2020年北航计组P4实验报告

冯张驰 19373573

一、CPU设计方案综述

（一）总体设计概述

本CPU为logisim实现的单周期MIPS - CPU，支持的指令集包含{addu, subu, ori, lw, sw, beq, lui, jalr,jr,nop,jal}。为了实现这些功能，CPU主要包含了IFU、GRF、ALU、DM、EXT、Controller六个模块。

（二）关键模块定义

1. GRF

端口定义

端口名	方向	描述
RA1[4:0]	I	RD1中数据的寄存器地址
RA2[4:0]	I	RD2中数据的寄存器地址
RD1[31:0]	O	RA1地址对应寄存器数值
RD2[31:0]	O	RA2地址对应寄存器数值
WA	I	写入数据的寄存器地址
WD	I	写入WA对应寄存器的数据
WE	I	写使能信号
clk	I	时钟信号
reset	I	异步复位信号

功能定义

1.复位。复位信号有效时所有寄存器的数值被设置为0x00000000.

2.读寄存器 根据当前输入的地址信号读出32位数据。

3.写寄存器。根据当前输入的地址信号，把输入的数据写入相应寄存器。

代码

```
`timescale 1ns / 1ps
module GRF(
    input [4:0] RA1,
    input [4:0] RA2,
    output [31:0] RD1,
    output [31:0] RD2,
    input [4:0] WA,
    input [31:0] WD,
    input WE,
    input reset,
    input clk,
     input [31:0] pc
    );
     reg [31:0] gpr [31:0];
     assign RD1=gpr[RA1];
     assign RD2=gpr[RA2];
     integer i=0;

     always@(posedge clk)begin
        if(reset)begin
            for(i=0;i<32;i=i+1)begin
                gpr[i]<=32'b0;
            end
        end
        else begin
            if(WE)begin
                if(WA)gpr[WA]<=WD;
                else gpr[WA]<=0;
            end
        end
     end

     /*always@(posedge clk)begin
        if(WE & ~reset)begin
            $display("@%h: $%d <= %h", pc, WA, WD);
        end
     end*/
endmodule
```

2. DM

端口定义

端口名	方向	描述
A[11:2]	I	数据的地址信号
WD[31:0]	I	当WE为高电平时，写入地址A的数据
RD[31:0]	O	当RE为高电平时，读出地址A的数据
RE	I	高电平时允许读数据
WE	I	高电平时允许写入数据
clk	I	时钟信号
reset	I	异步复位信号

功能定义

1. 复位：复位信号有效时，所有数据被设置为0x00000000。
2. 读：根据当前输入的寄存器地址读出数据
3. 写数据：根据输入地址，写入输入的数据。

代码

```
`timescale 1ns / 1ps
module DM(
    input [31:0] A,
    input [31:0] WD,
    output [31:0] RD,
    input RE,
    input WE,
    input clk,
    input reset,
     input [31:0] pc
    );
     reg [31:0] dm [0:1023];
     wire [11:2] addr;
     assign addr=A[11:2];
     integer i=0;
     assign RD=RE?dm[addr]:RD;

     always@(posedge clk)begin
         if(reset)begin
            for(i=0;i<=10'h3ff;i=i+1)begin
                dm[i]<=0;
            end
         end
         else begin
            if(WE)begin
                dm[addr]<=WD;
            end
            else begin
                dm[addr]<=dm[addr];
            end
         end
     end

     /*always@(posedge clk)begin
        if(WE & ~reset)begin
            $display("@%h: *%h <= %h", pc, A, WD);
        end
     end*/
endmodule
```

3. ALU

端口定义

端口名	方向	描述
A[31:0]	I	数据1
B[31:0]	I	数据2
aluCtrl[2:0]	I	控制信号000：与001：或010：加011：减101：比较大小
result	O	输出的数据
zero	O	判断A与B是否相等

功能定义

1.与：A&B

2.或：A|B

3.加：A+B

4.减：A-B

5.是否相等：A==B?

6.比较大小：A>B?

代码

```
`timescale 1ns / 1ps
module ALU(
    input [31:0] dataA,
    input [31:0] dataB,
    input [2:0] aluCtrl,
    output reg [31:0] result,
    output reg zero
    );
     always@(*)begin
        case(aluCtrl)
            3'b000:begin   //&
              result = dataA&dataB;
            end
            3'b001:begin     //|
              result = dataA|dataB;
            end
            3'b010:begin   //+
              result = dataA+dataB;
            end
            3'b011:begin   //-
              result = dataA-dataB;
            end
            default:begin
              result = result;
            end
        endcase
        zero=(dataA==dataB);
     end
endmodule
```

4. EXT

EXT端口定义

端口名	方向	描述
in[15:0]	I	16位立即数
extOp	I	扩展控制信号 0：符号扩展 1：无符号扩展
out[31:0]	O	扩展后输出的数据值

功能

根据控制信号extOp进行相应的符号扩展操作

代码

```
`timescale 1ns / 1ps
module EXT(
    input [15:0] in,
    input extOp,
    output reg [31:0] out
    );
     always@(*)begin
        case(extOp)
            0:begin
                out={{16{in[15]}},in};
            end
            1:begin
                out={{16{1'b0}},in};
            end
            default:out=out;
        endcase
     end
endmodule
```

5. IFU

本部分分为3个模块，PC,pcCalc,IM

PC模块端口定义

端口名	方向	描述
clk	I	时钟信号
reset	I	复位信号
next_pc [31:0]	I	下一时刻的PC数值
pc [31:0]	O	PC的当前值

代码

```
`timescale 1ns / 1ps
module PC(
    input clk,
    input reset,
    input [31:0] next_pc,
    output reg [31:0] pc
    );
     always@(posedge clk)begin
        if(reset)begin
          pc<=32'h00003000;
        end
        else begin
          pc<=next_pc;
        end
     end
endmodule
```

pcCalc模块端口定义

端口名	方向	描述
clk	I	时钟信号
reset	I	复位信号（同步复位）
zero	I	ALU计算结果是否为0
branch	I	当前指令是否为beq指令
instr	O	新的32位MIPS指令
imm[31:0]	I	32位地址（beq指令的立即数符号扩展后的数据）
pcPlus4	O	当前PC加4之后的值
jump	I	当前指令是否为j/jal指令
rsData [31:0]	I	GRF模块RD1输出端口的值
imm26 [25:0]	I	当前周期内指令的低26位立即数

代码

```
`timescale 1ns / 1ps
module pcCalc(
    input [31:0] pc,
    input [31:0] imm,
     input [31:0] rsData,
    output reg [31:0] npc,
    output reg [31:0] pcPlus4,
     input branch,
     input jump,
     input zero,
     input pcSrc,
     input [25:0] imm26
    );
    always@(*)begin
        pcPlus4=pc+4;
        case(pcSrc)
            0:begin
                if( zero&branch )begin
                    npc=pc+4+(imm<<2);
                end
                else if(jump)begin
                    npc={pc[31:28],imm26,{2{1'b0}}};
                end
                else begin
                    npc=pc+4;
                end
            end
            1:begin
                npc=rsData;
            end
        endcase
    end
endmodule
```

IM模块端口定义

端口名	方向	描述
pc [31:0]	I	PC当前值
instr [31:0]	O	当前PC值对应的32位指令

代码

```
`timescale 1ns / 1ps
module im(
    input [31:0] pc,
    output reg [31:0] instr
    );
     reg [31:0] instruction [0:1023];
     reg [9:0] addr;
     integer file_wr;
     initial begin
        $readmemh("code.txt",instruction);
     end
     always@(*)begin
        addr=pc[11:2];
        instr=instruction[addr];
     end
endmodule
```

IFU功能

1. 同步复位 复位信号有效时，PC被设置为**0x00003000**。
2. 取指令 根据**PC当前值**从**IM模块**中取出指令，并输出。
3. 计算下一条指令地址。
    - 当前指令是jal(pcSrc==1)，PC=PC[31:28]||imm26||0^2

- 如果当前指令是beq指令（branch==1)且ALU输出的zero信号为1，则 PC=PC+4+sign_extend(offset||0^2)
- 当前指令是jr指令，PC=GPR[RS]
- 以上三种情况之外，PC=PC+4；

6. Controller

端口定义

端口名	方向	描述
op[5:0]	I	6位opcode字段
funcode[5:0]	I	6位funcode字段
regDst[1:0]	O	GRF写地址控制信号 **2'b00**：选择rt端 **2'b01**：选择rd端 **2'b10**：选择31号寄存器
aluSrc	O	决定输入ALU模块的第二个输入端口数据是寄存器的输出值还是指令最低16位立即数扩展后的数值，**0，是前者**，**1，是后者**
regWrite	O	GRF写控制
memRead	O	DM读控制
memWrite	O	DM写控制
memToReg[1:0]	O	GRF写数据选择： **2'b00**：写入ALU的计算结果 **2'b01**：写入DM的读出数据 **2'b10**：写入lui计算所得的数据结果 **2'b11**：写入当前PC加4后的值
extOp	O	高位扩展方式选择信号 0：
branch	O	是否为beq指令
aluCtrl	O	ALU的控制信号
jump	O	传递给pcCalc模块，决定next_pc的值
pcSrc	O	传递给pcCalc模块，选择是否为jr指令

代码

```
`timescale 1ns / 1ps
module ctroller(
    input [5:0] op,
     input [5:0] func,
    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 branch,
    output reg jump,
    output reg [2:0] aluCtrl,
     output reg pcSrc
    );
     always@(*)begin
     case(op)
        6'b000000:begin  //R instruction
            regDst=1;
            aluSrc=0;
            memRead=0;
            memWrite=0;
            branch=0;
            extOp=0;
            case(func)
              6'b100001:begin  //addu
                 aluCtrl=3'b010;
                 jump=0;
                 pcSrc=0;
                 memToReg=2'b00;
                 regWrite=1;
              end
              6'b100011:begin  //subu
                 aluCtrl=3'b011;
                 jump=0;
                 pcSrc=0;
                 memToReg=2'b00;
                 regWrite=1;
              end
              6'b001000:begin  //jr
                 aluCtrl=aluCtrl;
                 jump=0;
                 pcSrc=1;
                 memToReg=memToReg;
                 regWrite=0;
              end
              6'b001001:begin  //jalr
                 aluCtrl=aluCtrl;
                 jump=1;
                 pcSrc=1;
                 memToReg=2'b11;
                 regWrite=1;
              end
              6'b000000:begin
                 jump=0;
                 pcSrc=0;
                 memToReg=0;
                 regWrite=0;
                 aluCtrl=0;
                 regDst=0;
                 aluSrc=0;
                 memRead=0;
                 memWrite=0;
                 branch=0;
                 extOp=0;
              end
            endcase
        end
        6'b100011:begin  //lw
            regDst=0;
            aluSrc=1;
```

```
            regWrite=1;
            memRead=1;
            memWrite=0;
            memToReg=2'b01;
            branch=0;
            aluCtrl=3'b010;
            jump=0;
            extOp=0;
            pcSrc=0;
        end
        6'b101011:begin  //sw
            regDst=0;
            aluSrc=1;
            regWrite=0;
            memRead=0;
            memWrite=1;
            memToReg=memToReg;
            branch=0;
            aluCtrl=3'b010;
            jump=0;
            extOp=0;
            pcSrc=0;
        end
        6'b000100:begin  //beq
            regDst=0;
            aluSrc=0;
            regWrite=0;
            memRead=0;
            memWrite=0;
            memToReg=memToReg;
            branch=1;
            aluCtrl=3'b011;
            jump=0;
            extOp=0;
            pcSrc=0;
        end
        6'b001111:begin  //lui
            regDst=0;
            aluSrc=aluSrc;
            regWrite=1;
            memRead=0;
            memWrite=0;
            memToReg=2'b10;
            branch=0;
            aluCtrl=aluCtrl;
            jump=0;
            extOp=0;
            pcSrc=0;
        end
        6'b001101:begin  //ori
            regDst=0;
            aluSrc=1;
            regWrite=1;
            memRead=0;
            memWrite=0;
            memToReg=2'b00;
            branch=0;
            aluCtrl=3'b001;
```

```
                jump=0;
                extOp=1;
                pcSrc=0;
            end
            6'b000011:begin  //jal
                regDst=2;
                aluSrc=aluSrc;
                regWrite=1;
                memRead=0;
                memWrite=0;
                memToReg=2'b11;
                branch=0;
                aluCtrl=aluCtrl;
                jump=1;
                extOp=0;
                pcSrc=0;
            end
            default:begin
                regDst=0;
                aluSrc=0;
                regWrite=0;
                memRead=0;
                memWrite=0;
                memToReg=0;
                branch=0;
                aluCtrl=0;
                jump=0;
                extOp=0;
                pcSrc=0;
            end
         endcase
         end
endmodule
```

（三）控制器设计思路

设计方法

根据每个指令的opcode,funcode信号和对应的控制信号间的关系列出真值表，在`always@(*)`块中使用用 `case` 语句块对译码和输出控制信号过程进行描述。详细代码在上一部分已展出。

主控单元译码电路真值表

端口名字\输出信号	addu	subu	jr	jalr	lw	sw	beq	lui	ori	jal
op	000000	000000	000000	000000	100011	101011	000100	001111	001101	000011
func	100001	100011	001000	001001	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx
regDst	01	01	01	01	00	xx	00	00	00	10
aluSrc	0	0	0	0	1	1	0	x	1	x
regWrite	1	1	0	1	1	0	0	1	1	1
memRead	0	0	0	0	1	0	0	0	0	0
memWrite	0	0	0	0	0	1	0	0	0	0
memToReg[1:0]	00	00	11	11	01	xx	xx	10	00	x
extOp	0	0	0	0	0	0	0	0	1	0
branch	0	0	0	0	0	0	1	0	0	0
aluCtrl[2:0]	010	011	xxx	xxx	010	010	011	xxx	001	xxx
jump	0	0	0	0	0	0	0	0	0	1
pcSrc	0	0	1	1	0	0	0	0	0	0

二、测试方案

典型测试样例

测试用例

注：此测试代码借鉴自评论区的一个同学发的资料。

```
subu $28 $28 $28
subu $29 $29 $29
ori $0, $0, 0
ori $1, $1, 1
ori $2, $2, 2
ori $3, $3, 3
ori $4, $4, 4
ori $5, $5, 5
ori $6, $6, 6
ori $7, $7, 7
ori $8, $8, 8
ori $9, $9, 9
ori $10, $10, 10
ori $11, $11, 11
ori $12, $12, 12
ori $13, $13, 13
ori $14, $14, 14
ori $15, $15, 15
ori $16, $16, 16
ori $17, $17, 17
ori $18, $18, 18
ori $19, $19, 19
ori $20, $20, 20
ori $21, $21, 21
ori $22, $22, 22
ori $23, $23, 23
ori $24, $24, 24
ori $25, $25, 25
ori $26, $26, 26
ori $27, $27, 27
ori $28, $28, 28
ori $29, $29, 29
ori $30, $30, 30
ori $31, $31, 31
ori $1, $1, 1
sw $1, 4($0)
sw $1, 8($0)
sw $1, 12($0)
sw $1, 16($0)
sw $1, 20($0)
sw $1, 24($0)
sw $1, 28($0)
sw $1, 32($0)
sw $1, 36($0)
sw $1, 40($0)
sw $1, 44($0)
sw $1, 48($0)
sw $1, 52($0)
sw $1, 56($0)
sw $1, 60($0)
sw $1, 64($0)
sw $1, 68($0)
sw $1, 72($0)
sw $1, 76($0)
sw $1, 80($0)
sw $1, 84($0)
sw $1, 88($0)
sw $1, 92($0)
sw $1, 96($0)
sw $1, 100($0)
sw $1, 104($0)
sw $1, 108($0)
sw $1, 112($0)
sw $1, 116($0)
sw $1, 120($0)
sw $1, 124($0)
ori $1, $1, 1
sw $1, 0($0)
lw $2, 0($0)
lw $3, 0($0)
lw $4, 0($0)
lw $5, 0($0)
lw $6, 0($0)
lw $7, 0($0)
lw $8, 0($0)
lw $9, 0($0)
lw $10, 0($0)
lw $11, 0($0)
lw $12, 0($0)
lw $13, 0($0)
lw $14, 0($0)
lw $15, 0($0)
lw $16, 0($0)
lw $17, 0($0)
lw $18, 0($0)
lw $19, 0($0)
lw $20, 0($0)
lw $21, 0($0)
lw $22, 0($0)
lw $23, 0($0)
lw $24, 0($0)
lw $25, 0($0)
lw $26, 0($0)
lw $27, 0($0)
lw $28, 0($0)
lw $29, 0($0)
lw $30, 0($0)
lw $31, 0($0)
ori $1 $1 907
sw $0 0($0)
lw $1 0($0)
sw $1 4($0)
lw $2 4($0)
sw $2 8($0)
lw $3 8($0)
sw $3 12($0)
lw $4 12($0)
sw $4 16($0)
lw $5 16($0)
sw $5 20($0)
lw $6 20($0)
sw $6 24($0)
lw $7 24($0)
sw $7 28($0)
lw $8 28($0)
sw $8 32($0)
lw $9 32($0)
sw $9 36($0)
lw $10 36($0)
sw $10 40($0)
lw $11 40($0)
sw $11 44($0)
lw $12 44($0)
sw $12 48($0)
lw $13 48($0)
sw $13 52($0)
lw $14 52($0)
sw $14 56($0)
lw $15 56($0)
sw $15 60($0)
lw $16 60($0)
sw $16 64($0)
lw $17 64($0)
sw $17 68($0)
lw $18 68($0)
sw $18 72($0)
lw $19 72($0)
sw $19 76($0)
lw $20 76($0)
sw $20 80($0)
lw $21 80($0)
sw $21 84($0)
lw $22 84($0)
sw $22 88($0)
lw $23 88($0)
sw $23 92($0)
lw $24 92($0)
sw $24 96($0)
lw $25 96($0)
sw $25 100($0)
```

```
lw $26 100($0)
sw $26 104($0)
lw $27 104($0)
sw $27 108($0)
lw $28 108($0)
sw $28 112($0)
lw $29 112($0)
sw $29 116($0)
lw $30 116($0)
sw $30 120($0)
lw $31 120($0)
sw $31 124($0)
lui $1 234
lui $2 234
lui $3 234
lui $4 234
lui $5 234
lui $6 234
lui $7 234
lui $8 234
lui $9 234
lui $10 234
lui $11 234
lui $12 234
lui $13 234
lui $14 234
lui $15 234
jal con
lui $1 222
subu $16 $16 $1
subu $17 $17 $1
subu $18 $18 $1
subu $19 $19 $1
subu $20 $20 $1
subu $21 $21 $1
subu $22 $22 $1
subu $23 $23 $1
subu $24 $24 $1
subu $25 $25 $1
subu $26 $26 $1
subu $27 $27 $1
subu $28 $28 $1
subu $29 $29 $1
subu $30 $30 $1
jal end
con:
addu $16 $16 $1
addu $17 $17 $2
addu $18 $18 $3
addu $19 $19 $4
addu $20 $20 $5
addu $21 $21 $6
addu $22 $22 $7
addu $23 $23 $8
addu $24 $24 $9
addu $25 $25 $10
addu $26 $26 $11
addu $27 $27 $12
addu $28 $28 $13
addu $29 $29 $14
addu $30 $30 $15
jr $31
end:
```

运行结果

```
@00003000: $28 <= 00000000
@00003004: $29 <= 00000000
@0000300c: $ 1 <= 00000001
@00003010: $ 2 <= 00000002
@00003014: $ 3 <= 00000003
@00003018: $ 4 <= 00000004
@0000301c: $ 5 <= 00000005
@00003020: $ 6 <= 00000006
@00003024: $ 7 <= 00000007
@00003028: $ 8 <= 00000008
@0000302c: $ 9 <= 00000009
@00003030: $10 <= 0000000a
@00003034: $11 <= 0000000b
@00003038: $12 <= 0000000c
@0000303c: $13 <= 0000000d
@00003040: $14 <= 0000000e
@00003044: $15 <= 0000000f
@00003048: $16 <= 00000010
@0000304c: $17 <= 00000011
@00003050: $18 <= 00000012
@00003054: $19 <= 00000013
@00003058: $20 <= 00000014
@0000305c: $21 <= 00000015
@00003060: $22 <= 00000016
@00003064: $23 <= 00000017
@00003068: $24 <= 00000018
@0000306c: $25 <= 00000019
@00003070: $26 <= 0000001a
@00003074: $27 <= 0000001b
@00003078: $28 <= 0000001c
@0000307c: $29 <= 0000001d
@00003080: $30 <= 0000001e
@00003084: $31 <= 0000001f
@00003088: $ 1 <= 00000001
@0000308c: *00000004 <= 00000001
@00003090: *00000008 <= 00000001
@00003094: *0000000c <= 00000001
@00003098: *00000010 <= 00000001
@0000309c: *00000014 <= 00000001
@000030a0: *00000018 <= 00000001
@000030a4: *0000001c <= 00000001
@000030a8: *00000020 <= 00000001
@000030ac: *00000024 <= 00000001
@000030b0: *00000028 <= 00000001
@000030b4: *0000002c <= 00000001
@000030b8: *00000030 <= 00000001
@000030bc: *00000034 <= 00000001
@000030c0: *00000038 <= 00000001
@000030c4: *0000003c <= 00000001
```

```
@000030c8: *00000040 <= 00000001
@000030cc: *00000044 <= 00000001
@000030d0: *00000048 <= 00000001
@000030d4: *0000004c <= 00000001
@000030d8: *00000050 <= 00000001
@000030dc: *00000054 <= 00000001
@000030e0: *00000058 <= 00000001
@000030e4: *0000005c <= 00000001
@000030e8: *00000060 <= 00000001
@000030ec: *00000064 <= 00000001
@000030f0: *00000068 <= 00000001
@000030f4: *0000006c <= 00000001
@000030f8: *00000070 <= 00000001
@000030fc: *00000074 <= 00000001
@00003100: *00000078 <= 00000001
@00003104: *0000007c <= 00000001
@00003108: $ 1 <= 00000001
@0000310c: *00000000 <= 00000001
@00003110: $ 2 <= 00000001
@00003114: $ 3 <= 00000001
@00003118: $ 4 <= 00000001
@0000311c: $ 5 <= 00000001
@00003120: $ 6 <= 00000001
@00003124: $ 7 <= 00000001
@00003128: $ 8 <= 00000001
@0000312c: $ 9 <= 00000001
@00003130: $10 <= 00000001
@00003134: $11 <= 00000001
@00003138: $12 <= 00000001
@0000313c: $13 <= 00000001
@00003140: $14 <= 00000001
@00003144: $15 <= 00000001
@00003148: $16 <= 00000001
@0000314c: $17 <= 00000001
@00003150: $18 <= 00000001
@00003154: $19 <= 00000001
@00003158: $20 <= 00000001
@0000315c: $21 <= 00000001
@00003160: $22 <= 00000001
@00003164: $23 <= 00000001
@00003168: $24 <= 00000001
@0000316c: $25 <= 00000001
@00003170: $26 <= 00000001
@00003174: $27 <= 00000001
@00003178: $28 <= 00000001
@0000317c: $29 <= 00000001
@00003180: $30 <= 00000001
@00003184: $31 <= 00000001
@00003188: $ 1 <= 0000038b
@0000318c: *00000000 <= 00000000
@00003190: $ 1 <= 00000000
@00003194: *00000004 <= 00000000
@00003198: $ 2 <= 00000000
@0000319c: *00000008 <= 00000000
@000031a0: $ 3 <= 00000000
@000031a4: *0000000c <= 00000000
@000031a8: $ 4 <= 00000000
@000031ac: *00000010 <= 00000000
```

```
@000031b0: $ 5 <= 00000000
@000031b4: *00000014 <= 00000000
@000031b8: $ 6 <= 00000000
@000031bc: *00000018 <= 00000000
@000031c0: $ 7 <= 00000000
@000031c4: *0000001c <= 00000000
@000031c8: $ 8 <= 00000000
@000031cc: *00000020 <= 00000000
@000031d0: $ 9 <= 00000000
@000031d4: *00000024 <= 00000000
@000031d8: $10 <= 00000000
@000031dc: *00000028 <= 00000000
@000031e0: $11 <= 00000000
@000031e4: *0000002c <= 00000000
@000031e8: $12 <= 00000000
@000031ec: *00000030 <= 00000000
@000031f0: $13 <= 00000000
@000031f4: *00000034 <= 00000000
@000031f8: $14 <= 00000000
@000031fc: *00000038 <= 00000000
@00003200: $15 <= 00000000
@00003204: *0000003c <= 00000000
@00003208: $16 <= 00000000
@0000320c: *00000040 <= 00000000
@00003210: $17 <= 00000000
@00003214: *00000044 <= 00000000
@00003218: $18 <= 00000000
@0000321c: *00000048 <= 00000000
@00003220: $19 <= 00000000
@00003224: *0000004c <= 00000000
@00003228: $20 <= 00000000
@0000322c: *00000050 <= 00000000
@00003230: $21 <= 00000000
@00003234: *00000054 <= 00000000
@00003238: $22 <= 00000000
@0000323c: *00000058 <= 00000000
@00003240: $23 <= 00000000
@00003244: *0000005c <= 00000000
@00003248: $24 <= 00000000
@0000324c: *00000060 <= 00000000
@00003250: $25 <= 00000000
@00003254: *00000064 <= 00000000
@00003258: $26 <= 00000000
@0000325c: *00000068 <= 00000000
@00003260: $27 <= 00000000
@00003264: *0000006c <= 00000000
@00003268: $28 <= 00000000
@0000326c: *00000070 <= 00000000
@00003270: $29 <= 00000000
@00003274: *00000074 <= 00000000
@00003278: $30 <= 00000000
@0000327c: *00000078 <= 00000000
@00003280: $31 <= 00000000
@00003284: *0000007c <= 00000000
@00003288: $ 1 <= 00ea0000
@0000328c: $ 2 <= 00ea0000
@00003290: $ 3 <= 00ea0000
@00003294: $ 4 <= 00ea0000
```

```
@00003298: $ 5 <= 00ea0000
@0000329c: $ 6 <= 00ea0000
@000032a0: $ 7 <= 00ea0000
@000032a4: $ 8 <= 00ea0000
@000032a8: $ 9 <= 00ea0000
@000032ac: $10 <= 00ea0000
@000032b0: $11 <= 00ea0000
@000032b4: $12 <= 00ea0000
@000032b8: $13 <= 00ea0000
@000032bc: $14 <= 00ea0000
@000032c0: $15 <= 00ea0000
@000032c4: $31 <= 000032c8
@0000330c: $16 <= 00ea0000
@00003310: $17 <= 00ea0000
@00003314: $18 <= 00ea0000
@00003318: $19 <= 00ea0000
@0000331c: $20 <= 00ea0000
@00003320: $21 <= 00ea0000
@00003324: $22 <= 00ea0000
@00003328: $23 <= 00ea0000
@0000332c: $24 <= 00ea0000
@00003330: $25 <= 00ea0000
@00003334: $26 <= 00ea0000
@00003338: $27 <= 00ea0000
@0000333c: $28 <= 00ea0000
@00003340: $29 <= 00ea0000
@00003344: $30 <= 00ea0000
@000032c8: $ 1 <= 00de0000
@000032cc: $16 <= 000c0000
@000032d0: $17 <= 000c0000
@000032d4: $18 <= 000c0000
@000032d8: $19 <= 000c0000
@000032dc: $20 <= 000c0000
@000032e0: $21 <= 000c0000
@000032e4: $22 <= 000c0000
@000032e8: $23 <= 000c0000
@000032ec: $24 <= 000c0000
@000032f0: $25 <= 000c0000
@000032f4: $26 <= 000c0000
@000032f8: $27 <= 000c0000
@000032fc: $28 <= 000c0000
@00003300: $29 <= 000c0000
@00003304: $30 <= 000c0000
@00003308: $31 <= 0000330c
```

此结果与魔改MARS比对一致，可以证明CPU设计成功。

自动化测试

此部分进行了简单的MIPS代码生成（未考虑跳转指令），这部分指令在其他样例中单独实现

需要的环境为**python3**环境，指令生成.py代码如下：

```
n=1
path=os.path.dirname(os.path.realpath(__file__))
os.chdir(path)
with open ("instr.asm","w") as f:
    for i in range(32):
        f.write("ori $%d $0 0\n"%(i))
    for i in range(1):
            for i in range(10):
                m=random.randint(0,31)
                n=random.randint(0,31)
                imm16=random.getrandbits(16)
                f.write("ori $%d $%d %d\n"%(m,n,imm16))
                f.write("lui $%d %d\n"%(m,imm16))
            for i in range(10):
                m=random.randint(0,31)
                n=random.randint(0,31)
                addr=random.randint(0,0x3ff)*4
                f.write("sw $%d %d($%d)\n"%(m,addr,n))
                f.write("lw $%d %d($%d)\n"%(m,addr,n))
            for i in range(10):
                m=random.randint(0,31)
                n=random.randint(0,31)
                k=random.randint(0,31)
                f.write("addu $%d $%d $%d\n"%(m,n,k))
                f.write("subu $%d $%d $%d\n"%(m,n,k))
f.close()
```

自动化测试的代码如下

注：本部分代码的verilog命令行运行部分参考了评论区的一位同学和历年学长的代码

```
import os
import re
import random

asmfilename=input("please input mips file name:")
xlinx="D:\\Xilinx\\14.7\\ISE_DS\\ISE"
time="10us"
os.environ['XILINX']=xlinx
path=os.path.dirname(os.path.realpath(__file__))
os.chdir(path)
filelist=os.walk(path)
with open("mips.prj","w") as prj:
    for folder in filelist:
        for file in folder[2]:
            if(len(file.split("."))>1 and file.split(".")[1]=="v"):
                prj.write("verilog work \""+folder[0]+"\\"+file+"\"\n")
with open("mips.tcl","w") as tcl:
    tcl.write("run "+time+";\nexit;\n")
os.system("java -jar Mars.jar a nc mc CompactDataAtZero dump .text HexText
code.txt "+asmfilename)
os.system("java -jar Mars.jar  nc mc CompactDataAtZero dump .text HexText
code.txt >ans.txt "+asmfilename)
os.system(xlinx+"\\bin\\nt64\\fuse "+"--nodebug  --prj mips.prj -o mips.exe
mipsAutoTest >log.txt")
os.system("mips.exe -nolog -tclbatch mips.tcl >my.txt")

process=0
with open("my.txt","r") as my:
    lines=my.readlines()
    if(len(lines)==0):
        print("fail to simulate")
        os._exit(1)
    if(lines[0][0]=='I'):
        process=1
n=0
while(1):
    if(lines[n][0]=="@"):
        break
    else:
        n=n+1
if(process):
    with open("my.txt","w") as my:
        my.writelines(lines[n:])
i=0
biao=0
with open("my.txt","r") as my:
    with open("ans.txt","r") as ans:
        while(1):
            i+=1
            l1=my.readline().strip()
            l2=ans.readline().strip()
            if((l1== "" or l1==None)and(l2=="" or l2==None)):
                break
            elif l1!=l2 and not "$ 0"in l2 and not "$ 0" in  l1:
                biao=1
                if l2=="" or l2 == None:
                    print("Wrong answer occur in line %d of code: "%(i)+"we got "+l1+" when we expected Nothing")
                else:
                    print("Wrong answer occur in line %d of code: "%(i)+"we got "+l1+" when we expected "+l2)

if biao==0:
    print("Accept on the point ")
else:
    os._exit(1)
```

三、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16位立即数以Byte为单位，而我们设计的 DM 模块是以 word 为单位的。我们通过ALU 运算出来的 MemAddress 是以Byte为单位的，所以要除以 4，也就是右移两位，才是真正的 MemAddress。因此取 ALU 输出的 32 位先右移两位，再取最低的10位，即取[11:2]才是所需要的真正的 MemAddress。所以这个 address 信号来自于ALU 的输出 32 位中的[11:2]这10位。

2.思考Verilog语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

第一种 用 always@(*) + case 语句实现操作码和控制信号的值之间的对应

本文使用的就是这种方法构建控制器，由于在控制器设计思路部分已经有完整代码，此处仅详细展示R型指令对应的代码。

```
always@(*)begin
     case(op)
        6'b000000:begin  //R instruction
            regDst=1;
            aluSrc=0;
            memRead=0;
            memWrite=0;
            branch=0;
            extOp=0;
            case(func)
              6'b100001:begin  //addu
                 aluCtrl=3'b010;
                 jump=0;
                 pcSrc=0;
                 memToReg=2'b00;
                 regWrite=1;
              end
              6'b100011:begin  //subu
                 aluCtrl=3'b011;
                 jump=0;
                 pcSrc=0;
                 memToReg=2'b00;
                 regWrite=1;
              end
              6'b001000:begin  //jr
                 aluCtrl=aluCtrl;
                 jump=0;
                 pcSrc=1;
                 memToReg=memToReg;
                 regWrite=0;
              end
              6'b001001:begin  //jalr
                 aluCtrl=aluCtrl;
                 jump=1;
                 pcSrc=1;
                 memToReg=2'b11;
                 regWrite=1;
              end
              default:begin
                 jump=0;
                 pcSrc=0;
                 memToReg=memToReg;
                 regWrite=0;
                 aluCtrl=aluCtrl;
              end
            endcase
        end
        6'b100011:begin  //lw
            ...
        end
        6'b101011:begin  //sw
            ...
        end
        6'b000100:begin  //beq
            ...
        end
        6'b001111:begin  //lui
            ...
        end
        6'b001101:begin  //ori
            ...
        end
        6'b000011:begin  //jal
            ...
        end
        default:begin
            ...
        end
     endcase
end
```

第二种方法 利用 assign 语句完成操作码和控制信号的值之间的对应

这种方法模仿了logisim单周期处理器中的与或门阵列，用assign语句对组合逻辑进行建模

先写出指令种类与opcode的对应关系，再写出各个控制信号和指令种类及funcode的综合关系

以pcSrc这一控制信号为例,应当写成如下形式：

```
//relationship between opcode and op type
assign jr= !op[5] & !op[4] & op[3] & !op[2] & !op[1] & !op[0];
assign jalr= !op[5] & !op[4] & op[3] & !op[2] & !op[1] & op[0];

//relationship between op type and control signal
pcSrc = jr | jalr;
```

其他的控制信号的原理和这个相同。

第三种方法 利用宏定义 + case语句 完成操作码和控制信号的值之间的对应

代码示例如下：

```
`define R 6'b000000
`define lw 6'b100011
`define sw 6'b101011
`define beq 6'b000100
`define lui 6'b001111
`define ori 6'b001101
`define jal 6'b000011
always@(*)begin
     case(op)
        `R:begin  //R instruction
            ...
        end
        `lw:begin  //lw
            ...
        end
        `sw:begin  //sw
            ...
        end
        `beq:begin  //beq
            ...
        end
        `lui:begin  //lui
            ...
        end
        `ori:begin  //ori
            ...
        end
        `jal:begin  //jal
            ...
        end
        default:begin
            ...
        end
     endcase
end
```

三种方法的优劣

第一种方法的 case 语句是常规的方式，代码有些冗长，未体现出对 6 位 opcode操作码与具体指令的关系，第三种方法中的宏定义就可以表新出这个关系，且也利用了 case 语句，与第一种方法的代码长度相仿，但思路和结构更加清晰

第二种方法 assign 代码量少，且是连续赋值，直接连线，本文采用模仿与或门阵列的方式，先取一些代表指令种类的变量如`assign jr= !op[5] & !op[4] & op[3] & !op[2] & !op[1] & !op[0];`再连接到输出的信号上（利用或门），代码简单易懂，直接体现了连线的关系，但不能直接清晰的看出具体输出信号是高电平还是低电平，而 1,3 两种方法则直接体现了真值表的内容，从这一点上来说，1、3两种方法思路更加清晰。

3.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?

三个需要重置的部件：PC,DM,GRF

- PC 复位到 0x00003000 这一初值，即程序重新开始运行
- DM 为内存空间，程序运行后可能向这里写入数据，不同程序执行时对内存空间写的值和位置可能是不一样的，复位时需要清零重置
- GRF 是寄存器文件，程序运行后可能向这里写入数据，不同程序执行时对寄存器文件写的值和寄对应寄存器编号可能是不一样的，复位应该清零重置

共同特点：不重置清零很可能会影响下一次程序的运行。

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，但实际上低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+ in[31]（in 表示进位）,没有则 temp32=GPR[rs]31+ GPR[rt]31+in[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。

同理，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)
```

跟前一对指令情况类似，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，然而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，时钟周期是一个常数，需要设置成足够长从而能够满足所需时间最长的指令，而大部分指令执行时时间是较短的，比如R 型指令不需要访问存储器，比 lw 指令所需时间短，类似这样的问题带来了时间效率上的消耗。
2. CPU设计中有三个加法器，一个用于 ALU，两个用于 PC 的逻辑（PC+4 和 beq 指令的跳转），而加法器是占用芯片面积较多，真正搭建时可能成本过高。
3. 采用独立的指令存储器 IM 和数据存储器 DM，在实际的CPU电路搭建过程中不太容易实现。