Project4 Verilog 完成单周期处理器开发实验报告

一．整体结构

**1.处理器为32位处理器。**

**2.处理器应支持的指令集为：** **处理器应支持指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}。**

**3.nop机器码为0x00000000， 即空指令，不进行任何有效行为（修改寄存器等）**

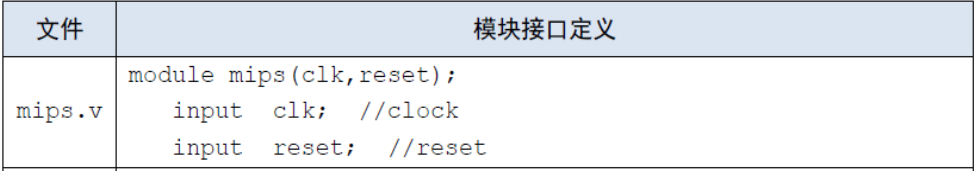
4.addu,subu可以不支持溢出。

5.处理器为**单周期**设计。

6.**不**需要考虑**延迟槽**。

7.需要采用**模块化**和**层次化**设计。

8.顶层文件为mips.v，接口定义如下：



二．模块规格

1.pc.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| pc.v | module pc(  input [31:0] next\_pc,  input clk,  input reset,  output reg[31:0] pc  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号  1：复位  0：无效 |
| next\_pc | I | 更新的PC（时钟上升沿更新） |
| Pc | I | PC 程序计数器 |

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0x00003000 |
| 2 | 更新pc | 时钟上升沿时改变pc=next\_pc |

module pc(

input [31:0] next\_pc,

input clk,

input reset,

output reg[31:0] pc

);

always@(posedge clk) begin

if(reset) begin

pc <= 32'h00003000;

end

else begin

pc <= next\_pc;

end

end

endmodule

2.im.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| im.v | module im(  input [31:0] PC,  output[31:0] instruction  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| PC[31:0] | I | 32位PC |
| Instruction[31:0] | O | 32位当前指令 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 取指令 | 根据 PC 从 IM 中取出指令 |

module im(

input [31:0] PC,

output[31:0] instruction

);

reg[31:0] im[0:1023];

initial begin

$readmemh("code.txt",im);

end

assign instruction = im[PC[11:2]]; //直接读 不用等时钟

endmodule

3.grf.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| grf.v | module grf(  input clk,  input reset,  input RegWrite,  input [4:0] RA1,  input [4:0] RA2,  input [4:0] WA,  input [31:0] WD,  input [31:0] PC,  output [31:0] RD1,  output [31:0] RD2  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| WD[31:0] | I | 写入数据的输入 |
| RA1[4:0] | I | 读寄存器地址 1 |
| RA2[4:0] | I | 读寄存器地址 2 |
| WA[4:0] | I | 写寄存器地址 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号  1：复位  0：无效 |
| PC[31:0] | I | 当前PC |
| RegWrite | I | 是否可以写入控制信号(随时都可以读出)  1：可以写  0：不可以写 |
| RD1[31:0] | O | 32 位数据输出 1 |
| RD2[31:0] | O | 32 位数据输出 2 |

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，所有寄存器被设置为 0x00000000 |
| 2 | 读寄存器 | 根据输入的寄存器地址读出32位数据 |
| 3 | 写寄存器 | 根据输入的地址，把输入的数据写进所选的寄存器 |

module grf(

input clk,

input reset,

input RegWrite,

input [4:0] RA1,

input [4:0] RA2,

input [4:0] WA,

input [31:0] WD,

input [31:0] PC,

output [31:0] RD1,

output [31:0] RD2

);

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

integer i;

initial begin

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

register[i] <= 0;

end

end

assign RD1 = register[RA1]; //读是组合逻辑

assign RD2 = register[RA2];

always@(posedge clk) begin //写要在上升沿进行

if(reset) begin //同步复位

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

register[i] <= 0;

end

end

else begin

if(RegWrite&&WA!=5'b00000) begin //0号不写

register[WA] <= WD;

$display("@%h: $%d <= %h", PC, WA,WD);

end

end

end

endmodule

4.alu.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| alu.v | module alu(  input [31:0] A,  input [31:0] B,  input [2:0] ALUCtrl,  output reg[31:0] Result,  output Zero  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| A[31:0] | I | 32 位输入数据 1 |
| B[31:0] | I | 32 位输入数据 2 |
| ALUCtrl[2:0] | I | 控制信号  000：与  001：或  010：加  011：减 |
| Result[31:0] | O | 32 位数据输出 |
| Zero | O | A,B是否相等的标志信号  1：相等  0：不相等 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 与 | A&B |
| 2 | 或 | A|B |
| 3 | 加 | A+B |
| 4 | 减 | A-B |
| 5 | 判零 | A=?B |

module alu(

input [31:0] A,

input [31:0] B,

input [2:0] ALUCtrl,

output reg[31:0] Result,

output Zero

);

always@(\*) begin

case(ALUCtrl)

3'b000: begin

Result = A & B;

end

3'b001: begin

Result = A | B;

end

3'b010: begin

Result = A + B;

end

3'b011: begin

Result = A - B;

end

3'b100: begin

Result = 0;

end

3'b101: begin

Result = 0;

end

3'b110: begin

Result = 0;

end

3'b111: begin

Result = 0;

end

default: begin

Result = 0;

end

endcase

end

assign Zero = (A==B) ? 1'b1 : 1'b0;

endmodule

5.dm.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| dm.v | module dm(  input clk,  input reset,  input MemWrite,  input MemRead,  input [31:0] MemAddr,  input [31:0] WD,  input [31:0] PC,  output [31:0] RD  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号  1：复位  0：无效 |
| MemWrite | I | 读写控制信号  1：写操作 |
| MemRead | I | 读写控制信号  1：读操作 |
| MemAddr[31:0] | I | 操作寄存器地址 |
| WD[31:0] | I | 输入（写入内存）的32位数据 |
| PC[31:0] | I | 当前PC |
| RD[31:0] | O | 32 位数据输出 |

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，所有数据被设置为 0x00000000 |
| 2 | 读 | 根据输入的寄存器地址读出数据 |
| 3 | 写 | 根据输入的地址，把输入的数据写入 |

module dm(

input clk,

input reset,

input MemWrite,

input MemRead,

input [31:0] MemAddr,

input [31:0] WD,

input [31:0] PC,

output [31:0] RD

);

reg[31:0] dm[0:1023];

integer i;

initial begin

for(i=0;i<1024;i=i+1) begin

dm[i]<=0;

end

end

assign RD = MemRead ? dm[MemAddr[11:2]] : 0; //直接读 ？：

always@(posedge clk) begin //写在上升沿

if(reset) begin //同步复位

for(i=0;i<1024;i=i+1) begin

dm[i]<=0;

end

end

else begin

if(MemWrite) begin

dm[MemAddr[11:2]] <= WD;

$display("@%h: \*%h <= %h",PC, MemAddr,WD);

end

end

end

endmodule

6. ext.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| ext.v | module ext(  input [15:0] in,  output[31:0] out,  input ExtOp  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| In[15:0] | I | 16 位数据输入 |
| Out[31:0] | O | 32 位数据输出 |
| ExtOp | I | 控制信号  0：高位符号扩展  1：高位0扩展 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 高位符号扩展 | 高16位补符号位 |
| 2 | 高位0扩展 | 高16位补0 |

module ext(

input [15:0] in,

output[31:0] out,

input ExtOp

);

//always@(\*) begin

// case(ExtOp)

// 1'b1: out <= {{16{1'b0}},in};

// 1'b0: out <= {{16{in[15]}},in};

// endcase

//end

assign out = ExtOp ? {{16{1'b0}},in} : {{16{in[15]}},in};

endmodule

7. controller.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| controller.v | module controller(  input [5:0] op,  output reg[1:0] RegDst,  output reg ALUSrc,  output reg RegWrite,  output reg MemRead,  output reg MemWrite,  output reg [1:0] MemtoReg,  output reg ExtOp,  output reg Branch1,  output reg Branch2,  output reg [2:0] ALUOp  ); |

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| Op[5:0] | I | 6位opcode段 |
| Func[5:0] | I | 6位func段 |
| RegDst[1:0] | O | 写地址控制 选择RT,RD |
| ALUSrc | O | ALU第二操作数选择控制 |
| RegWrite | O | GRF 写入控制 |
| MemRead | O | DM读信号 |
| MemWrite | O | DM写信号 |
| MemToReg[1:0] | O | GRF写入数据的选择信号 |
| ExtOp | O | 高位扩展方式选择信号 |
| Branch1 | O | 判断是否为beq指令的信号 是则为1 |
| ALUCtrl[2:0] | O | ALU的控制信号 |
| Branch2 | O | 判断是不是jal/j指令 是则为1 |
| Branch3 | O | 判断是不是jr指令 是则为1 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 产生控制信号 | 产生控制信号 |

三．控制器设计思路

数据通路如下

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | Adder | | PC | IM.A | GRF | | | | ALU | | DM | | EXT | Nadd | | Shift |
| A | B | RA1 | RA2 | WA | WD | ALU | B | A | WD | A | B |  |
| R型 | PC | 4 | Adder | PC | Rs | Rt | Rd | ALU | RF.RD1 | RF.RD2 |  |  |  |  |  |  |
| lw | PC | 4 | Adder | PC | Rs |  | Rt | DM.RD | RF.RD1 | sign\_ext | ALU |  | imm16 |  |  |  |
| sw | PC | 4 | Adder | PC | Rs | Rt |  |  | RF.RD1 | sign\_ext | ALU | RF.RD2 | imm16 |  |  |  |
| beq | PC | 4 | Adder  /Nadd | PC | Rs | Rt |  |  | RF.RD1 | RF.RD2 |  |  | imm16 | Adder | Shift | sign\_ext |
| ori | PC | 4 | Adder | PC | Rs |  | Rt | ALU | RF.RD1 | zero\_ext |  |  | imm16 |  |  |  |
| lui | PC | 4 | Adder | PC |  |  | Rt | imm+016 |  |  |  |  |  |  |  |  |
| nop | PC | 4 | Adder | PC |  |  |  |  |  |  |  |  |  |  |  |  |
| jal | PC | 4 | PCj | PC |  |  | 0x1f |  |  |  |  |  |  |  |  |  |
| jr  R型 | PC | 4 | ALU | PC | RS | Rt | Rd |  | RF.RD1 | RF.RD2 |  |  |  |  |  |  |

由此可见需要以下几个MUX多路选择器

1.beq指令 PC有两种选择 PC=Adder输出或者Nadd的输出 选择信号为Branch1

2.GRF的WA端选择Rd,Rt需要一个MUX，控制信号RegDst[1:0]

3.GRF的WD输入端，有三种选择：RF.RD2，ALU的输出，lui指令直接对imm16后边补16位0，需要2选4MUX,选择信号MemToReg[1:0]

4.两种扩展方式的选择（符号扩展，0扩展）选择信号EXTOp

5.ALU的B端两种选择，RF.RD2或EXT的输出，选择信号ALUSrc

6.j/jal指令 跳转地址的选择 Branch2

7.jr指令 跳转地址的选择 Branch3

除了上述Branch1, ALUSrc, EXTOp, MemToReg[1:0], RegDst[1:0],Branch2, Branch3,还有三个读写控制信号，RegWrite是GRF写入信号，MemRead, MemWrite是DM读写信号，ALUCtrl[2:0]是ALU控制信号，所以控制器Controller需要设计这11个控制信号。

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| Op[5:0] | I | 6位opcode段 |
| Func[5:0] | I | 6位func段 |
| RegDst[1:0] | O | 写地址控制 选择RT,RD |
| ALUSrc | O | ALU第二操作数选择控制 |
| RegWrite | O | GRF 写入控制 |
| MemRead | O | DM读信号 |
| MemWrite | O | DM写信号 |
| MemToReg[1:0] | O | GRF写入数据的选择信号 |
| ExtOp | O | 高位扩展方式选择信号 |
| Branch1 | O | 判断是否为beq指令的信号 是则为1 |
| ALUCtrl[2:0] | O | ALU的控制信号 |
| Branch2 | O | 判断是不是jal/j指令 是则为1 |
| Branch3 | O | 判断是不是jr指令 是则为1 |

可以绘制如下表格

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| name | lw | sw | beq | lui | ori | nop | jal | addu | subu | jr |
| Op5 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Op4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Op3 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Op2 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Op1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| Op0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| Func5 |  |  |  |  |  |  |  | 1 | 1 | 0 |
| Func4 |  |  |  |  |  |  |  | 0 | 0 | 0 |
| Func3 |  |  |  |  |  |  |  | 0 | 0 | 1 |
| Func2 |  |  |  |  |  |  |  | 0 | 0 | 0 |
| Func1 |  |  |  |  |  |  |  | 0 | 1 | 0 |
| Func0 |  |  |  |  |  |  |  | 1 | 1 | 0 |
| RegDst[1:0] | 00 | 0x | 0x | 00 | 00 | xx | 10 | 01 | 01 | 01 |
| ALUSrc | 1 | 1 | 0 | x | 1 | x | x | 0 | 0 | 0 |
| RegWrite | 1 | 0 | 0 | 1 | 1 | x | 1 | 1 | 1 | 1 |
| MemRead | 1 | 0 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 |
| MemWrite | 0 | 1 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 |
| MemToReg[1:0] | 10 | 00 | 00 | 01 | 00 | xx | 11 | 00 | 00 | 00 |
| EXTOp | 0 | 0 | 0 | 0 | 1 | x | x | 0 | 0 | 0 |
| Branch1 | 0 | 0 | 1 | 0 | 0 | x | 0 | 0 | 0 | 0 |
| ALUCtrl[2:0] | 010 | 010 | 011 | 111 | 001 | xxx | 111 | 010 | 011 | 010 |
| Branch2 | 0 | 0 | 0 | 0 | 0 | x | 1 | 0 | 0 | 0 |
| Branch3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

module new\_controller(

input [5:0] op,

input [5:0] func,

output reg[2:0] ALUCtrl,

output reg[1:0] RegDst,

output reg ALUSrc,

output reg RegWrite,

output reg MemRead,

output reg MemWrite,

output reg [1:0] MemtoReg,

output reg ExtOp,

output reg Branch1,

output reg Branch2,

output reg Branch3

);

always@(\*)

begin

case (op)

6'b000000://R

begin

case(func)

6'b100001: begin //addu

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b100011: begin //subu

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

6'b001000: begin //jr

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=1;

end

default: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

ALUCtrl<=3'b111;

end

endcase

end

6'b100011://lw

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=1;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b101011://sw

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=1;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b000100://beq

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=0;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=1;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

6'b001111://lui

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=0;

Branch3<=0;

end

6'b001101://ori

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=1;

Branch1<=0;

ALUCtrl<=3'b001;

Branch2<=0;

Branch3<=0;

end

6'b000011://jal

begin

RegDst[1]<=1;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=1;

Branch3<=0;

end

endcase

end

endmodule

四．主程序，数据通路设计，tb

数据通路如下

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | Adder | | PC | IM.A | GRF | | | | ALU | | DM | | EXT | Nadd | | Shift |
| A | B | RA1 | RA2 | WA | WD | ALU | B | A | WD | A | B |  |
| R型 | PC | 4 | Adder | PC | Rs | Rt | Rd | ALU | RF.RD1 | RF.RD2 |  |  |  |  |  |  |
| lw | PC | 4 | Adder | PC | Rs |  | Rt | DM.RD | RF.RD1 | sign\_ext | ALU |  | imm16 |  |  |  |
| sw | PC | 4 | Adder | PC | Rs | Rt |  |  | RF.RD1 | sign\_ext | ALU | RF.RD2 | imm16 |  |  |  |
| beq | PC | 4 | Adder  /Nadd | PC | Rs | Rt |  |  | RF.RD1 | RF.RD2 |  |  | imm16 | Adder | Shift | sign\_ext |
| ori | PC | 4 | Adder | PC | Rs |  | Rt | ALU | RF.RD1 | zero\_ext |  |  | imm16 |  |  |  |
| lui | PC | 4 | Adder | PC |  |  | Rt | imm+016 |  |  |  |  |  |  |  |  |
| nop | PC | 4 | Adder | PC |  |  |  |  |  |  |  |  |  |  |  |  |
| jal | PC | 4 | PCj | PC |  |  | 0x1f |  |  |  |  |  |  |  |  |  |
| jr  R型 | PC | 4 | ALU | PC | RS | Rt | Rd |  | RF.RD1 | RF.RD2 |  |  |  |  |  |  |

由此可见需要以下几个MUX多路选择器

1.beq指令 PC有两种选择 PC=Adder输出或者Nadd的输出 选择信号为Branch1

2.GRF的WA端选择Rd,Rt需要一个MUX，控制信号RegDst[1:0]

3.GRF的WD输入端，有三种选择：RF.RD2，ALU的输出，lui指令直接对imm16后边补16位0，需要2选4MUX,选择信号MemToReg[1:0]

4.两种扩展方式的选择（符号扩展，0扩展）选择信号EXTOp

5.ALU的B端两种选择，RF.RD2或EXT的输出，选择信号ALUSrc

6.j/jal指令 跳转地址的选择 Branch2

1.mux.v

模块接口

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| mux.v | module mux(  input [4:0] rt,  input [4:0] rd,  input [31:0] RD2,  input [31:0] imm32,  input [31:0] Result,  input [15:0] imm16,  input [31:0] RD,  input [31:0] PC,  input [1:0] RegDst,  input ALUSrc,  input [1:0] MemToReg,  input [31:0] PC4,  input [31:0] PCbeq,  input [31:0] PCj,  input Zero,  input Branch1,  input Branch2,  input [5:0] op,  input [5:0] func,  output reg[4:0] WA,  output reg[31:0] B,  output reg[31:0] WD,  output reg[31:0] next\_pc  ); |

module mux(

input [4:0] rt,

input [4:0] rd,

input [31:0] RD2,

input [31:0] imm32,

input [31:0] Result,

input [15:0] imm16,

input [31:0] RD,

input [31:0] PC,

input [1:0] RegDst,

input ALUSrc,

input [1:0] MemToReg,

input [31:0] PC4,

input [31:0] PCbeq,

input [31:0] PCj,

input Zero,

input Branch1,

input Branch2,

input Branch3,

input [5:0] op,

input [5:0] func,

output reg[4:0] WA,

output reg[31:0] B,

output reg[31:0] WD,

output reg[31:0] next\_pc

);

reg [31:0] Choice1;

reg [31:0] Choice2;

always@(\*) begin

case(RegDst)

2'b00: WA<=rt;

2'b01: WA<=rd;

2'b10: WA<=5'b11111;

2'b11: WA<=0;

endcase

case(ALUSrc)

1'b0: B<=RD2;

1'b1: B<=imm32;

endcase

case(MemToReg)

2'b00: WD<=Result;

2'b01: WD<={imm16,{16{1'b0}}};

2'b10: WD<=RD;

2'b11: WD<=PC+4;

endcase

if(Zero&&Branch1) begin

Choice1 <= PCbeq;

end

else begin

Choice1 <= PC4;

end

if(Branch2) begin

Choice2 <= PCj;

end

else begin

Choice2 <= Choice1;

end

if(Branch3) begin

next\_pc <= Result;

end

else begin

next\_pc <= Choice2;

end

end

endmodule

2.mips.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| mips.v | module mips(  input clk,  input reset  ); |

module mips(

input clk,

input reset

);

wire [5:0] op;

wire [4:0] rs;

wire [4:0] rt;

wire [4:0] rd;

wire [4:0] shamt;

wire [15:0] imm16;

wire [5:0] func;

wire [1:0] RegDst;

wire ALUSrc;

wire RegWrite;

wire MemRead;

wire MemWrite;

wire [1:0] MemToReg;

wire ExtOp;

wire Branch1;

wire Branch2;

wire Branch3;

wire [31:0] Instr;

wire [31:0] imm32;

wire [4:0] RA1;

wire [4:0] RA2;

wire [31:0] RD1;

wire [31:0] RD2;

wire Zero;

wire [2:0] ALUCtrl;

wire [4:0] WA;

wire [31:0] WD;

wire [31:0] MemAddr;

wire [31:0] MemData;

wire [31:0] RD;

wire [31:0] A;

wire [31:0] B;

wire [31:0] Result;

wire [31:0] PC;

wire [31:0] PC4;

wire [31:0] next\_pc;

wire [31:0] PCbeq;

wire [31:0] PCj;

assign op = Instr[31:26];

assign rs = Instr[25:21];

assign rt = Instr[20:16];

assign rd = Instr[15:11];

assign shamt = Instr[10:6];

assign func = Instr[5:0];

assign imm16 = Instr[15:0];

assign RA1 = rs;

assign RA2 = rt;

assign A = RD1;

assign MemAddr = Result;

assign MemData = RD2;

assign PC4 = PC+4;

assign PCbeq = PC+4+{{14{imm16[15]}},imm16,2'b00};

assign PCj = {PC[31:28],Instr[25:0],2'b00};

pc my\_pc(next\_pc,clk,reset,PC);

im my\_im(PC,Instr);

new\_controller my\_controller(op,func,ALUCtrl,RegDst,ALUSrc,RegWrite,MemRead,MemWrite,MemToReg,ExtOp,Branch1,Branch2,Branch3);

grf my\_grf(clk,reset,RegWrite,RA1,RA2,WA,WD,PC,RD1,RD2);

ext my\_ext(imm16,imm32,ExtOp);

dm my\_dm(clk,reset,MemWrite,MemRead,MemAddr,MemData,PC,RD);

alu my\_alu(A,B,ALUCtrl,Result,Zero);

mux my\_mux(rt,rd,RD2,imm32,Result,imm16,RD,PC,RegDst,ALUSrc,MemToReg,PC4,PCbeq,PCj,Zero,Branch1,Branch2,Branch3,op,func,WA,B,WD,next\_pc);

endmodule

3.tb

module test;

// Inputs

reg clk;

reg reset;

// Instantiate the Unit Under Test (UUT)

mips uut (

.clk(clk),

.reset(reset)

);

initial begin

clk = 0;

reset = 1;

#12 reset = 0;

end

always #10 clk = ~clk;

endmodule

五．测试程序

ori $a0,$0,1999

ori $a1,$a0,111

lui $a2,12345

lui $a3,0xffff

nop

ori $a3,$a3,0xffff

addu $s0,$a0,$a1

addu $s1,$a3,$a3

addu $s2,$a3,$s0

beq $s2,$s3,eee

subu $s0,$a0,$s2

subu $s1,$a3,$a3

eee:

subu $s2,$a3,$a0

subu $s3,$s2,$s1

ori $t0,$0,0x0000

sw $a0,0($t0)

nop

sw $a1,4($t0)

sw $s0,8($t0)

sw $s1,12($t0)

sw $s2,16($t0)

sw $s5,20($t0)

lw $t1,20($t0)

lw $t7,0($t0)

lw $t6,20($t0)

sw $t6,24($t0)

lw $t5,12($t0)

jal end

ori $t0,$t0,1

ori $t1,$t1,1

ori $t2,$t2,2

beq $t0,$t2,eee

lui $t3,1111

jal out

end:

addu $t0,$t0,$t7

jr $ra

out:

addu $t0,$t0,$t3

ori $t2,$t0,0

beq $t0,$t2,qqq

lui $v0,10

qqq:

lui $v0,11

机器码

340407cf

3485006f

3c063039

3c07ffff

00000000

34e7ffff

00858021

00e78821

00f09021

12530002

00928023

00e78823

00e49023

02519823

34080000

ad040000

00000000

ad050004

ad100008

ad11000c

ad120010

ad150014

8d090014

8d0f0000

8d0e0014

ad0e0018

8d0d000c

0c000c22

35080001

35290001

354a0002

110affec

3c0b0457

0c000c24

010f4021

03e00008

010b4021

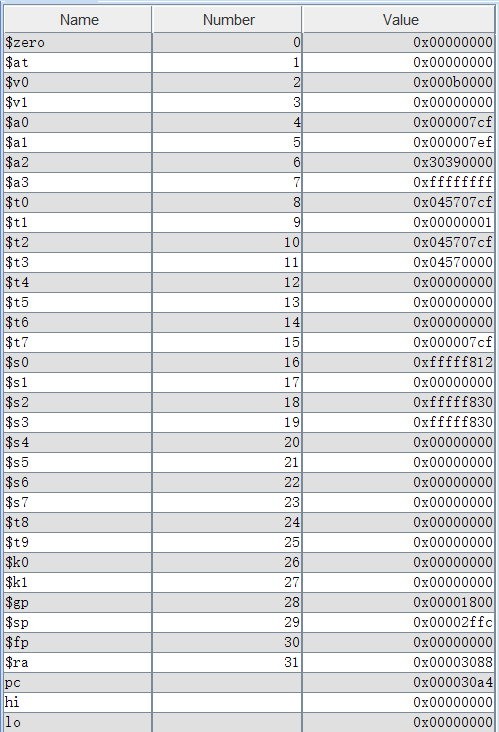
350a0000

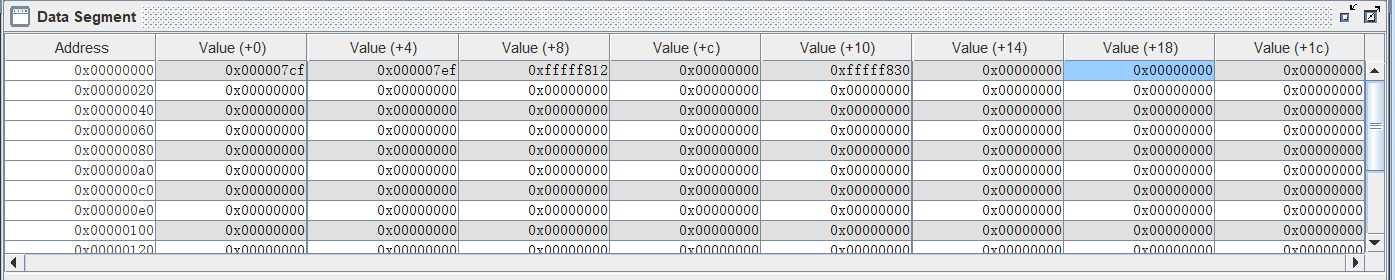
110a0001

3c02000a

3c02000b

MARS结果





ISE输出

@00003000: $ 4 <= 000007cf

@00003004: $ 5 <= 000007ef

@00003008: $ 6 <= 30390000

@0000300c: $ 7 <= ffff0000

@00003014: $ 7 <= ffffffff

@00003018: $16 <= 00000fbe

@0000301c: $17 <= fffffffe

@00003020: $18 <= 00000fbd

@00003028: $16 <= fffff812

@0000302c: $17 <= 00000000

@00003030: $18 <= fffff830

@00003034: $19 <= fffff830

@00003038: $ 8 <= 00000000

@0000303c: \*00000000 <= 000007cf

@00003044: \*00000004 <= 000007ef

@00003048: \*00000008 <= fffff812

@0000304c: \*0000000c <= 00000000

@00003050: \*00000010 <= fffff830

@00003054: \*00000014 <= 00000000

@00003058: $ 9 <= 00000000

@0000305c: $15 <= 000007cf

@00003060: $14 <= 00000000

@00003064: \*00000018 <= 00000000

@00003068: $13 <= 00000000

@0000306c: $31 <= 00003070

@00003088: $ 8 <= 000007cf

@00003070: $ 8 <= 000007cf

@00003074: $ 9 <= 00000001

@00003078: $10 <= 00000002

@00003080: $11 <= 04570000

@00003084: $31 <= 00003088

@00003090: $ 8 <= 045707cf

@00003094: $10 <= 045707cf

@000030a0: $ 2 <= 000b0000

六．思考题

数据通路设计（L0.T2）

1、根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？



**lw,sw的立即数是以字节为单位的，而设计的DM是以字为单位的，我们通过ALU运算出来的MemAddr是以字节为单位的，所以要除以4，也就是右移两位，才是真正的MemAddr，取ALU输出32位的低10位作为地址输入，即[9:0],但是需要右移两位，也就是取[11:2]才是所需要的真正的MemAddr。**这个addr信号来自于ALU的输出32位，取[11:2]。

2、在相应的部件中，**reset的优先级**比其他控制信号（不包括clk信号）都要**高**，且相应的设计都是**同步复位**。清零信号reset是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

PC,DM,GRF

PC复位要回到0x00003000处，即重现开始

DM存储了程序运行后向内存sw的数据，复位需要清空

GRF存储了程序向寄存器堆写入的数据，复位应该清空

不清空就可能影响下一次程序的执行。

控制器设计（L0.T4）

1. 列举出用Verilog语言设计控制器的几种编码方式（至少三种），并给出代码示例。

第一种 直接用case语句实现操作码和控制信号的值之间的对应

module new\_controller(

input [5:0] op,

input [5:0] func,

output reg[2:0] ALUCtrl,

output reg[1:0] RegDst,

output reg ALUSrc,

output reg RegWrite,

output reg MemRead,

output reg MemWrite,

output reg [1:0] MemtoReg,

output reg ExtOp,

output reg Branch1,

output reg Branch2,

output reg Branch3

);

always@(\*)

begin

case (op)

6'b000000://R

begin

case(func)

6'b100001: begin //addu

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b100011: begin //subu

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

6'b001000: begin //jr

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=1;

end

default: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

ALUCtrl<=3'b111;

end

endcase

end

6'b100011://lw

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=1;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b101011://sw

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=1;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

6'b000100://beq

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=0;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=1;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

6'b001111://lui

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=0;

Branch3<=0;

end

6'b001101://ori

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=1;

Branch1<=0;

ALUCtrl<=3'b001;

Branch2<=0;

Branch3<=0;

end

6'b000011://jal

begin

RegDst[1]<=1;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=1;

Branch3<=0;

end

endcase

end

endmodule

第二种 利用assign语句完成操作码和控制信号的值之间的对应；

下边的方法是模仿与或门阵列的

甚至于可以用真值表写表达式然后assign

module new\_controller2(

input [5:0] op,

input [5:0] func,

output [2:0] ALUCtrl,

output [1:0] RegDst,

output ALUSrc,

output RegWrite,

output MemRead,

output MemWrite,

output [1:0] MemtoReg,

output ExtOp,

output Branch1,

output Branch2,

output Branch3

);

wire r,lw,sw,beq,lui,ori,jal,jr,addu,subu;

assign r = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5];

assign lw = op[0]&&op[1]&&!op[2]&&!op[3]&&!op[4]&&op[5];

assign sw = op[0]&&op[1]&&!op[2]&&op[3]&&!op[4]&&op[5];

assign beq = !op[0]&&!op[1]&&op[2]&&!op[3]&&!op[4]&&!op[5];

assign lui = op[0]&&op[1]&&op[2]&&op[3]&&!op[4]&&!op[5];

assign ori = op[0]&&!op[1]&&op[2]&&op[3]&&!op[4]&&!op[5];

assign jal = op[0]&&op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5];

assign addu = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5]&&func[5]&&!func[4]&&!func[3]&&!func[2]&&!func[1]&&func[0];

assign subu = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5]&&func[5]&&!func[4]&&!func[3]&&!func[2]&&func[1]&&func[0];

assign jr = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5]&&!func[5]&&!func[4]&&func[3]&&!func[2]&&!func[1]&&!func[0];

assign RegDst[1] = jal;

assign RegDst[0] = r;

assign ALUSrc = lw||sw||ori;

assign RegWrite = r||lui||ori||lw||jal;

assign MemRead = lw;

assign MemWrite = sw;

assign MemtoReg[1] = lw||jal;

assign MemtoReg[0] = lui||jal;

assign ExtOp = ori;

assign Branch1 = beq;

assign Branch2 = jal;

assign Branch3 = jr;

assign ALUCtrl[2] = jal||lui;

assign ALUCtrl[1] = lw||sw||beq||lui||addu||subu||jr||jal;

assign ALUCtrl[0] = beq||lui||ori||subu||jal;

endmodule

第三种利用宏定义

module new\_controller3(

input [5:0] op,

input [5:0] func,

output reg[2:0] ALUCtrl,

output reg[1:0] RegDst,

output reg ALUSrc,

output reg RegWrite,

output reg MemRead,

output reg MemWrite,

output reg [1:0] MemtoReg,

output reg ExtOp,

output reg Branch1,

output reg Branch2,

output reg Branch3

);

`define R 6'b000000

`define lw 6'b100011

`define sw 6'b101011

`define lui 6'b001111

`define ori 6'b001101

`define beq 6'b000100

`define jal 6'b000011

`define addu 6'b100001

`define subu 6'b100011

`define jr 6'b001000

always@(\*)

begin

case (op)

`R:

begin

case(func)

`addu: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

`subu: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

`jr: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=1;

end

default: begin

RegDst[1]<=0;

RegDst[0]<=1;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

ALUCtrl<=3'b111;

end

endcase

end

`lw:

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=1;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

`sw:

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=0;

MemRead<=0;

MemWrite<=1;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b010;

Branch2<=0;

Branch3<=0;

end

`beq:

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=0;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=0;

Branch1<=1;

ALUCtrl<=3'b011;

Branch2<=0;

Branch3<=0;

end

`lui:

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=0;

Branch3<=0;

end

`ori:

begin

RegDst[1]<=0;

RegDst[0]<=0;

ALUSrc<=1;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=0;

MemtoReg[0]<=0;

ExtOp<=1;

Branch1<=0;

ALUCtrl<=3'b001;

Branch2<=0;

Branch3<=0;

end

`jal:

begin

RegDst[1]<=1;

RegDst[0]<=0;

ALUSrc<=0;

RegWrite<=1;

MemRead<=0;

MemWrite<=0;

MemtoReg[1]<=1;

MemtoReg[0]<=1;

ExtOp<=0;

Branch1<=0;

ALUCtrl<=3'b111;

Branch2<=1;

Branch3<=0;

end

endcase

end

endmodule

1. 根据你所列举的编码方式，说明他们的优缺点。

第一种case语句常规方式，代码略微冗长，没体现出来对应的6位op与具体指令的关系，而第三种宏定义就可以`define R 6'b000000，同时利用case语句一起，代码长度相仿，但更加清晰，第二种assign代码很短直接连线，可以采用真值表求表达式的方式，也可以像我写的这个**模仿与或门阵列**的方式，先取一些临时变量assign r = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5]; 再连接到输出的信号上（利用或门），assign ALUSrc = lw||sw||ori; 代码简单易读，直接体现了连线的关系，但不能直接清晰的看出具体信号是1还是0，而1,3两种方式则直接体现了真值表的内容，更加清晰。

在线测试相关信息（L0.T5）

1.C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分 。

add指令操作如下

temp  (GPR[rs]31||GPR[rs]) + (GPR[rt]31||GPR[rt])

if temp32 ≠ temp31 then

SignalException(IntegerOverflow)

else

GPR[rd] ← temp 31..0

Endif

addu指令操作如下

GPR[rd] ← GPR[rs] + GPR[rt]

add指令把两个操作数的最高位当做第33位，实现的33位加法，但实际上前32位的结果只跟GPR[rs]，GPR[rt]有关，即两者之和，如果有进位1，则temp32=1+ GPR[rs]31+ GPR[rt]31,没有则temp32=GPR[rs]31+ GPR[rt]31，temp31是只跟GPR[rs]，GPR[rt]有关的，计算出temp31,temp32后可以用来判断是否溢出，但如果忽略溢出，add指令保留的GPR[rd] ← temp 31..0 也就是addu所保留的GPR[rd] ← GPR[rs] + GPR[rt]，即rs,rt两个寄存器的和，跟溢出无关，所以在忽略溢出的前提下add与addu是等价的。

同样的

addi 操作为

temp ← (GPR[rs]31||GPR[rs]) + sign\_extend(immediate)

if temp 32 ≠ temp 31 then

SignalException(IntegerOverflow)

else

GPR[rt] ← temp 31..0

endif

addiu操作为

GPR[rt] ← GPR[rs] + sign\_extend(immediate)

跟上边类似，低32位的加法只跟GPR[rs] ，sign\_extend(immediate)有关，所以如果忽略溢出，addi指令保留的GPR[rd] ← temp 31..0 也就是addiu所保留的GPR[rd] ← GPR[rs] + sign\_extend(immediate)，即rs寄存器和sign\_extend(immediate)的和，跟溢出无关，所以在忽略溢出的前提下addi与addiu是等价的。

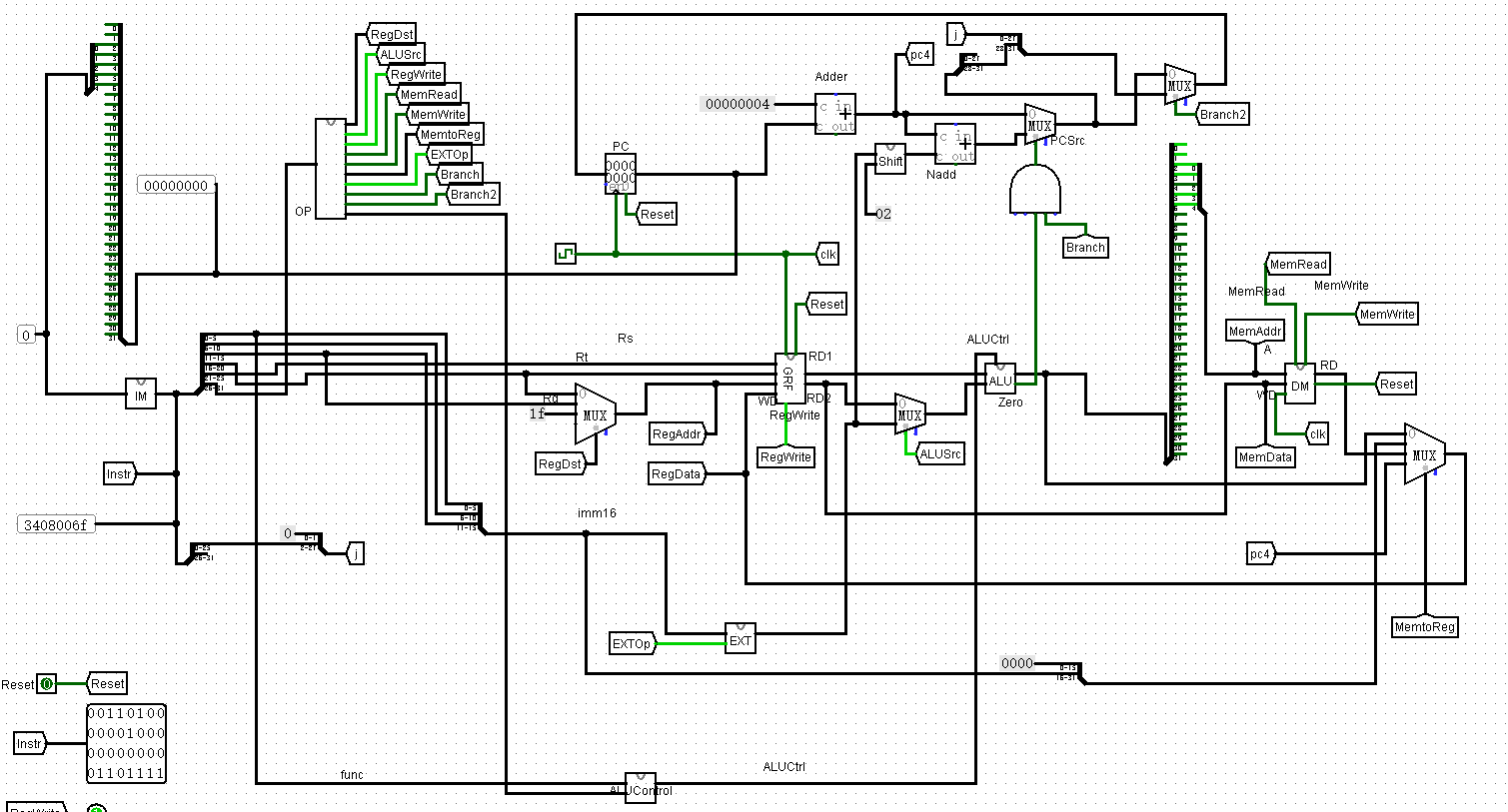
2.根据自己的设计说明单周期处理器的优缺点。

优点：设计简单，结构简单，都由统一时钟控制

缺点: （1）所有指令都在一个周期内完成，但是不同类型的指令可能具有不同的指令周期，这就导致了单周期处理器速度慢，吞吐量低，同步时钟的设计，时钟周期是常数，需要足够长以满足最慢的指令，而大部分指令执行时比较快的，比如R型不需要访问存储器，比lw要快，这就很浪费时间。（2）有三个加法器，一个用于ALU，两个用于PC的逻辑（PC+4和beq指令的跳转），而加法器是比较占用芯片面积的电路。（3）采用独立的指令存储器IM和数据存储器DM，在实际系统中不太现实。

3.简要说明jal、jr和堆栈的关系。

jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。程序调用函数，当函数调用结束后需要重新继续执行原来的程序，所以在调用函数之前，必须先存储函数返回起始点地址，存储在$ra也就是31号寄存器中。栈是用来存储局部变量的，$sp栈指针是MIPS特定的寄存器，jal调用函数，为了避免不必要的寄存器被修改，要把它先存入栈中，jr $ra返回后，再把寄存器的值从堆栈取出来，调用递归函数时更是如此。



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| name | lw | sw | beq | lui | ori | nop | jal | addu | subu | jr | blez |
| Op5 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Op4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Op3 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Op2 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| Op1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| Op0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Func5 |  |  |  |  |  |  |  | 1 | 1 | 0 |  |
| Func4 |  |  |  |  |  |  |  | 0 | 0 | 0 |  |
| Func3 |  |  |  |  |  |  |  | 0 | 0 | 1 |  |
| Func2 |  |  |  |  |  |  |  | 0 | 0 | 0 |  |
| Func1 |  |  |  |  |  |  |  | 0 | 1 | 0 |  |
| Func0 |  |  |  |  |  |  |  | 1 | 1 | 0 |  |
| RegDst[1:0] | 00 | 0x | 0x | 00 | 00 | xx | 10 | 01 | 01 | 01 | 0x |
| ALUSrc | 1 | 1 | 0 | x | 1 | x | x | 0 | 0 | 0 | 0 |
| RegWrite | 1 | 0 | 0 | 1 | 1 | x | 1 | 1 | 1 | 1 | 0 |
| MemRead | 1 | 0 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 |
| MemWrite | 0 | 1 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 |
| MemToReg[1:0] | 10 | 00 | 00 | 01 | 00 | xx | 11 | 00 | 00 | 00 | 00 |
| EXTOp | 0 | 0 | 0 | 0 | 1 | x | x | 0 | 0 | 0 | 0 |
| Branch1 | 0 | 0 | 1 | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 |
| ALUCtrl[2:0] | 010 | 010 | 011 | 111 | 001 | xxx | 111 | 010 | 011 | 010 | 011 |
| Branch2 | 0 | 0 | 0 | 0 | 0 | x | 1 | 0 | 0 | 0 | 0 |
| Branch3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Branch4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |