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：

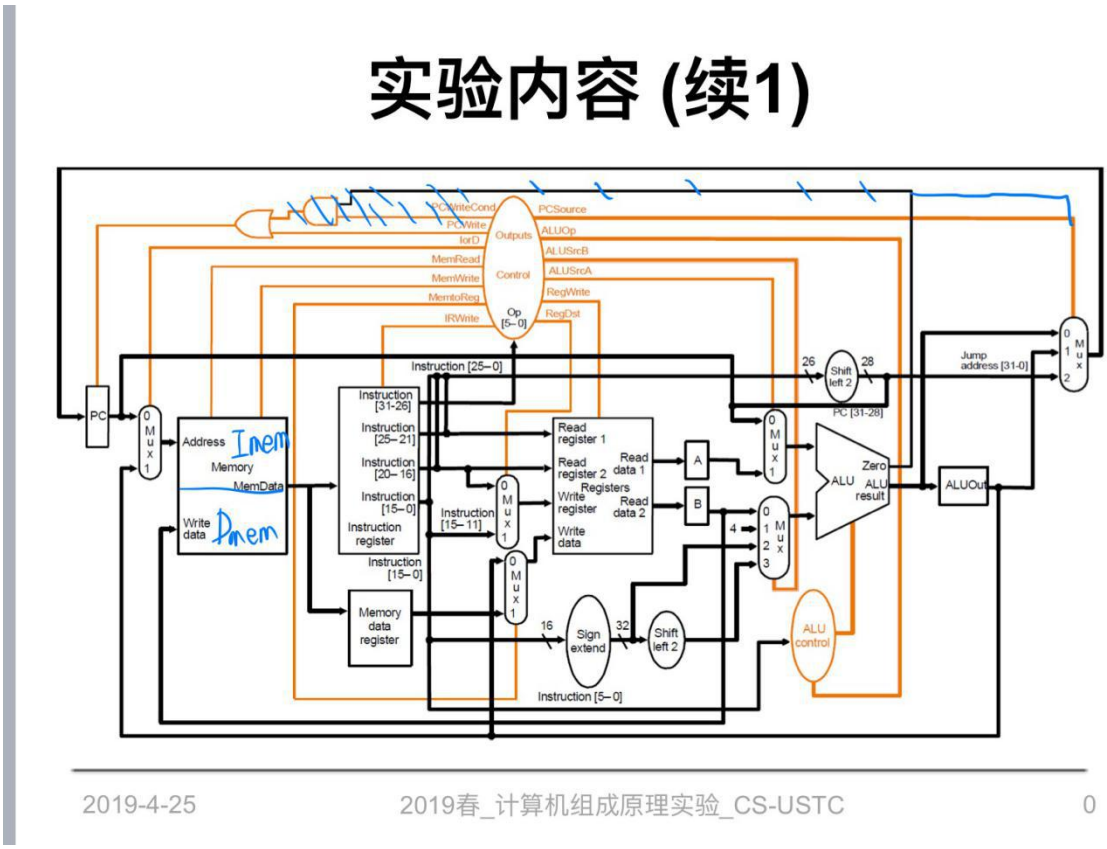

图 1：多周期 cpu 数据通路

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,IorD,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。

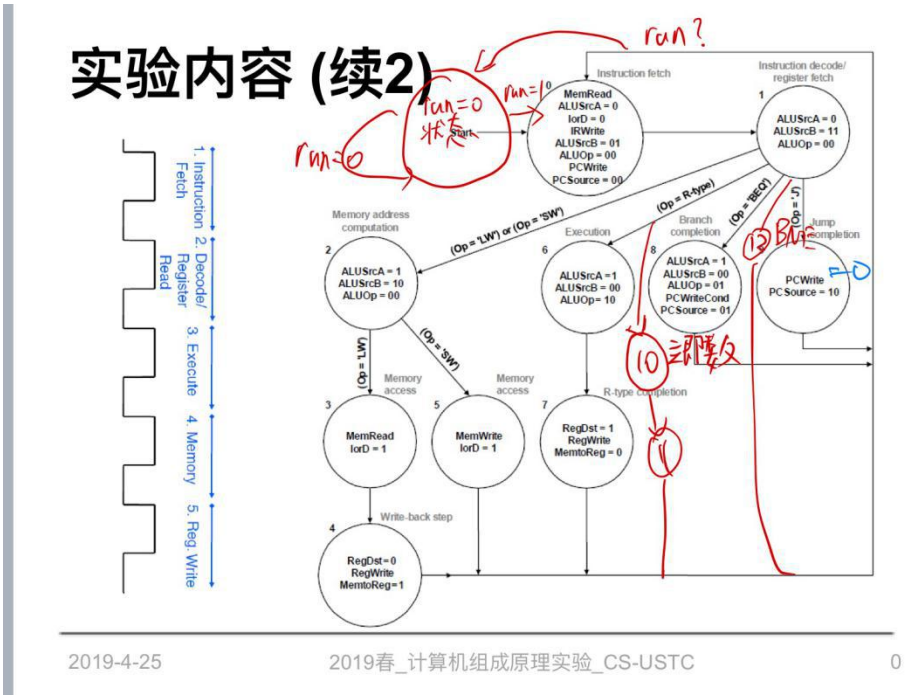

图 2：状态图

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

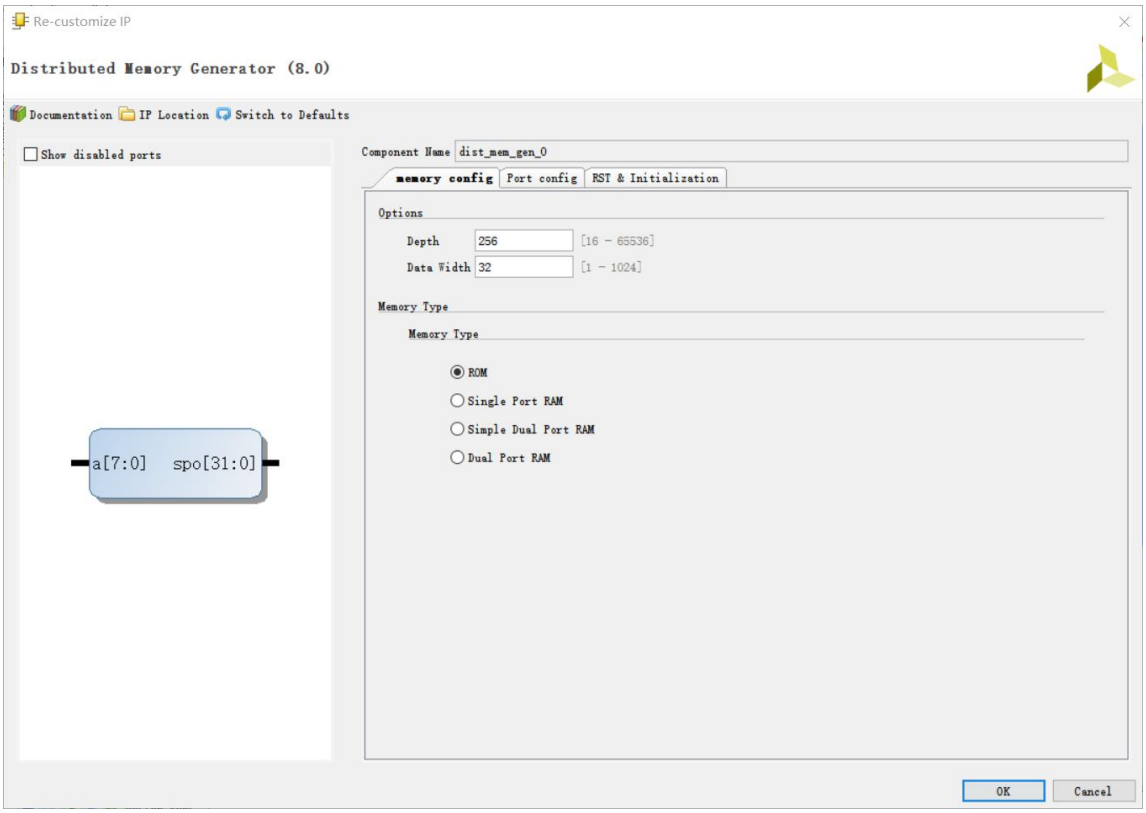

图 3：IP 核设置

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'hffffffff8;
        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：

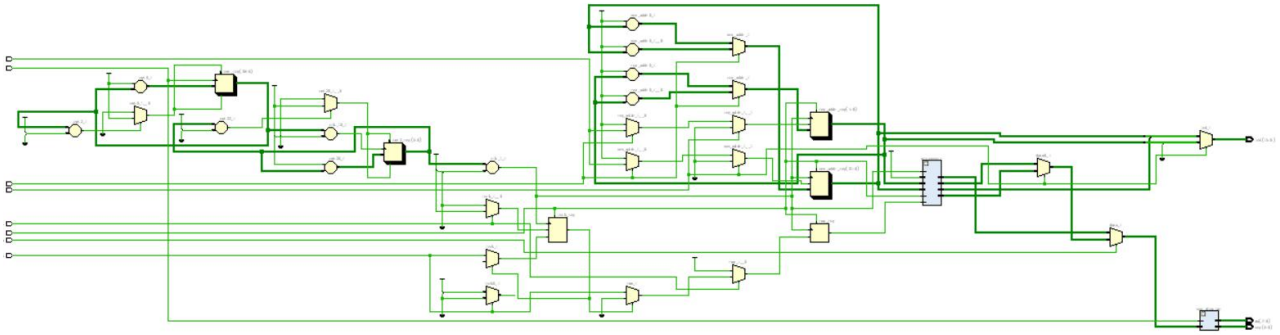

图 4：schematic 结果

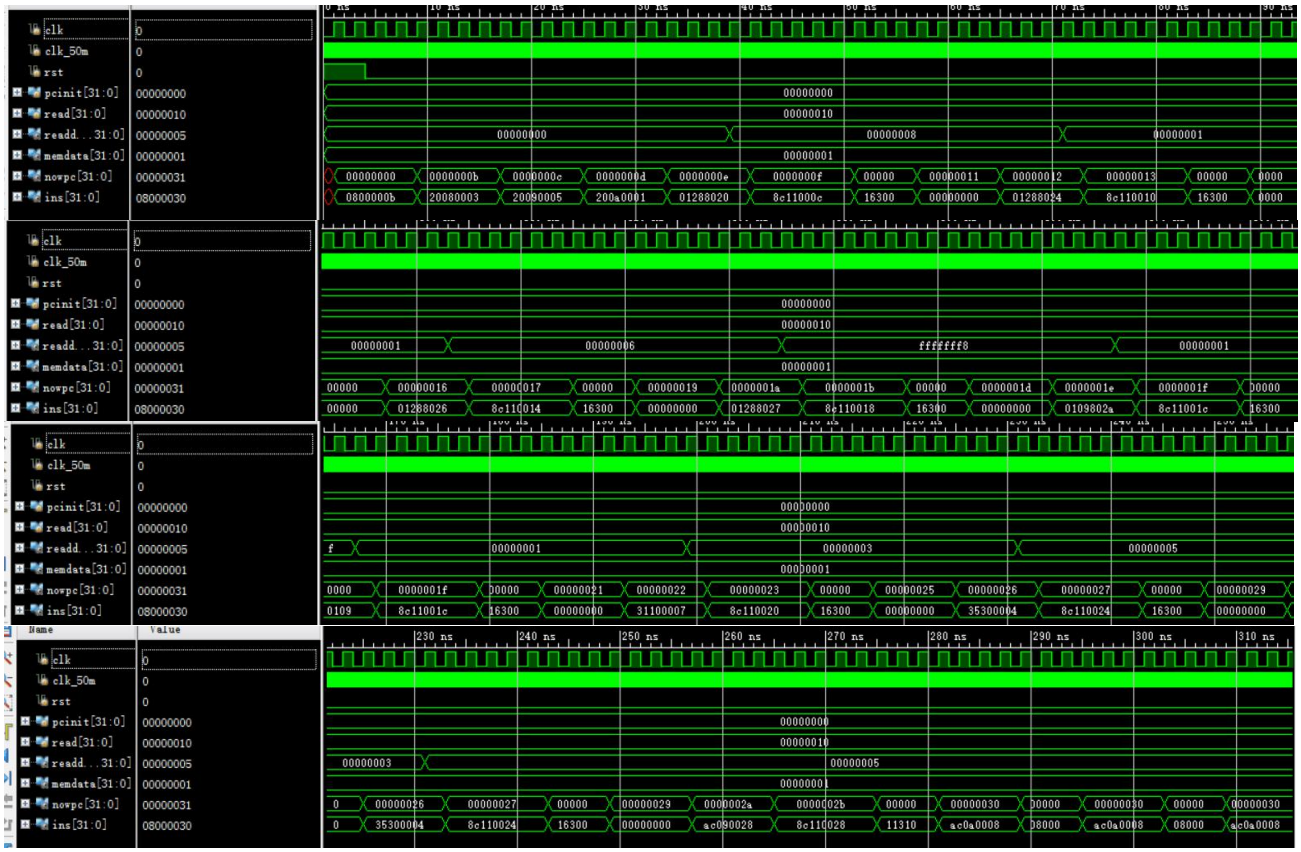

图 5：仿真测试结果

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。

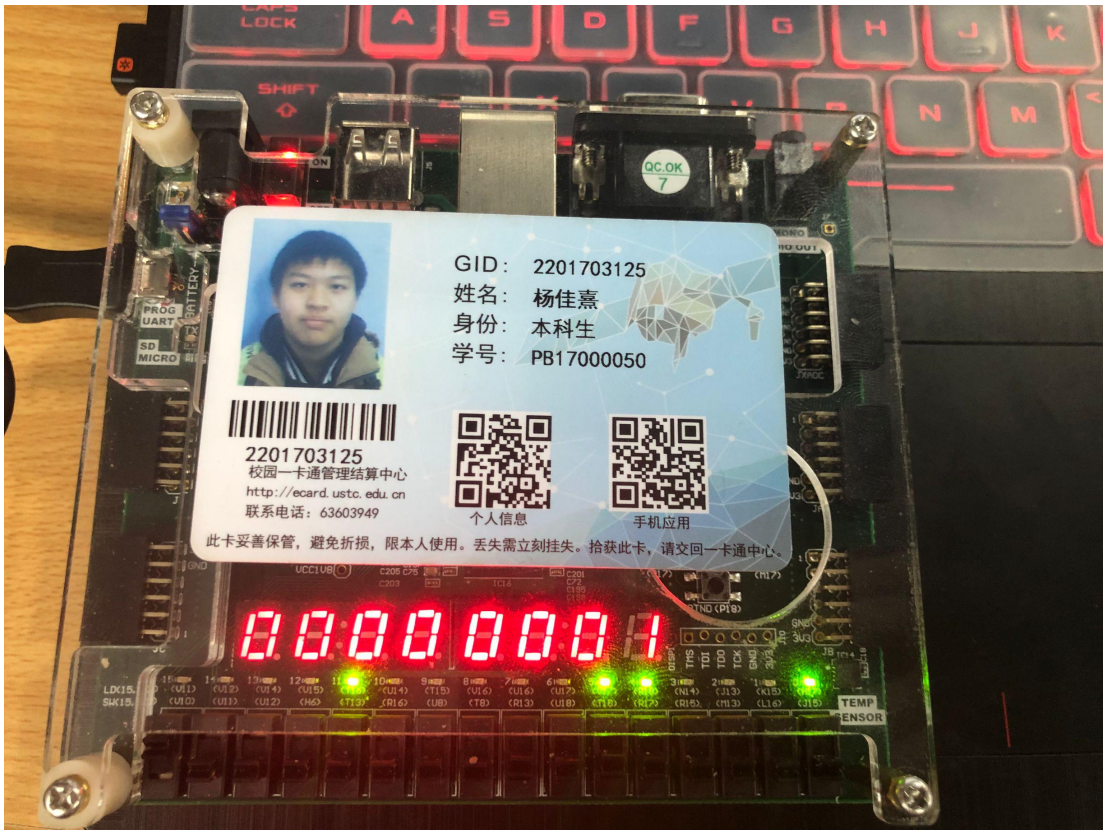

图 6：下载结果

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。