**计算机组成原理 实验报告**

姓名： 杨佳熹 学号：PB17000050 实验日期：2019-5-16

**一、实验题目：**

Lab05 多周期MIPS-cpu

**二、实验目的：**

设计实现多周期MIPS-CPU，可执行如下指令：

add, sub, and, or, xor, nor, slt

addi, andi, ori, xori, slti

lw, sw

beq, bne, j

**三、实验平台：**

Vivado

**四、实验过程：**

本实验分为两部分，分别为DDU和processor，以DDU为顶层模块并与外界接口连接，processor为多周期cpu模块。

DDU模块：

module DDU(clk,rst,in,cont,step,mem,inc,dec,led,an,seg);

input in;

input clk;//100M

input rst;

input cont,step;//cpu运行方式

input mem;//查看MEM/RF

input inc,dec;//增加/减小待查看地址

output [15:0]led;

output [7:0]an;//数码管使能

output [6:0]seg;

//output o\_dp;

reg [4:0] reg\_addr;

reg run;//运行cpu

reg [31:0]mem\_addr;

reg lock;

wire [31:0] data;

//wire clk\_run;//可被冻结的时钟

wire [31:0] pc,pc\_init;

wire [31:0] mem\_data,reg\_data;

wire clk\_10;

wire clk\_1;

reg [30:0]cnt;

reg [3:0] cnt2;

wire [31:0]ins;

wire clk\_r;

reg flag;

always @(posedge clk)

cnt <= (cnt!= 10\_000\_000) ? cnt + 1: 1;

assign clk\_10 = (cnt <= 5\_000\_000);

always @(posedge clk\_10)

cnt2 <= (cnt2 != 10) ? cnt2 + 1 : 1;

assign clk\_1 = (cnt2 <= 5);

seg\_display seg\_display(clk,data,seg,an);

processor processor (clk\_1,rst,pc\_init,pc,ins,reg\_addr,reg\_data,mem\_addr,mem\_data,run);

assign pc\_init = 32'hffffffff;

assign led = (mem)? {mem\_addr[7:0],pc[7:0]}:{3'b0,reg\_addr,pc[7:0]};

assign data = (in)? ins:(mem)? mem\_data : reg\_data;

always @(posedge clk\_1)

if(rst)

reg\_addr <= 5'b0;

else if (mem == 0)

if(inc)

reg\_addr <= reg\_addr + 1;

else if(dec)

reg\_addr <= reg\_addr - 1;

always @(posedge clk\_1)

if(rst)

mem\_addr <= 32'b0;

else if (mem == 1)

if(inc)

mem\_addr <= mem\_addr + 1;

else if(dec)

mem\_addr <= mem\_addr - 1;

//assign clk\_r=(run)? clk\_1:0;

//always@(step or cont or pc)

//begin

// if(flag&&step&&~cont)

// run=1;

// else if (~flag&&~cont)

// run=0;

// else if(cont)

// run=1;

//end

//always@(run or rst)

//begin

//if(rst)

//flag=1;

//if(run)

//flag=0;

//else

//flag=1;

//end

always@(posedge clk\_1,posedge rst)

if (rst) begin

run <= 0;

lock <= 0;

end

else if (cont) begin

run <= 1;

//lock <= lock;

end

else if( ~lock) begin//not locked

if( step ) begin

run <= 1;

lock <= 1;

end

else begin

run <= 0;

lock <= 0;

end

end

else begin//locked

run <= 0;

lock <= (step) ? 1 : 0;//lock retore

end

endmodule

值得注意的是，需要实现一个小的“锁”，控制run有效时间。其余部分按照实验指导实现，另外加入一个显示指令功能以便更好地检查错误。

时钟选用了两个，一快一满。快时钟用于分频显示，慢时钟用于指令连续执行。具体时钟降速参考前几次实验。

分频显示模块：

module seg\_display(clk,i\_data,o\_seg,o\_an);

input clk;//100M

input [31:0]i\_data;

output reg [6:0]o\_seg;

output reg [7:0]o\_an;

wire clk\_5hz;

reg [2:0]state;

reg [25:0]cnt;

reg [3:0]number;

always @(posedge clk)

cnt <= (cnt!= 200\_000) ? cnt + 1: 1;

assign clk\_5hz = (cnt!=1);

always @(posedge clk\_5hz)

begin

o\_an <= ~( 1 << state);

state <= state + 1;

end

always @(posedge clk\_5hz)

case(state)

3'd0: number <= i\_data[3:0];

3'd1: number <= i\_data[7:4];

3'd2: number <= i\_data[11:8];

3'd3: number <= i\_data[15:12];

3'd4: number <= i\_data[19:16];

3'd5: number <= i\_data[23:20];

3'd6: number <= i\_data[27:24];

3'd7: number <= i\_data[31:28];

endcase

always @(\*)

case(number)

4'h0 : o\_seg = 7'b100\_0000; //显示"0"

4'h1 : o\_seg = 7'b111\_1001; //显示"1"

4'h2 : o\_seg = 7'b010\_0100; //显示"2"

4'h3 : o\_seg = 7'b011\_0000; //显示"3"

4'h4 : o\_seg = 7'b001\_1001; //显示"4"

4'h5 : o\_seg = 7'b001\_0010; //显示"5"

4'h6 : o\_seg= 7'b000\_0010; //显示"6"

4'h7 : o\_seg = 7'b111\_1000; //显示"7"

4'h8 : o\_seg = 7'b000\_0000; //显示"8"

4'h9 : o\_seg = 7'b001\_0000; //显示"9"

4'ha : o\_seg = 7'b000\_0100;//显示"A"

4'hb : o\_seg = 7'b000\_0011;//显示"b"

4'hc : o\_seg = 7'b100\_0110;//显示"c"

4'hd : o\_seg = 7'b010\_0001;//显示"d"

4'he: o\_seg = 7'b000\_0110;//显示"E"

4'hf: o\_seg = 7'b000\_1110;//显示"F"

endcase

endmodule

较之前的有所改进，把时钟频率改成100mhz

Processor部分：

按照实验指导书给出的数据通路和所有模块的示意图，设计出多周期cpu，主要分为如下模块，下图为改进的数据通路图：

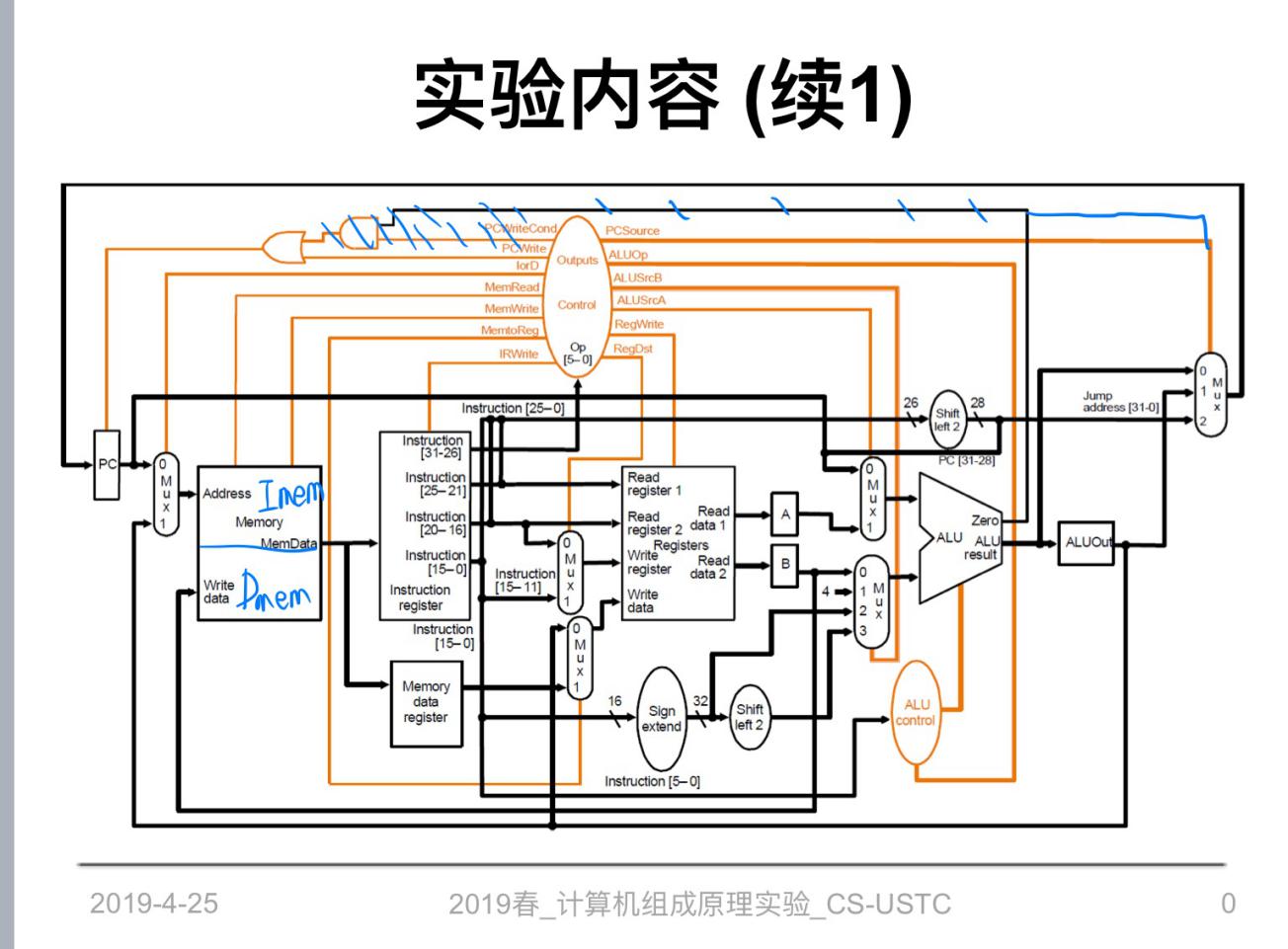


图1：多周期cpu数据通路

PC模块：

module PC(clk,zero,PCWriteCont,PCWrite,ipc,opc,pcinit,rst);

input clk,zero,PCWriteCont,PCWrite,rst;

input [31:0] ipc,pcinit;

output reg [31:0] opc;

wire enpc;

reg [31:0] ippc;

assign enpc=(PCWrite) ;//| (PCWriteCont & zero)

always@(posedge enpc or posedge rst )

begin

if(rst)

opc=pcinit;

else if(enpc)

opc=ipc;

else if(!enpc)

opc=opc;

end

Endmodule

主要用于pc的改变，这里pc为地址并可以在指令存储器中读出指令。我这里只检测PCWrite的变化来改变pc，与原理图有些不同，主要是避免pc提前改变。pcinit设为第一条指令的前一个地址来保证jump顺利执行。

Controlunit模块：

module controlunit(rst,clk,OpCode,PCWriteCond,PCWrite,lorD,MemRead,MemWrite,MemtoReg,IRWrite,PCSource,ALUOp,ALUSrcB,ALUSrcA,RegWrite,RegDst,run);

input rst,clk,run;

input [5:0] OpCode;

output reg PCWriteCond,PCWrite,lorD,MemRead,MemWrite,MemtoReg,IRWrite;

output reg ALUSrcA,RegWrite,RegDst;

output reg [1:0] ALUOp,ALUSrcB,PCSource;

reg [3:0]next\_state,state;

always @(posedge clk) begin

if(rst) state <= 4'hf;

else state <= next\_state;

end

always @(state,run )

begin

case (state)

4'hf: next\_state = (run) ? 4'h0 : 4'hf;//idle

4'h0: next\_state = 4'h1;

4'h1: begin

case (OpCode)

6'b000000: next\_state = 4'h6;//r\_type

6'b100011: next\_state = 4'h2;//lw

6'b101011: next\_state = 4'h2;//sw

6'b000100: next\_state = 4'h8;//BEQ

6'b000101: next\_state = 4'hc;//BNE

6'b000010: next\_state = 4'h9;//J

default: next\_state = 4'ha;//immediate

endcase

end

4'h2:begin

case (OpCode)

6'b100011: next\_state = 4'h3;//lw

6'b101011: next\_state = 4'h5;//sw

endcase

end

4'h3: next\_state = 4'h4;

4'h4: next\_state = (run) ? 4'h0 : 4'hf;

4'h5: next\_state = (run) ? 4'h0 : 4'hf;

4'h6: next\_state = 4'h7;

4'h7: next\_state = (run) ? 4'h0 : 4'hf;

4'h8: next\_state = (run) ? 4'h0 : 4'hf;

4'h9: next\_state = (run) ? 4'h0 : 4'hf;

4'ha: next\_state = 4'hb;

4'hb: next\_state = (run) ? 4'h0 : 4'hf;

4'hc: next\_state = (run) ? 4'h0 : 4'hf;

default next\_state = (run) ? 4'h0 : 4'hf;

endcase

end

always@(posedge clk)

begin

// PCWriteCond=0;

// PCWrite=0;

// lorD=0;

// MemRead=0;

// MemWrite=0;

// MemtoReg=0;

// IRWrite=0;

// ALUSrcA=0;

// RegWrite=0;

// RegDst=0;

// PCSource=2'b0;

// ALUOp=2'b00;

// ALUSrcB=2'b00;

case(next\_state)

4'h0:begin

PCWriteCond <=0;

PCWrite <=1;

lorD <=0;

MemRead <=1;

MemWrite <=0;

IRWrite <=1;

PCSource <=2'b00;

ALUOp <=2'b00;

ALUSrcB <=2'b01;

ALUSrcA <=0;

RegWrite <=0;

end

4'h1:begin

ALUOp <=2'b00;

ALUSrcB <=2'b11;

ALUSrcA <=0;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'h2:begin

ALUOp <=2'b00;

ALUSrcB <=2'b10;

ALUSrcA <=1;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'h3:begin

lorD <=1;

MemRead <=1;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'h4:begin

MemtoReg <=1;

RegWrite <=1;

RegDst <=0;

PCSource <=2'b00;

PCWrite <=0;

MemWrite <=0;

end

4'h5:begin

lorD <=1;

MemWrite <=1;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

end

4'h6:begin

ALUOp <=2'b10;

ALUSrcB <=2'b00;

ALUSrcA <=1;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'h7:begin

MemtoReg <=0;

RegWrite <=1;

RegDst <=1;

PCSource <=2'b00;

PCWrite <=0;

MemWrite <=0;

end

4'h8:begin

PCWriteCond <=1;

PCSource <=2'b01;

ALUOp <=2'b01;

ALUSrcB <=2'b00;

ALUSrcA <=1;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'h9:begin

PCWrite <=0;

PCSource <=2'b10;

RegWrite <=0;

MemWrite <=0;

end

4'ha:begin

ALUOp <=2'b10;

ALUSrcB <=2'b10;

ALUSrcA <=1;

PCSource <=2'b00;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

4'hb:begin

MemtoReg <=0;

RegWrite <=1;

RegDst <=0;

PCSource <=2'b00;

PCWrite <=0;

MemWrite <=0;

end

4'hc:begin

PCWriteCond <=1;

PCSource <=2'b01;

ALUOp <=2'b11;

ALUSrcB <=2'b00;

ALUSrcA <=1;

PCWrite <=0;

RegWrite <=0;

MemWrite <=0;

end

default PCWrite <= 0;

endcase

end

//NextState NextState(cstate,OpCode,nstate);

//OutPutFunc OutPutFunc(clk,cstate,PCWriteCond,PCWrite,lorD,MemRead,MemWrite,MemtoReg,IRWrite,PCSource,ALUOp,ALUSrcB,ALUSrcA,RegWrite,RegDst);

Endmodule

按照实验指导书的时序逻辑状态图实现，总共设出13个状态，在指导书的基础上加入了bne跳转状态，run=0状态实现单步执行，和立即数操作状态。用最基本的三段式编写状态机，如下图所示。

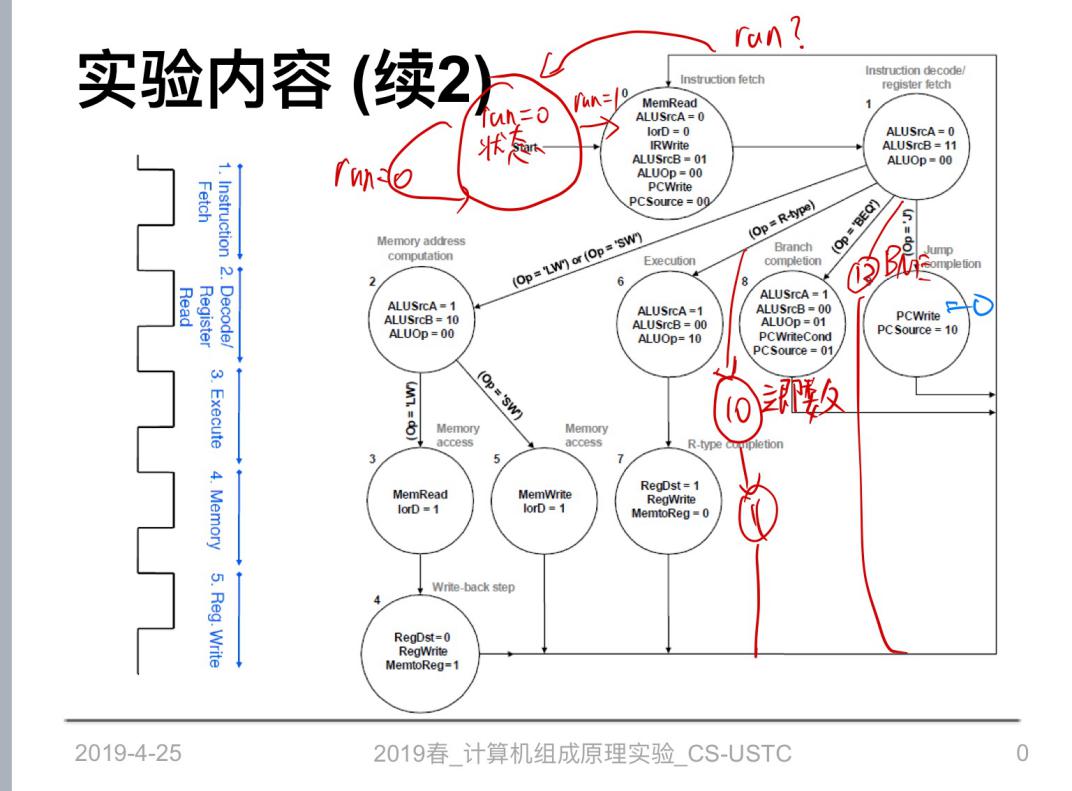


图2：状态图

Imem：

module iMem(address,MemData);

input [31:0]address;

output [31:0]MemData;

dist\_mem\_gen\_0 dist\_mem\_gen\_0(

.a (address[7:0]),

.spo (MemData)

);

endmodule

利用只读存储器存指令，将提供的测试程序翻译成机器码改成coe文件并存入。IP核设置如下图

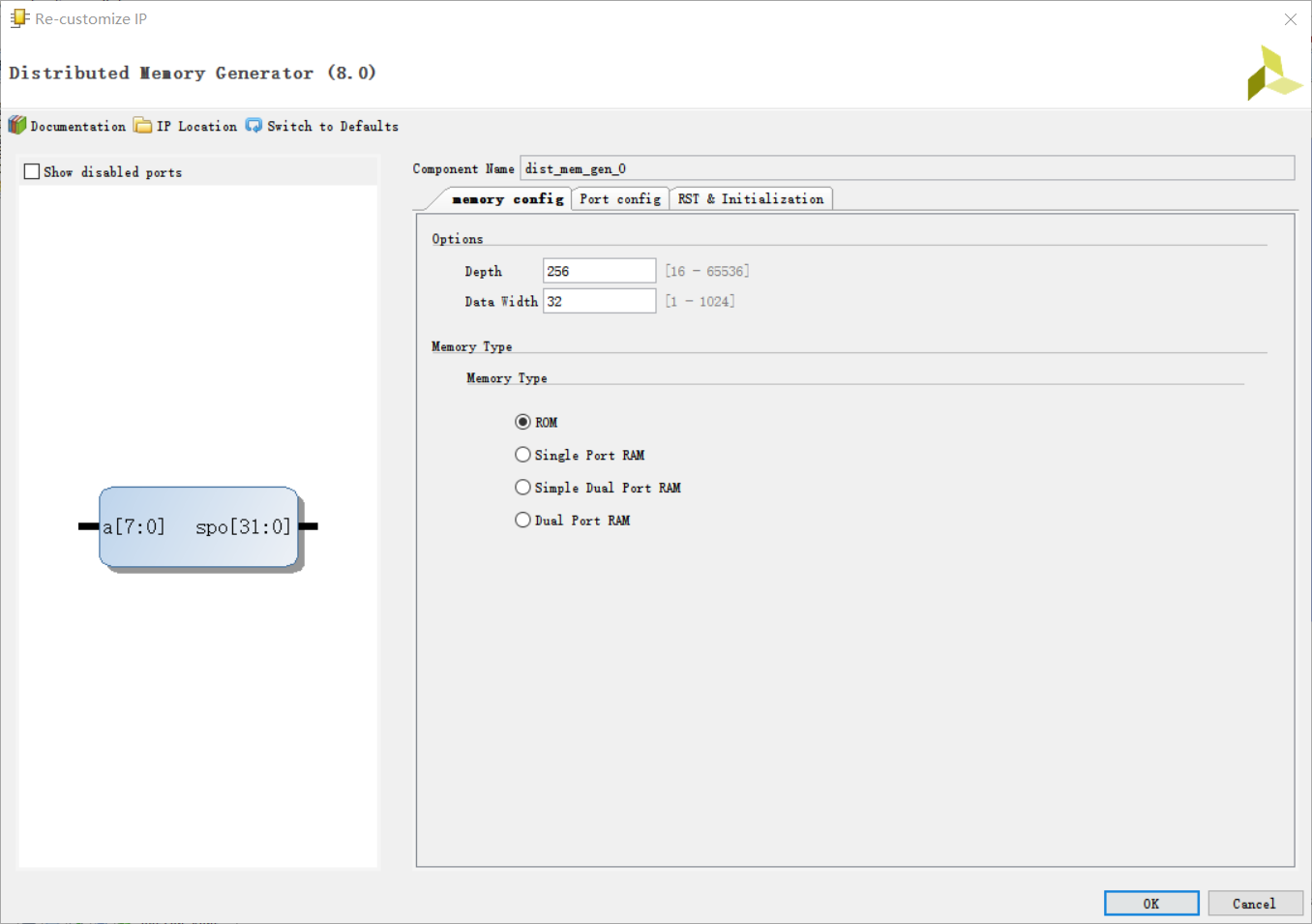


图3：IP核设置

Dmem模块：

module dMem(rst,lord,address,MemRead,MemWrite,Writedata,MemData,mem\_addr,mem\_data);

input lord;

input rst;

input MemRead,MemWrite;

input [31:0] address,mem\_addr;

input [31:0] Writedata;

output [31:0] MemData,mem\_data;

wire [31:0] MemData2;

reg [31:0] MemData1;

reg [31:0] dmem[255:0];

integer i;

always@(\*)

if (rst)

begin

for(i = 0;i < 256;i = i + 1)

dmem[i] = 0;

dmem[12] = 32'h8;

dmem[16] = 32'h1;

dmem[20] = 32'h6;

dmem[24] = 32'hfffffff8;

dmem[28] = 32'h1;

dmem[32] = 32'h3;

dmem[36] = 32'h5;

end

else if(MemWrite)

dmem[ address[7:0] ] = Writedata;

always@(\*)

begin

if(MemRead)

MemData1 = dmem[ address[7:0] ];

end

assign MemData = MemData1;

assign mem\_data= dmem[mem\_addr];

endmodule

初始时利用rst设置初始值，因为测试时需要数据内存中的数据进行比较。

除此之外，加入异步读操作，支持从外界读取。

InsReg模块：

module InsReg (IRWrite,ins,in1,in2,in3,in4);

input IRWrite;

input [31:0]ins;

output [5:0] in1;

output [4:0] in2,in3;

output [15:0]in4;

reg [31:0]instruction;

always@(ins)

if(1)

instruction=ins;

assign in1=instruction[31:26];

assign in2=instruction[25:21];

assign in3=instruction[20:16];

assign in4=instruction[15:0];

Endmodule

这一模块主要用于指令的分割，分别用于立即数计算和alucontrol信号的选择，运用assign实现。

数据分割模块：

module A\_DR(clk,Ai,Ao);

input clk;

input [31:0]Ai;

output reg [31:0]Ao;

always@(posedge clk)

Ao=Ai;

endmodule

module B\_DR(clk,Bi,Bo);

input clk;

input [31:0]Bi;

output reg [31:0]Bo;

always@(posedge clk)

Bo=Bi;

endmodule

module ALU\_DR(clk,ALUi,ALUo);

input clk;

input [31:0]ALUi;

output reg [31:0]ALUo;

always@(posedge clk)

ALUo=ALUi;

endmodule

module MemData\_DR(clk,MemDatai,MemDatao);

input clk;

input [31:0]MemDatai;

output reg [31:0]MemDatao;

always@(posedge clk)

MemDatao=MemDatai;

Endmodule

根据实验指导书上的原理图，需要加入几个用于存储数据的buffer来等时钟沿让数据流通，以便实现多周期。实现方法是always时钟上升沿敏感数据赋值。

数据选择器：

module DataSelector\_3to1\_32(A, B, C, Control, Result, zero);

input [31:0] A, B, C;

input zero;

input [1:0] Control;

output reg[31:0] Result;

always @(Control or A or B or C or zero)

begin

case({Control,zero})

3'b001:Result = A;

3'b011:Result = B;

3'b101:Result = C;

3'b000:Result = A;

3'b010:Result = A;

3'b100:Result = C;

endcase

end

endmodule

module DataSelector\_4to1(A, B, C, D, Control, Result);

input [31:0] A, B, C, D;

input [1:0]Control;

output reg[31:0] Result;

always @(Control or A or B or C or D) begin

case(Control)

2'b00: Result = A;

2'b01: Result = B;

2'b10: Result = C;

2'b11: Result = D;

default: Result = 0;

endcase

end

endmodule

module DataSelector\_2to1\_32(A, B, Control, Result);

input [31:0] A, B;

input Control;

output [31:0] Result;

assign Result = (Control == 1'b0 ? A : B);

endmodule

module DataSelector\_2to1\_5(A, B, Control, Result);

input [4:0] A, B;

input Control;

output [4:0] Result;

assign Result = (Control == 1'b0 ? A : B);

Endmodule

用到了四个数据选择器，分别为32位三选一，四选一，二选一和5位二选一。其中三选一用于pc选择，根据pcsource和zero变量的取值选择适当的接口输入到pc中。与指导书上的数据通路不同的是，我把zero加入到选择器中直接控制pc来源，如果zero等于2来源为pc+1，beq/bne，jump，如果zero等于0，来源为pc+1，pc+1，jump，这样可以保证pc跳转准确。

寄存器模块：

module RegFile (rs, rt, rd, i\_data, RegWrite, o\_data\_1, o\_data\_2, read, readdata, rst, clk);

input clk;

input [4:0] rs, rt, rd, read;

input rst;

input [31:0] i\_data;

input RegWrite;

output [31:0] o\_data\_1, o\_data\_2,readdata;

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

integer i;

// always@(rst)

// begin

// begin

// // for(i=0;i<32;i = i + 1)

// register[i] = 0;

// end

// end

assign o\_data\_1 = register[rs];

assign o\_data\_2 = register[rt];

assign readdata = register[read];

always @(posedge clk or posedge rst) begin

if(rst)

begin

// 只需要确定零号寄存器的值就好，$0恒等于0

for(i=0;i<32;i = i + 1)

register[i] = 0;

end

else if ((rd != 0) && (RegWrite == 1)) begin

register[rd] = i\_data;

end

end

endmodule

寄存器模块是根据之前实验进行改变后的代码，除了两端口异步读和一端口同步写之外，加入了DDU模块需要的读数据操作。

移位模块：

module Shiftleft\_1(Ai,Ao);

input [25:0]Ai;

output [27:0]Ao;

assign Ao=Ai;

endmodule

module Shiftleft\_2(Ai,Ao);

input [31:0]Ai;

output [31:0]Ao;

assign Ao=Ai<<2;

Endmodule

利用verilog的运算符求出移位后的数据。

ALU模块：

module ALUcontrol(ALUOp,opcode,ALUControl);

input [1:0] ALUOp;

input [31:0] opcode;

output reg [2:0] ALUControl;

wire [5:0]alucontrol;

assign alucontrol[0]=opcode[0];

assign alucontrol[1]=opcode[1];

assign alucontrol[2]=opcode[2];

assign alucontrol[3]=opcode[3];

assign alucontrol[4]=opcode[4];

assign alucontrol[5]=opcode[5];

always@(ALUOp)

begin

case(ALUOp)

2'b00:ALUControl=3'b000;//jia

2'b01:ALUControl=3'b001;//beq

2'b10:

begin

case(opcode[31:26])

6'b000000:

begin

case(alucontrol)

6'b100000:ALUControl=3'b000;//add

6'b100010:ALUControl=3'b001;//sub

6'b100100:ALUControl=3'b110;//and

6'b101010:ALUControl=3'b011;//slt

6'b100101:ALUControl=3'b101;//or

6'b100110:ALUControl=3'b111;//xor

6'b100111:ALUControl=3'b010;//nor

endcase

end

6'b001000:ALUControl=3'b000;//addi

6'b001100:ALUControl=3'b110;//andi

6'b001101:ALUControl=3'b101;//ori

6'b001110:ALUControl=3'b010;//xori

6'b001010:ALUControl=3'b011;//slti

endcase

end

2'b11:ALUControl=3'b100;//bne

endcase

end

endmodule

module ALU(A, B, ALUControl, zero, result);

input [31:0] A, B;

input [2:0] ALUControl;

output zero;

output reg [31:0] result;

assign zero = (result? 0 : 1);

always @(A or B or ALUControl) begin

case(ALUControl)

3'b000: result = A + B;

3'b001: result = A - B;

3'b010: result = ~(A | B);

// 3'b010: begin

// if (A < B &&(( A[31] == 0 && B[31]==0) || (A[31] == 1 && B[31]==1))) result = 1;

// else if (A[31] == 0 && B[31]==1) result = 0;

// else if (A[31] == 1 && B[31]==0) result = 1;

// else result = 0;

// end

3'b011: result = (A < B ? 1 : 0);

3'b100: result = (A == B ? 1 : 0);

3'b101: result = A | B;

3'b110: result = A & B;

3'b111: result = (~A & B) | (A & ~B);

default: result = 0;

endcase

end

Endmodule

为实现立即数和rtype指令一起实现，把指令的全部32位输入，再利用ALUcontrol算出对应的ALU控制指令，并进行计算。另外包括BNE和BEQ的判等或判不等运算，并对应得出zero的值。

**五、实验结果：**

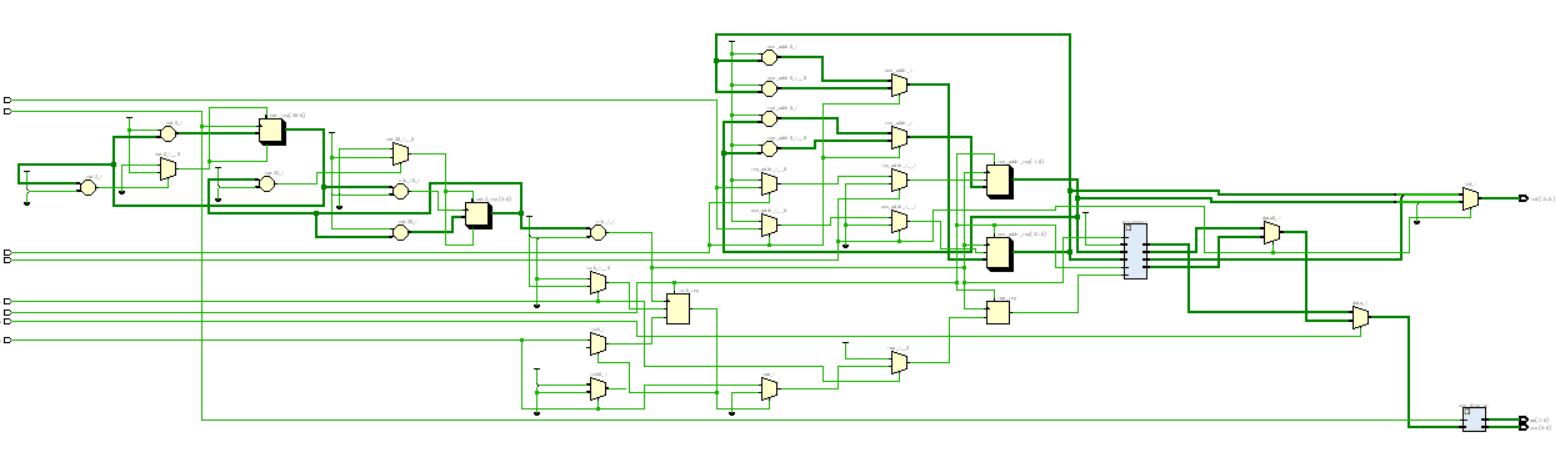
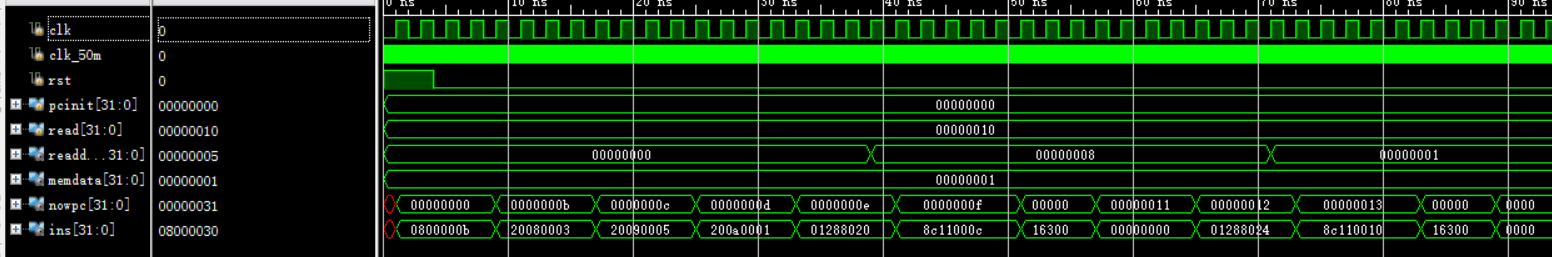
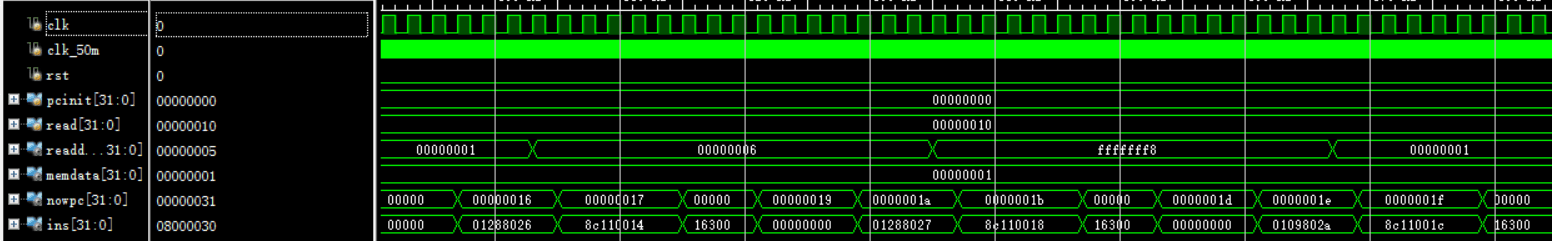
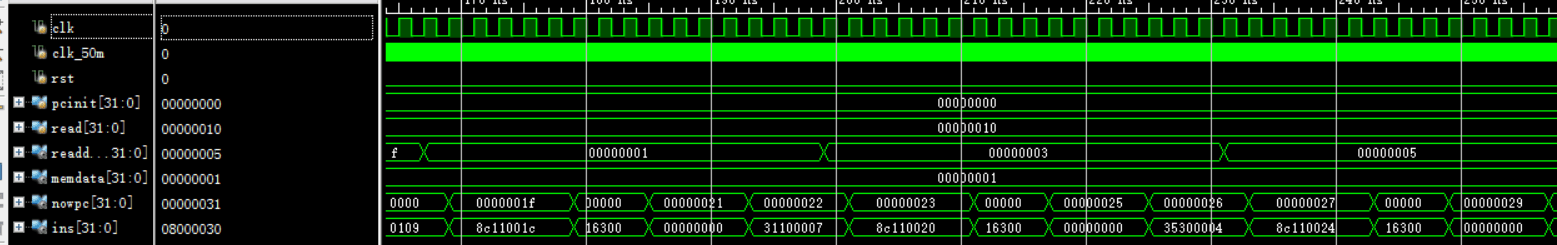


图4：schematic结果







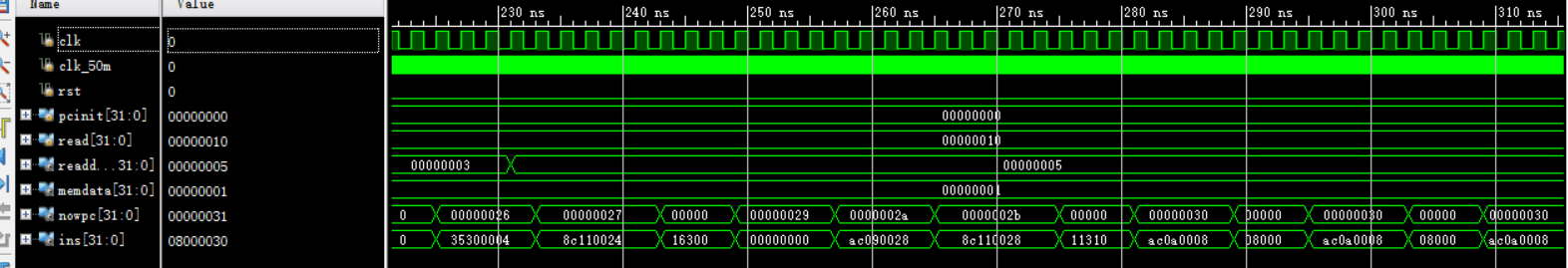


图5：仿真测试结果

结果分析：当指令为0是当作rtype指令执行，不会影响内存以及寄存器中的数据，其他类型指令都可以在仿真结果中正确体现，并可以看到最后可以在success的语段中循环执行。

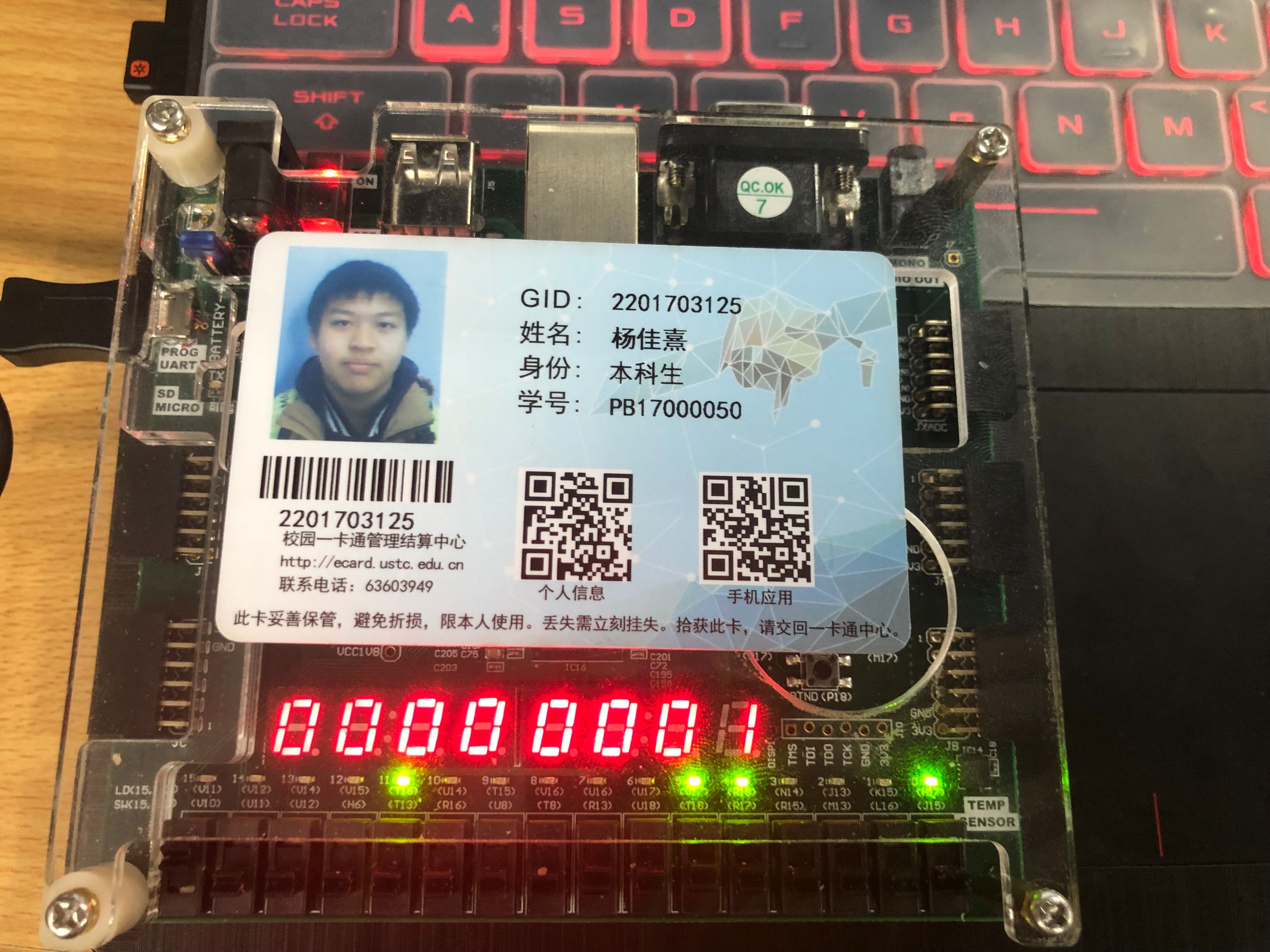


图6：下载结果

如图所示最后内存中地址为8的数据为一，是程序运行成功标识。

**六、心得体会：**

本次实验难度提升一个层次，考查了我们对多周期cpuu基本结构的掌握程度。相比较于之前的实验，这次需要的模块多很多，并且需要分成多个源文件存放以便更好地debug。根据指导书上面的数据通路和时序状态机，基本的框架可以很直观的建立，包括其中多周期需要的数据分割，但是由于已给的状态图指令不全，需要以此基础上加入三个状态，这也让我们复习了状态机的基本结构。所以通过本次实验，我们更加熟悉了多周期cpu的功能模块，模拟了cpu中含有的寄存器，alu等模块，另外我们也更加了解多周期cpu的工作原理。