;

wire [31:0] watch\_wb\_data;

wire [31:0] watch\_imm32;

wire [31:0] watch\_lhs;

wire [31:0] watch\_rhs;

wire [3:0] watch\_alu\_op;

wire [31:0] watch\_alu\_f;

wire [3:0] watch\_alu\_flags;

wire [31:0] watch\_mdr;

wire [1:0] watch\_wb\_data\_sel;

wire [1:0] watch\_pc\_update\_sel;

wire [31:0] inst\_addr;

wire [31:0] inst;

wire dm\_write;

wire [31:0] dm\_addr;

wire [31:0] dm\_data\_in;

wire [31:0] dm\_data\_out;

CPU cpu(

cpu\_clk,

rst,

inst\_addr,

inst,

dm\_write,

dm\_addr,

dm\_data\_in,

dm\_data\_out,

watch\_stat,

watch\_pc,

watch\_ir,

watch\_rs1,

watch\_rs2,

watch\_rd,

watch\_wb\_data,

watch\_imm32,

watch\_lhs,

watch\_rhs,

watch\_alu\_op,

watch\_alu\_f,

watch\_alu\_flags,

watch\_mdr,

watch\_wb\_data\_sel,

watch\_pc\_update\_sel

);

IM4jb im(

cpu\_clk,

inst\_addr,

inst

);

DM dm(

cpu\_clk,

dm\_write,

dm\_addr,

dm\_data\_in,

dm\_data\_out

);

wire [31:0] mux\_out;

MuxOut mux(

watch\_stat,

watch\_pc,

watch\_ir,

watch\_rs1,

watch\_rs2,

watch\_rd,

watch\_wb\_data,

watch\_imm32,

watch\_lhs,

watch\_rhs,

watch\_alu\_op,

watch\_alu\_f,

watch\_alu\_flags,

watch\_mdr,

watch\_wb\_data\_sel,

watch\_pc\_update\_sel,

led\_mux\_sel,

mux\_out

);

LedDisplay led(

led\_clk,

mux\_out,

1'b1,

sel,

seg

);

endmodule

1. FPGA管脚约束代码

（引脚约束文件的内容，备注使用的具体设备）

led\_clk使用全局时钟信号。cpu\_clk使用按键模拟。rst使用按键。led\_mux\_sel[3:0]使用一组逻辑开关。sel[3:0]和seg[7:0]共同驱动数码管

set\_property BITSTREAM.GENERAL.COMPRESS TRUE [current\_design]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[7]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[6]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[5]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[4]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[3]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[2]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[1]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {seg[0]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {sel[3]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {sel[2]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {sel[1]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {sel[0]}]

set\_property PACKAGE\_PIN H19 [get\_ports {seg[7]}]

set\_property PACKAGE\_PIN G20 [get\_ports {seg[6]}]

set\_property PACKAGE\_PIN J22 [get\_ports {seg[5]}]

set\_property PACKAGE\_PIN K22 [get\_ports {seg[4]}]

set\_property PACKAGE\_PIN K21 [get\_ports {seg[3]}]

set\_property PACKAGE\_PIN H20 [get\_ports {seg[2]}]

set\_property PACKAGE\_PIN H22 [get\_ports {seg[1]}]

set\_property PACKAGE\_PIN J21 [get\_ports {seg[0]}]

set\_property PACKAGE\_PIN L21 [get\_ports {sel[3]}]

set\_property PACKAGE\_PIN M22 [get\_ports {sel[2]}]

set\_property PACKAGE\_PIN M21 [get\_ports {sel[1]}]

set\_property PACKAGE\_PIN N22 [get\_ports {sel[0]}]

set\_property -dict {IOSTANDARD LVCMOS18 PACKAGE\_PIN H4} [get\_ports led\_clk]

set\_property PACKAGE\_PIN R4 [get\_ports cpu\_clk]

set\_property IOSTANDARD LVCMOS18 [get\_ports cpu\_clk]

set\_property PACKAGE\_PIN AA4 [get\_ports rst]

set\_property IOSTANDARD LVCMOS18 [get\_ports rst]

set\_property PACKAGE\_PIN T3 [get\_ports {led\_mux\_sel[3]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {led\_mux\_sel[3]}]

set\_property PULLDOWN true [get\_ports {led\_mux\_sel[3]}]

set\_property PACKAGE\_PIN U3 [get\_ports {led\_mux\_sel[2]}]

set\_property PACKAGE\_PIN T4 [get\_ports {led\_mux\_sel[1]}]

set\_property PACKAGE\_PIN V3 [get\_ports {led\_mux\_sel[0]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {led\_mux\_sel[2]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {led\_mux\_sel[1]}]

set\_property IOSTANDARD LVCMOS18 [get\_ports {led\_mux\_sel[0]}]

set\_property PULLDOWN true [get\_ports {led\_mux\_sel[2]}]

set\_property PULLDOWN true [get\_ports {led\_mux\_sel[1]}]

set\_property PULLDOWN true [get\_ports {led\_mux\_sel[0]}]

# 三、测试程序

（10分）

1、CPU测试程序

（汇编语言程序、机器指令代码及对应主存地址、每条指令执行结果）

main:

addi a0, zero, 0x10; #a0=0000\_0010H，数据区域（数组）首址

ori a1, zero, 3 #a1=0000\_0003H，累加的数据个数

xori a2, zero, 0x30; #a2=0000\_0030H，累加和存放的单元

jal BankSum #子程序调用

lw s0, 0(a2) #读出累加和

BankSum:

add t0, a0, zero; #t0=数据区域首址

or t1, a1, zero; #t1=计数器，初始为累加的数据个数

and t2, zero, zero; #t2=累加和，初始清零

L: lw t3, 0(t0); #t3=取出数据

add t2, t2, t3 #累加

addi t0, t0, 4; #移动数据区指针

addi t1, t1, -1; #计数器-1

beq t1, zero, exit; #计数值=0，累加完成，退出循环

j L #计数值≠0，继续累加，跳转至循环体首部

exit: sw t2, 0(a2); #累加和，存到指定单元

jr ra #子程序返回

riulsjb.o: file format elf32-littleriscv

Disassembly of section .text:

00000000 <main>:

0: 01000513 li a0,16

4: 00306593 or a1,zero,3

8: 03004613 xor a2,zero,48

c: 008000ef jal 14 <BankSum>

10: 00062403 lw s0,0(a2)

00000014 <BankSum>:

14: 000502b3 add t0,a0,zero

18: 0005e333 or t1,a1,zero

1c: 000073b3 and t2,zero,zero

00000020 <L>:

20: 0002ae03 lw t3,0(t0)

24: 01c383b3 add t2,t2,t3

28: 00428293 add t0,t0,4

2c: fff30313 add t1,t1,-1

30: 00030463 beqz t1,38 <exit>

34: fedff06f j 20 <L>

00000038 <exit>:

38: 00762023 sw t2,0(a2)

3c: 00008067 ret

2、存储器内容

（指令存储器和数据存储器内容，可附coe文件内容）

//dm.coe

memory\_initialization\_radix=16;

memory\_initialization\_vector= 00000000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A 0000000B 0000000C 0000000D 0000000E 0000000F 00000010 00000011 00000012 00000013 00000014 00000015 00000016 00000017 00000018 00000019 0000001A 0000001B 0000001C 0000001D 0000001E 0000001F 00000020 00000021 00000022 00000023 00000024 00000025 00000026 00000027 00000028 00000029 0000002A 0000002B 0000002C 0000002D 0000002E 0000002F 00000030 00000031 00000032 00000033 00000034 00000035 00000036 00000037 00000038 00000039 0000003A 0000003B 0000003C 0000003D 0000003E 0000003F;

//inst.coe

memory\_initialization\_radix=16;

memory\_initialization\_vector=01000513 00306593 03004613 008000ef 00062403 000502b3 0005e333 000073b3 0002ae03 01c383b3 00428293 fff30313 00030463 fedff06f 00762023 00008067;

# 四、实验结果分析

（20分，仿真验证可以只对CPU模块仿真）

1. 仿真代码

（含仿真源代码、仿真验证方案）

`timescale 1ns / 1ps

`include "./jb/im\_jb.v"

module test4jb();

reg clk = 0;

reg rst = 0;

initial begin

forever begin

#2 clk = ~clk;

end

end

wire [3:0] watch\_stat;

wire [31:0] watch\_pc;

wire [31:0] watch\_ir;

wire [4:0] watch\_rs1;

wire [4:0] watch\_rs2;

wire [4:0] watch\_rd;

wire [31:0] watch\_wb\_data;

wire [31:0] watch\_imm32;

wire [31:0] watch\_lhs;

wire [31:0] watch\_rhs;

wire [3:0] watch\_alu\_op;

wire [31:0] watch\_alu\_f;

wire [3:0] watch\_alu\_flags;

wire [31:0] watch\_mdr;

wire [1:0] watch\_wb\_data\_sel;

wire [1:0] watch\_pc\_update\_sel;

wire [31:0] inst\_addr;

wire [31:0] inst;

wire dm\_write;

wire [31:0] dm\_addr;

wire [31:0] dm\_data\_in;

wire [31:0] dm\_data\_out;

CPU cpu(

clk,

rst,

inst\_addr,

inst,

dm\_write,

dm\_addr,

dm\_data\_in,

dm\_data\_out,

watch\_stat,

watch\_pc,

watch\_ir,

watch\_rs1,

watch\_rs2,

watch\_rd,

watch\_wb\_data,

watch\_imm32,

watch\_lhs,

watch\_rhs,

watch\_alu\_op,

watch\_alu\_f,

watch\_alu\_flags,

watch\_mdr,

watch\_wb\_data\_sel,

watch\_pc\_update\_sel

);

IM4jb im(

clk,

inst\_addr,

inst

);

DM dm(

clk,

dm\_write,

dm\_addr,

dm\_data\_in,

dm\_data\_out

);

initial begin

#10 rst = 1;

#10 rst = 0;

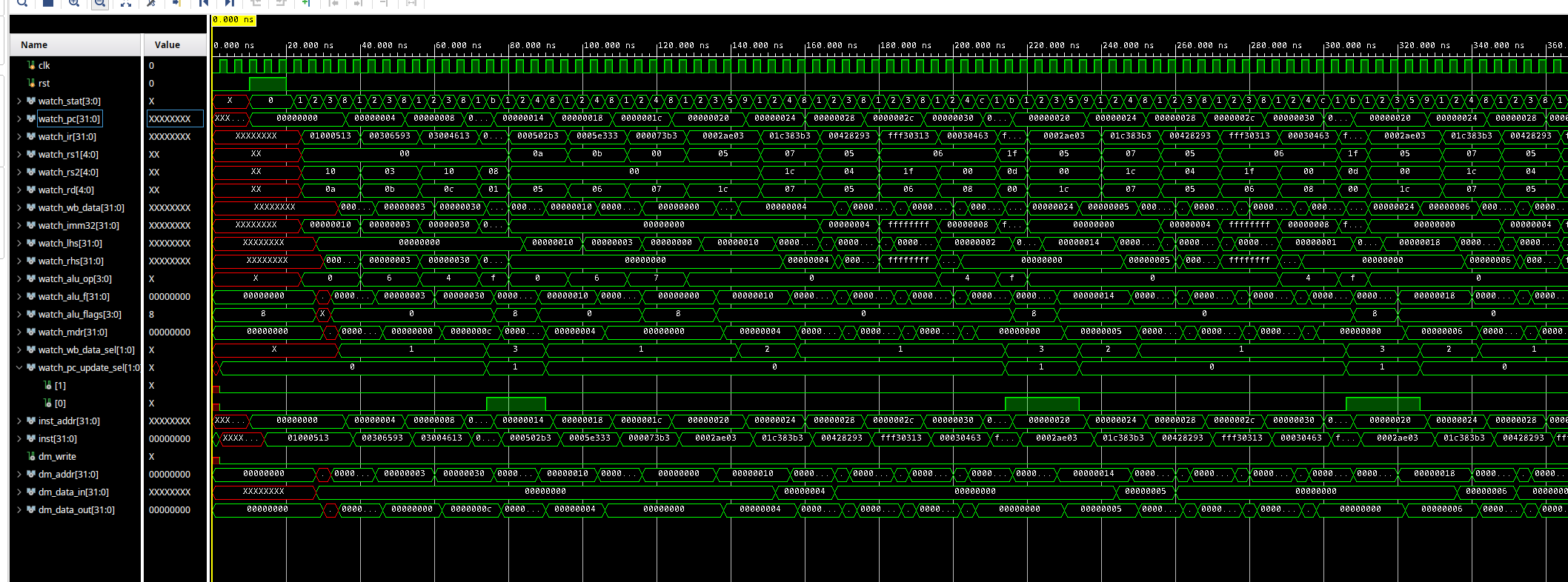
#4000 $finish;

end

endmodule

1. 仿真波形及结果分析

（仿真波形截图，并对仿真波形进行分析）



1. 板级实验操作说明

由上图可知，仿真波形基本符合实验要求

1. 板级实验结果记录

（以表格形式展现，具体见教材上实验结果记录表）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | pc | ir | 汇编代码 | 执行结果 | 下条指令地址 |
| 00 | 00 | 0100\_0513 | addi a0,zero,0x10 | a0=0x10指向数据区域首地址 | 04 |
| 01 | 04 | 0030\_6593 | ori a1,zero,3 | a1=3累加单元个数 | 08 |
| 02 | 08 | 0300\_4613 | xori a2,zero,0x30 | a2=0x30,指向存放数据的地址 | 0C |
| 03 | 0C | 0080\_00ef | jal banksum | pc+=0x8, pc=0x14,调用子程序 | 14 |
| 04 | 14 | 0005\_02b3 | add t0,a0,zero | t0=a0, ,t0=0x10,把a0数据写入t0寄存器 | 18 |
| 05 | 18 | 0005\_e333 | add t1,a1,zero | t1=a1,t1=3把a1数据写入t1寄存器, t1作为计数器 | 1C |
| 06 | 1C | 0000\_73b3 | add t2,zero,zero | t2=0,初始化t2寄存器为0,作为累加器 | 20 |
| 07 | 20 | 0002\_ae03 | lw t3,0(t0) | t3=DM[(t0)], t3=4,访问t0所指向的内存字，把数据存放在t3 | 24 |
| 08 | 24 | 01c3\_83b3 | add t2,t2,t3 | t2+=t3, t2=4把t3中的数据累加到t2 | 28 |
| 09 | 28 | 0042\_8293 | add t0,t0,4 | t0+=4,t0=0x14,指向下一个内存字 | 2C |
| 10 | 2C | fff3\_0313 | add t1,t1,-1 | t1-=1,t1=2,计数器减1 | 30 |
| 11 | 30 | 0003\_0463 | beq t1,zero,exit | 如果t1为0跳转至循环出口 | 34 |
| 12 | 34 | fedf\_f06f | j L | 跳转到循环体首部 | 20 |
| 13 | 20 | 0002\_ae03 | lw t3,0(t0) | t3=DM[(t0)], t3=5,访问t0所指向的内存字，把数据存放在t3 | 24 |
| 14 | 24 | 01c3\_83b3 | add t2,t2,t3 | t2+=t3, t2=9把t3中的数据累加到t2 | 28 |
| 15 | 28 | 0042\_8293 | add t0,t0,4 | t0+=4,t0=0x18,指向下一个内存字 | 2C |
| 16 | 2C | fff3\_0313 | add t1,t1,-1 | t1-=1,t1=1,计数器减1 | 30 |
| 17 | 30 | 0003\_0463 | beq t1,zero,exit | 如果t1为0跳转至循环出口 | 34 |
| 18 | 34 | fedf\_f06f | j L | 跳转到循环体首部 | 20 |
| 19 | 20 | 0002\_ae03 | lw t3,0(t0) | t3=DM[(t0)], t3=6,访问t0所指向的内存字，把数据存放在t3 | 24 |
| 20 | 24 | 01c3\_83b3 | add t2,t2,t3 | t2+=t3, t2=15把t3中的数据累加到t2 | 28 |
| 21 | 28 | 0042\_8293 | add t0,t0,4 | t0+=4,t0=0x1C,指向下一个内存字 | 2C |
| 22 | 2C | fff3\_0313 | add t1,t1,-1 | t1-=1,t1=0,计数器减1 | 30 |
| 23 | 30 | 0003\_0463 | beq t1,zero,exit | 如果t1为0跳转至循环出口 | 38 |
| 24 | 38 | 0076\_2023 | sw t2, 0(a2) | DM[a2]=15把t2累加器中的数据写入a2指向的内存单元 | 3C |
| 25 | 3C | 0000\_8067 | ret | 返回父程序 | 10 |
| 26 | 10 | 0006\_2403 | lw s0,0(a2) | a0=15 读出a0所指向的内存单元的数据(累加和) |  |

1. 实验结果分析与结论

（分析实验结果，给出实验结论）

发现本次实验的jb型指令实现了子程序调用和循环功能说明JB型的设计符合要求，本次设计实验成功

# 五、思考与探索

（10分）

1. 问题与解决方案：

（整个实验过程中发生了什么问题？你是如何解决的。）

1. 仿真结果不符合预期: 我的pc寄存器在取码译码阶段的上升沿更新。而指令存储器也在取码译码阶段的上升沿读出指令。这两者发生了数据竞态使数据出错。改正方法: 让pc寄存器在取码译码阶段的下降沿更新

1. 思考题：

（力所能及，尝试实践或回答教材上的思考与探索题目，至少完成1道）

1. 是。在我的测试程序里cpu先运行到了jal bankmove处。发生了子程序调用。pc跳转到了0x14处继续执行。一直运行到beq t1, zero, exit不成立。在下一处指令j L无条件跳转到了L标签所在的位置。之后一直循环了2次。到t1寄存器数据为0时beq分支跳转跳转成功。跳转到了exit标签处。然后jr ra返回父程序。lw s0，0(a2)读出累加和后有效程序结束

2. jal相对跳转实现的: jal x0, -20;

jalr实现 jalr x0, 32(x0);

# 六、实验心得体会、意见建议

（如果是小组合作，请给出具体分工、分别给出心得体会）

在这个实验中，我学习了如何设计一个基于RISC-V指令集架构的CPU，特别是关于条件分支指令的实现和程序控制的设计。通过实验，我深入了解了程序计数器的作用和流水线架构的原理，并了解了如何实现跳转和分支等指令。

在实验过程中，我遇到了一些挑战。例如，我需要仔细研究RISC-V指令集架构的细节和指令的含义，了解不同的指令格式和操作码。我还需要考虑如何在CPU中实现条件分支的控制流，并解决可能出现的数据冲突和控制信号的生成问题，以确保CPU的正确性和高效性。

通过不断的尝试和调试，我最终成功地实现了JB型指令的CPU设计，并且验证了其正确性和可靠性。这个过程让我感受到了从无到有的设计过程，提高了我的设计和调试能力，并且让我更深入地了解了计算机组成原理的核心概念和基本原理。

总之，这个实验让我受益匪浅，深刻理解了计算机组成原理的基本概念和基本原理，增强了我的计算机系统设计和调试能力，为我的未来学习和职业发展打下了坚实的基础。