**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 4](#_Toc107995492)

[1 总体数据通路结构设计 4](#_Toc107995493)

[1.1 总体数据通路结构图 4](#_Toc107995494)

[2 模块定义 7](#_Toc107995495)

[2.1 GPR模块定义 7](#_Toc107995496)

[2.2 ALU模块定义 10](#_Toc107995502)

[2.3 EXT模块定义 12](#_Toc107995507)

[2.4 DM模块定义 14](#_Toc107995512)

[2.5 Controller模块定义 16](#_Toc107995517)

[2.6 PC模块定义 21](#_Toc107995522)

[2.7 NextPC模块定义 23](#_Toc107995527)

[2.8 IM模块定义 25](#_Toc107995532)

[2.9 MUX1模块定义 26](#_Toc107995537)

[2.10 MUX2模块定义 27](#_Toc107995542)

[2.11 MUX3模块定义 29](#_Toc107995547)

[2.12 im\_reg模块定义 30](#_Toc107995552)

[2.13 dm\_reg模块定义 31](#_Toc107995557)

[2.14 A\_reg模块定义 32](#_Toc107995562)

[2.15 B\_reg模块定义 33](#_Toc107995567)

[2.16 ALU\_reg模块定义 34](#_Toc107995572)

[2.16 LB模块定义 35](#_Toc107995577)

[2.17 SB模块定义 36](#_Toc107995582)

[3 设计的机器指令描述 37](#_Toc107995587)

[4 状态转移图 38](#_Toc107995588)

[5 测试程序 38](#_Toc107995589)

[6 测试结果 41](#_Toc107995590)

[6.1 GPR运行结果 41](#_Toc107995591)

[6.2 DM运行结果 42](#_Toc107995592)

[6.3 波形图 43](#_Toc107995593)

[Project2 44](#_Toc107995594)

[1 总体数据通路结构设计 44](#_Toc107995595)

[1.1 总体数据通路结构图 44](#_Toc107995596)

[2 模块定义 45](#_Toc107995597)

[2.1 cp0模块定义 45](#_Toc107995598)

[2.2 bridge模块定义 48](#_Toc107995603)

[2.3 timer模块定义 50](#_Toc107995608)

[2.4 inputdev模块定义 53](#_Toc107995613)

[2.5 outputdev模块定义 54](#_Toc107995618)

[3 设计的机器指令描述 55](#_Toc107995623)

[4 状态转移图 55](#_Toc107995624)

[5 测试程序 56](#_Toc107995625)

[6 测试结果 58](#_Toc107995626)

[7 总结与收获 58](#_Toc107995627)

# Project1

# 1 总体数据通路结构设计

## 1.1 总体数据通路结构图

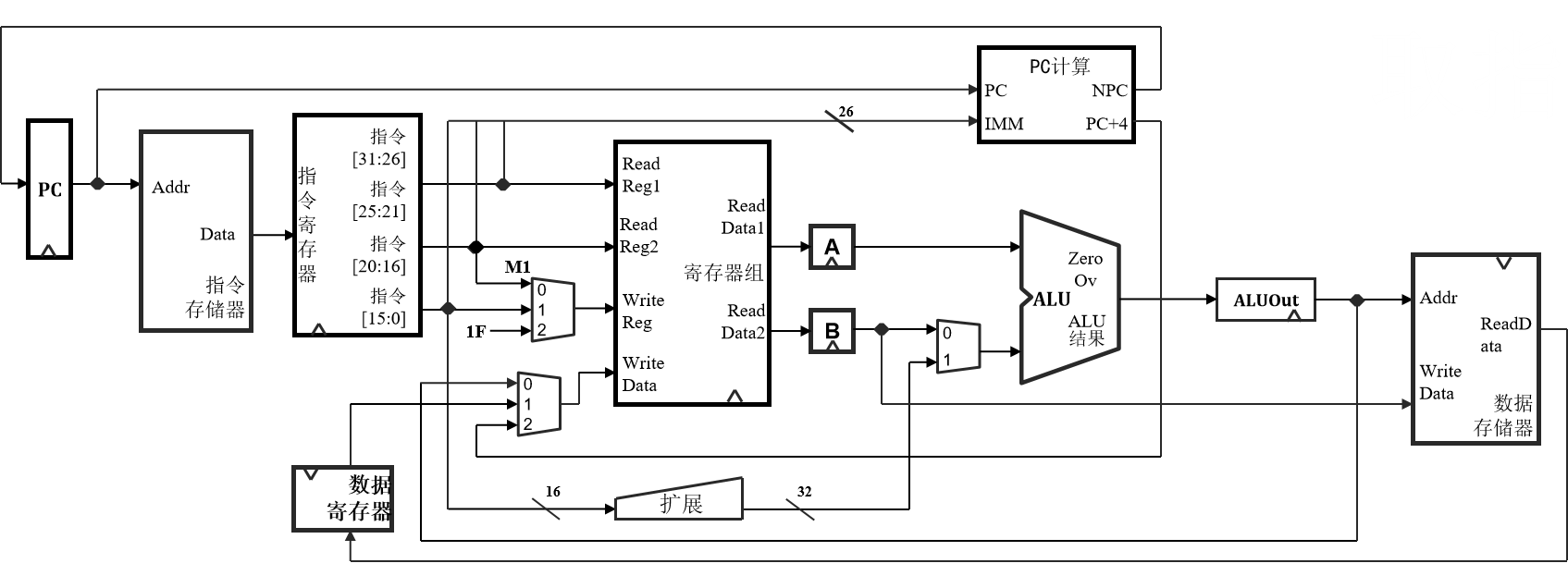


图 1：总体数据通路结构图

|  |
| --- |
| 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

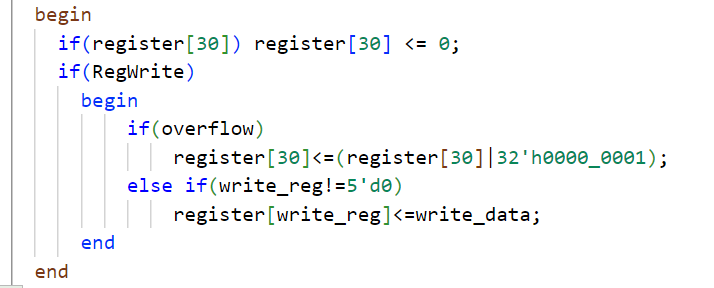
|  |  |  |
| --- | --- | --- |
| **信号名** | **方向** | **描述** |
| RegWrite | I | 寄存器写使能。  1：当前指令写入寄存器  0：当前指令不写入寄存器 |
| reset | I | 复位信号。  1：复位  0：无效 |
| clk | I | 时钟信号。 |
| ins[31:0] | I | 传入的32位指令。 |
| overflow | I | 溢出标志。  1：溢出  0：未溢出 |
| write\_reg[4:0] | I | 当前指令需要写入的寄存器地址。 |
| write\_data[31:0] | 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：



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

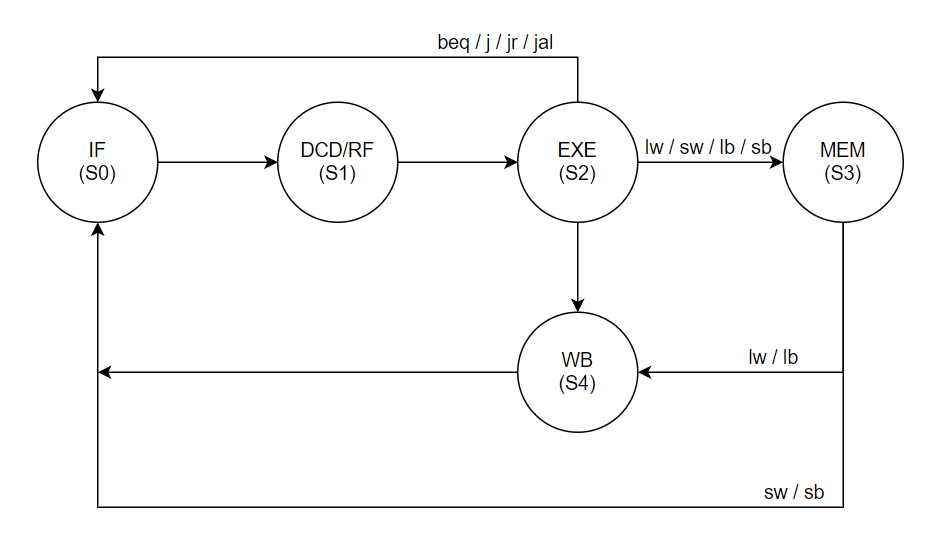


图 3：FSM有限状态机图

# 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

|  |  |
| --- | --- |
| 图 4：Mars中的GPR | 图 5：modelsim中的GPR |

## 6.2 DM运行结果

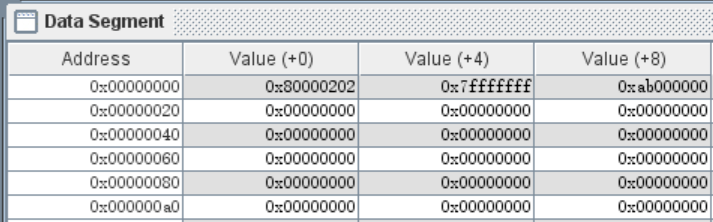


图 6：Mars中的DM

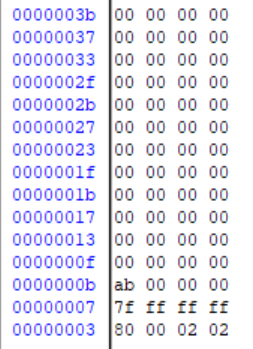




图 7：modelsim中的DM

## 6.3 波形图

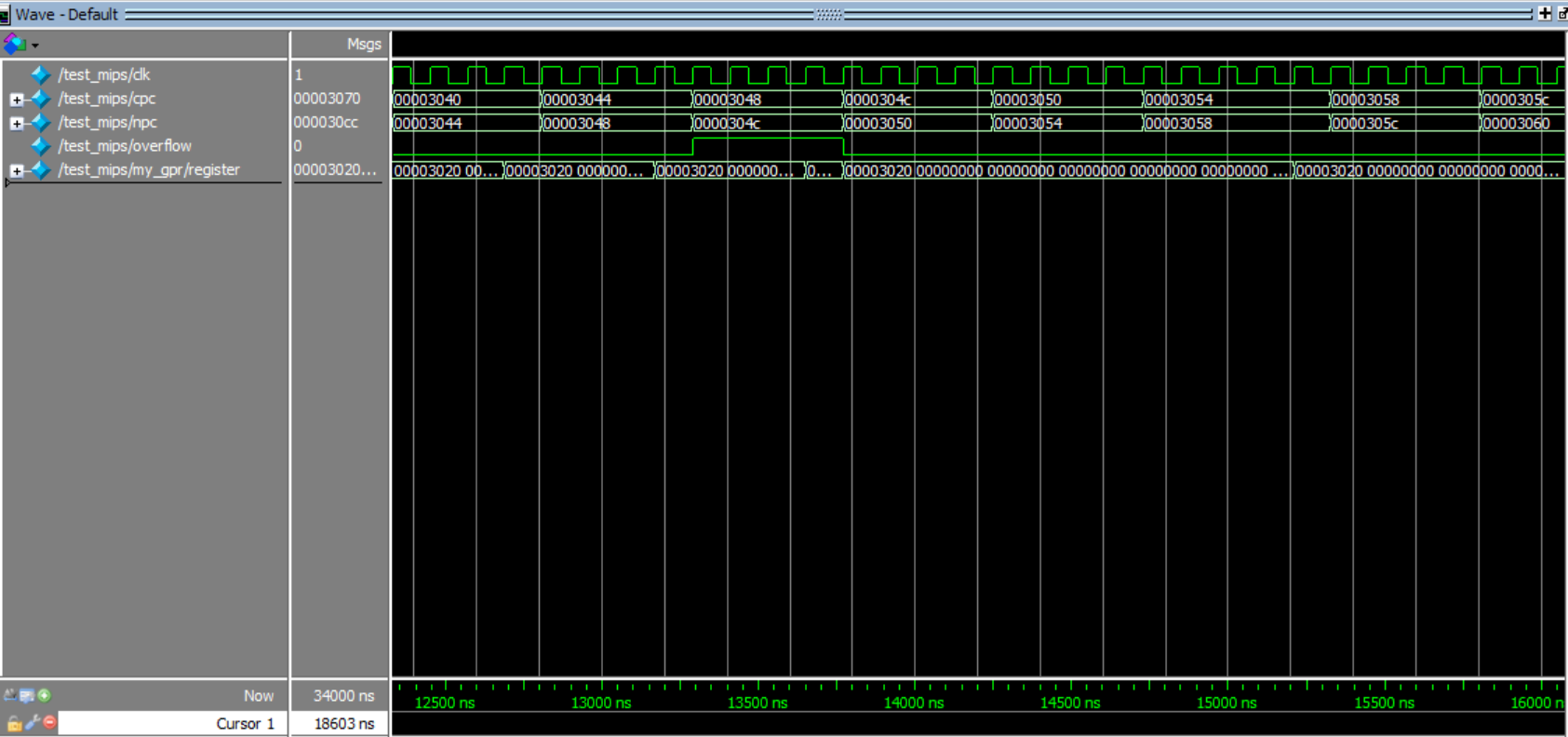


图 8：局部仿真结果截图

# Project2

# 1 总体数据通路结构设计

## 1.1 总体数据通路结构图

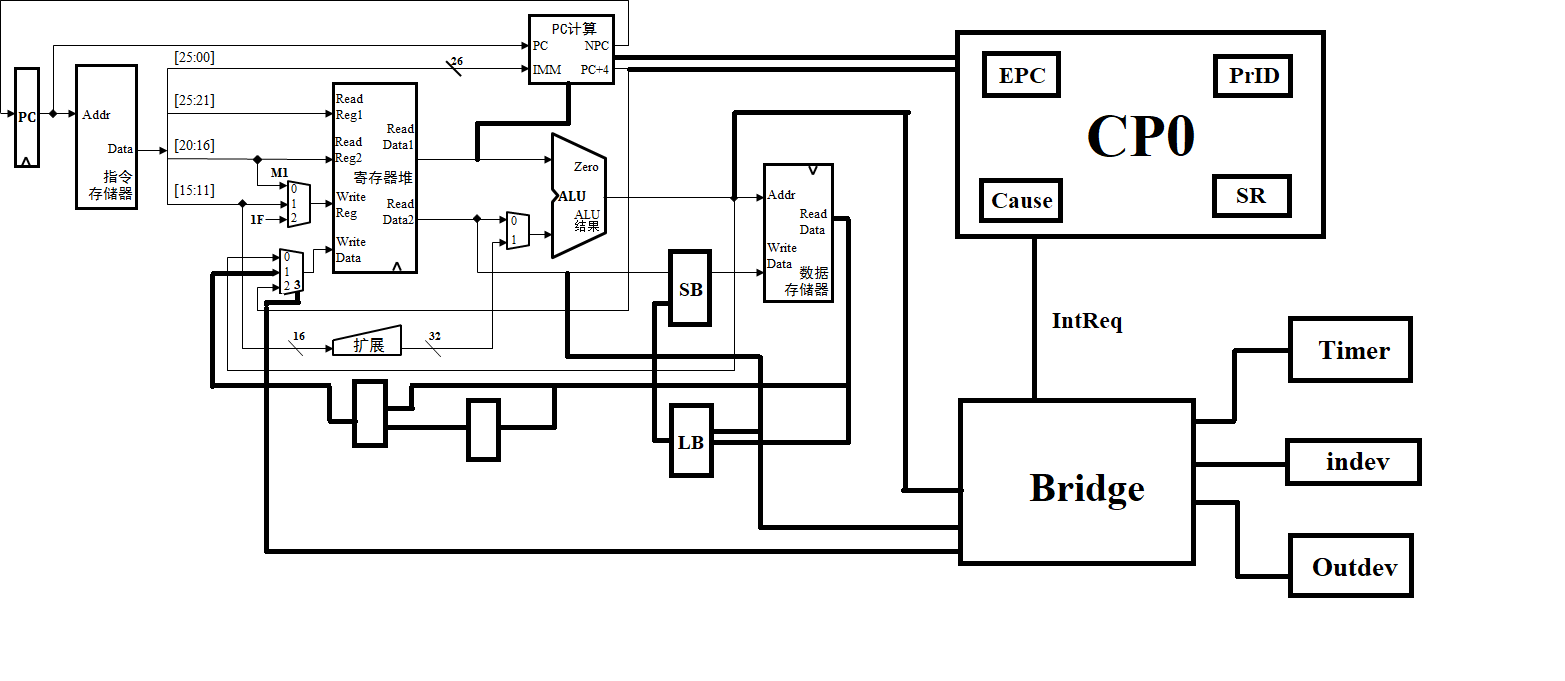


图 1：总体数据通路结构图

|  |
| --- |
| 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

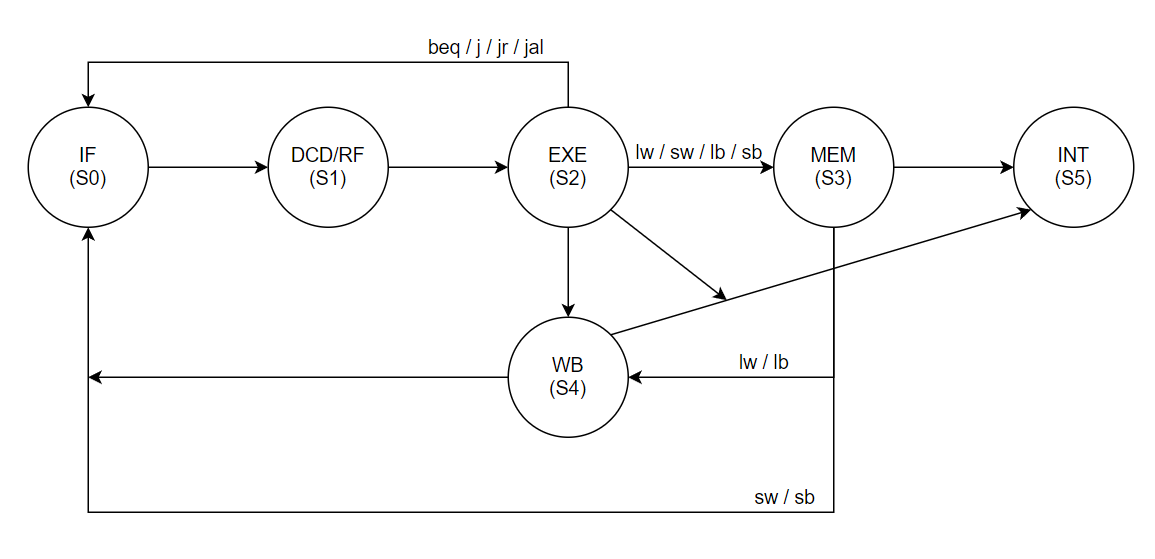
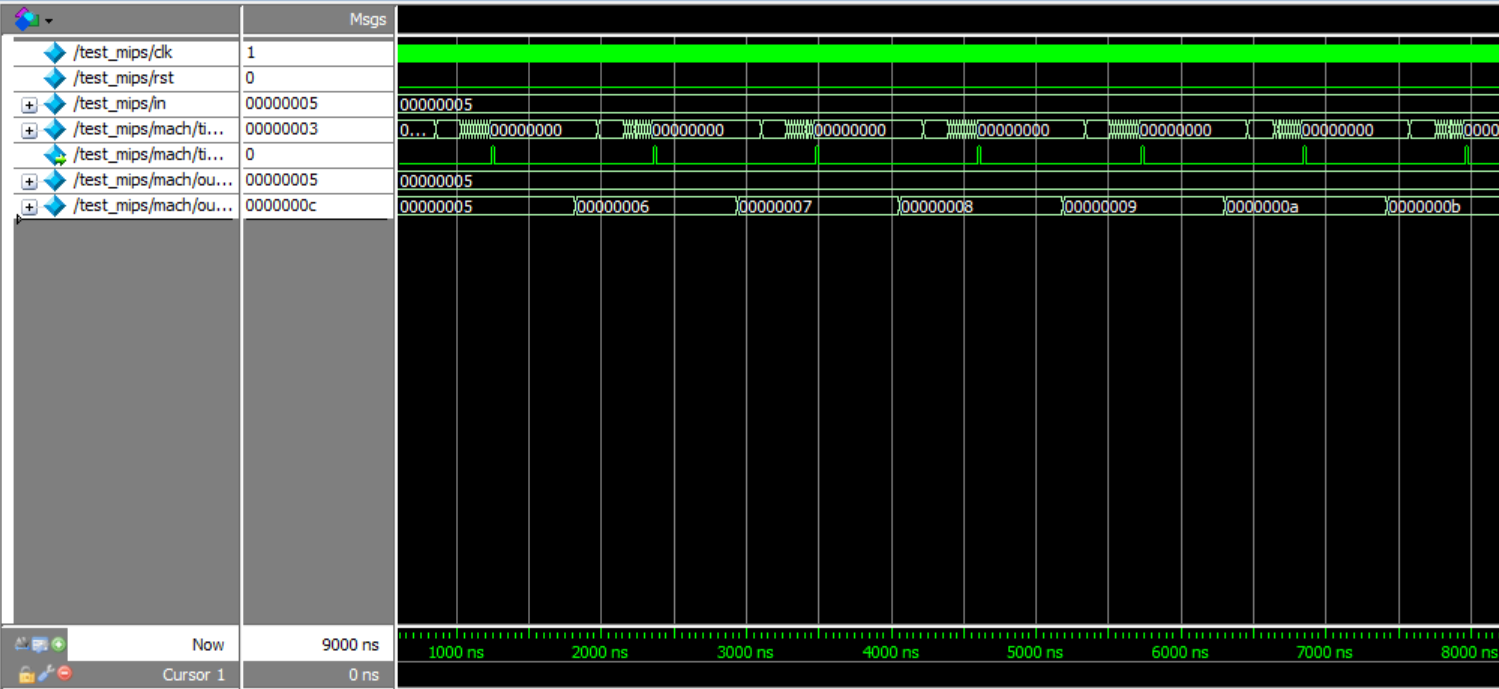


图 3：FSM有限状态机图

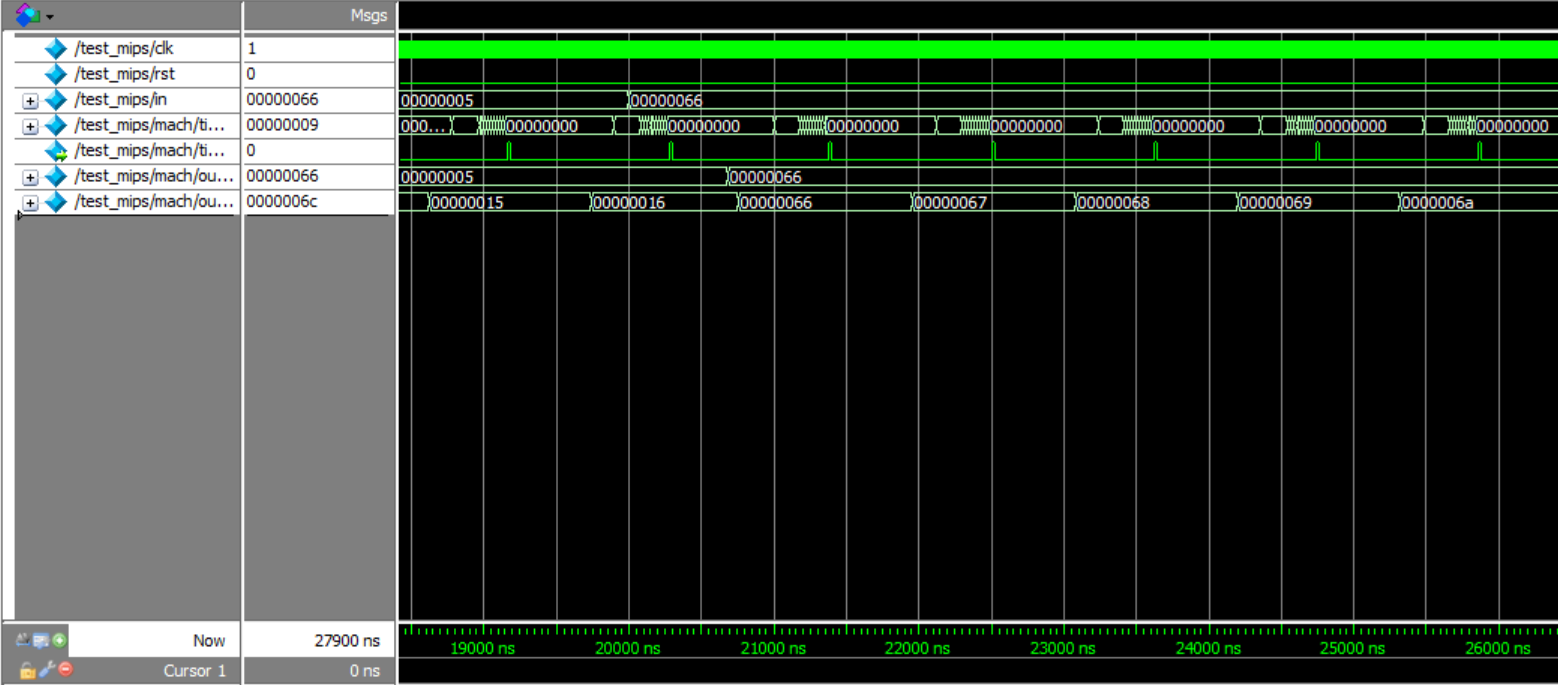
# 5 测试程序

|  |  |
| --- | --- |
| **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。