

### 本科实验报告

课程名称： 计算机组成与设计

姓 名： 梁晨

学 院： 计算机科学与技术学院专 业： 计算机

邮 箱： liangchenwater@outlook.com

QQ 号： 1749612503

电 话： 15901260806

指导教师： 马德

报告日期： 2021 年 5 月 14 日

# 浙江大学实验报告

# 实验项目名称： lab4-CPU设计

# 学生姓名：梁晨

# 学号： 3180102160

# 实验地点： 紫金港东四 509 室

# 实验日期：2021 年 5 月 14 日

##### 一、实验目的和原理

##### 1.实验目的

1. 运用寄存器传输控制技术

2. 掌握CPU的核心：数据通路组成与控制器

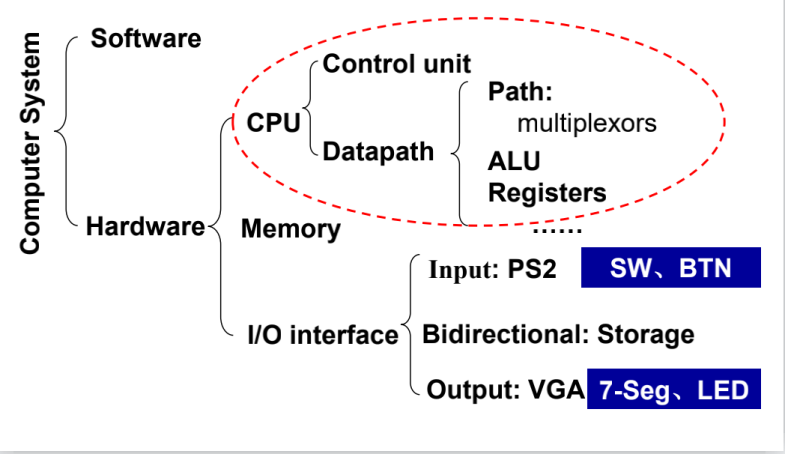
3. 学习测试方案的设计

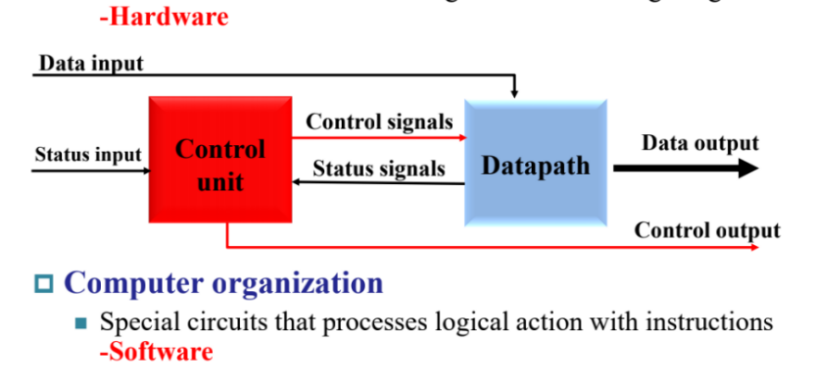
4. 学习测试程序的设计

5. 熟练掌握IP核的使用方法

##### 实验原理

##### 2.1CPU组成结构





##### 2.2CPU主要部件：数据通路Data-path

##### 2.2.1基本功能：

1.具有通用计算功能的算术逻辑部件

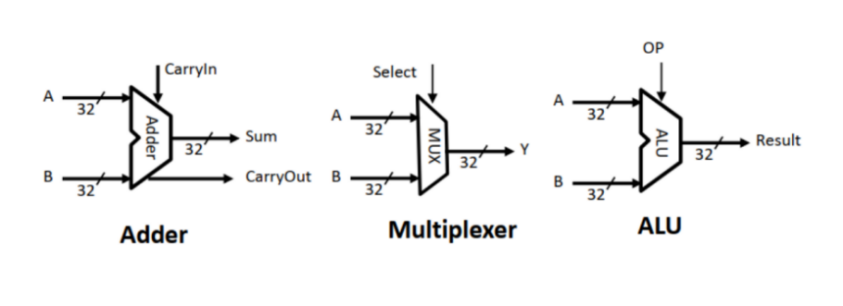
2.具有通用目的寄存器

3.具有通用计数所需的尽可能的路径

##### 截屏2021-05-29 下午8.08.43

##### 2.2.2数据通路部件

1.组合逻辑单元： 加法器、多路选择器、ALU算数运算单元

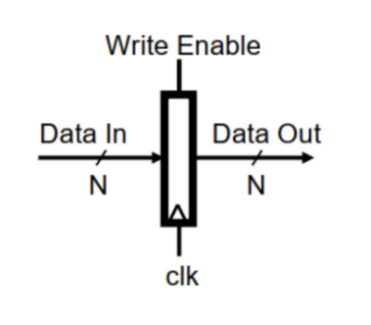


2.时序逻辑单元（状态元件）：Register

• Write Enable:

– 置0，数据输出保持原状态不变

– 置1，在有效时钟边沿到来，数据输出为数据输入值



时序逻辑单元： 寄存器堆

• Register files: 由32个register构成

– 两个32位的数据输出端口：busA、busB

– 一个32位的数据输入端口：busW

• 寄存器读写操作:

– RA (number) 作为地址选择register存储的数据传输到输出端口busA（读）

– RB (number) 作为地址选择register存储的数据传输到输出端口busB（读）

– RW (number) 作为地址选择register接收输入端口busW的数据，当Write

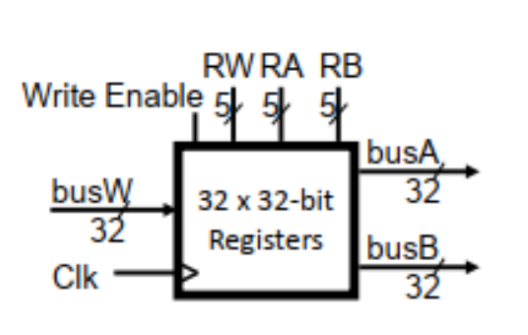
Enable=1且时钟边沿有效时（写）

• 时钟（clk）:

– 写操作时，clk是影响因子，在有效的时钟边沿到来，数据被写入

– 读操作时，clk非影响因子，只要RA、RB有效，经过短暂的器件延迟，数据

即从busA、busB输出（此时属于组合逻辑操作）



时序逻辑单元：

• 存储器

– 一个数据输入端口: DataIn

– 一个数据输出端口: DataOut

• 读写操作:

– 读：Address 作为地址选择存储的数据输出到端口DataOut

– 写：Address 作为地址接收端口DataIn输入的数据，当Write Enable = 1且时钟

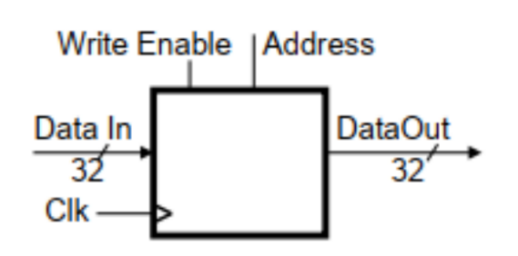
边沿有效

• 时钟(CLK)

– 写操作时，clk是影响因子，在有效的时钟边沿到来，数据被写入

– 读操作时，clk非影响因子，只要Address有效，经过短暂的器件延迟，数据即

从DataOut输出（此时属于组合逻辑操作）



##### 2.3CPU主要部件：控制器SCPU\_ctrl

##### 2.3.1基本功能：

1.指令译码

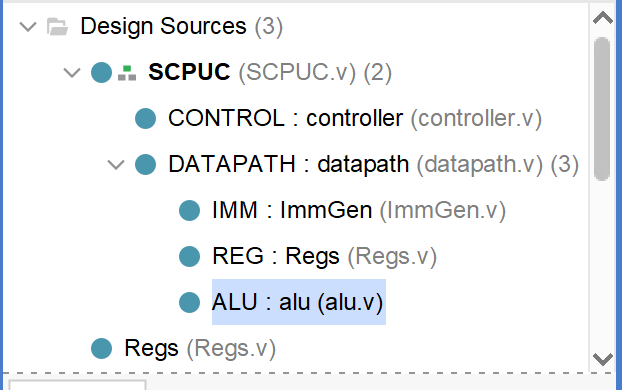
2.产生操作控制信号：ALU运算控制

3.产生指令所需的路径选择



##### 二、实验内容

##### top结构



##### 2.Datapath

##### 2.1ALU

module alu(

input wire [31:0] A,B,

input wire [2:0] ALU\_operation,

output reg [31:0] res,

output reg zero

);

always@(\*) begin

case(ALU\_operation)

3'b000:begin

res = A&B; end

3'b001:begin

res= A|B; end

3'b010:begin

res=A+B; end

3'b011:begin

res=A^B;end

3'b100:begin

res=~(A|B);end

3'b101:begin

res=A >> B; end

3'b110:begin

res=A-B;end

3'b111:begin

res= $signed(A) < $signed(B);end

endcase

zero = (res==0)?1:0;

end

endmodule

##### 2.2Regs

module Regs( input clk,

input rst,

input [4:0] Rs1\_addr,

input [4:0] Rs2\_addr,

input [4:0] Wt\_addr,

input [31:0]Wt\_data,

input RegWrite,

output [31:0] Rs1\_data,

output [31:0] Rs2\_data

);

reg[31:0] register[1:31];

integer i;

assign Rs1\_data=(Rs1\_addr==0)?0:register[Rs1\_addr];

assign Rs2\_data=(Rs2\_addr==0)?0:register[Rs2\_addr];

always@(posedge clk or posedge rst)

begin if(rst==1) for (i=1;i<32;i=i+1) register[i]<=0;

else if ((Wt\_addr!=0)&&(RegWrite==1)) register[Wt\_addr]<=Wt\_data;

end

endmodule

##### 2.3 ImmGen

module ImmGen(

input wire [1:0] ImmSel,

input wire [31:0] inst\_field,

output reg [31:0] Imm\_out

);

always @ \* begin

case(ImmSel)

2'b00: Imm\_out={ {20 {inst\_field[31]} }, inst\_field[31:20]}; //addi/lw (I)

2'b01: Imm\_out={ {20 {inst\_field[31]} }, inst\_field[31:25], inst\_field[11:7]}; //sw(S)

2'b10: Imm\_out={ {19 {inst\_field[31]} }, inst\_field[31], inst\_field[7], inst\_field[30:25], inst\_field[11:8], 1'b0}; //beq(SB)

2'b11: Imm\_out={ {11 {inst\_field[31]} }, inst\_field[31], inst\_field[19:12], inst\_field[20], inst\_field[30:21],1'b0}; // jal(UJ)

endcase

end

endmodule

##### 2.4 Datapath

module datapath(

input wire clk,

input wire rst,

//control signals

input wire Jump,

input wire Branch,

input wire [1:0] MemtoReg,

input wire ALUSrc\_B,

input wire RegWrite,

input wire [1:0] ImmSel,

input wire [2:0] ALU\_operation,

//input data and inst

input wire [31:0] Data\_in,

input wire [31:0] Inst\_in,

//output

output reg [31:0] PC\_out,

output reg [31:0] Data\_out,

output reg [31:0] ALU\_out

);

//generate immdiates

wire [31:0] Imm\_out;

ImmGen IMM(.ImmSel(ImmSel),

.inst\_field(Inst\_in),

.Imm\_out(Imm\_out));

//calculate PC

wire [31:0] PC\_next;

wire [31:0] PC\_jump;

wire rs\_rt\_equal;

assign PC\_next = PC\_out+4;

assign PC\_jump = PC\_out+Imm\_out;

always @ (posedge clk or posedge rst) begin

if(rst) begin

PC\_out<=0; end

else if(Jump) begin

PC\_out<= PC\_jump;end

else if(Branch) begin

PC\_out<= rs\_rt\_equal? PC\_jump : PC\_next; end

else begin

PC\_out <= PC\_next; end

end

//reg

wire[4:0] Rs1\_addr;

wire[4:0] Rs2\_addr;

wire[4:0] Wt\_addr;

assign

Rs1\_addr=Inst\_in[19:15],

Rs2\_addr=Inst\_in[24:20],

Wt\_addr=Inst\_in[11:7];

reg [31:0] Wt\_data;

wire [31:0] Rs1\_data;

wire [31:0] Rs2\_data;

always @ (\*) begin

case(MemtoReg)

2'b00: Wt\_data = ALU\_out;

2'b01: Wt\_data = Data\_in;

2'b10: Wt\_data=PC\_next;

2'b11: Wt\_data=PC\_next;

endcase

end

Regs REG(

.clk(clk),

.rst(rst),

.Rs1\_addr(Rs1\_addr),

.Rs2\_addr(Rs2\_addr),

.Wt\_addr(Wt\_addr),

.Wt\_data(Wt\_data),

.RegWrite(RegWrite),

.Rs1\_data(Rs1\_data),

.Rs2\_data(Rs2\_data)

);

assign rs\_rt\_equal = (Rs1\_data==Rs2\_data);

//alu

reg[31:0] alu\_a;

reg[31:0] alu\_b;

wire zero;

wire [31:0] alu\_out;

always@(\*)begin

alu\_a=Rs1\_data;

case(ALUSrc\_B)

1'b0:alu\_b=Rs2\_data;

1'b1: alu\_b=Imm\_out;

endcase

end

alu ALU(

.A(alu\_a),

.B(alu\_b),

.ALU\_operation(ALU\_operation),

.res(alu\_out),

.zero(zero)

);

//Data\_out and ALU\_out

always@(\*) begin

Data\_out = Rs2\_data;

ALU\_out = alu\_out;

end

endmodule

##### controller

module controller(

input [4:0] OPcode,

input [2:0] Fun3,

input Fun7,

input MIO\_ready,

output reg[1:0] ImmSel,

output reg ALUSrc\_B,

output reg[1:0] MemtoReg,

output reg Jump,

output reg Branch,

output reg RegWrite,

output reg MemRW,

output reg [2:0] ALU\_Control,

output reg CPU\_MIO

);

reg [1:0] ALUop;

`define CPU\_ctrl\_signals {ALUSrc\_B, MemtoReg, RegWrite, MemRW, Branch ,Jump,ALUop, ImmSel}

always @\*begin

case(OPcode)

5'b01100: begin `CPU\_ctrl\_signals={1'b0,2'b00,1'b1,1'b0,1'b0,1'b0,2'b00,2'b00}; end //ALU

5'b00000: begin `CPU\_ctrl\_signals={1'b1,2'b01,1'b1,1'b0,1'b0,1'b0,2'b01,2'b00}; end //Load

5'b01000: begin `CPU\_ctrl\_signals={1'b1,2'b00,1'b0,1'b1,1'b0,1'b0,2'b10,2'b01}; end //Store

5'b11000: begin `CPU\_ctrl\_signals={1'b0,2'b00,1'b0,1'b0,1'b1,1'b0,2'b11,2'b10}; end //Beq

5'b11011: begin `CPU\_ctrl\_signals={1'b0,2'b10,1'b1,1'b0,1'b0,1'b1,2'b00,2'b11}; end //Jump

5'b00100: begin `CPU\_ctrl\_signals={1'b1,2'b00,1'b1,1'b0,1'b0,1'b0,2'b01,2'b00}; end //Imm

endcase

end

wire[3:0] Fun;

assign Fun={Fun3,Fun7};

always@ \* begin

case(ALUop)

2'b10:ALU\_Control=3'b010; //addr addition

2'b11:ALU\_Control= 3'b110 ; //sub for beq

2'b00:

case(Fun)

4'b0000: ALU\_Control=3'b010; //add

4'b0001: ALU\_Control=3'b110 ;//sub

4'b1110: ALU\_Control=3'b000;//and

4'b1100: ALU\_Control=3'b001;//or

4'b0100: ALU\_Control=3'b111; //slt

4'b1010: ALU\_Control=3'b101; //srl

4'b1000: ALU\_Control=3'b011; //xor

default: ALU\_Control=3'bx;

endcase

2'b01:

case(OPcode)

5'b00000: ALU\_Control=3'b010;//load

5'b00100: //Imm

case(Fun3)

3'b010: ALU\_Control=3'b111;//slti

3'b000: ALU\_Control=3'b010;//addi

3'b100: ALU\_Control=3'b011;//xori

3'b110: ALU\_Control=3'b001;//ori

3'b111: ALU\_Control=3'b000;//andi

3'b101: ALU\_Control=3'b101;//srli

//jalr

default:ALU\_Control=3'bx;

endcase

default: ALU\_Control=3'bx;

endcase

endcase

end

endmodule

##### SCPU

module SCPUC(

input wire clk,

input wire rst,

input wire [31:0] Inst\_in,

input wire [31:0] Data\_in,

input wire MIO\_ready,

output wire MemRW,

output wire CPU\_MIO,

output wire [31:0] Addr\_out,

output wire [31:0] Data\_out,

output wire [31:0] PC\_out

);

wire[4:0] OPcode;

wire[2:0] Fun3;

wire Fun7;

assign OPcode=Inst\_in[6:2],

Fun3=Inst\_in[14:12],

Fun7=Inst\_in[30];

wire[1:0] ImmSel;

wire ALUSrc\_B;

wire[1:0] MemtoReg;

wire Jump;

wire Branch;

wire RegWrite;

wire[2:0] ALU\_Control;

controller CONTROL(

.OPcode(OPcode),

.Fun3(Fun3),

.Fun7(Fun7),

.MIO\_ready(MIO\_ready),

.ImmSel(ImmSel),

.ALUSrc\_B(ALUSrc\_B),

.MemtoReg(MemtoReg),

.Jump(Jump),

.Branch(Branch),

.RegWrite(RegWrite),

.MemRW(MemRW),

.ALU\_Control(ALU\_Control),

.CPU\_MIO(CPU\_MIO)

);

datapath DATAPATH(

.ImmSel(ImmSel),

.ALUSrc\_B(ALUSrc\_B),

.MemtoReg(MemtoReg),

.Jump(Jump),

.Branch(Branch),

.RegWrite(RegWrite),

.ALU\_operation(ALU\_Control),

.Inst\_in(Inst\_in),

.Data\_in(Data\_in),

.clk(clk),

.rst(rst),

.ALU\_out(Addr\_out),

.Data\_out(Data\_out),

.PC\_out(PC\_out)

);

endmodule

##### 仿真模拟

##### 测试代码：

start:addi x1, x0, 1 //x1=1;

slt x2, x0, x1 //x2=1;

add x3,x2,x2 //x3=2

add x4,x3,x3 //x4=4

add x5, x4, x2 //x5=5

add x6, x5, x5 //x6=a

sub x7, x6, x5 //x7=5

and x8,x7,x5 //x8=5

sub x10, x8,x6 //x10=ffff fffb

or x11, x5, x6 // x11= f

or x12, x11, x7 //x12= f

slt x13,x7, x7 //x13 = 0

lw x5,0x34(x0) // x5=5555 5555

lw x5,0X48(x0) //x5 = aaaa aaaa

sw x5, 0x34(x0)

lw x14 0x34(x0) //x14=aaaa aaaa

and x1, x14,x14 //x1=aaaa aaaa

jal x1, here//x1为pc\_next

add x0,x0,x0

add x0,x0,x0

add x0,x0,x0

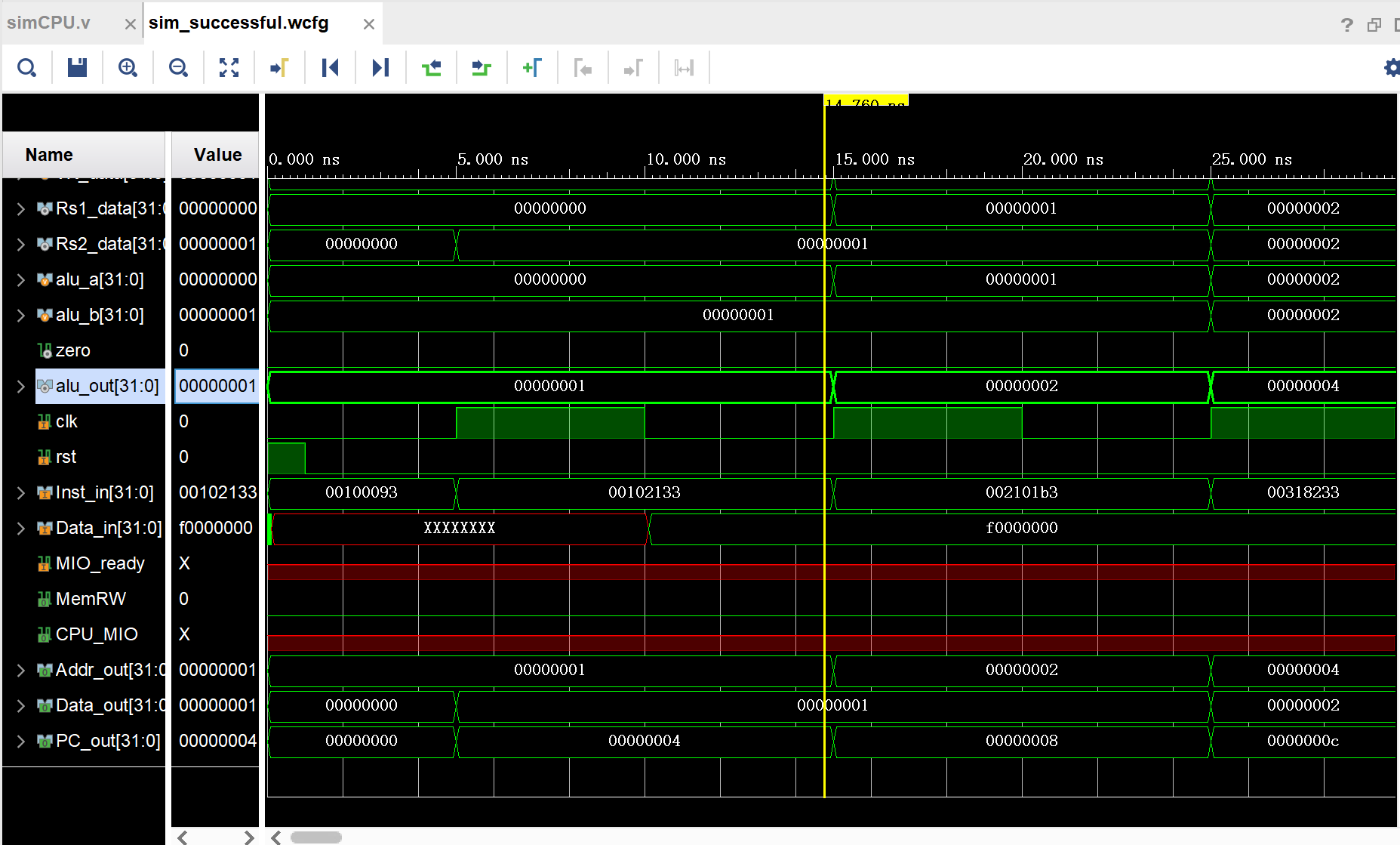
here: addi x1, x1, 1// x1=49

beq x0,x0 start

##### 仿真结果：

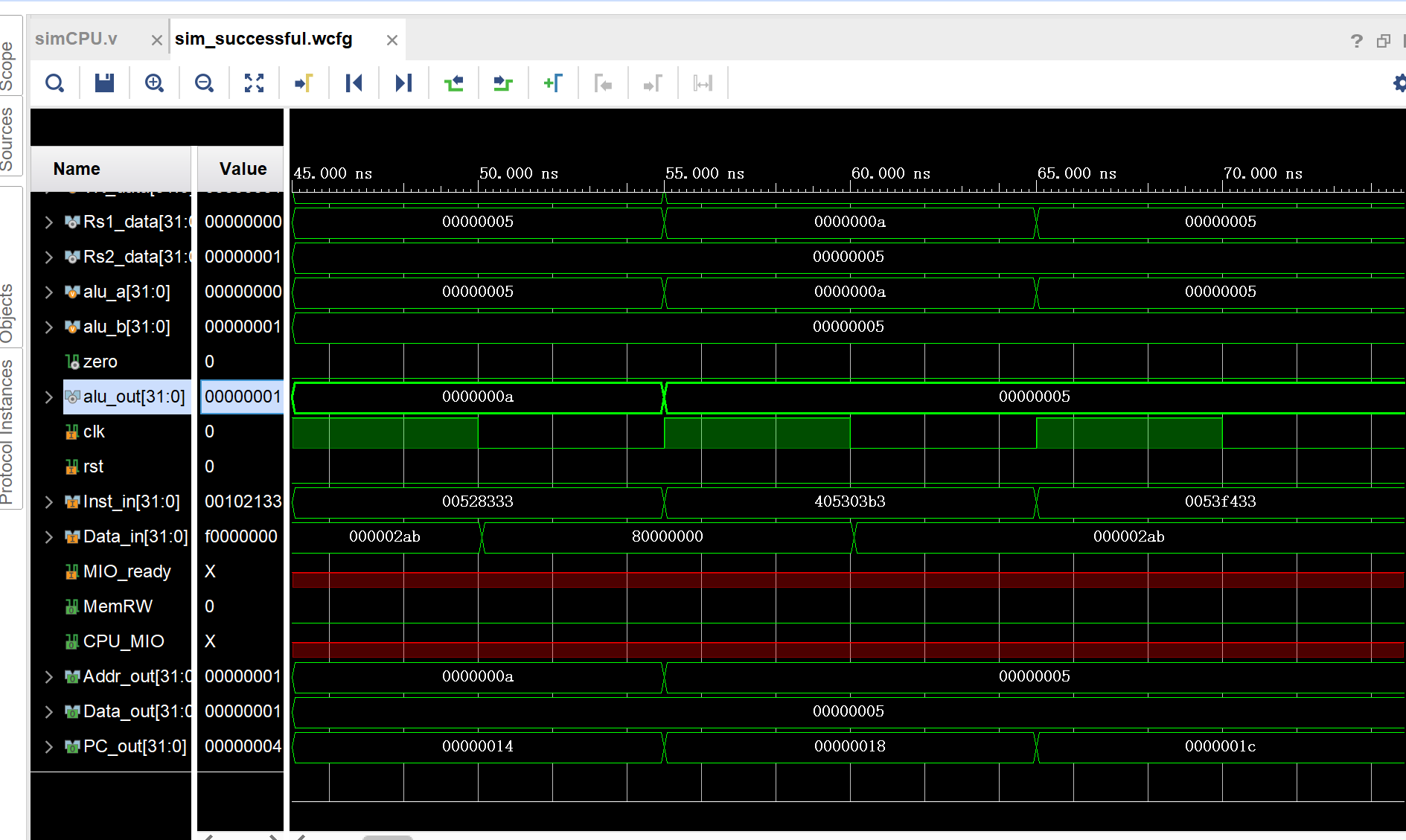
addi指令，aluout输出1

add指令，aluout输出2，4 结果正确

add x6, x5, x5 //x6=a

sub x7, x6, x5 //x7=5

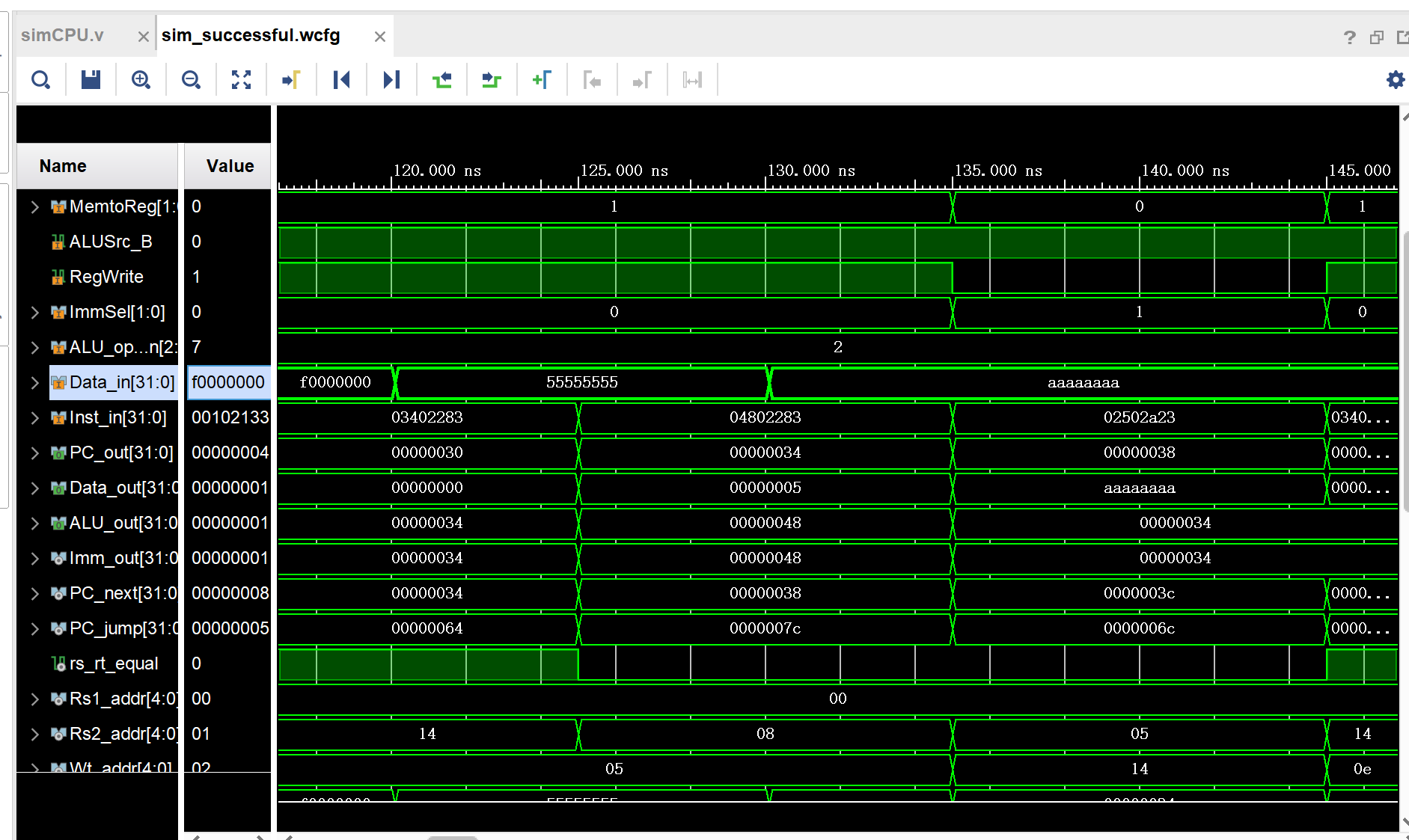
aluout输出a，5 结果正确



lw x5,0x34(x0) // x5=5555 5555

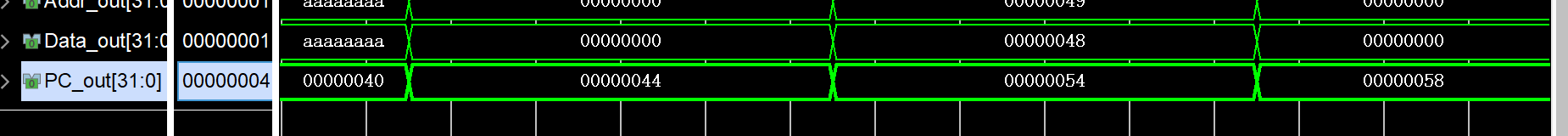
lw x5,0X48(x0) //x5 = aaaa aaaa

datain输出55555555，aaaaaaaa 结果正确

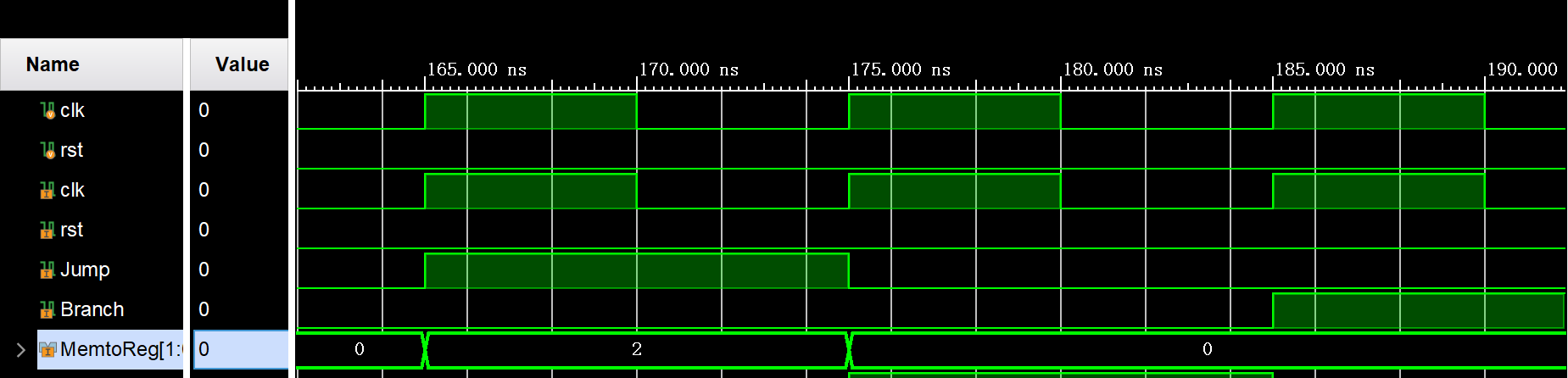


jal x1, here//x1为pc\_next

pcout从44跳转到54

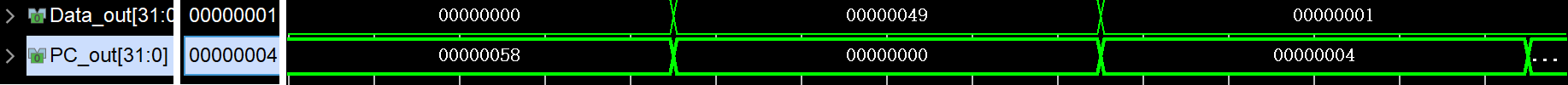


memtoreg变成2，其余时间都是0或1



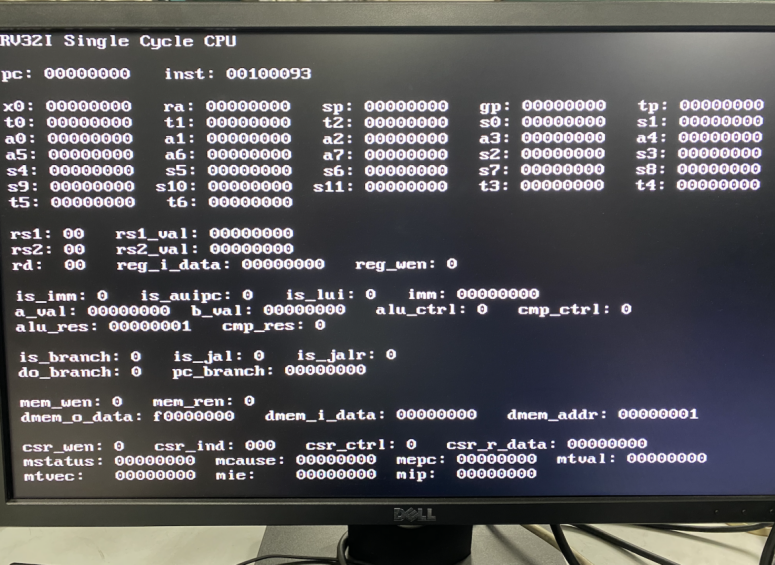
beq x0,x0 start

pcout跳转至0

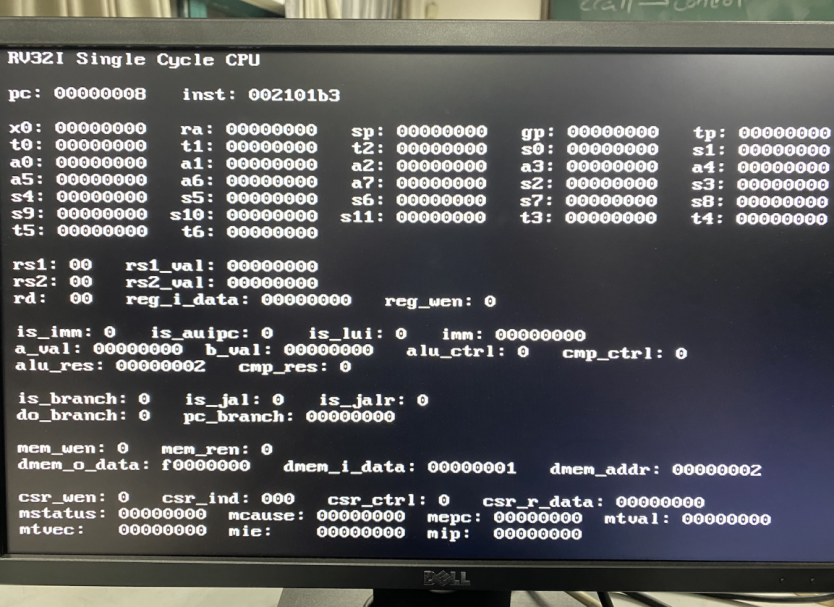


##### 实验结果

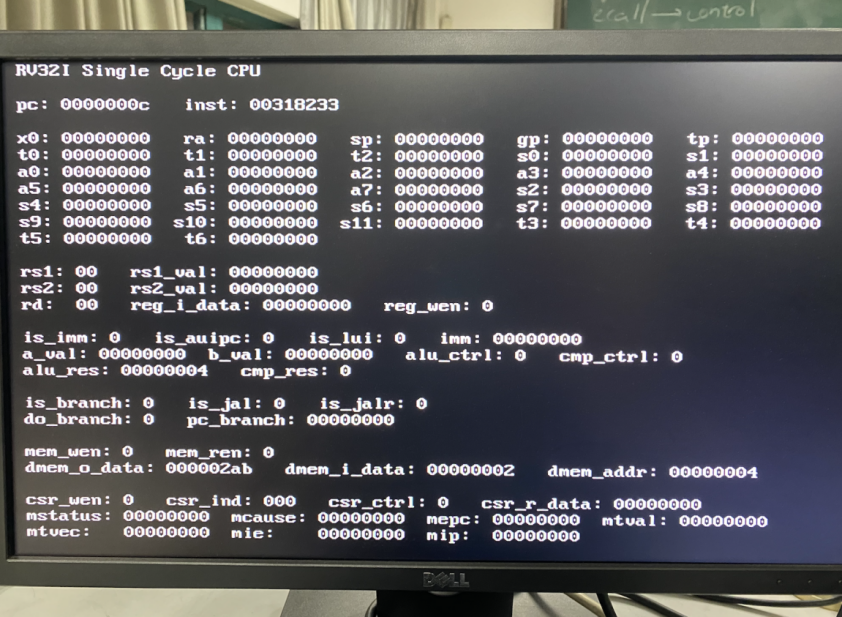
addi x1, x0, 1 //x1=1;



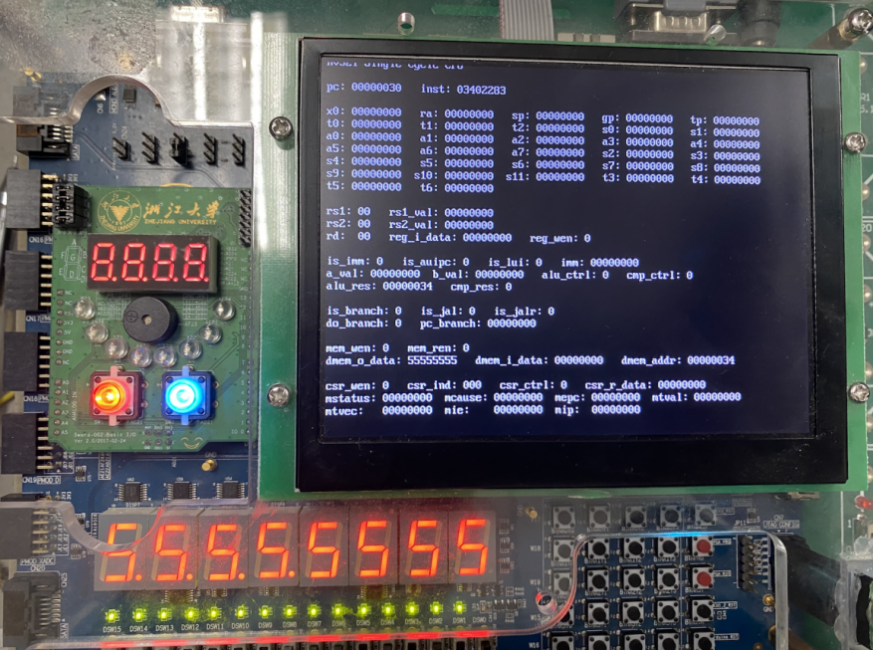
add x3,x2,x2 //x3=2



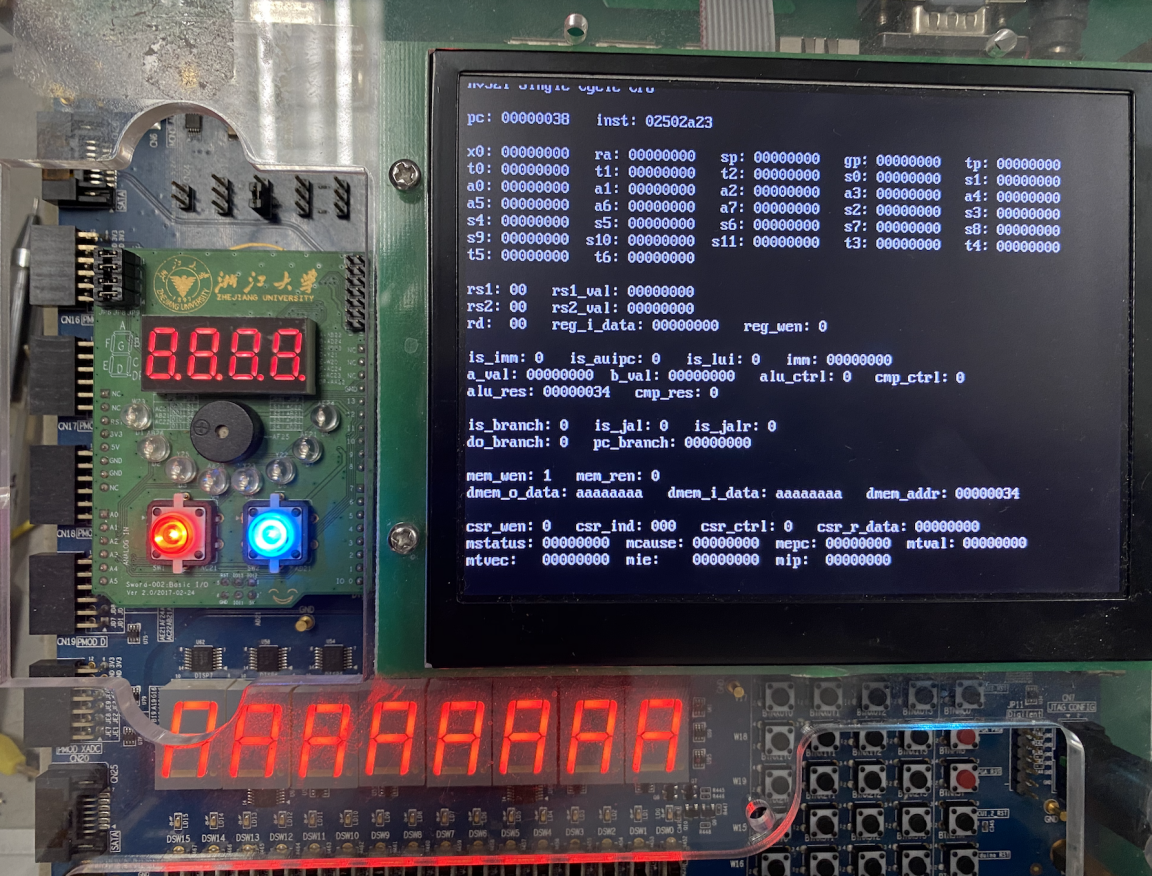
add x4,x3,x3 //x4=4



lw x5,0x34(x0) // x5=5555 5555



lw x5,0X48(x0) //x5 = aaaa aaaa



##### 心得体会

我个人认为本实验是课程中最为重要、也是难度最大的一个实验。其一是因为从之前的实验到本次实验跨度较大。实验0-3都是“热身”实验，并不涉及复杂的硬件电路设计，模块较少，调试也并不费力，而实验四则较为综合、过程趋于复杂，不但要求我们对理论知识了然于心，而且要求我们熟练掌握vivado的使用和verilog硬件设计语言的调试技巧。其二是因为之后的流水线、Cache实验都要在单周期CPU的基础上完成，所以在做实验四时必须用心，为今后的学习打好基础。

本次实验我遇到的最大的困难是助教提供的vivado 17的相关IP和我使用的vivado 20不兼容。用设计原理图搭建好Datapath之后会报各种奇奇怪怪的错误，调试这些bug费了很长时间，最终无果，故全换成verilog代码来实现。

对比原理图，verilog代码不但简洁清晰、方便理解，也省下了开发的时间。