计算机组成原理

课程设计报告

学　　号_____20074221_____

姓　　名_____游佳慧_____

指导教师_____魏坚华_____

提交日期_____2022.7.6_____

成绩评价表

报告内容	报告结构	报告最终成绩
☐丰富正确 ☐基本正确 ☐有一些问题 ☐问题很大	☐完全符合要求 ☐基本符合要求 ☐有比较多的缺陷 ☐完全不符合要求	
报告与 Project 功能一致性	报告图表	总体评价
☐完全一致 ☐基本一致 ☐基本不一致	☐符合规范 ☐基本符合规范 ☐有一些错误 ☐完全不正确	

教师签字:_____

目录

Project1

1 总体数据通路结构设计

1.1 总体数据通路结构图

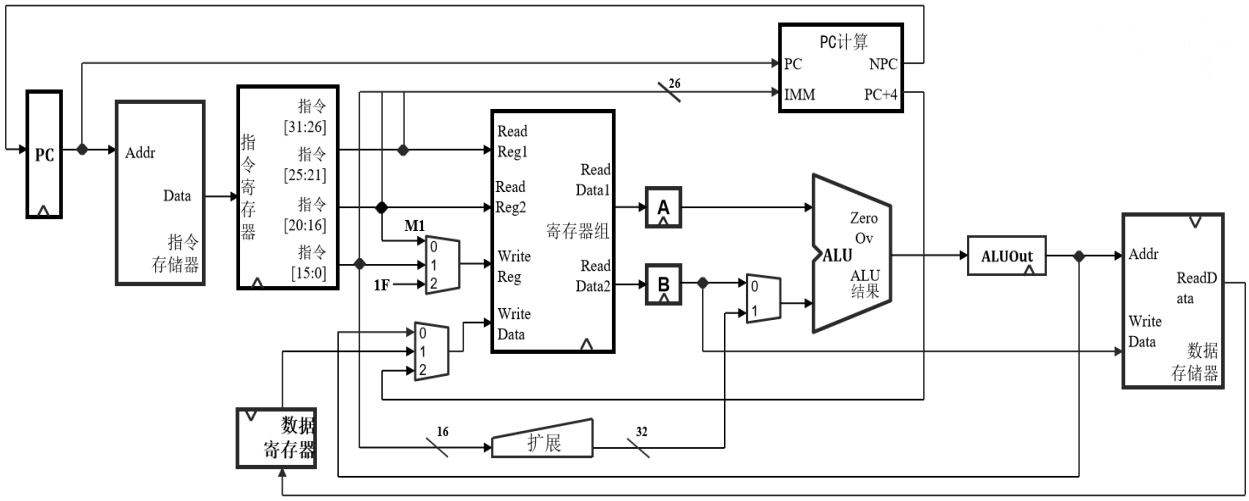


图 1：总体数据通路结构图

```
module mips(clk,rst);
      input clk,rst;
      wire [31:0] ins;
      wire [31:0] ins_reg;
      wire [31:0] npc;
      wire [31:0] cpc;
      wire [31:0] memout;
      wire [31:0] memout_reg;
      wire [31:0] din;
      wire [31:0] dout;
      wire [31:0] SB_out;
      wire [31:0] LB_out;
      wire [31:0] write_data;
      wire [31:0] bushA,bushB;
```

```
    wire [31:0] bushA_reg,bushB_reg;
    wire [31:0] extout;
    wire [31:0] alu_out;
    wire [31:0] alu_out_reg;
    wire [31:0] sltout;
    wire [31:0] jalPC;
    wire [31:0] b;
    wire [1:0] MemtoReg;    //写入寄存器数据 t
    wire [1:0] regdst;      //写寄存器选择 00:rt 01:rd 10:$31
    wire [1:0] extop;       //扩展方法 00:zero 01:sign 10:lui
    wire [1:0] aluctr;      //ALU 计算方法
    wire alusrc;            //B 端输入数据 0: busB 1: imm16
    wire MemWrite;          //DM 写使能
    wire RegWrite;          //GPR 写使能
    wire IrWrite;
    wire PcWrite;
    wire if_jr;
    wire if_beq;
    wire if_j;
    wire if_lb;
    wire if_sb;
    wire overflow;
    wire zero;
    wire [4:0] m1out;      //write reg
    wire [9:0] im_addr;
    wire [9:0] dm_addr;
    wire [9:0] dm_addr_reg;
    integer i;

    controller my_controller(ins_reg,clk,rst,if_jr,if_beq,if_j,MemWrite,
MemtoReg,RegWrite,regdst,alusrc,aluctr,extop,if_lb,if_sb,PcWrite,IrWrite,zero);
    pc my_pc(clk,rst,npc,cpc,im_addr,PcWrite);
    im_lk my_im_lk(im_addr,ins);
    NextPC my_NextPC(cpc,rst,ins_reg,if_beq,zero,if_j,
                     npc,if_jr,jalPC,bushA_reg);
    gpr my_gpr(clk,rst,RegWrite,overflow,ins_reg,
                m1out,write_data,bushA,bushB);
    ext my_ext(extop,ins_reg,extout);
    assign din = (if_sb)? SB_out : bushB_reg;
    assign memout = (if_lb)? LB_out : dout;
```

```
    sb my_sb(bushB_reg,dout,SB_out);
    lb my_lb(dout,LB_out);
    dm_1k my_dm_1k(dm_addr,din,MemWrite,clk,dout);
    ALU my_ALU(bushA_reg,b,aluctr,alu_out,
               zero,overflow,sltout,dm_addr_reg);
    mux1 my_mux1(regdst,ins_reg,m1out);
    mux2 my_mux2(MemtoReg,write_data,alu_out_reg,memout_reg,jalPC,sltout);
    mux3 my_mux3(alusrc,bushB_reg,extout,b);
    im_reg my_im_reg(clk,ins,ins_reg,IrWrite);
    A_reg my_A_reg(clk,bushA,bushA_reg);
    B_reg my_B_reg(clk,bushB,bushB_reg);
    ALU_reg my_ALU_reg(clk,alu_out,alu_out_reg,dm_addr,dm_addr_reg);
    dm_reg my_dm_reg(clk,memout,memout_reg);
endmodule
```

图 2：顶层设计

2 模块定义

2.1 GPR 模块定义

2.1.1 模块设计

```
module gpr(clk,reset,RegWrite,overflow,ins,write_reg,write_data,bushA,bushB);
    input clk;
    input reset;
    input RegWrite;
    input overflow;
    input[4:0] write_reg;       //write address
    input [31:0] write_data;
    input [31:0] ins;          //32-bit instruct
    output [31:0] bushA,bushB;  //read data
    reg [31:0] register[31:0];  //32 32-bit register
    integer i;
    assign bushA = register[ins[25:21]];
    assign bushB = register[ins[20:16]];
    always@(posedge clk or posedge reset or overflow)
    begin
      if(reset)  begin
            for(i=0;i<32;i=i+1)
              register[i]<=32'd0; end
      else begin
          if(register[30]) register[30] <= 0;
          if(RegWrite)
            begin
                if(overflow)
            //if overflow, the destination register is not modified
                    register[30]<=(register[30]|32'h0000_0001);
                else if(write_reg!=5'd0)
              //if $0, the destination register is not modified
                    register[write_reg]<=write_data;
            end
        end
    end
endmodule
```

2.1.2 基本描述

GPR 是由 32 个 32 位寄存器组成的寄存器组模块，包含修改寄存器内容、读取寄存器内容功能，寄存器的读输出总是对应于读寄存器号，不需要其他控制信号；写寄存器必须明确写使能控制信号。

2.1.3 模块接口

信号名	方向	描述
RegWrite	I	寄存器写使能。 1：当前指令写入寄存器 0：当前指令不写入寄存器
reset	I	复位信号。 1：复位 0：无效
clk	I	时钟信号。
ins[31:0]	I	传入的 32 位指令。
overflow	I	溢出标志。 1：溢出 0：未溢出
write_reg[4:0]	I	当前指令需要写入的寄存器地址。
write_data[31:0]	I	当前指令需要写入的数据。
bushA[31:0]	O	当前指令读出的数据 1。
bushB[31:0]	O	当前指令读出的数据 2。

2.1.4 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，所有寄存器数据清零。
2	取数据	根据 rs 和 rt 的地址从寄存器中取出数据。
3	写数据	若 30 号寄存器为 1，则置为 0； RegWrite 有效时，若 overflow 为 1，则将 1 写入$30，否则根据 write_reg 的地址将数据写入该地址所对应寄存器中。

2.1.5 功能说明

最开始设计 GPR 模块时未考虑到存入溢出位后将 30 号寄存器重新清零的问题，经改进后该部分代码如下：

```
begin
  if(register[30]) register[30] <= 0;
  if(RegWrite)
    begin
      if(overflow)
        register[30]<=(register[30]|32'h0000_0001);
      else if(write_reg!=5'd0)
        register[write_reg]<=write_data;
    end
end
```

每次判断 30 号寄存器是否不为 0，若是则将其清零。而后当寄存器写使能有效时，判断溢出标志是否有效，若是则将溢出位存入 30 号寄存器，若未发生溢出且所存寄存器不是 0 号寄存器时，将数据写入对应寄存器。

从而实现每次溢出时能实时将溢出位存入 30 号寄存器，且下一条指令未溢出就将 30 号寄存器清零。

2.2 ALU 模块定义

2.2.1 模块设计

```
module ALU(a,b,alu_ctr,alu_out,zero,overflow,sltout,dm_addr);
   input [31:0] a,b;         //32-bit input data
   input [1:0] alu_ctr;      //00 add; 01 sub; 10 or; 11 addi
   output zero;
   output overflow;
   output reg [31:0] alu_out;
   output reg [31:0] sltout;
   output [13:0] dm_addr;
   assign dm_addr=alu_out[13:0];
   reg signed [31:0] signed_a,signed_b;

   always@(*)
   begin
      case(alu_ctr)
         2'b00:
               alu_out <= a+b;
               sltout <= 32'd0;
         2'b01:
               alu_out <= a-b;
               signed_a <= a;
               signed_b <= b;
               if(signed_a<signed_b) sltout <= 32'd1;
               else sltout <= 32'd0;
         2'b10:
               alu_out <= a|b;
               sltout <= 32'd0;
         2'b11:
               alu_out <= a+b;
               sltout <= 32'd0;
      endcase
      end
   assign overflow=((alu_out[31] && (!a[31]) &&
(!b[31]))||((~alu_out[31]) && a[31] && b[31])) ? 1 : 0;
   assign zero = (alu_out == 0);
endmodule
```

2.2.2 基本描述

ALU 的主要功能是完成算术运算和逻辑运算，根据 ALU 控制信号判断 ALU 应进行的运算，产生运算结果并生成零标志信号 zero 和溢出信号 overflow。

2.2.3 模块接口

信号名	方向	描述
a[31:0]	I	参与运算的第一个输入数据。
b[31:0]	I	参与运算的第二个输入数据
alu_ctr[1:0]	I	ALU 控制信号。 00：无溢出加 01：减法运算 10：或运算 11：带溢出加
alu_out [31:0]	O	ALU 运算结果。
zero	O	运算结果是否为零的标志位。 1：运算结果为 0 0：运算结果非 0
overflow	O	溢出标志位。 1：溢出 0：未溢出
sltout[31:0]	O	slt 指令结果输出
dm_addr[9:0]	O	输出 dm 地址

2.2.4 功能定义

序号	功能名称	功能描述
1	算术运算	两数无溢出加、减、或、带溢出加
2	dm 地址	输出对应 dm 地址

2.3 EXT 模块定义

2.3.1 模块设计

```
module ext(extop,ins,extout);
    input [1:0] extop;
    input [31:0] ins;
    output reg [31:0] extout;

    always@(ins or extop)
    begin
      case(extop)
          2'b00: extout = {16'h0000,ins[15:0]};       //zero extend
          2'b01: extout = {{16{ins[15]}},ins[15:0]}; //sign extend
          2'b10: extout = {ins[15:0],16'h0000};       //lui
          default: extout=0;
      endcase
    end
endmodule
```

2.3.2 基本描述

EXT 的主要功能是完成 16 位立即数扩展，根据 EXTop 信号的不同值分别进行 0 扩展、符号位扩展或 lui 指令高位复制扩展，扩展为 32 位立即数输出。

2.3.3 模块接口

信号名	方向	描述
ins[31:0]	I	传入的 32 位指令。
extop[1:0]	I	符号扩展控制信号。 00：高位 0 扩展 01：符号位扩展 10：低 16 位补零扩展
extout [31:0]	O	完成扩展的 32 位立即数。

2.3.4 功能定义

序号	功能名称	功能描述
1	0 扩展	16 位->32 位，高 16 位补零
2	符号位扩展	16 位->32 位，最高有效位（符号位）复制填满高 16 位
3	低 16 位补零扩展	16 位->32 位，低 16 位补零

2.4 DM 模块定义

2.4.1 模块设计

```
module dm_1k(addr, din, we, clk, dout);
    input [9:0] addr ;
    input [31:0] din ;   // 32-bit input data
    input we,clk;          // dm write enable & clock
    output [31:0] dout ;  // 32-bit dm output
    reg[7:0] dm[1023:0] ;
    integer i;
    initial begin
    for (i = 0;i < 1024;i = i+1)
       dm[i]<=8'b0; end
    always@(posedge clk) begin
        if(we) {dm[addr+3],dm[addr+2],dm[addr+1],dm[addr]}<=din; end
    assign dout={dm[addr+3],dm[addr+2],dm[addr+1],dm[addr]};
endmodule
```

2.4.2 基本描述

DM 是数据存储器，主要功能是完成存储器读写。当写入使能有效时，根据输入的地址将输入的数据写入存储器的相应位置，或输出从该地址读取的数据。

2.4.3 模块接口

信号名	方向	描述
addr[9:0]	I	需要读或写的存储器地址。
din[31:0]	I	需要写入的数据。
clk	I	时钟信号。
we	I	写入使能信号。 1：允许写入 0：不允许写入
dout[31:0]	O	从输入地址读出的数据。

2.4.5 功能定义

序号	功能名称	功能描述
1	读数据	根据输入的寄存器地址读出数据。
2	写数据	根据输入的地址将输入数据写入存储器的相应位置。

2.5 Controller 模块定义

2.5.1 模块设计

```
module controller(ins,clk,reset,if_jr,if_beq,if_j,MemWrite,MemtoReg,
RegWrite,regdst,alusrc,alustr,extop,if_lb,if_sb,PcWrite,IrWrite,zero);
  parameter
s0=0,s1=1,s2=2,s3=3,s4=4,s5=5,s6=6,s7=7,s8=8,s9=9,s10=10,s11=11,s12=12,s13=13;
  wire addu,subu,ori,lw,sw,beq,lui,j,addiu,addi,slt,jal,jr,lb,sb;
  input clk,reset,zero;
  input[31:0]ins;                    //32-bit instruct
  reg[2:0] current,next;
  output if_lb,if_sb,if_jr,if_beq,if_j,alusrc;
  output reg PcWrite,IrWrite,MemWrite,RegWrite;
  output[1:0] MemtoReg,regdst,alustr,extop;

  //根据 opcode 和 funct 字段确定指令类型
    assign addu = (ins[31:26]==6'd0 && ins[5:0]==6'b100001)?1:0;
    assign subu = (ins[31:26]==6'd0 && ins[5:0]==6'b100011)?1:0;
    assign slt  = (ins[31:26]==6'd0 && ins[5:0]==6'b101010)?1:0;
    assign jr   = (ins[31:26]==6'd0 && ins[5:0]==6'b001000)?1:0;
    assign j    = (ins[31:26]==6'b000010)?1:0;
    assign jal  = (ins[31:26]==6'b000011)?1:0;
    assign beq  = (ins[31:26]==6'b000100)?1:0;
    assign addi = (ins[31:26]==6'b001000)?1:0;
    assign addiu= (ins[31:26]==6'b001001)?1:0;
    assign ori  = (ins[31:26]==6'b001101)?1:0;
    assign lw   = (ins[31:26]==6'b100011)?1:0;
    assign sw   = (ins[31:26]==6'b101011)?1:0;
    assign lui  = (ins[31:26]==6'b001111)?1:0;
    assign lb   = (ins[31:26]==6'b100000)?1:0;
    assign sb   = (ins[31:26]==6'b101000)?1:0;

    //设置控制信号
    assign if_jr  = jr && (current!=s0);          //1 为 jr 指令
    assign if_beq = beq && (current!=s0);         //1 为 beq 指令
    assign if_j   = (j||jal) && (current!=s0);    //1 为 j 指令
    assign if_lb  = lb;                           //1 为 lb 指令
    assign if_sb  = sb;                           //1 为 sb 指令
```

```
  assign MemtoReg = {(slt||jal),(lw||lb||slt)};  //选择传入寄存器数据
  assign regdst   = {jal,(addu||subu||slt)};          //选择写入的寄存器
  assign alusrc   = ori||lw||sw||lui||addiu||addi||lb||sb;
  //选择 ALU 第二个操作数 0 bushB;1 extout
  assign alustr   = {(ori||lui||addi),(subu||beq||addi||slt)};
  //选择 ALU 计算类型 00 add; 01 sub; 10 or; 11 addi
  assign extop    = {lui,(lw||sw||addiu||addi||lb||sb)};
   //选择扩展方法 00 0 扩展；01 符号扩展；10 lui 扩展

always@(posedge clk or posedge reset)
begin
  if(reset)
    begin
      current <= 0;
      next <= 0;
    end
  else
    begin
      current <= next;
    end
end

always@(current,addu,subu,ori,lw,sw,
         beq,lui,j,addiu,addi,slt,jal,jr,lb,sb)
begin
  case(current)
    s0:
        next<=s1;
    s1:
        next<=s2;
    s2:
        if(beq||j||jr||jal) next <= s0;
        else if(lw||sw||lb||sb) next <= s3;
        else next <= s4;
    s3:
        if(lw||lb) next <= s4;
        else next<=s0;
    s4:
        next<=s0;
    default: next<=s0;
```

```
    endcase
  end

  always@(current,addu,subu,ori,lw,sw,beq,lui,j,addiu,addi,slt,
jal,jr,lb,sb,zero)
  begin
    PcWrite  = (current==s0)||((current==s2) && jal)||((current==s2) && beq &&
zero)||((current==s2) && j)||((current==s2) && jr);
    //1 PC 写使能有效
    IrWrite  = (current==s0);
    //1 IR 写使能有效
    RegWrite = (current==s4)||((current==s2) && jal);
    //1 寄存器堆写使能有效
    MemWrite = ((current==s3) && (sw||sb));
    //1 数据存储器写使能有效
  end
endmodule
```

2.5.2 基本描述

Controller 主要功能是完成对指令功能的判断和确定每条指令对应的控制信号以及状态机的功能执行。根据输入指令的操作码和功能码判断指令，并实现将一条指令的大周期划分成若干个小周期的执行过程，在不同小周期内，即不同状态内设置相应的控制信号。

2.5.3 模块接口

信号名	方向	描述
ins[31:0]	I	传入的 32 位指令。
clk	I	时钟信号。
reset	I	复位信号。
zero	I	运算结果是否为零的标志位。 1：运算结果为 0 0：运算结果非 0
PcWrite	O	PC 写使能。

IrWrite	O	IR 写使能。
aluctr[1:0]	O	ALU 控制信号。 00：无溢出加 01：减法运算 10：或运算 11：带溢出加
alusrc	O	选择第二个 ALU 操作数。 0：操作数为寄存器取出的值 1：操作数为经 EXT 扩展后的 32 位立即数
extop[1:0]	O	控制 EXT 的扩展方式
if_beq	O	beq 指令标志。 1：是 beq 指令 0：非 beq 指令
if_j	O	j 指令标志。 1：是 j 指令 0：非 j 指令
if_jr	O	jr 指令标志。 1：是 jr 指令 0：非 jr 指令
if_lb	O	lb 指令标志。 1：是 lb 指令 0：非 lb 指令
if_sb	O	sb 指令标志。 1：是 sb 指令 0：非 sb 指令
MemWrite	O	DM 写使能信号。
RegWrite	O	GPR 写使能信号。
MemtoReg	O	选择写入寄存器的数据。 0：写入的数据是 ALU 计算输出结果 1：写入的数据是 DM 输出结果 2：写入的数据是 PC+4 3：写入的数据是 sltout
regdst	O	写入寄存器的目标寄存器号来源。

2.5.4 功能定义

序号	功能名称	功能描述
1	译码	将 ins[31:0]转换成对应指令。
2	产生控制信号	对输入指令的所有控制信号赋值。
3	状态机	设置状态，以满足不同指令实现不同阶段的跳转。

2.6 PC 模块定义

2.6.1 模块设计

```
module pc(clk,reset,npc,cpc,im_addr,PcWrite);
  input clk,reset,PcWrite;
  input[31:0] npc;
  output reg[31:0] cpc;
  output[9:0] im_addr;
  assign im_addr=cpc[9:0];
  always@(posedge clk or posedge reset)
  begin
    if(reset)
    begin
      cpc<=32'h0000_3000;
    end
    else if(PcWrite)
      cpc<=npc;
  end
    endmodule
```

2.6.2 基本描述

PC 主要功能是完成储存指令地址的功能。将要执行的指令编码对应的地址储存起来，然后在下一个 clk 上升沿到来且写使能有效时将储存的指令编码对应的地址送出去，并继续更新指令编码对应的地址。

2.6.3 模块接口

信号名	方向	描述
clk	I	时钟信号
reset	I	复位信号
PcWrite	I	PC 写使能
npc[31:0]	I	下一条指令编码对应的地址
cpc[31:0]	O	输出即将执行的指令编码对应的地址

addr[9:0]	O	IM 地址

2.6.4 功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，PC 被设置为 0x0000_3000，即第一条指令编码对应的地址。
2	写入并储存下一条指令编码对应的地址	当译码阶段的 clk 上升沿到来时，写入并存储下一条指令编码对应的地址 NPC 的值。
3	送出下一条指令地址	将即将执行的指令编码对应的地址传送给 IM 模块。

2.7 NextPC 模块定义

2.7.1 模块设计

```
module NextPC(cpc,reset,ins,if_beq,zero,if_j,npc,if_jr,jalPC,bushA);
  input [31:0] ins;        //32-bit instruct
  input [31:0] cpc;        //now PC
  input [31:0] bushA;     //target address in GPR rs
  input if_beq,if_j,zero,if_jr,reset;
  output reg [31:0] npc;
  output [31:0] jalPC;
  reg beq_jump;
  reg [2:0] choose;
  assign jalPC = cpc+4;

  always@(posedge reset)
  begin
    if(reset) npc<=32'h0000_3004;
  end

  always@(choose,ins,if_j,beq_jump,if_beq,zero,if_jr,bushA)
  begin
    beq_jump = if_beq && zero;
    choose={if_j,beq_jump,if_jr};
    case(choose)
      3'b000: npc = cpc+32'h4;                //pc=pc+4
      3'b010: npc = cpc+({{16{ins[15]}},ins[15:0]}<<2);    //beq
      3'b100: npc = {cpc[31:28],ins[25:0],2'b0};          //j jal
      3'b001: npc = bushA;                                //jr
      default:npc = 32'h00003004;
    endcase
  end
endmodule
```

2.7.2 基本描述

根据输入的控制信号，计算下一条指令地址。

2.7.3 模块接口

信号名	方向	描述
ins[31:0]	I	输出 32 位 MIPS 指令
cpc[31:0]	I	PC 模块传入的地址
if_beq	I	beq 指令标志。 1：是 beq 指令 0：非 beq 指令
if_j	I	j 指令标志。 1：是 j 指令 0：非 j 指令
if_jr	I	jr 指令标志。 1：是 jr 指令 0：非 jr 指令
zero	I	ALU 计算结果为 0 标志。 1：计算结果为 0 0：计算结果非 0
bushA[31:0]	I	jr 指令的跳转地址
jalPC[31:0]	O	输出当前指令地址+4
npc[31:0]	O	输出下一条指令编码对应的地址给 PC

2.7.4 功能定义

序号	功能名称	功能描述
1	计算下一条指令地址	如果当前指令是 beq 指令，并且 zero 为 1，则 PC←PC+4+(sign_ext(ins[15:0])<<2)； 否则如果当前指令是 J 类型指令，则 PC←{cpc[31:28],ins[25:0],2'b0}； 否则如果当前指令是 Jr 指令，则 PC←bushA； 否则 PC←PC+4。

2.8 IM 模块定义

2.8.1 模块设计

```
module im_lk(addr,dout);
    input [9:0] addr;
    output [31:0] dout;
    reg [7:0] im[1023:0];
    assign dout={im[addr],im[addr+1],im[addr+2],im[addr+3]};
endmodule
```

2.8.2 基本描述

IM 的主要功能是作为指令存储器存储指令，根据输入的指令地址取出相应的指令。

2.8.3 模块接口

信号名	方向	描述
addr[9:0]	I	指令选择地址
dout[31:0]	O	输出 32 位指令

2.8.4 功能定义

序号	功能名称	功能描述
1	取指令	根据译码地址 addr 从 IM 中取出指令。

2.9 MUX1 模块定义

2.9.1 模块设计

```
module mux1(regdst,ins,m1out); //choose write register
    input [1:0] regdst;
    input [31:0] ins;
    output reg[4:0] m1out;
    always@(regdst or ins)
    begin
      case(regdst)
          2'b00: m1out = ins[20:16]; //rt
          2'b01: m1out = ins[15:11]; //rd
          2'b10: m1out = 5'd31;     //$31
          default: m1out = 5'd0;
      endcase
    end
endmodule
```

2.9.2 基本描述

多路选择器，为 gpr 选择写入的寄存器。

2.9.3 模块接口

信号名	方向	描述
regdst[1:0]	I	寄存器选择端 0：ins[20:16]；1：ins[15:11] 2：$31
ins[31:0]	I	32 位指令
m1out[40]	O	选择结果输出

2.9.4 功能定义

序号	功能名称	功能描述
1	选择器	选择将数据写入哪个寄存器

2.10 MUX2 模块定义

2.10.1 模块设计

```
module mux2(MemtoReg,write_data,alu_out,dm_out,jalPC,sltout);
//choose write data in register
    input [1:0] MemtoReg;
    input [31:0] alu_out;
    input [31:0] dm_out;
    input [31:0] jalPC;
    input [31:0] sltout;
    output reg[31:0] write_data;

    always@(MemtoReg or alu_out or dm_out or jalPC or sltout)
    begin
      case(MemtoReg)
          2'd0: write_data = alu_out;
          2'd1: write_data = dm_out;
          2'd2: write_data = jalPC;
          2'd3: write_data = sltout;
          default: write_data = 32'd0;
      endcase
    end
endmodule
```

2.10.2 基本描述

多路选择器，选择写入寄存器的数据。

2.10.3 模块接口

信号名	方向	描述
MemtoReg[1:0]	I	选择写入寄存器的数据。 0：写入的数据是 ALU 计算输出结果 1：写入的数据是 DM 输出结果 2：写入的数据是 PC+4 3：写入的数据是 sltout
alu_out [31:0]	I	alu 计算结果

dm_out [4:0]	I	dm 输出内容
jalPC[31:0]	I	PC+4
sltout[31:0]	I	slt 指令结果
write_data[31:0]	O	选择结果输出

2.10.4 功能定义

序号	功能名称	功能描述
1	选择器	选择写入寄存器的数据。

2.11 MUX3 模块定义

2.11.1 模块设计

```
module mux3(alusrc,bushB,extout,b);    //choose the second input to ALU
    input [31:0] bushB;
    input [31:0] extout;
    input alusrc;
    output reg[31:0] b;
    always@(alusrc or bushB or extout)
    begin
      case(alusrc)
          1'd0:b=bushB;
          1'd1:b=extout;
      endcase
    end
    endmodule
```

2.11.2 基本描述

多路选择器，选择 ALU 的第二个输入数据。

2.11.3 模块接口

信号名	方向	描述
alusrc[1:0]	I	寄存器选择端 0：bushB；1：extout
bushB[31:0]	I	rt 寄存器数值
extout[31:0]	I	扩展器输出
b[31:0]	O	输出数据

2.11.4 功能定义

序号	功能名称	功能描述
1	选择器	选择 ALU 的第二个输入数据。

2.12 im_reg 模块定义

2.12.1 模块设计

```
module im_reg(clk,ins,ins_reg,IrWrite);
  input clk,IrWrite;
  input[31:0] ins;
  output reg[31:0] ins_reg;
  always@(posedge clk)
    if(IrWrite)
      ins_reg<=ins;
endmodule
```

2.12.2 基本描述

主要功能是锁存正在执行的指令编码。

2.12.3 模块接口

信号名	方向	描述
clk	I	时钟信号。
ins[31:0]	I	传入的 32 位指令。
IrWrite	I	IR 写使能。
ins_reg[31:0]	O	传出的即将执行的 32 位指令编码。

2.12.4 功能定义

序号	功能名称	功能描述
1	锁存器	锁存当前执行指令。

2.13 dm_reg 模块定义

2.13.1 模块设计

```
module dm_reg(clk,memout,memout_reg);  //存储 dm 输出数据
  input clk;
  input [31:0] memout;
  output reg [31:0] memout_reg;

  always@(posedge clk)
    memout_reg<=memout;
    endmodule
```

2.13.2 基本描述

主要功能是储存 dm 输出数据。

2.13.3 模块接口

信号名	方向	描述
clk	I	时钟信号。
memout[31:0]	I	dm 输出数据。
memout_reg[31:0]	O	dm 输出锁存后数据。

2.13.4 功能定义

序号	功能名称	功能描述
1	锁存器	锁存 dm 输出数据。

2.14 A_reg 模块定义

2.14.1 模块设计

```
module A_reg(clk,a,a_reg);
  input clk;
  input[31:0] a;
  output reg[31:0] a_reg;
  always@(posedge clk)
    a_reg<=a;
    endmodule
```

2.14.2 基本描述

用于存储 gpr 的 bushA 端数据的寄存器。

2.14.3 模块接口

信号名	方向	描述
clk	I	时钟信号。
a[31:0]	I	gpr 的 bushA 端输出数据。
a_reg[31:0]	O	gpr 的 bushA 端锁存后数据。

2.14.4 功能定义

序号	功能名称	功能描述
1	锁存器	锁存 gpr 的 bushA 端数据。

2.15 B_reg 模块定义

2.15.1 模块设计

```
module B_reg(clk,b,b_reg);
  input clk;
  input[31:0] b;
  output reg[31:0] b_reg;
  always@(posedge clk)
    b_reg<=b;
    endmodule
```

2.15.2 基本描述

用于存储 gpr 的 bushB 端数据的寄存器。

2.15.3 模块接口

信号名	方向	描述
clk	I	时钟信号。
b[31:0]	I	gpr 的 bushB 端输出数据。
b_reg[31:0]	O	gpr 的 bushB 端锁存后数据。

2.15.4 功能定义

序号	功能名称	功能描述
1	锁存器	锁存 gpr 的 bushB 端数据。

2.16 ALU_reg 模块定义

2.16.1 模块设计

```
module ALU_reg(clk,alu_out,alu_out_reg,dm_addr,dm_addr_reg);    //存储 ALU 计算结果和 dm 地址
  input clk;
  input[31:0] alu_out;
  output reg[31:0] alu_out_reg;
  input[9:0] dm_addr;
  output reg[9:0] dm_addr_reg;
  always@(posedge clk)  begin
    alu_out_reg<=alu_out;
    dm_addr_reg<=dm_addr; end
endmodule
```

2.15.2 基本描述

用于存储 ALU 计算结果以及 dm 地址的寄存器。

2.15.3 模块接口

信号名	方向	描述
clk	I	时钟信号。
alu_out[31:0]	I	ALU 计算结果。
alu_out_reg[31:0]	O	ALU 计算结果锁存后数据。
dm_addr[9:0]	I	dm 地址。
dm_addr_reg[9:0]	O	dm 地址锁存后数据。

2.15.4 功能定义

序号	功能名称	功能描述
1	锁存器	锁存 ALU 计算结果以及 dm 地址。

2.16 LB 模块定义

2.16.1 模块设计

```
module lb(dout,LB_out);
    input [31:0] dout;
    output [31:0] LB_out;
    assign LB_out = {{24{dout[7]}},dout[7:0]}; //取一个字节，按符号位扩展
    endmodule
```

2.16.2 基本描述

LB 主要功能是完成 DM 中字节读出。取出访存地址对应的 DM 的 dout，然后取低八位符号扩展后形成新数据 LB_out，送入 DR。

2.16.3 模块接口

信号名	方向	描述
dout[31:0]	I	DM 中访存地址对应的字单元数据。
LB_out[31:0]	O	dout 的低八位符号扩展后形成新数据 LB_out。

2.16.4 功能定义

序号	功能名称	功能描述
1	DM 字节读出	实现字节读出访存地址对应的 DM 中的字节单元。

2.17 SB 模块定义

2.17.1 模块设计

```
module sb(bushB,dout,SB_out);
    input [31:0] bushB;
    input [31:0] dout;
    output [31:0] SB_out;
    assign SB_out = {dout[31:8],bushB[7:0]};
    endmodule
```

2.17.2 基本描述

SB 的主要功能是完成字节写入 DM。先取出访存地址对应的 DM 的 dout，然后 bushB 的低八位覆盖 dout 的低八位，形成新数据 SB_out，送回 DM 的 din。

2.17.3 模块接口

信号名	方向	描述
dout[31:0]	I	DM 中访存地址对应的字单元数据。
bushB[31:0]	I	GPR 输出值经 B_reg 模块缓存后的输出值.
SB_out	O	bushB 的低八位覆盖 dout 的低八位形成的新数据 SB_out。

2.17.4 功能定义

序号	功能名称	功能描述
1	字节写入 DM	实现字节写入访存地址对应的 DM 中的字节单元。

3 设计的机器指令描述

指令操作码助记符	机器指令代码		指令功能
	opcode	funct	
addu	000000	100001	分别从 rs 和 rt 寄存器中取出两个数做无符号数加法，结果放入 rd 寄存器中。
subu	000000	100011	分别从 rs 和 rt 寄存器中取出两个数做无符号数减法，结果放入 rd 寄存器中。
ori	001101	—	从 rs 寄存器中取出一个数与高位零扩展后的 16 位立即数做或运算，结果放入 rt 寄存器。
lui	110000	—	从 rs 寄存器中取出一个数与低位补零扩展后的 16 位立即数做或运算，结果放入 rt 寄存器。
sw	101011	—	根据基地址+偏移量算出地址，将 rt 寄存器内容写入该地址对应的内存单元中。
lw	100011	—	根据基地址+偏移量算出地址，从该地址对应存储器中取出一个数存入 rt 寄存器中。
beq	000100	—	分别从 s 和 t 寄存器中取出两个数比较是否相等，若相等则进行分支跳转，PC<-PC+4+符号扩展 imm16,若不相等则不跳转，PC<-PC+4。
j	000010	—	无条件跳转，PC<-{PC+4[31:28] imm26}。
addi	001000	—	支持溢出的立即数加法，若溢出，则将$30 第 0 位置 1，否则进行正常加法操作。
addiu	001001	—	不支持溢出的立即数加法。
slt	000000	101010	如果 gpr[rs]<gpr[rt],则 gpr[rd]=1，否则为 0。
jal	000011	—	将 PC+4 存入$31 寄存器并跳转到对应地址。
jr	000000	001000	跳转到 gpr[rs]中对应地址。
lb	100000	—	将 dm 对应处一字节内容符号扩展后写入寄存器。
sb	101000	—	将输入数据低 8 位写入 dm 对应地址处。

4 状态转移图

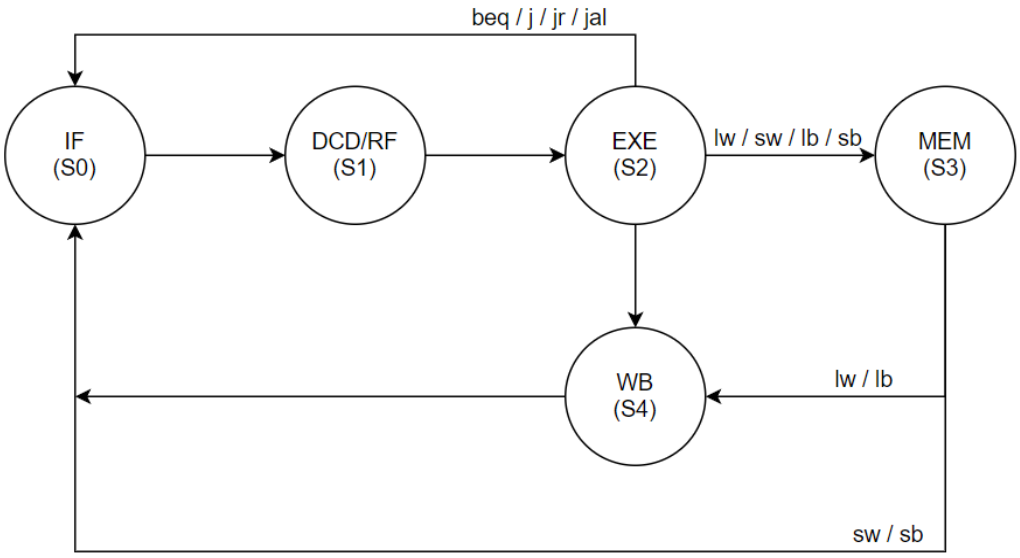


图 3：FSM 有限状态机图

5 测试程序

机器码	指令	注释说明
34100001	ori $16, $0, 1	将 16 号寄存器赋值 1
34110003	ori $17, $0, 3	将 17 号寄存器赋值 3
34080001	ori $8, $0, 1	将 8 号寄存器赋值 1
340cabab	ori $12, $0,0xabab	将 12 号寄存器赋值 0xabab
3c0d000a	lui $13, 10	将 13 号寄存器高 16 位赋值 10
00102021	start:addu $4, $0,$16	将 0 号寄存器与 16 号寄存器内容相加放入 4 号寄存器
00082821	addu $5, $0,$8	将 0 号寄存器与 8 号寄存器内容相加放入 5 号寄存器
0c000c32	jal newadd	跳转到 newadd 处，并将下条指令地址存入 31 号寄存器
00028021	addu $16, $0, $2	将 0 号寄存器与 2 号寄存器内容相加放入 16 号寄存器
02288823	subu $17,$17,$8	将 17 号寄存器内容与 8 号寄存器内容相减放入 17 号寄

		存器
1211fffa	beq $16, $17, start	若 16 号寄存器内容与 17 号寄存器内容相等则跳转到 start，否则向下执行
34080004	ori $8, $0,4	将 8 号寄存器赋值 4
3c017fff	addiu $24,$0,0x7fffffff	将 0 号寄存器内容与 0x7fffffff 相加放入 24 号寄存器
27090003	addiu $9,$24,3	将 24 号寄存器内容+3 放入 9 号寄存器（无溢出加法）
270a0005	addiu $10,$24,5	将 24 号寄存器内容+5 放入 10 号寄存器（无溢出加法）
00000021	addu $0,$0,$0	将 0 号寄存器与 0 号寄存器内容相加放入 0 号寄存器
	#addi $22,$24,6	将 24 号寄存器内容+6，若无溢出，则将结果赋给 22 号寄存器，否则将 30 号寄存器第 0 位置 1
ad09fffc	start2:sw $9, -4($8)	将 9 号寄存器内容存到 dm[[$8]-4]处
ad010000	sw $1, 0($8)	将 1 号寄存器内容存到 dm[[$8]]处
810e0003	lb $14, 3($8)	将 dm[[$8]+3]处一字节内容符号扩展赋给 14 号寄存器
a10c0007	sb $12,7($8)	将 12 号寄存器低 8 位内容存到 dm[[$8]+7]处
8d0f0004	lw $15,4($8)	将 dm[[$8]+4]处值赋给 15 号寄存器
a104fffd	sb $4, -3($8)	将 4 号寄存器低 8 位内容存到 dm[[$8]-3]处
8112ffff	lb $18, -1($8)	将 dm[[$8]-1]处一字节内容符号扩展赋给 18 号寄存器
00082021	addu $4,$0,$8	将 0 号寄存器与 8 号寄存器内容相加放入 4 号寄存器
00092821	addu $5,$0,$9	将 0 号寄存器与 9 号寄存器内容相加放入 5 号寄存器
0c000c33	jal newadd	跳转到 newadd 处，并将下条指令地址放入 31 号寄存器
0148c82a	slt $25,$10,$8	如果[$10]<[$8],则[$25]=1,否则为 0
13200018	beq $25, $0,end2	若[$25]=0 则跳转到 end2，否则顺序执行
0184a02a	slt $20,$12,$4	如果[$12]<[$4],则[$20]=1,否则为 0
12800001	beq $20, $0, end1	若[$20]=0 则跳转到 end1，否则顺序执行
3c0cffff	lui $12, 65535	给 12 号寄存器高 16 位赋值 65535
34000001	end1:ori $0, $0,1	不执行任何操作

3c13efef	lui $19, 0xefef	给 19 号寄存器高 16 位赋值 0xefef
3c01abab	addiu $3,$0,0xababcdcd	将 0 号寄存器与 0xababcdcd 相加放入 3 号寄存器
24640002	start3:addiu $4, $3, 2	将 3 号寄存器内容+2 赋给 4 号寄存器（无溢出加法）
20770005	addi $23, $3, 5	将 3 号寄存器内容+5，若无溢出，则将结果赋给 23 号寄存器，否则将 30 号寄存器第 0 位置 1
0c000c33	jal newadd	跳转到 newadd 处，并将下条指令地址放入 31 号寄存器
00024021	addu $8, $0, $2	把 0 号寄存器与 2 号寄存器内容相加赋给 8 号寄存器
00082021	addu $4, $0, $8	把 0 号寄存器与 8 号寄存器内容相加赋给 4 号寄存器
00092821	addu $5, $0, $9	把 0 号寄存器与 9 号寄存器内容相加赋给 5 号寄存器
0c000c33	jal newadd	跳转到 newadd 处，并将下条指令地址放入 31 号寄存器
00024821	addu $9, $0, $2	把 0 号寄存器与 2 号寄存器内容相加赋给 9 号寄存器
01004821	addu $9, $8, $0	把 8 号寄存器与 0 号寄存器内容相加赋给 9 号寄存器
3c0a0069	lui $10, 0x69	给 10 号寄存器高 16 位赋值 0x69
11090001	beq $8, $9, start4	若[$8]=[$9]则跳转到 start1，否则顺序执行
1000fff4	beq $0, $0, start3	跳转到 start3
08000c37	start4:j end	跳转到 end
00851021	newadd:addu $2, $4, $5	将 4 号寄存器内容与 5 号寄存器内容相加赋给 2 号寄存器（无溢出加法）
21801234	addi $0,$12,0x1234	将 12 号寄存器内容+0x1234，若溢出，将 30 号寄存器第 0 位置 1
03e00008	jr $31	跳转至 31 号寄存器中存储的地址
201a5678	end2:addi $26,$0,0x5678	给 26 号寄存器赋值 0x5678
	end:	代码结束

6 测试结果

6.1 GPR 运行结果

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xababcdcd
$v0	2	0xababcdd3
$v1	3	0xababcdcd
$a0	4	0x2babcdd1
$a1	5	0x80000002
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x2babcdd1
$t1	9	0x2babcdd1
$t2	10	0x00690000
$t3	11	0x00000000
$t4	12	0x0000abab
$t5	13	0x000a0000
$t6	14	0x0000007f
$t7	15	0xab000000
$s0	16	0x00000003
$s1	17	0x00000001
$s2	18	0xffffff80
$s3	19	0xefef0000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0xababcdd2
$t8	24	0x7fffffff
$t9	25	0x00000001
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x000030b4
pc		0x000030dc
hi		0x00000000
lo		0x00000000

图 4：Mars 中的 GPR

Memory Data - /test_mips/my_gpr/

31	000030b4
30	00000000
29	00000000
28	00000000
27	00000000
26	00000000
25	00000001
24	7fffffff
23	ababcdd2
22	00000000
21	00000000
20	00000000
19	efef0000
18	ffffff80
17	00000001
16	00000003
15	ab000000
14	0000007f
13	000a0000
12	0000abab
11	00000000
10	00690000
9	2babcdd1
8	2babcdd1
7	00000000
6	00000000
5	80000002
4	2babcdd1
3	ababcdcd
2	ababcdd3
1	ababcdcd
0	00000000

图 5：modelsim 中的 GPR

6.2 DM 运行结果

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x00000000	0x80000202	0x7fffffff	0xab000000
0x00000020	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000

图 6：Mars 中的 DM

```
0000003b 00 00 00 00
00000037 00 00 00 00
00000033 00 00 00 00
0000002f 00 00 00 00
0000002b 00 00 00 00
00000027 00 00 00 00
00000023 00 00 00 00
0000001f 00 00 00 00
0000001b 00 00 00 00
00000017 00 00 00 00
00000013 00 00 00 00
0000000f 00 00 00 00
0000000b ab 00 00 00
00000007 7f ff ff ff
00000003 80 00 02 02
```

图 7：modelsim 中的 DM

6.3 波形图

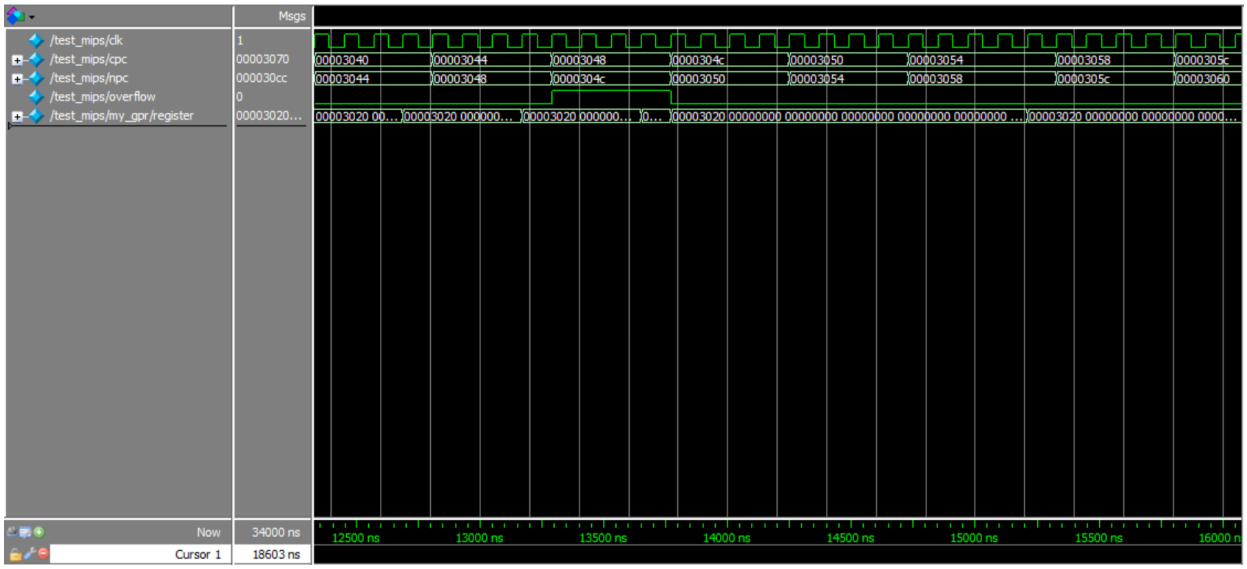


图 8：局部仿真结果截图

Project2

1 总体数据通路结构设计

1.1 总体数据通路结构图

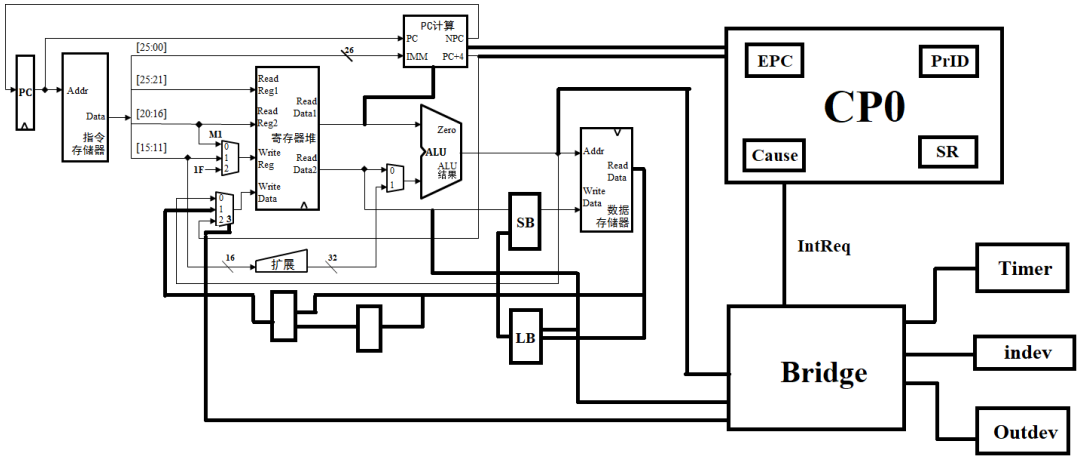


图 1：总体数据通路结构图

```
module mach(clk,rst,in);
    input clk,rst;
    input [31:0] in;
    wire [31:0] praddr,prrd,prwd;
    wire [31:0] dev0_rd,dev1_rd,dev2_rd;
    wire [5:0] hwint;
    wire [1:0] dev_addr;
    wire dev_wen;
    mips mips(clk,rst,praddr,prrd,prwd,dev_wen,hwint);
    bridge bridge(praddr,prrd,prwd,dev_addr,dev0_rd,dev1_rd,
                  dev2_rd,dev_wd,dev_wen,weTimer,weOut,IRQ,hwint);
    timer timer(clk,rst,weTimer,dev_addr,prwd,dev0_rd,IRQ);
    outputdev outputdev(clk,weOut,prwd,dev_addr,dev1_rd);
    inputdev inputdev(in,dev2_rd);
endmodule
```

图 2：顶层设计

2 模块定义

由于 P2 是在 P1 基础上增加了 bridge、cp0 以及三个外设，并修改了 im 以及 dm 的容量，只需更改一下 im 以及 dm 寄存器数组大小以及输入地址位数，以及写入寄存器堆的多路选择器扩展为 5 路，加上 cp0out 和 din，只需修改 MUX2 模块引脚。故以下只列出 P2 相对于 P1 新增的五个模块。

2.1 cp0 模块定义

2.1.1 模块设计

```
module
cp0(pc,din,hwint,sel,cp0wr,exlset,exlclr,clk,reset,intreq,epc,epcwr,dout);
    input [31:0] pc;        //用于保存 pc
    input [31:0] din;       //cp0 寄存器的写入数据：GPR 中 rt 寄存器读出数据
    input [5:0] hwint;      //6 个设备中断，从 bridg 传递过来
    input [4:0] sel;        //选择 cp0 内部寄存器
    input epcwr;            //epc 写使能
    input cp0wr;            //cp0 写使能
    input exlset;           //置位 SR 的 EXL 位
    input exlclr;           //清除 SR 的 EXL 位，执行 eret 指令产生
    input clk,reset;
    output intreq;          //中断请求信号
    output [31:0] dout;     //cp0 寄存器的输出数据
    output reg [31:0] epc;        //epc 寄存器输出至 npc
    reg [15:10] hwint_pend;       //cause 中 6 位寄存器,锁存 hwint
    reg [15:10] im;  //SR
    reg exl,ie;              //exl 标记中断状态：ie 全局中断使能 1 允许中断

     always@(posedge clk or posedge reset)
     begin
        if(reset) begin
            exl=0; ie=0; im=0; hwint_pend=0;
        end
        else begin
            if(epcwr)
                epc <= pc;                  //保存中断时的 pc
            if(cp0wr && (sel==5'd12))    //cp0 写使能有效且为 sr 寄存器,给 SR 赋初值
```

```
            {im,exl,ie} <= {din[15:10],din[1],din[0]};
        if(exlset)                                  //关中断，防止再次进入
            exl<=1'b1;
        if(exlclr) begin                            //开中断
            exl<=1'b0;
            hwint_pend=0;
        end
        else hwint_pend<=hwint;
    end
 end
assign intreq=|(hwint & im) & ie & !exl;        //产生中断请求

//写入 cpu 寄存器的数据
//12:SR 13:CAUSE 14:EPC 15:PrID
assign dout=(sel==5'd12)? {16'b0,im,8'b0,exl,ie}:
            (sel==5'd13)? {16'b0,hwint_pend,10'b0}:
            (sel==5'd14)? epc:
            (sel==5'd15)? 32'h20074221:
            32'd0;
endmodule
```

2.1.2 基本描述

CP0 的主要功能是处理外设发出的中断请求，中断信号由 bridge 模块传入到 CP0，经过处理后传入 CPU 的 controller 模块。处理中断的过程包含使用 MTC0 与 MFC0 指令：完成 SR 寄存器的预设；完成主程序中断位置的下一条指令对应的地址写入 EPC 寄存器，保护现场，并在 ERET 指令时回写到 NPC，返回主程序。CP0 内部主要用到了四个寄存器：SR 寄存器（控制是否响应中断）、CAUSE 寄存器（锁存中断原因）、EPC 寄存器（存储中断位置下一条指令地址进行保护）、PrID 寄存器（显示独有设计标识）。

2.1.3 模块接口

信号名	方向	描述
reset	I	复位信号。 1：复位 0：无效
clk	I	时钟信号。

pc[31:0]	I	中断位置下一条指令的地址。
din[31:0]	I	CP0 内寄存器的写入数据。
hwint[5:0]	I	6 路设备中断（最低位由定时器产生）。
sel[4:0]	I	CP0 内部寄存器的选择信号。
epcwr	I	Epc 寄存器写使能
cp0wr	I	cp0 内部寄存器写使能。
exlset	I	用于置位 SR 的 EXL 位(EXL 为 1)。
exlclr	I	用于清除 SR 的 EXL 位(EXL 为 0)。
intreq	O	中断请求信号。
dout[31:0]	O	CP0 内寄存器的输出数据。
epc[31:0]	O	执行完中断后返回地址。

2.1.4 功能定义

序号	功能名称	功能描述
1	与 CPU 传输数据	通过 MTC0/MFC0 指令，CPU 的寄存器与 CP0 的寄存器进行数据传输。
2	处理中断请求	处理外设传来的中断请求信号，判断是否需要响应。若响应该中断请求，则将中断位置的下一条指令地址写入 EPC，进行现场保护。最后 ERET 指令将 EPC 内指令地址写回 NPC，返回主程序。

2.2 bridge 模块定义

2.2.1 模块设计

```
module bridge(praddr,prrd,prwd,dev_addr,dev0_rd,dev1_rd,dev2_rd,dev_wd,
                weCPU,weTimer,weOut,IRQ,hwint);
    input [31:0] praddr;      //CPU 传入的访存地址
    input [31:0] prwd;        //CPU 写入外设的数据
    input [31:0] dev0_rd;     //定时器读出的数据
    input [31:0] dev1_rd;     //输出设备读出的数据
    input [31:0] dev2_rd;     //输入设备读出的数据
    input weCPU;              //CPU 传入的外设写使能
    input IRQ;                //定时器传入的中断信号
    output [31:0] prrd;       //外设写入 CPU 的数据
    output [31:0] dev_wd;     //写入外设的数据
    output [1:0] dev_addr;    //选择外设
    output [5:0] hwint;      //6 个外设的中断请求信号
    output weTimer,weOut;   //定时器和输出设备的写使能

    wire hitdev_timer,hitdev_out,hitdev_in;    //设备译码信号

    assign hwint = {5'd0,IRQ};                 //定时器的中断信号存到第 0 位
    assign dev_wd = prwd;                     //写入外设的数据
    assign dev_addr = praddr[3:2];
    assign hitdev_timer = (praddr[31:4] == 28'h0000_7f0);
    assign hitdev_out   = (praddr[31:4] == 28'h0000_7f1);
    assign hitdev_in    = (praddr[31:4] == 28'h0000_7f2);

    assign weTimer = weCPU && hitdev_timer;
    assign weOut   = weCPU && hitdev_out;

    assign prrd = (hitdev_timer) ? dev0_rd:
                  (hitdev_out)   ? dev1_rd:
                  (hitdev_in)    ? dev2_rd:
                  32'h20074221;
    endmodule
```

2.2.2 基本描述

Bridge 作为连接 CPU、CP0、外设的桥梁，主要功能是完成外设和 CPU 的数据传送以及向 CP0 发送外设的中断请求，从而保证中断能够被响应。

2.2.3 模块接口

信号名	方向	描述
praddr[31:0]	I	CPU 传入的访存地址。
prwd[31:0]	I	CPU 写入外设的数据。
prrd[31:0]	O	外设写入 CPU 的数据。
weCPU	I	CPU 传入的外设写使能。
hwint[5:0]	O	6 路设备的中断请求信号。
IRQ	I	定时器传入的中断信号。
dev_addr[1:0]	O	用于外设内部寄存器的选择。
dev0_rd[31:0]	I	定时器读出的数据。
dev1_rd [31:0]	I	输出设备读出的数据。
dev2_rd t[31:0]	I	输入设备读出的数据。
dev_wd[31:0]	O	写入外设的数据。
weTimer	O	定时器的写使能。
oweOut	O	输出设备的写使能。

2.2.4 功能定义

序号	功能名称	功能描述
1	数据传输枢纽	Bridge 作为桥梁连接 CPU、CP0、外设，负责数据的中转与传递。
2	中断请求上报	将外设的中断请求传递给 CP0 处理。

2.3 timer 模块定义

2.3.1 模块设计

```
module timer(CLK_I,RST_I,WE_I,ADD_I,DAT_I,DAT_O,IRQ);
    input CLK_I;
    input RST_I;
    input WE_I;            //写使能
    input [3:2] ADD_I;     //选择寄存器
    input [31:0] DAT_I;    //输入数据
    output [31:0] DAT_O;   //输出数据
    output reg IRQ;        //中断请求
    reg [31:0] ctrl;       // 控制计数起停
    reg [31:0] preset;     // 保存初值
    reg [31:0] count;      // 计数

    assign DAT_O = (ADD_I==2'b00)? ctrl:        //选择通过 bridg 写入 cpu 的数据
                   (ADD_I==2'b01)? preset:
                   (ADD_I==2'b10)? count: DAT_O;

    always@(posedge CLK_I or posedge RST_I)
    begin
        if(RST_I)
        begin
            ctrl <= 32'd0;
            preset <= 32'd0;
            count <= 32'd0;
            IRQ <= 1'b0;
        end
        else
        begin
            if(WE_I)                    //写使能有效，从 cp0 输入的数据存入寄存器
            begin
                ctrl <= (ADD_I==2'b00)? DAT_I : ctrl;
                preset <= (ADD_I==2'b01)? DAT_I : preset;
                count <= (ADD_I==2'b01)? DAT_I : count;
                 //初值寄存器重写后重新倒计时
            end
            if(IRQ==1) IRQ<=0;    //清除中断信号
```

```
            if(ctrl[0])  //计数器使能为 1 允许计数
            begin
                if(ctrl[2:1]==2'b00)  //模式 0
                begin
                    if(count == 2'b0)
                    begin
                        ctrl[0] <= 0;    //倒计数为 0 后，计数器停止计数，使能为 0
                        if(ctrl[3]==1) begin IRQ <= 1'b1; ctrl[3] <=0;
end   //如果中断允许，产生中断请求
                    end
                    else count <= count - 1;        //倒计数不为 0 则继续计数
                end
                else if(ctrl[2:1]==2'b01)  //模式 1
                begin
                    if(count == 2'b0)
                        begin
                            count <= preset;
                          //倒计数为 0 后，计数器自动加载初值，继续计数
                        end
                    else count <= count - 1;
                end
            end
        end
    end

    endmodule
```

2.3.2 基本描述

定时器的主要功能是实现倒计数功能。通过控制寄存器 CTRL 的[2:1]位来决定工作模式，即模式 0 和模式 1。模式 0：当计数器倒计数为 0 后，计数器停止计数，当初值寄存器再次被外部写入后，初值寄存器值再次被加载至计数器，计数器重新启动倒计数。模式 1：当计数器倒计数为 0 后，初值寄存器值被自动加载至计数器，计数器继续倒计数。当计数器工作在模式 0 并且在中断允许的前提下，当计数器计数值为 0 时，中断产生逻辑产生中断请求(IRQ 为 1)。

2.3.3 模块接口

信号名	方向	描述
CLK_I	I	时钟信号。
RST_I	I	复位信号。
WE_I	I	定时器写使能。
ADD_I[3:2]	I	定时器内部寄存器选择信号。
DAT_I[31:0]	I	CPU 通过 bridge 传入的数据。
DAT_O[31:0]	O	定时器通过 bridge 传入 CPU 的数据。
IRQ	O	定时器的中断请求信号。

2.3.4 功能定义

序号	功能名称	功能描述
1	倒计数	有模式 0 和模式 1 两种工作模式。
2	中断请求	当计数器工作在模式 0 并且在中断允许的前提下，当计数器计数值为 0 时，中断产生逻辑产生中断请求。

2.4 inputdev 模块定义

2.4.1 模块设计

```
module inputdev(din,dout);
    input [31:0] din;
    output [31:0] dout;
    reg [31:0] temp;
    always@(*) temp=din;
    assign dout=temp;
endmodule
```

2.4.2 基本描述

输入设备的主要功能是接入存储外部传进来的数据。

2.4.3 模块接口

信号名	方向	描述
din[31:0]	I	外部传进来的数据。
dout[31:0]	O	通过 bridge 写入 CPU 的数据。

2.4.4 功能定义

序号	功能名称	功能描述
1	传送外部数据	传送存储外部传进来的数据。

2.5 outputdev 模块定义

2.5.1 模块设计

```
module outputdev(clk,weOut,din,addr,dout);
    input clk;
    input weOut;
    input [31:0] din;         //CPU 传进来的数
    input [1:0] addr;        //选择输出设备的内部寄存器
    output [31:0] dout;      //通过 bridge 写入 CPU 的数
    reg [31:0] temp1,temp2;     //temp1 存放上一秒输入的数；temp1 存放当前输出
    assign dout=(addr==2'b00)? temp1:
                (addr==2'b01)? temp2: dout;    //输出数据
    always@(posedge clk) begin                 //写入数据
        if(weOut) begin
            if(addr==2'b00) temp1=din;
            if(addr==2'b01) temp2=din;
        end
    end
    endmodule
```

2.5.2 基本描述

输出设备的主要功能是输出显示 CPU 或者其他外设传过来的数据。

2.5.3 模块接口

信号名	方向	描述
din[31:0]	I	CPU 传进来的数据。
clk	I	时钟信号。
weOut	I	输出设备写使能。
addr[1:0]	I	选择输出设备的内部寄存器。
dout[31:0]	O	通过 bridge 写入 CPU 的数据。

2.5.4 功能定义

序号	功能名称	功能描述
1	存储并显示外显数据	存储 CPU 或其他外设写进来的数，并显示输出到外显设备。

3 设计的机器指令描述

指令操作码助记符	机器指令代码		指令功能
	opcode	funct	
Eret	010000	011000	返回中断前下一条地址处
Mtc0	010000	Rs=01000	把 rt 寄存器数据传给选中的 cp0 寄存器中
Mfc0	010000	Rs=00000	把选中的 cp0 寄存器中数据传给 rt 寄存器

4 状态转移图

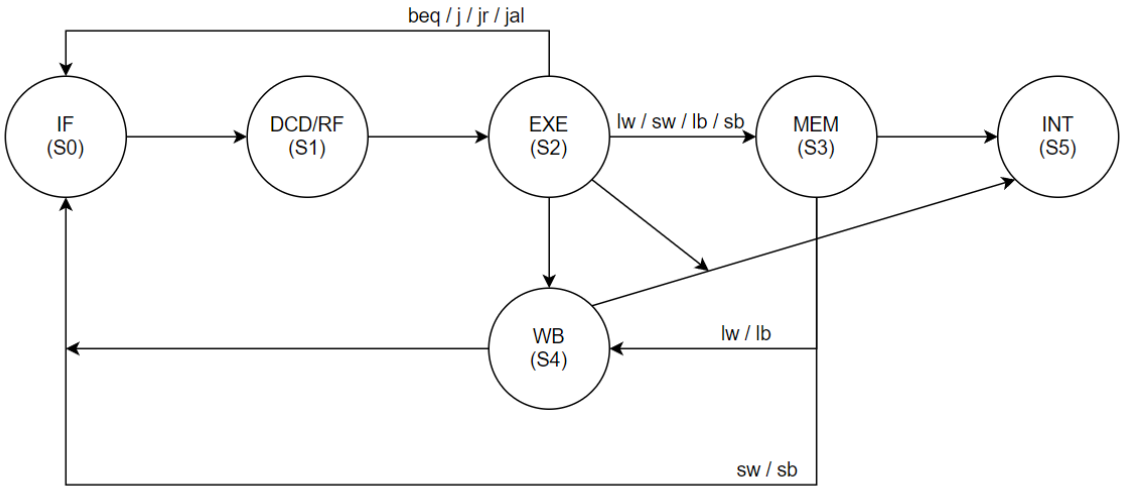


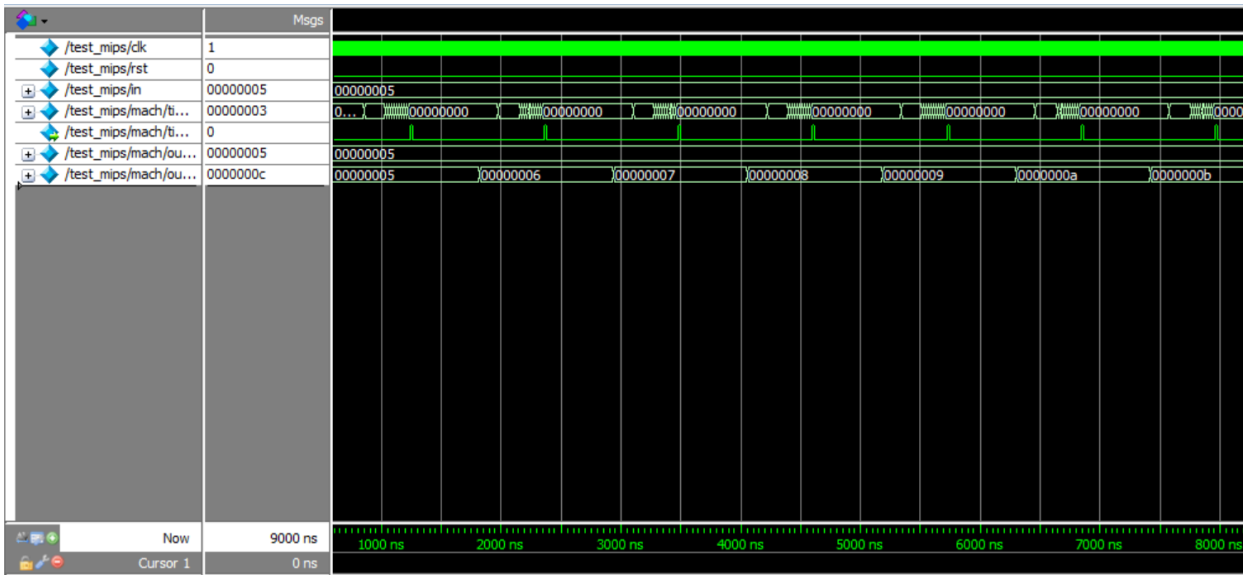
图 3：FSM 有限状态机图

5 测试程序

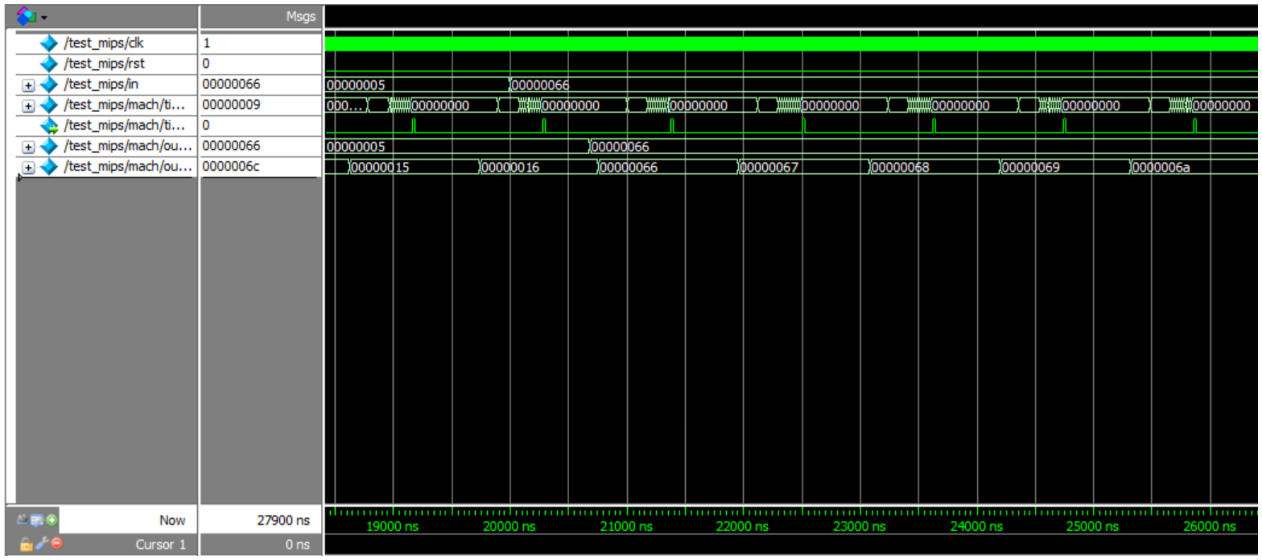
指令	注释说明
主程序	
ori $s0,$0,0x7f00	将定时器中寄存器的基地址放入$s0
ori $s1,$0,0x7f10	将输出设备中寄存器的基地址放入$s1
ori $s2,$0,0x7f20	将输入设备中寄存器的基地址放入$s2
lw $t0,($s2)	将 inputdev 数据写入$t0
sw $t0,($s1)	将$t0 数据写入 outputdev 中寄存器 temp1
sw $t0,4($s1)	将$t0 数据写入 outputdev 中寄存器 temp2
ori $t1,$0,0x0401	将 0x0000_0401 写入$t1
mtc0 $t1,$12	将$t1 的值送入 sr 寄存器
mfc0 $s3,$15	将 prid 的值送入$s3 寄存器
ori $t2,$0,10	将$t2 赋值 10
sw $t2,4($s0)	将$t2 送 preset 寄存器中
ori $t3,$0,9	将$t3 赋值 9
sw $t3,($s0)	将$t3 送 ctrl 寄存器中
loop:j loop	进入死循环等待中断信号的产生
子程序	
lw $t0,($s2)	将 inputdev 中寄存器 din 的值写入$t0
lw $t1,($s1)	将 outputdev 中寄存器 temp1 的值写入$t1
beq $t0,$t1,equal	判断$t0，$t1 的值是否相等，是则跳转至 equal
sw $t0,($s1)	将$t0 中的值送入 outdev 中的寄存器 temp1
sw $t0,4($s1)	将$t0 中的值送入 outdev 中的寄存器 temp2
beq $0,$0,exit	跳转至 exit

equal:lw $t2,4($s1)	将寄存器 temp2 的数据送入$t2
addiu $t2,$t2,1	将$t2 加一
sw $t2,4($s1)	将$t2 送回寄存器 temp2
exit:ori $t3,$0,10	将$t3 赋值 10
sw $t3,4($s0)	将$t3 的值送进 timer 的 preset 寄存器
ori $t4,$0,9	将$t4 赋值 9
sw $t4,($s0)	将$t4 送入 timer 的 ctrl 寄存器
eret	返回主程序

6 测试结果



输入不变时，outputdev 显示数据每秒+1。



输入改变时，outputdev 显示 inoutdev 当前数据。

7 总结与收获

通过课设的 p1，我学习了如何进行多周期状态机的设置，使得一个大周期被划分为若干个小周期，不同指令只走对应的几个小周期，从而缩短关键路径的长度与执行时间；而课设的 p2 难度则较之前有极大提升，不单单是只有 CPU，而是包含着外设、Bridge、CP0 的微系统，更是真正意义上的实现了中断请求、中断允许、中断响应、中断返回的全过程。在做课设的过程中我也真正将课堂所学知识运用到实践中，深刻理解了 CPU 和外设间的通讯过程和 bridge

的重要作用以及由外设产生的中断的处理过程，学会了计算机微系统的工作机理，更加懂得了学以致用，用以巩学的道理。在今后的学习生活中，我也将继续保持着认真、严谨、求实的态度。