**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}。**

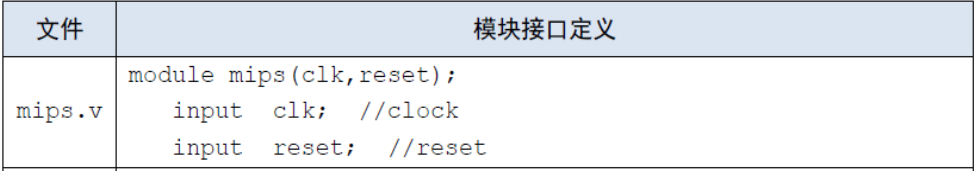
**3. nop机器码为0x00000000， 即空指令，不进行任何有效行为（修改寄存器等)**

**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.IFU.v

(参考往届学长 Roife 的设计，将 PC , Splitter , IM 三大模块功能合并在同一模块中完成)

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| IFU.v | module IFU(      input [31:0] PC\_,      input clk,      input reset,      output [31:0] Instr,      output [5:0] Op,      output [5:0] Funct,      output [4:0] shamt,      output [4:0] rd,      output [4:0] rs,      output [4:0] rt,      output [15:0] Imm,      output [25:0] addr,      output reg [31:0] PC      ); |

**a.模块接口**

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| PC\_[31:0] | I | 更新后的PC |
| clk | I | 时钟信号 |
| reset | I | 同步复位信号 |
| Instr[31:0] | I | Instruction |
| Op[5:0] | O | Opcode |
| Funct[5:0] | O | Function |
| shamt[4:0] | O | Shift Amount |
| rd[4:0] | O | rd |
| rs[4:0] | O | rs |
| rt[4:0] | O | Rt |
| Imm[15:0] | O | Imm |
| Addr[25:0] | O | Addr |
| PC[31:0] | O | PC |

**b.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 = PC\_ |
| 3 | 分位 | 将指令位拆开 |

**c.代码片段**

module IFU(

    input [31:0] PC\_,

    input clk,

    input reset,

    output [31:0] Instr,

    output [5:0] Op,

    output [5:0] Funct,

    output [4:0] shamt,

    output [4:0] rd,

    output [4:0] rs,

    output [4:0] rt,

    output [15:0] Imm,

    output [25:0] addr,

    output reg [31:0] PC

    );

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

assign Instr = IM[PC[11:2]];

assign Op = Instr[31:26];

assign Funct = Instr[5:0];

assign shamt = Instr[10:6];

assign rd = Instr[15:11];

assign rt = Instr[20:16];

assign rs = Instr[25:21];

assign Imm = Instr[15:0];

assign addr = Instr[25:0];

always@(posedge clk) begin

    if(reset) begin

        PC <= 32'h00003000;

        $readmemh("code.txt", IM);

    end

    else begin

        PC <= PC\_;

    end

end

endmodule

## 2.GRF.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| GRF.v | module GRF(      input [31:0] PC, *// Requirement*      input reset,      input clk,      input we,      input [4:0] A1,      input [4:0] A2,      input [4:0] A3,      input [31:0] WD3,      output [31:0] RD1,      output [31:0] RD2      ); |

**a.模块接口**

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| PC [31:0] | I | PC |
| clk | I | 时钟信号 |
| reset | I | 同步复位信号 |
| we | I | Writing Enable |
| A1[4:0] | I | Address 1(For Reading) |
| A2[4:0] | I | Address 2(For Reading) |
| A3[4:0] | I | Address 3(For Writing) |
| WD3[31:0] | I | Writing Data 3(For Writing) |
| RD1[31:0] | O | Reading Data 1(A1对应数据) |
| RD2[31:0] | O | Reading Data 1(A2 对应数据) |

**b.功能定义**

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 同步复位 | 当复位信号有效时，PC被设置为0x00003000 |
| 2 | 读取寄存器文件中数据 | 组合逻辑，不受时钟信号控制 |
| 3 | 写入寄存器文件 | 当we = 1，且时钟处于上升沿状态时，写入 |

**c.代码片段**

module GRF(

    input [31:0] PC, *// Requirement*

    input reset,

    input clk,

    input we,

    input [4:0] A1,

    input [4:0] A2,

    input [4:0] A3,

    input [31:0] WD3,

    output [31:0] RD1,

    output [31:0] RD2

    );

integer i;

reg [31:0] RF [31:0];

assign RD1 = RF[A1];

assign RD2 = RF[A2];

always@(posedge clk) begin

    if(reset) begin

        for(i = 0; i < 32; i = i + 1)

        RF[i] <= 32'b0;

    end

    else begin

        if(we) begin

            if(A3 != 5'b0) begin

                RF[A3] <= WD3;

                $display("@%h: $%d <= %h", PC, A3, WD3);

            end

            else RF[0] <= 32'b0;

        end

        else RF[0] <= 32'b0;

    end

end

endmodule

## 3.alu.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| alu.v | module ALU(      input [31:0] SrcA,      input [31:0] SrcB,      input [2:0] ALUControl,      output [31:0] ALUResult      ); |

**模块接口**

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| SrcA[31:0] | I | 32 位输入数据 1 |
| SrcB[31:0] | I | 32 位输入数据 2 |
| ALUControl[2:0] | I | 控制信号  000：+  001：-  010：&  011：|  100: <<  101: >>  110: >>>  111: ^(异或) |
| ALUResult[31:0] | O | 32 位数据输出 |

**代码片段**

module ALU(

    input [31:0] SrcA,

    input [31:0] SrcB,

    input [2:0] ALUControl,

    output [31:0] ALUResult

    );

assign ALUResult = (ALUControl == `ALU\_add)? SrcA + SrcB :

                   (ALUControl == `ALU\_sub)? SrcA - SrcB :

                   (ALUControl == `ALU\_and)? SrcA & SrcB :

                   (ALUControl == `ALU\_or)?  SrcA | SrcB :

                   (ALUControl == `ALU\_sll)? SrcA << SrcB :

                   (ALUControl == `ALU\_srl)? SrcA >> SrcB :

                   (ALUControl == `ALU\_sra)? SrcA >>> SrcB :

                   (ALUControl == `ALU\_xor)? SrcA ^ SrcB : 32'b0;

endmodule

## 5.DM.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| DM.v | module DM(      input  [31:0] PC, *// Requirement*      input  clk,      input  reset,      input  [31:0] A,      input  [31:0] WD,      input  we,      input  [2:0] DMType,      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 |
| we | I | 读写控制信号  1：写操作 |
| A[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 |
| DMType[2:0] | I | 判断待写入/读出数据类型  000: word  001: halfword  010: Byte  011:unsigned\_halfword  100:unsigned\_Byte |
| 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 | 处理对应地址，把输入的数据写入 |

**代码片段**

`include "const.v"

module DM(

    input  [31:0] PC, *// Requirement*

    input  clk,

    input  reset,

    input  [31:0] A,

    input  [31:0] WD,

    input  we,

    input  [2:0] DMType,

    output [31:0] RD

    );

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

wire [9:0] waddr = A[11:2];

integer i;

assign RD  =    (DMType == `DMType\_word)?  DataMemory[waddr] :

                (DMType == `DMType\_half)? {{16{DataMemory[waddr][16 \* A[1] + 15]}},{DataMemory[waddr][16 \* A[1]  +: 16]}} :

                (DMType == `DMType\_byte)? {{24{DataMemory[waddr][8 \*  A[1:0] + 7 ]}},{DataMemory[waddr][8 \* A[1:0] +:  8]}} :

                (DMType == `DMType\_uhalf)? {16'b0,{DataMemory[waddr][16 \* A[1]  +: 16]}} :

                (DMType == `DMType\_ubyte)? {24'b0,{DataMemory[waddr][8 \* A[1:0] +:  8]}} : DataMemory[waddr];

always@(posedge clk) begin

    if(reset) begin

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

        DataMemory[i] <= 32'b0;

    end

    else begin

        if(we) begin

            case(DMType)

                `DMType\_word: begin

                    DataMemory[waddr]  <=  WD;

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

                end

                `DMType\_half: begin

                    DataMemory[waddr][16 \* A[1] +: 16]  <=  WD[15:0];

                    $display("@%h: \*%h <= %h", PC, A, {16'b0,WD[15:0]});

                end

                `DMType\_byte: begin

                    DataMemory[waddr][8 \* A[1:0] +: 8]  <=  WD[7:0];

                    $display("@%h: \*%h <= %h", PC, A, {24'b0,WD[7:0]});

                end

                     default: ;

            endcase

        end

        else ;

    end

end

endmodule

## 6. EXT.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| EXT.v | module EXT(      input [15:0] Imm,      input EXTOp,      output [31:0] ExtendedImm      ); |

**模块接口**

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

**代码片段**

module EXT(

    input [15:0] Imm,

    input EXTOp,

    output [31:0] ExtendedImm

    );

assign  ExtendedImm = (EXTOp == 1)? {{16{Imm[15]}}, {Imm}} : {16'b0, {Imm}};

endmodule

## 7. CU.v

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| CU.v | module CU(      input [5:0] Op,      input [5:0] Funct,      output EXTOp,      output [1:0] JumpOp,      output MemtoReg,      output ALUSrc,      output RegDst,      output RegWrite,      output MemWrite,      output [2:0] ALUControl,      output beq,      output [2:0] DMType,      output lui,      output jal      ); |

**模块接口**

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 功能描述 |
| Op[5:0] | I | Operator, Instruction[31:25] |
| Funct[5:0] | I | Function, Instruction[5:0] |
| EXTOp | O | ExtenderOperator |
| JumpOp[1:0] | O | JumpOperator |
| MemtoReg | O | Memory\_to\_register\_Multiplexer\_Op |
| ALUSrc | O | Control\_the\_source\_of\_SrcB |
| RegDst | O | Write\_Data\_to\_rs or rt |
| RegWrite | O | WriteEnable\_of\_GRF |
| MemWrite | O | WriteEnable\_of\_DM |
| ALUControl[2:0] | O | Details can be Reffered to in ALU Unit |
| beq | O | set 1 if the instruction is beq , Input of Branchif Unit |
| DMType[2:0] | O | 用于处理 lw、sw、lh、sh、lb、sb、lhu、 lbu等指令 |
| lui | O | set 1 if the instruction is lui |

**代码片段**

`include "const.v"

module CU(

    input [5:0] Op,

    input [5:0] Funct,

    output EXTOp,

    output [1:0] JumpOp,

    output MemtoReg,

    output ALUSrc,

    output RegDst,

    output RegWrite,

    output MemWrite,

    output [2:0] ALUControl,

    output beq,

    output [2:0] DMType,

    output lui,

    output jal

    );

wire is\_R    = (Op == 6'b000000)? 1 : 0;

wire is\_addu = (is\_R & Funct == 6'b100001)? 1 : 0;

wire is\_subu = (is\_R & Funct == 6'b100011)? 1 : 0;

wire is\_jr   = (is\_R & Funct == 6'b001000)? 1 : 0;

wire is\_ori  = (Op == 6'b001101)? 1 : 0;

wire is\_lw   = (Op == 6'b100011)? 1 : 0;

wire is\_sw   = (Op == 6'b101011)? 1 : 0;

wire is\_beq  = (Op == 6'b000100)? 1 : 0;

wire is\_lui  = (Op == 6'b001111)? 1 : 0;

wire is\_j    = (Op == 6'b000010)? 1 : 0;

wire is\_jal  = (Op == 6'b000011)? 1 : 0;

*//ALUcontrol[2:0]*

assign ALUControl = (is\_ori)?                               `ALU\_or  :

                              (is\_lw || is\_sw || is\_addu)?  `ALU\_add :

                              (is\_beq || is\_subu)?              `ALU\_sub : `ALU\_and;

*//JumpOp[1:0]*

assign JumpOp = (is\_j)?     `JumpOp\_j   :

                        (is\_jal)?   `JumpOp\_jal :

                        (is\_jr)?    `JumpOp\_jr  :   2'b0;

*//DMType[2:0]*

assign DMType     = (is\_sw || is\_lw)?     `DMType\_word    :

*/\*(is\_sh || is\_lh)?   `DMType\_half    :*

*(is\_sb || is\_lb)?   `DMType\_byte    :*

*(is\_lbu)?             `DMType\_ubyte   :*

*(is\_lhu)?             `DMType\_uhalf   : \*/* `DMType\_word;

*//1-bit signal*

assign lui = is\_lui;

assign EXTOp = is\_sw || is\_lw || is\_beq;

assign MemtoReg = is\_lw;

assign ALUSrc = is\_ori || is\_sw || is\_lw;

assign RegDst = is\_addu || is\_subu;             *// rd*

assign RegWrite = is\_addu || is\_subu || is\_ori || is\_lw || is\_lui || is\_jal;

assign MemWrite = is\_sw;

assign beq = is\_beq;

assign lui = is\_lui;

assign jal  = is\_jal;

endmodule

## 三．控制器设计思路(控制信号与指令对应表见末尾)

**代码如下**

module mips(

    input clk,

    input reset

    );

*// WriteData -> RD2/ ReadData -> RD 直接删去，使模块简洁化（DM, Prereading, Prewriting已合体）*

wire [31:0] PC\_, PC, Instr, RD1, RD2, WD3, ExtendedImm, ALUResult, RD;

wire [25:0] addr;

wire [15:0] Imm;

wire [5:0]  Op, Funct;

wire [4:0]  shamt, rd, rt, rs;

wire [2:0]  ALUControl, DMType;

wire [1:0]  JumpOp;

wire  RegWrite, RegDst, BranchtoJump, EXTOp, lui, MemtoReg, beq, jal, MemWrite;

ALU Alu (

        .SrcA(RD1),

        .SrcB((ALUSrc == 1)? ExtendedImm : RD2),

        .ALUControl(ALUControl),

        .ALUResult(ALUResult)

    );

CU Cu (

        .Op(Op),

        .Funct(Funct),

        .EXTOp(EXTOp),

        .JumpOp(JumpOp),

        .MemtoReg(MemtoReg),

        .ALUSrc(ALUSrc),

        .RegDst(RegDst),

        .RegWrite(RegWrite),

        .MemWrite(MemWrite),

        .ALUControl(ALUControl),

        .beq(beq),

        .DMType(DMType),

        .lui(lui),

        .jal(jal)

    );

BranchIf Branchif (

        .beq(beq),

        .ALUResult(ALUResult),

        .BranchtoJump(BranchtoJump)

    );

DM Dm (

        .PC(PC),

        .clk(clk),

        .reset(reset),

        .A(ALUResult),

        .WD(RD2),

        .we(MemWrite),

        .DMType(DMType),

        .RD(RD)

    );

EXT Ext (

        .Imm(Imm),

        .EXTOp(EXTOp),

        .ExtendedImm(ExtendedImm)

    );

GRF Grf (

        .PC(PC),

        .reset(reset),

        .clk(clk),

        .we(RegWrite),

        .A1(rs),

        .A2(rt),

        .A3(  (jal == 1)?    5'd31 :

              (RegDst == 1)?   rd : rt),

        .WD3( (jal == 1)?               PC + 4 :

              (lui == 1)?        {Imm, 16'b0} :

              (MemtoReg == 1)?             RD : ALUResult),

        .RD1(RD1),

        .RD2(RD2)

    );

IFU Ifu (

        .PC\_(PC\_),

        .clk(clk),

        .reset(reset),

        .Instr(Instr),

        .Op(Op),

        .Funct(Funct),

        .shamt(shamt),

        .rd(rd),

        .rs(rs),

        .rt(rt),

        .Imm(Imm),

        .addr(addr),

        .PC(PC)

    );

NPC Npc (

        .PC(PC),

        .ExtendedImm(ExtendedImm),

        .JumpOp(JumpOp),

        .BranchtoJump(BranchtoJump),

        .addr(addr),

        .RD1(RD1),

        .PC\_(PC\_)

    );

endmodule

五．测试程序

见附件

六．思考题

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所驱动的部件具有什么共同特点？

PC,DM,GRF

PC复位要回到0x00003000处，即重新开始

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。

3.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。根据你所列举的编码方式，说明他们的优缺点。

**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

**第三种直接将读入Opcode与对应指令码进行比较**

module **CU**(

    input [5:0] Op,

    input [5:0] Funct,

    output EXTOp,

    output [1:0] JumpOp,

    output MemtoReg,

    output ALUSrc,

    output RegDst,

    output RegWrite,

    output MemWrite,

    output [2:0] ALUControl,

    output beq,

    output [2:0] DMType,

    output lui,

    output jal

    );

wire is\_R     = (Op == 6'b000000)? 1 : 0;

wire is\_addu  = (is\_R & Funct == 6'b100001)? 1 : 0;

wire is\_subu  = (is\_R & Funct == 6'b100011)? 1 : 0;

wire is\_jr    = (is\_R & Funct == 6'b001000)? 1 : 0;

wire is\_ori   = (Op == 6'b001101)? 1 : 0;

wire is\_lw    = (Op == 6'b100011)? 1 : 0;

wire is\_sw    = (Op == 6'b101011)? 1 : 0;

wire is\_beq   = (Op == 6'b000100)? 1 : 0;

wire is\_lui   = (Op == 6'b001111)? 1 : 0;

wire is\_j     = (Op == 6'b000010)? 1 : 0;

wire is\_jal   = (Op == 6'b000011)? 1 : 0;

wire is\_lb    = (Op == 6'b100000)? 1 : 0;

wire is\_sb    = (Op == 6'b101000)? 1 : 0;

wire is\_lh    = (Op == 6'b100001)? 1 : 0;

wire is\_sh    = (Op == 6'b101001)? 1 : 0;

wire is\_lbu   = (Op == 6'b100100)? 1 : 0;

wire is\_lhu   = (Op == 6'b100101)? 1 : 0;

wire is\_lboez = (Op == 6'b110110) ? 1 : 0;

*//ALUcontrol[2:0]*

assign ALUControl =  (is\_ori)?                                                                                              `ALU\_or  :

                     (is\_lw || is\_sw || is\_addu || is\_lh || is\_sh || is\_lhu || is\_lb || is\_sb || is\_lbu || is\_lboez)?       `ALU\_add :

                     (is\_beq || is\_subu)?                                                                                   `ALU\_sub : `ALU\_and;

*//JumpOp[1:0]*

assign JumpOp     =     (is\_j)?     `JumpOp\_j   :

                        (is\_jal)?   `JumpOp\_jal :

                        (is\_jr)?    `JumpOp\_jr  :   2'b0;

*//DMType[2:0]*

assign DMType     =     (is\_sw || is\_lw)?       `DMType\_word    :

                        (is\_sh || is\_lh)?       `DMType\_half    :

                        (is\_sb || is\_lb)?       `DMType\_byte    :

                        (is\_lbu)?               `DMType\_byteu   :

                        (is\_lhu)?               `DMType\_halfu   :

                        (is\_lboez)?             `DMType\_lboez   :   `DMType\_word;

*//1-bit signal*

assign EXTOp = is\_sw || is\_lw || is\_beq || is\_lh || is\_sh || is\_lhu || is\_lb || is\_sb || is\_lbu || is\_lboez;

assign MemtoReg = is\_lw || is\_lh || is\_lhu || is\_lb || is\_lbu || is\_lboez;

assign ALUSrc = is\_ori || is\_sw || is\_lw || is\_sh || is\_sb || is\_lboez;

assign RegDst = is\_addu || is\_subu;             *// rd*

assign RegWrite = is\_addu || is\_subu || is\_ori || is\_lw || is\_lui || is\_jal || is\_lh || is\_lhu || is\_lb || is\_lbu || is\_lboez;

assign MemWrite = is\_sw || is\_sh || is\_sb;

assign beq = is\_beq;

assign lui = is\_lui;

assign jal = is\_jal;

endmodule

CU总共可分为两部分：

第一部分是指令判断部分。Logisim中利用与门阵列实现，Verilog中直接进行比较即可；

第二部分是控制信取值部分。Logisim中利用或门阵列实现，Verilog中可以通过case语句直接针对某一特定指令将全部控制信号赋值，与真值表对应较好，理解起来也更为直观，但指令过多时代码过于冗长；还有一种办法，单位信号使用或逻辑运算，多位信号使用assign语句赋值，代码较为简洁。

4.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

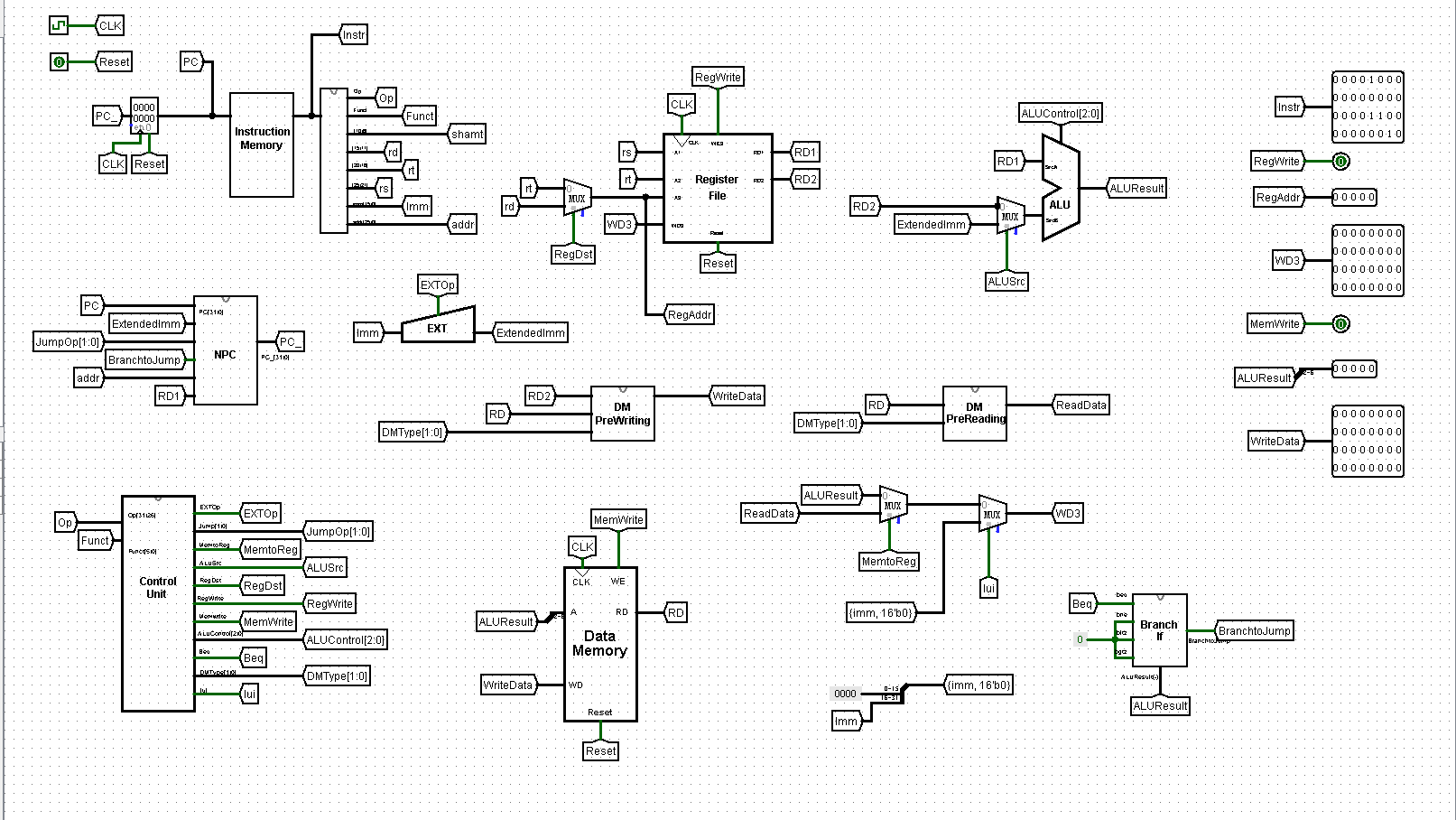
GPR[rt] ← GPR[rs] + sign\_extend(immediate)

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。

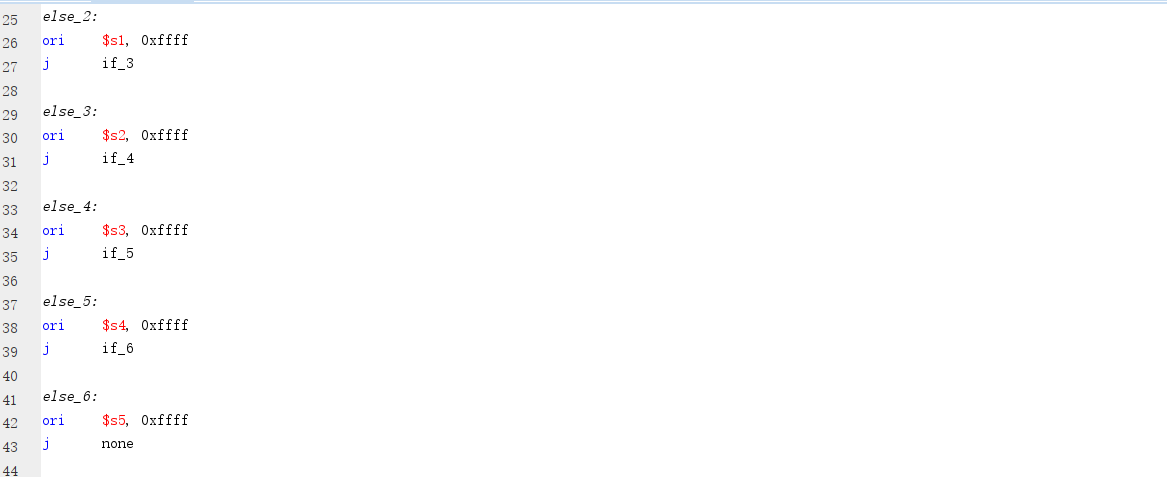
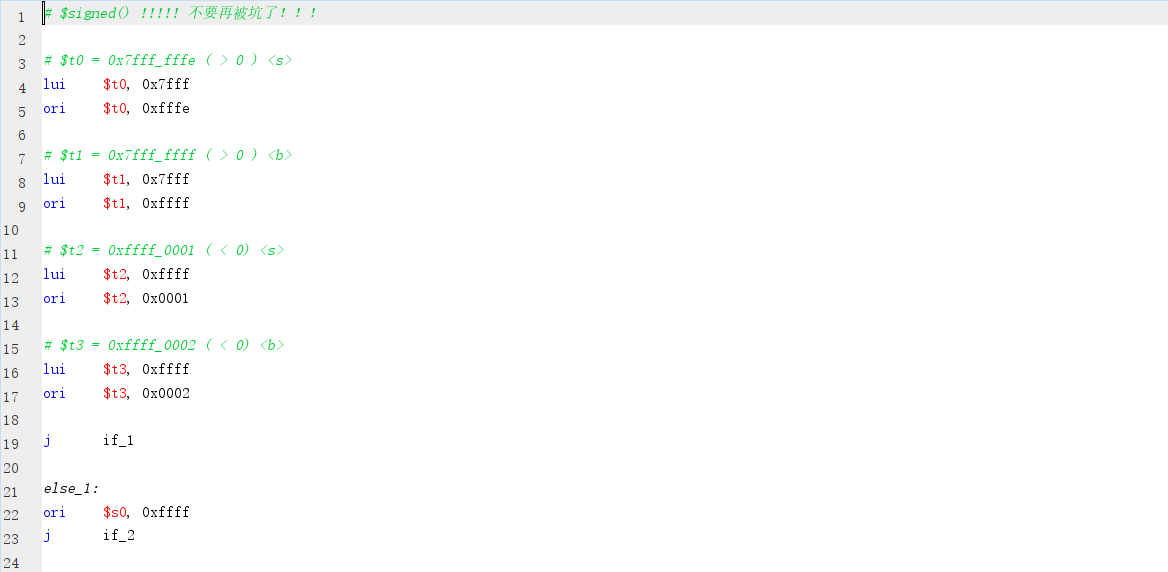
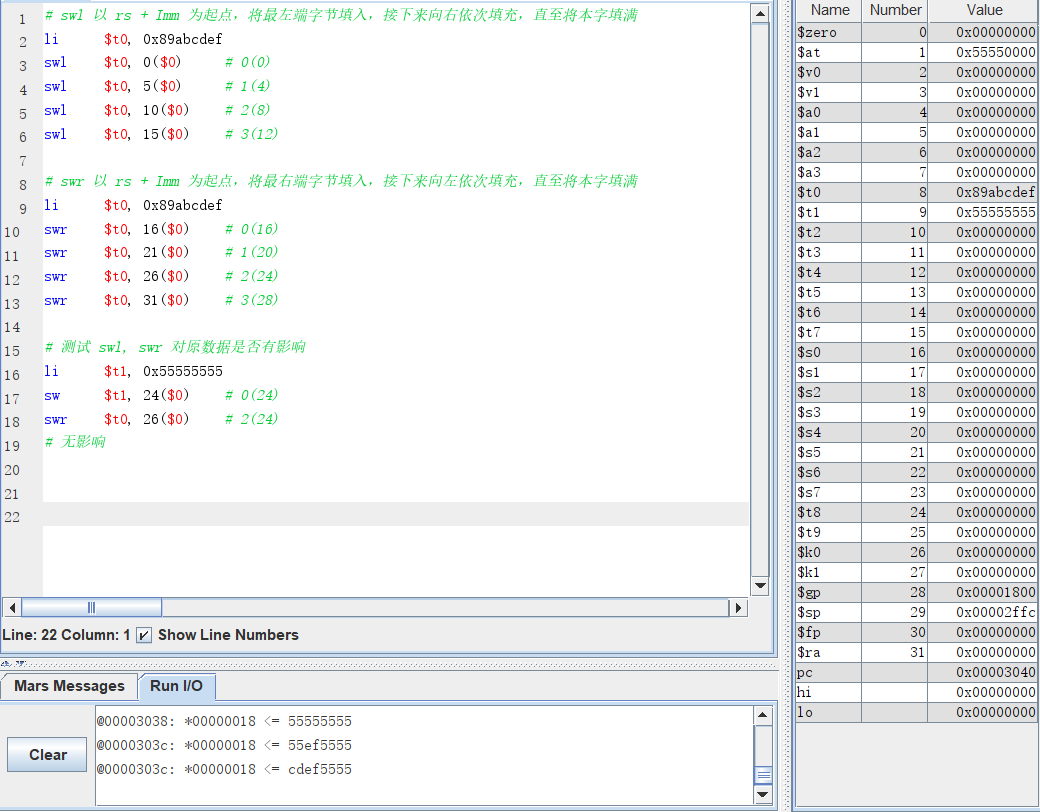
5.根据自己的设计说明单周期处理器的优缺点。

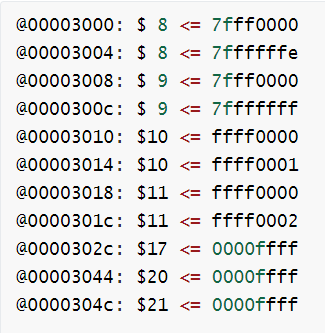
优点：设计简单，结构简单，都由统一时钟控制

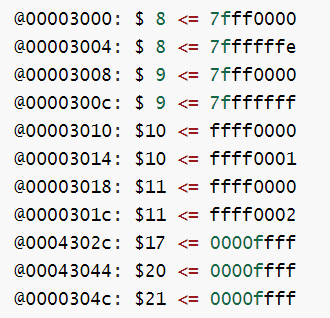
缺点: （1）所有指令都在一个周期内完成，但是不同类型的指令可能具有不同的指令周期，这就导致了单周期处理器速度慢，吞吐量低，同步时钟的设计，时钟周期是常数，需要足够长以满足最慢的指令，而大部分指令执行时比较快的，比如R型不需要访问存储器，比lw要快，这就很浪费时间。（2）有三个加法器，一个用于ALU，两个用于PC的逻辑（PC+4和beq指令的跳转），而加法器是比较占用芯片面积的电路。（3）采用独立的指令存储器IM和数据存储器DM，在实际系统中不太现实。

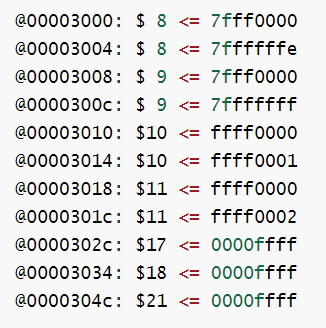
****

| **Instruction** | **Opcode** | **Funct** | **EXTOp** | **JumpOp[1:0]** | **MemtoReg** | **ALUSrc** | **RegDst** | **RegWrite** | **MemWrite** | **ALUControl[2:0]** | **Beq** | **DMType[1:0]** | **lui** | **jal** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| addu | 000000 | 100001 | X | 00 | 0 | 0 | 1 | 1 | 0 | 010 | 0 | 0 | 00 | 0 |
| addu | 000000 | 100001 | X | 00 | 0 | 0 | 1 | 1 | 0 | 010 | 0 | 0 | 00 | 0 |
| subu | 000000 | 100011 | X | 00 | 0 | 0 | 1 | 1 | 0 | 011 | 0 | 0 | 00 | 0 |
| ori | 001101 | X | 0 | 00 | 0 | 1 | 0 | 1 | 0 | 001 | 0 | 00 | 0 | 0 |
| sw | 101011 | X | 1 | 00 | X | 1 | X | 0 | 1 | 010 | 0 | 00 | 0 | 0 |
| lw | 100011 | X | 1 | 00 | 1 | 1 | 0 | 1 | 0 | 010 | 0 | 00 | 0 | 0 |
| beq | 000100 | X | 1 | 00 | X | 0 | X | 0 | 0 | 011 | 1 | X | 0 | 0 |
| lui | 001111 | X | X | 00 | X | X | 0 | 1 | 0 | X | 0 | X | 1 | 0 |
| Jal | 000011 | X | X | 01 | X | X | X | 1 | 0 | X | 0 | X | 0 | 1 |
| j | 000010 | X | 0 | 10 | X | X | X | 0 | 0 | X | 0 | X | 0 | 0 |
| jr | 000000 | 001000 | X | 11 | X | X | X | 0 | 0 | X | 0 | X | 0 | 0 |



****

****

****