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：

文件	模块接口定义

mips.v	module mips(clk,reset);
	input clk; //clock
	input reset; //reset

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？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

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]&&!f
unc[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]&&!f
unc[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]&&f
unc[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

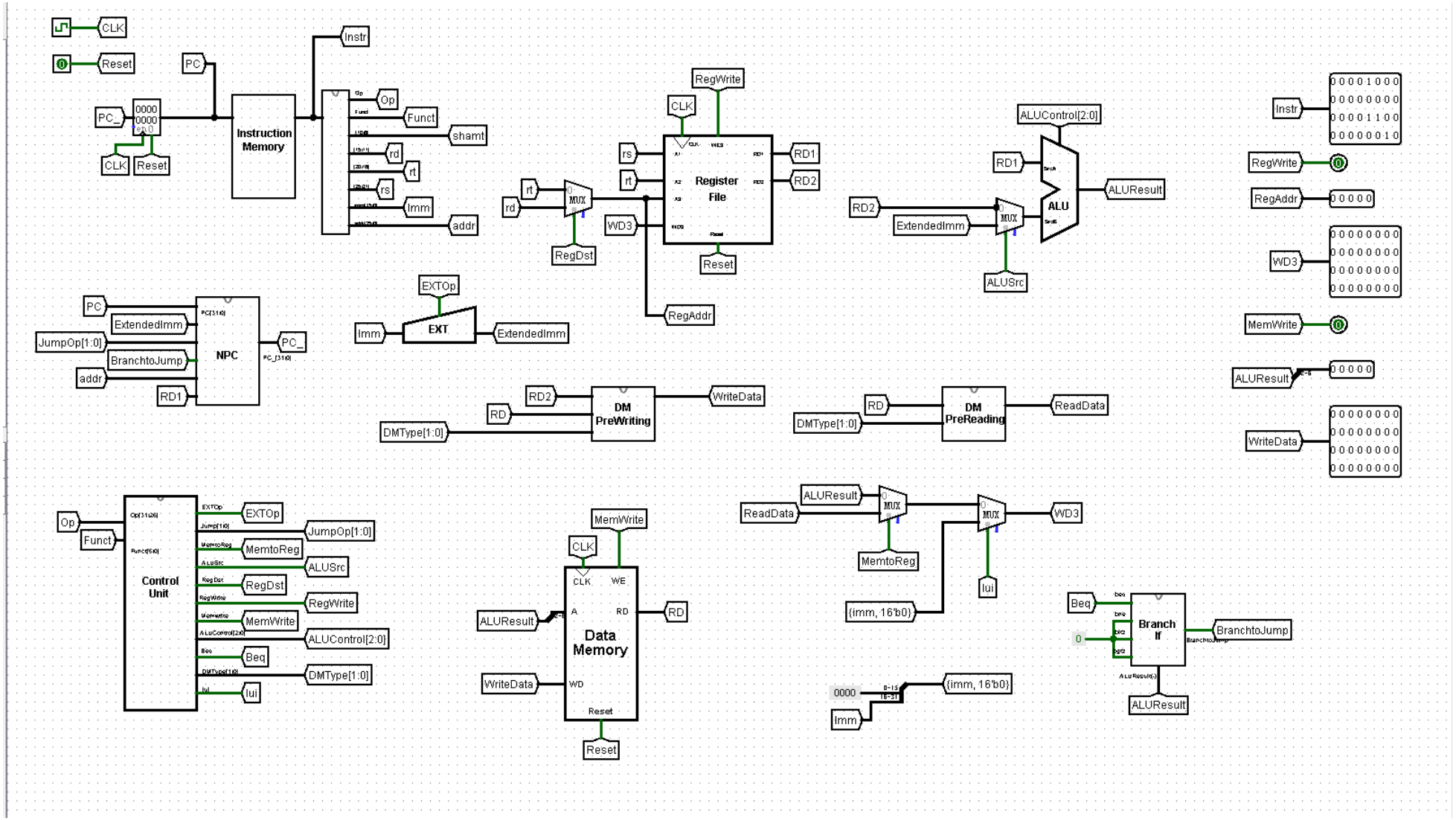
```
GPR[rt] ← GPR[rs] + sign_extend(immediate)
```

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。


CLK
Reset
PC
PC_
0000
0000
00
CLK
Reset
Instruction Memory
Instr
Op
Funct
shamt
rd
rt
rs
Imm
addr
RegWrite
CLK
rs
rt
WD3
Register File
RD1
RD2
Reset
rt
rd
MUX
RegDst
RegAddr
ALUControl[2:0]
RD1
RD2
ExtendedImm
MUX
ALU
ALUResult
ALUSrc
Instr
RegWrite
RegAddr
WD3
MemWrite
ALUResult
WriteData
PC
ExtendedImm
JumpOp[1:0]
BranchtoJump
addr
RD1
NPC
PC_
EXTOp
Imm
EXT
ExtendedImm
RD2
RD
DMType[1:0]
DM PreWriting
WriteData
RD
DMType[1:0]
DM PreReading
ReadData
Op
Funct
Control Unit
EXTOp
JumpOp[1:0]
MemtoReg
ALUSrc
RegDst
RegWrite
MemWrite
ALUControl[2:0]
Beq
DMType[1:0]
lui
MemWrite
CLK
ALUResult
WriteData
CLK
WE
A
RD
WD
Reset
Data Memory
RD
Reset
ReadData
ALUResult
MUX
MemtoReg
{imm, 16'b0}
MUX
lui
WD3
0000
Imm
{imm, 16'b0}
Beq
0
Branch If
BranchtoJump
ALUResult

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

```asm
# swl 以 rs + Imm 为起点，将最左端字节填入，接下来向右依次填充，直至将本字填满
li      $t0, 0x89abcdef
swl     $t0, 0($0)      # 0(0)
swl     $t0, 5($0)      # 1(4)
swl     $t0, 10($0)     # 2(8)
swl     $t0, 15($0)     # 3(12)

# swr 以 rs + Imm 为起点，将最右端字节填入，接下来向左依次填充，直至将本字填满
li      $t0, 0x89abcdef
swr     $t0, 16($0)     # 0(16)
swr     $t0, 21($0)     # 1(20)
swr     $t0, 26($0)     # 2(24)
swr     $t0, 31($0)     # 3(28)

# 测试 swl, swr 对原数据是否有影响
li      $t1, 0x55555555
sw      $t1, 24($0)     # 0(24)
swr     $t0, 26($0)     # 2(24)
# 无影响
```

Line: 22 Column: 1 ☑ Show Line Numbers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x55550000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x89abcdef
$t1	9	0x55555555
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00003040
hi		0x00000000
lo		0x00000000

Mars Messages | Run I/O

Clear

```
@00003038: *00000018 <= 55555555
@0000303c: *00000018 <= 55ef5555
@0000303c: *00000018 <= cdef5555
```

```asm
# $signed() !!!!! 不要再被坑了！！！

# $t0 = 0x7fff_fffe ( > 0 ) <s>
lui     $t0, 0x7fff
ori     $t0, 0xfffe

# $t1 = 0x7fff_ffff ( > 0 ) <b>
lui     $t1, 0x7fff
ori     $t1, 0xffff

# $t2 = 0xffff_0001 ( < 0) <s>
lui     $t2, 0xffff
ori     $t2, 0x0001

# $t3 = 0xffff_0002 ( < 0) <b>
lui     $t3, 0xffff
ori     $t3, 0x0002

j       if_1

else_1:
ori     $s0, 0xffff
j       if_2

else_2:
ori     $s1, 0xffff
j       if_3

else_3:
ori     $s2, 0xffff
j       if_4

else_4:
ori     $s3, 0xffff
j       if_5

else_5:
ori     $s4, 0xffff
j       if_6

else_6:
ori     $s5, 0xffff
j       none

if_1:
bne     $t0, $t0, else_1

if_2:
bgtz    $t1, else_2

if_3:
bgtz    $t2, else_3

if_4:
bltz    $t1, else_4

if_5:
bltz    $t2, else_5

if_6:
beq     $t1, $t1, else_6

none:
nop
```

```
@00003000: $ 8 <= 7fff0000
@00003004: $ 8 <= 7ffffffe
@00003008: $ 9 <= 7fff0000
@0000300c: $ 9 <= 7fffffff
@00003010: $10 <= ffff0000
@00003014: $10 <= ffff0001
@00003018: $11 <= ffff0000
@0000301c: $11 <= ffff0002
@0000302c: $17 <= 0000ffff
@00003044: $20 <= 0000ffff
@0000304c: $21 <= 0000ffff
```

```
@00003000: $ 8 <= 7fff0000
@00003004: $ 8 <= 7ffffffe
@00003008: $ 9 <= 7fff0000
@0000300c: $ 9 <= 7fffffff
@00003010: $10 <= ffff0000
@00003014: $10 <= ffff0001
@00003018: $11 <= ffff0000
@0000301c: $11 <= ffff0002
@0004302c: $17 <= 0000ffff
@00043044: $20 <= 0000ffff
@0000304c: $21 <= 0000ffff
```

```
@00003000: $ 8 <= 7fff0000
@00003004: $ 8 <= 7ffffffe
@00003008: $ 9 <= 7fff0000
@0000300c: $ 9 <= 7fffffff
@00003010: $10 <= ffff0000
@00003014: $10 <= ffff0001
@00003018: $11 <= ffff0000
@0000301c: $11 <= ffff0002
@0000302c: $17 <= 0000ffff
@00003034: $18 <= 0000ffff
@0000304c: $21 <= 0000ffff
```