中山大学数据科学与计算机学院本科生实验报告

（2019 学年秋季学期）

课程名称：**计算机组成原理实验**　　任课教师：郭雪梅　　助教：丁文、汪庭葳

年级&班级	2019 级 04 班	专业(方向)	计算机科学与技术（超算方向）
学号	19335112	姓名	李钰
电话	19847352856	Email	1643589912@qq.com
开始日期	2020 年 11 月 20 日	完成日期	2020 年 11 月 27 日

一、实验题目

单时钟周期 CPU 的设计实验

二、实验目的

1. 理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现。
2. 通过对单周期 CPU 的运行状况进行观察和分析，进一步加深理解。

三、实验内容

1. 实验原理

单时钟周期 CPU

单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由于每个时钟上升沿时更新地址，因此要在上升沿到来之前完成所有运算，而这所有的运算除可以利用一个下降沿外，只能通过组合逻辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。

单周期 CPU 在每个 CLK 上升沿时更新 PC，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。

CPU 的顶层结构实现。主要器件有程序计数器 PC、程序存储器、寄存器堆、ALU、数据存储器和控制部件等。所有的控制信号简单地说明如下：

其中，控制单元(Ctrl Unit)定义如下：

（1）JUMP：为 1 时，选择跳转目标地址；为 0 时，选择由 Branch 选出的地址；

（2）MemToReg：为 1 时，选择存储器数据；为 0 时，选择 ALU 输出的数据；

（3）Branch：为 1 时，选择转移目标地址；为 0 时，选择 PC +4（图中的 NextPC）；

（4）MemWrite：为 1 时写入存储器。存储器地址由 ALU 的输出决定，写入数据为寄存器 rt 的内容；

（5）ALUOP：ALU 控制码；

（6）ALUSrc：ALU 操作数 B 的选择，为 1 时，选择扩展的立即数；为 0 时，选择寄存器数据；

（7）RegWrite：为 1 时写入寄存器堆，目的寄存器号是由 RegDst 选出的 rt 或 rd，写入数据是由 MemToReg 选出的存储器数据或 ALU 的输出结果；

（8）ExtOp：符号扩展。为 1 时，符号扩展；为 0 时，0 扩展；

（9）RegDst：目的地址，为 1 时，选择 rd；为 0 时，选择 rt。

MIPS 指令集

本次实验共涉及三种类型的 MIPS 指令，分别为 R 型、I 型和 J 型，三种类型的 MIPS 指令格式定义如下：

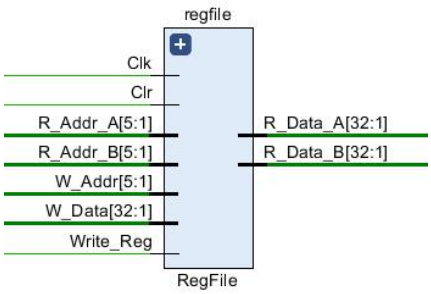
- R（register）类型的指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆；
- I（immediate）类型的指令使用一个 16 位的立即数作为一个源操作数；
- J（jump）类型的指令使用一个 26 位立即数作为跳转的目标地址（target address）；

2. 实验步骤

各模块设计

① 寄存器模块

寄存器组是指令操作的主要对象，MIPS 处理器里一共有 32 个 32 位的寄存器，本实验中只声明一个包含 15 个 32 位的寄存器数组。读寄存器时需要 Rs，Rd 的地址，得到其数据。写寄存器 Rd 时需要所写地址，所写数据，同时需要写使能。以上所有操作需要在时钟和复位信号控制下操作。故寄存器组设计如下：

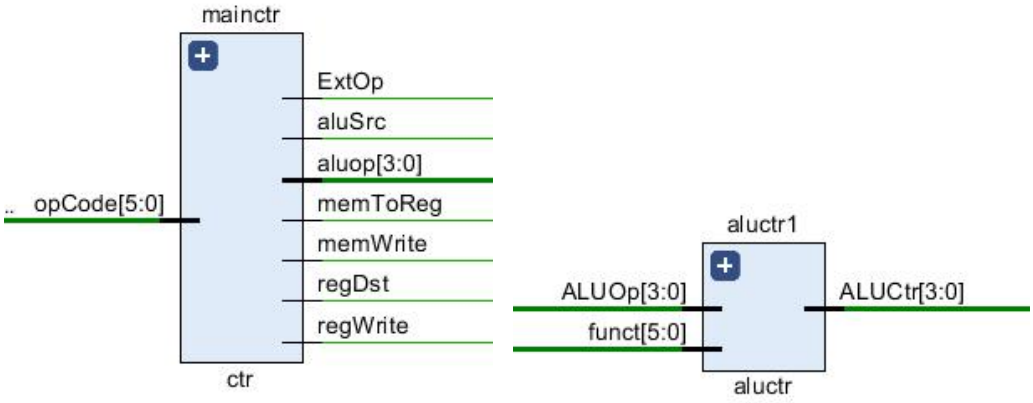


② **控制器模块**

根据指令中的指令码（op）和功能码（funct）的不同组合输出相应的控制信号

aluop[3:0]	运算	ALUCtr[3:0]
XXXX	J型指令	x
0000	and	0000
0000	or	0001
0010	ori	0001
0011	Lw、sw、addi	0010
0000	add	0010
0110	bne	0011
1000	xori	0100
0000	nor	0101
0000	sub	0110
0101	beq	0110
0000	slt	0111
0000	sll	1000
0000	srl	1001
0000	sra	1010
0000	srlv	1011
0111	Lui	1101
0000	srav	1110

控制器设计如下：



③ ALU控制译码模块

ALU 主要执行5 种操作：与，或，加，减，小于设置。这五种操作可以使用四位的编码表示：0000，0001，0010，0110，0111。指令不同，则对应的ALU 运算不同，所以该模块需要根据指令来控制 ALU 进行正确的运算。

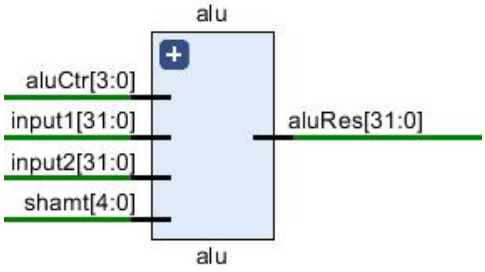
lw，sw，addi 指令均要求 ALU 执行加操作，则可分为一类，aluop编码 0011；

beq 指令要求ALU 执行减操作，则分为一类，编码0101；

最后一类是 R 型指令，可以编码为 0000；但不同的R 型指令对应不同的 ALU 运算，故需要再通过指令的功能码进一步确定 ALU 的运算。

最终该模块即实现4 位操作码以及6 位功能码输出4 位ALU 控制信号码。

④ ALU运算器模块



input1：操作数 32 位，输入；
input2：操作数，32 位，输入；
ALUCtr：4 位操作码，输入；
aluRes：运算结果，32 位，输出；
zero：零标志，1位；当运算结果为0时，该位为1，否则为0；

⑤ 符号扩展模块

根据符号补充符号位

如果符号位为 1，则补充 16 个 1，即 16'h ffff

如果符号位为 0，则补充 16 个 0

⑥ 指令存储器模块

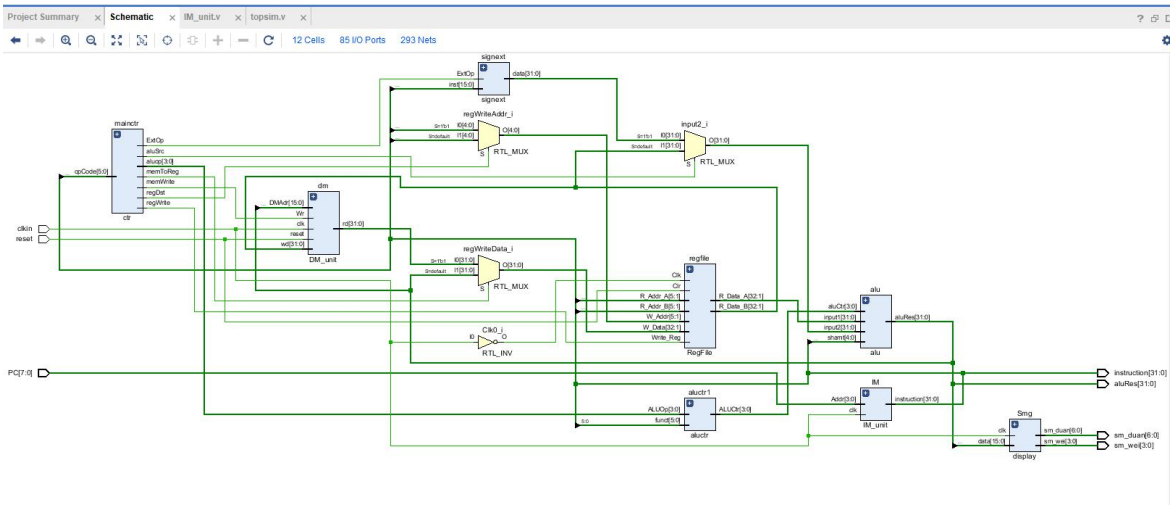
该模块有4位地址输入和32位数据输出，首先将16条指令写入存储单元中，然后根据4位地址输入选择相应的单元指令内容，将数据写入到输出变量中。

⑦ 数据存储器

⑧ 译码管显示模块

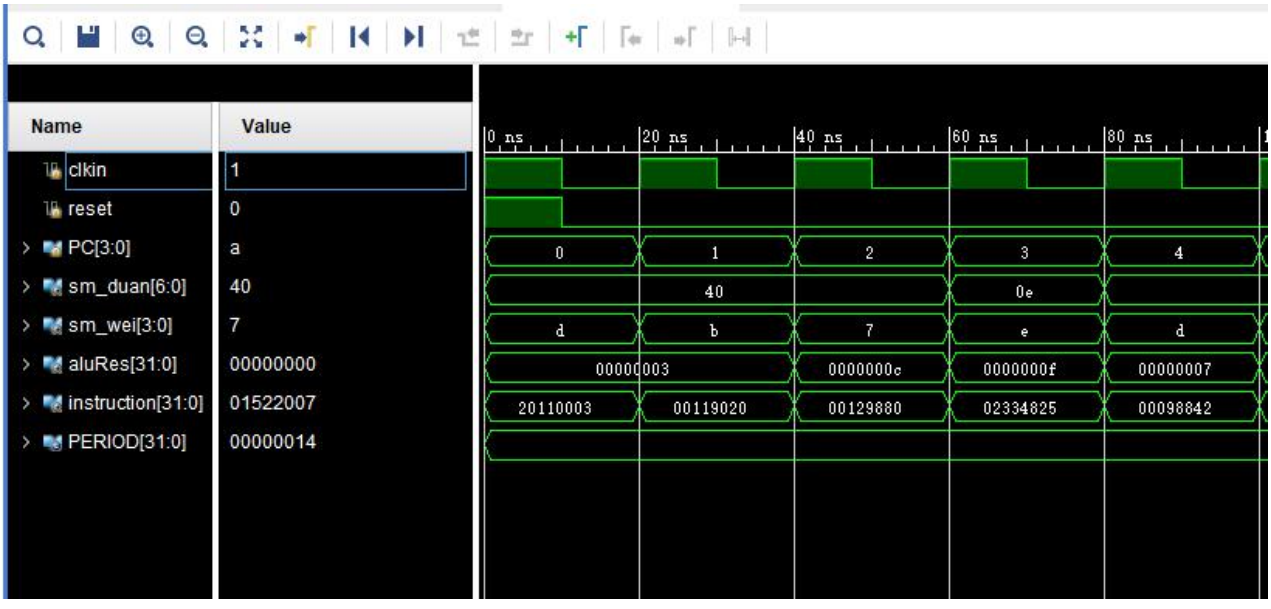
⑨ PC模块 top文件，组装各个模块

最终结构原理图



四、实验结果

仿真图波形分析：



指令 1： 20110003 -> addi $s1, $0, 3

运算： s1 = 0 + 3 = 3

aluRes = 00000003

指令 2：00119020 -> add $s2, $0, $s1

运算: s2 = 0 + 3 = 3

aluRes = 00000003

指令 3: 00129880 -> sll $s3, $s2, 2

运算: s3 = s2 << 2

aluRes = 0000_0000_0000_0000_0000_0000_0000_1100 = 0000000c

指令 4: 02334825 -> or $t1, $s1, $s3

运算: t1 = 0000_0000_0000_0000_0000_0000_0000_0011 |

0000_0000_0000_0000_0000_0000_0000_1100

= 0000_0000_0000_0000_0000_0000_0000_1111

=0000000f

aluRes = 0000000f

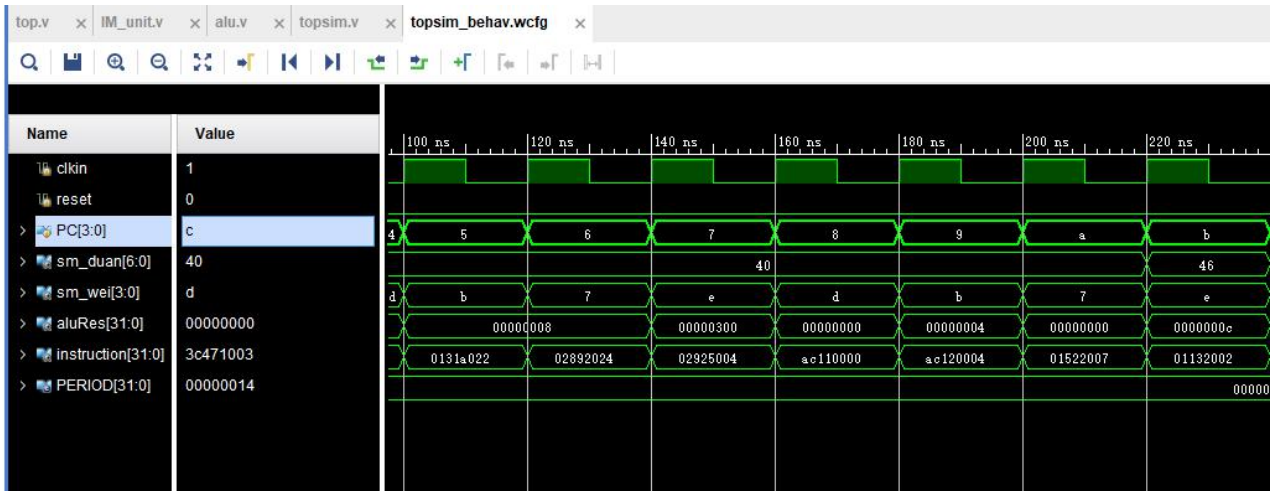
指令 5: 00098842 -> srl $s1, $t1, 1

运算: s1 = t1 >> 1;

aluRes = 0000_0000_0000_0000_0000_0000_0000_1111 >> 1

= 0000_0000_0000_0000_0000_0000_0000_0111

=00000007;



指令 6:　0131a022-> sub $s4, $t1, $s1

运算: s4 = t1 - s1 = 00000000f - 00000007 = 00000008

aluRes = 00000008

指令 7: 02892024 -> and $4, $s4, $t1

运算: $4 = s4 & t1 = 00000008 & 0000000f

= 0000_0000_0000_0000_0000_0000_0000_1000 &

= 0000_0000_0000_0000_0000_0000_0000_1100

=0000_0000_0000_0000_0000_0000_0000_1000

= 00000008

aluRes = 00000008

指令 8: 02925004 -> sllv $t2, $s4, $s2

运算: t2 = s4 << s2

= 00000003 << 8

= 0000_0000_0000_0000_0000_0000_0000_0110 <<8

= 0000_0000_0000_0000_0000_0011_0000_0000

=00000300

aluRes = 00000300

指令 9: ac110000 -> lw $0, 0($s1)

aluRes = 00000000

指令 10: ac120004 -> sw $s2, 4($0)

aluRes = 00000004

指令 11: 01522007 -> srav $4, $t2, $s2

运算: $4 = s2 >> t2

= 00000003 >> 00000300

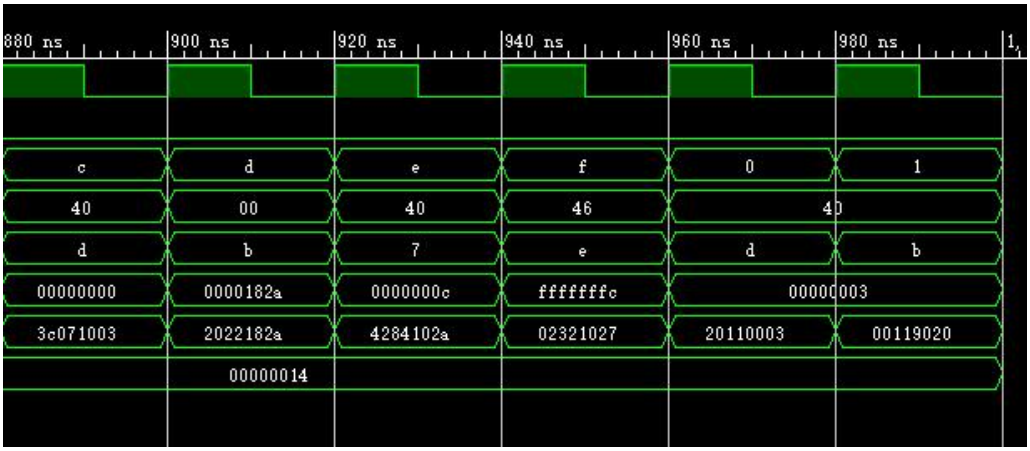
= 00000000

aluRes = 00000000

指令 12: 11320002 -> beq $s3, $4, 2

运算: $s3 - $4 = 0000000c - 00000000 =0000000c

aluRes = 0000000c



指令 13: 3c071003 -> lui

aluRes = 00000000

指令 14: 2c22182a -> addi $1, $2, 182a

运算: $1 = $2 + 182a = 182a

aluRes = 0000182a

指令 15: 4284102a -> slt $2, $s4, $4

运算　s4 - $4 = 0000000c -00000000 = 0000000c > 0

aluRes = 0000000c

指令 16: 02321027 -> nor $2, $s1, $s2

! S2 = ~(00000007 | 00000003)

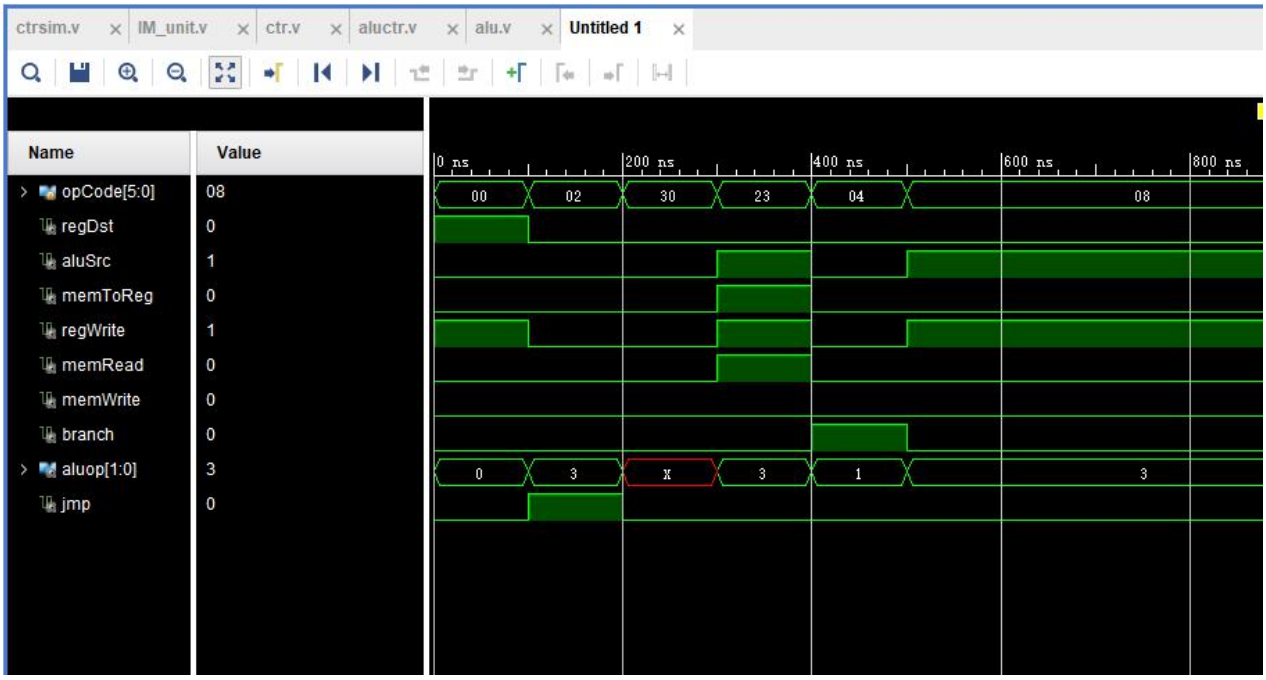
=~ (0000_0000_0000_0000_0000_0000_0000_0011 |

0000_0000_0000_0000_0000_0000_0000_0011)

= 1111_1111_1111_1111_1111_1111_1111_1100 =fffffffc

aluRes = fffffffc

控制器仿真



分析:

仿真部分代码

```
initial begin
// Initialize Inputs
opCode = 6'b000000;
#100;
opCode = 6'b000010;
// Wait 100 ns for global reset to finish
#100;
opCode = 6'b110000;
#100;
opCode = 6'b100011;
#100;
opCode = 6'b000100;
#100;
opCode = 6'b001000;
end
```

第一条 测试opCode = 000000, 对应ctr中的 R型指令设置, 相应结果正确

```
6'b000000: begin
regDst = 1;  aluSrc = 0; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000; jmp = 0; ExtOp = 1;
end // "R 型" 指令操作码: 000000
```

第三条 测试 opCode = 110000, 因为 ctr中未定义此种op,故aluop显示X ,无对应项

其他测试,对应于ctr文件中,输出波形与结果均正确

五、实验感想

通过这次实验,我对 CPU 内部结构之间的关系更加明了,对指令的构成形式运算过程更加熟悉，巩固了单周期 CPU 的相关知识，发现了一些之前没有掌握的很好的知识,比如 aluop func 和 aluctr 之间的对应关系，看书之后理清了，收获还是很多的.

附录（流程图，注释过的代码）:

① 寄存器模块

```
`timescale 1ns / 1ps//寄存器堆模块
module RegFile(
Clk,Clr,Write_Reg,R_Addr_A,R_Addr_B,W_Addr,W_Data,R_Data_A,R_Data_B
);
    parameter ADDR = 5;//寄存器编码/地址位宽
    parameter NUMB = 1<<ADDR;//寄存器个数
    parameter SIZE = 32;//寄存器数据位宽
    input Clk;//写入时钟信号
    input Clr;//清零信号
    input Write_Reg;//写控制信号
    input [ADDR:1]R_Addr_A;//A端口读寄存器地址
    input [ADDR:1]R_Addr_B;//B端口读寄存器地址
    input [ADDR:1]W_Addr;//写寄存器地址
    input [SIZE:1]W_Data;//写入数据
    output [SIZE:1]R_Data_A;//A端口读出数据
    output [SIZE:1]R_Data_B;//B端口读出数据
    reg [SIZE:1]REG_Files[0:NUMB-1];//寄存器堆本体
    integer i;//用于遍历NUMB个寄存器
        initial//初始化NUMB个寄存器，全为0
        for(i=0;i<NUMB;i=i+1)
REG_Files[i]<=0;
    always@(posedge Clk or posedge Clr)//时钟信号或清零信号上升沿
    begin
        if(Clr)//清零
                for(i=0;i<NUMB;i=i+1)
REG_Files[i]<=0;
        else//不清零,检测写控制，高电平则写入寄存器
                if(Write_Reg)
REG_Files[W_Addr]<=W_Data;
   end        //读操作没有使能或时钟信号控制，使用数据流建模(组合逻辑电路,读不需要时钟控制)
    assign R_Data_A=REG_Files[R_Addr_A];
    assign R_Data_B=REG_Files[R_Addr_B];
endmodule
```

② 控制器模块

```
`timescale 1ns / 1ps
module ctr(
input [5:0] opCode, output reg regDst, output reg aluSrc, output reg memToReg, output reg regWrite, output reg memRes
output reg ExtOp, //符号扩展方式，1 为 sign-extend，0 为 zero-extend
output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
output reg jmp
);

always@(opCode) begin
// 操作码改变时改变控制信号
case(opCode)
6'b000010: begin
regDst = 0;  aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111; jmp = 1; ExtOp = 1;
end //'J 型' 指令操作码：000010，无需 ALU

6'b000000: begin
regDst = 1;  aluSrc = 0; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000; jmp = 0; ExtOp = 1;
end //'R 型' 指令操作码：000000

//'I'型指令操作码
6'b100011: begin
regDst = 0;  aluSrc = 1; memToReg = 1;
regWrite = 1; memRead = 1; memWrite = 0; branch = 0; aluop = 4'b0011; jmp = 0; ExtOp = 1;
end //'lw' 指令操作码：100011

6'b101011: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 1; branch = 0; aluop = 4'b0011; jmp = 0; ExtOp = 1;
end //'sw' 指令操作码：101011

6'b000100: begin
regDst = 0;  aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0101; jmp = 0; ExtOp = 1;
end //'beq' 指令操作码：000100
```

```
6'b000101: begin
regDst = 0;  aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0110; jmp = 0; ExtOp = 1;
end //'bne' 指令操作码: 000101
6'b001000: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0;
branch = 0; aluop = 4'b0011;jmp = 0;
ExtOp = 1;
end //'addi' 指令操作码: 001000
6'b001100: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0001; jmp = 0; ExtOp = 0;
end //'andi' 指令操作码: 001100
6'b001101: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0010; jmp = 0; ExtOp = 0;
end //'ori' 指令操作码: 001101
6'b001110: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1000; jmp = 0; ExtOp = 0;
end //'xori' 指令操作码: 001110
6'b001000: // addi指令
begin
regDst = 0; aluSrc = 1; memToReg = 0; regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0011; jmp = 0;ExtOp = 1;
end //'addi' 指令操作码: 001000
6'b001010: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0100; jmp = 0; ExtOp = 1;
end //'slti' 指令操作码: 001010
6'b001111: begin
regDst = 0;  aluSrc = 1; memToReg = 0;
regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111; jmp = 0; ExtOp = 0;
end //'lui' 指令操作码: 001111
default: begin
regDst = 0;  aluSrc = 0; memToReg = 0;
regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 3'b0xxx; jmp = 0; ExtOp = 0;
end // 默认设置
endcase end
```

③ ALU控制译码模块

```
`timescale 1ns / 1ps
module aluctr(
input [3:0] ALUOp, input [5:0] funct, output reg [3:0]  ALUCtr
);

always @(ALUOp or funct) //  如果操作码或者功能码变化执行操作
casex({ALUOp, funct}) // 拼接操作码和功能码便于下一步的判断
10'b0000_100100: ALUCtr = 4'b0000; // and

10'b0000_100101: ALUCtr = 4'b0001; // or
10'b0010_xxxxxx: ALUCtr = 4'b0001; // ori

10'b0011_xxxxxx: ALUCtr = 4'b0010; // lw, sw, addi
10'b0000_100000: ALUCtr = 4'b0010; // add

10'b0110_xxxxxx: ALUCtr = 4'b0011; // bne
10'b1000_xxxxxx: ALUCtr = 4'b0100; // xori
10'b0000_100111: ALUCtr = 4'b0101; // nor

10'b0000_100010: ALUCtr = 4'b0110; // sub
10'b0101_xxxxxx: ALUCtr = 4'b0110; //beq

10'b0000_101010: ALUCtr = 4'b0111; // slt  小于设置
10'b0000_000000: ALUCtr = 4'b1000; // sll  左移
10'b0000_000010: ALUCtr = 4'b1001; // srl  逻辑右移
10'b0000_000011: ALUCtr = 4'b1010; // sra  算术右移
10'b0000_000100: ALUCtr = 4'b1011;// sllv  左移
10'b0000_000110: ALUCtr = 4'b1100; // srlv  逻辑右移
10'b0111_001111: ALUCtr = 4'b1101; //lui   将16位立即数放到目的寄存器高16位，目的寄存器的低16位填0
10'b0000_000111: ALUCtr = 4'b1110;//srav   算术右移
default:ALUCtr = 4'b1111;
endcase
endmodule
```

④ ALU运算器模块

```
`timescale 1ns / 1ps
module alu(
input [4:0] shamt,
input [31:0] input1,
input [31:0] input2,
input [3:0] aluCtr,
output reg[31:0] aluRes,
output reg ZF,
output reg CF,OF
);
always @(input1 or input2 or aluCtr) // 运算数或控制码变化时操作
begin
case(aluCtr) 4'b0000: //  与 and
begin
aluRes = input1 & input2;
if(aluRes==0) ZF=1;
else ZF=0;
end

4'b0001: // 或 or
begin
aluRes = input1 | input2;
if(aluRes==0) ZF=1;
else ZF=0;
end

4'b0001://ori
begin
aluRes=input1|input2;
if(aluRes==0) ZF=1;
else ZF=0;
end
```

```
4'b0001://ori
begin
aluRes=input1|input2;
if(aluRes==0) ZF=1;
else ZF=0;
end

4'b0010: // 加addi add lw sw
begin
{CF,aluRes} = input1 + input2;
if(aluRes==0) ZF=1; else ZF=0;
OF = input1[31]^input2[31]^aluRes[31]^CF;//溢出标志公式
end

 4'b0011://bne
begin
aluRes=input1-input2;
if(aluRes==0) ZF=0;//这里的zero是指不为0，不相等
 else ZF=1;
end


4'b0100://xori
begin
aluRes=(~input1&input2)|(input1&~input2);
if(aluRes==0) ZF=1;
else ZF=0;
end

 4'b0010://andi
 begin
  aluRes=input1&input2;
 if(aluRes==0) ZF=1;
 else ZF=0;
 end



 4'b0101: // 或非nor
 begin
  aluRes = ~(input1 | input2);
 if(aluRes==0) ZF=1;
 else ZF=0;
 end

 4'b0110: // 减sub
 begin
  {CF,aluRes} = input1 - input2;
 if(aluRes == 0)ZF = 1;
 else ZF = 0;
  OF = input1[31]^input2[31]^aluRes[31]^CF;//溢出标志公式
 end

 4'b0110: // beq
 begin
  aluRes = input1 - input2;
 if(aluRes == 0) ZF = 1;
 else ZF = 0;
 end
```

```
4'b0111: // 小于设置slt
begin
if(input1<input2) aluRes = 1;
else aluRes=0; if(aluRes==0) ZF=1; else ZF=0;
end


4'b1000://sll
begin
  aluRes=input2<<shamt;
  if(aluRes==0) ZF=1; else ZF=0;
end

4'b1001://srl
begin
  aluRes=input2>>shamt;
if(aluRes==0) ZF=1;
else ZF=0;
end

4'b1010://sra
begin
  aluRes = input2 >>> shamt;
if(aluRes == 0) ZF = 1;
else ZF = 0;
end
```

```
4'b1011://sllv
begin
 aluRes = input2 << input1;
if(aluRes == 0)
 ZF = 1;
else ZF = 0;
end

4'b1100://srlv
begin
 aluRes = input2 >> input1;
if(aluRes == 0)
 ZF = 1;
else ZF = 0;
end

4'b1101://lui
begin
 aluRes={input2[15:0],16'b0000_0000_0000_0000};
end

4'b1110://srav
begin
 aluRes = input2 >>> input1;
if(aluRes == 0)
 ZF = 1;
else ZF = 0;
end

default:
begin
 aluRes = 0; ZF=0;OF=0;
 end
endcase
 end
endmodule
```

⑤ 符号扩展模块

```
`timescale 1ns / 1ps
module signext(
input [15:0] inst, // 输入16位
input  ExtOp,
output [31:0] data // 输出32位
);
// 根据符号补充符号位
 //  如果符号位为1，则补充16个1，即16'h ffff
// 如果符号位为0，则补充16个0
 assign data= inst[15:15]&ExtOp?{16'hffff,inst}:{16'h0000,inst};

endmodule
```

⑥ 指令存储器模块

```
`timescale 1ns / 1ps
module IM_unit( input clk,
input [3:0] Addr,       //指令存储器地址编码
output reg [31:0]  instruction// 寄存器的值
);
//寄存器地址都是 4 位二进制数，因为寄存器只有 16 个，4 位就能表示所有寄存器
reg [31:0] regs [0:15]; // 寄存器组

initial
       begin

                        regs[0]  = 32'h20110003;// addi $s1,$0,3
                        regs[1]  = 32'h00119020;// add $s2,$0,$s1
                        regs[2]  = 32'h00129880;// sll $s3, $s2, 2
                        regs[3]  = 32'h02334825;// or $t1,$s1,$s3

                        regs[4]  = 32'h00098842;// srl $s1,$t1,1
                        regs[5]  = 32'h0131a022;// sub $s4,$t1,$s1,
                        regs[6]  = 32'h02892024;// and $4,$s4,$t1
                        regs[7]  = 32'h02925004;//sllv $t2,$s4,$s2
                        regs[8]  = 32'h8c110000;// lw
                        regs[9]  = 32'hac120004;// sw
                        regs[10]  = 32'h01522007;//srav $4,$t2,$s2
                        regs[11]  = 32'h1132002;//beq $s3,$4,2
                        regs[12]  = 32'h3c071003;// lui
                        regs[13]  = 32'b0010_0000_0010_0010_0001_1000_0010_1010;// addi $1, $2, 182a
                        regs[14]  = 32'h4284102a;//slt $2,$s4,$4
                        regs[15]  = 32'h02321027;//nor $2,$s1,$s2

end
always @( posedge clk ) // 时钟上升沿操作

instruction=regs[Addr] ; //  取指令
endmodule
```

⑦ 数据存储器

```
module DM_unit(input clk, Wr,
               input reset,
              input [15:0] DMAdr,
               input [31:0] wd,
               output [31:0] rd);
              reg [31:0] RAM[15:0];
//read
assign rd=RAM[DMAdr];
//write
integer i;
always @ (posedge clk,posedge reset)
begin
    if(reset)begin
        for(i = 0; i < 256; i = i + 1)
            RAM[i]=0;
    end
    else if (Wr) begin
      RAM[DMAdr] =wd;
        end
    end
 endmodule
```

⑧ 译码管显示模块

```
`timescale 1ns / 1ps
module display(clk,data,sm_wei,sm_duan);
input clk;
input [15:0] data;
output [3:0] sm_wei;
output [6:0] sm_duan;
//分频
integer clk_cnt;
reg clk_400Hz;
always @(posedge clk)
 if(clk_cnt==32'd1)
begin clk_cnt <= 1'b0; clk_400Hz <= ~clk_400Hz;
end else clk_cnt <= clk_cnt + 1'b1;
//位控制
reg [3:0]wei_ctrl=4'b1110;
always @(posedge clk)
wei_ctrl <= {wei_ctrl[2:0],wei_ctrl[3]}; //位控制
reg [3:0]duan_ctrl;
always @(wei_ctrl or data)
case(wei_ctrl)
4'b1110:duan_ctrl=data[3:0];
4'b1101:duan_ctrl=data[7:4];
4'b1011:duan_ctrl=data[11:8];
4'b0111:duan_ctrl=data[15:12];
default:duan_ctrl=4'hf;
 endcase
//解码模块
reg [6:0]duan;
```

```
always @(duan_ctrl)
case(duan_ctrl)
4'h0:duan=7'b100_0000;//0
4'h1:duan=7'b111_1001;//1
4'h2:duan=7'b010_0100;//2
4'h3:duan=7'b011_0000;//3
4'h4:duan=7'b001_1001;//4
4'h5:duan=7'b001_0010;//5
4'h6:duan=7'b000_0010;//6
4'h7:duan=7'b111_1000;//7
4'h8:duan=7'b000_0000;//8
4'h9:duan=7'b001_0000;//9
4'ha:duan=7'b000_1000;//a
4'hb:duan=7'b000_0011;//b
4'hc:duan=7'b100_0110;//c
4'hd:duan=7'b010_0001;//d
4'he:duan=7'b000_0111;//e
4'hf:duan=7'b000_1110;//f
// 4'hf:duan=7'b111_1111;//不显示
default : duan = 7'b100_0000;//0
endcase

//-------------------------------------------------------
assign sm_wei = wei_ctrl;
assign sm_duan = duan;
endmodule
```

⑨ PC模块 top文件，组装各个模块

```
`timescale 1ns / 1ps
module top(
input clkin, input reset,
input [7:0]PC,
output [6:0] sm_duan,//段码
output [3:0] sm_wei,//哪个数码管
output [31:0]aluRes,
output [31:0]instruction
);
// 复用器信号线
//wire[31:0] expand2, mux4, mux5, address, jmpaddr;
//数据存储器
wire[31:0] memreaddata;
// 指令存储器
//wire [31:0] instruction;
reg[7:0] Addr;
// CPU 控制信号线
wire reg_dst,jmp,branch, memread, memwrite, memtoreg,alu_src,ExtOp;
wire[3:0] aluop;
wire regwrite;
// ALU 控制信号线
wire ZF,OF,CF; //alu运算为零标志
//wire[31:0] aluRes; //alu运算结果
// ALU控制信号线
wire[3:0] aluCtr;//根据aluop和指令后6位 选择alu运算类型
//wire[31:0] aluRes; //alu运算结果
wire[31:0] input2;
wire [15:0]data;
// 寄存器信号线
wire[31:0] RsData, RtData;
wire[31:0] expand; wire[4:0] shamt;
wire [4:0]regWriteAddr;
wire[31:0]regWriteData;
```

```
assign shamt=instruction[10:6];
assign regWriteAddr = reg_dst ? instruction[15:11] : instruction[20:16];//写寄存器的目标寄存器来自rt或rd
assign data=aluRes[15:0];
assign regWriteData = memtoreg ? memreaddata : aluRes; //写入寄存器的数据来自ALU或数据寄存器
assign input2 = alu_src ? expand : RtData; //ALU的第二个操作数来自寄存器堆输出或指令低16位的符号扩展
// 例化指令存储器
IM_unit IM ( .clk(clkin), .Addr(PC), .instruction(instruction) );

// 实例化控制器模块
ctr mainctr(
.opCode(instruction[31:26]),
.regDst(reg_dst),
.aluSrc(alu_src),
.memToReg(memtoreg),
.regWrite(regwrite),
.memRead(memread),
.memWrite(memwrite),
.branch(branch),
.ExtOp(ExtOp),
.aluop(aluop),
.jmp(jmp));
//  实例化 ALU 控制模块
aluctr aluctr1(
.ALUOp(aluop),
.funct(instruction[5:0]),
.ALUCtr(aluCtr));
```

```
// . . . . . . . . . . . . . . . . . . . . . . . . . . . . 实例化寄存器模块
RegFile regfile(
.R_Addr_A(instruction[25:21]),
.R_Addr_B(instruction[20:16]),
.Clk(!clkin),
.Clr(reset),
.W_Addr(regWriteAddr),
.W_Data(regWriteData),
.Write_Reg(regwrite),
.R_Data_A(RsData),
.R_Data_B(RtData)
);

// . . . . . . . . . . . . . . . . . . . . . . . . 实例化ALU模块
alu alu(
.shamt(shamt),
.input1(RsData), //写入alu的第一个操作数必是Rs
.input2(input2),
.aluCtr(aluCtr),
.ZF(ZF),
.OF(OF),
.CF(CF),
.aluRes(aluRes));
//实例化符号扩展模块
signext signext(.inst(instruction[15:0]),.ExtOp(ExtOp), .data(expand));
//实例化数据存储器
 DM_unit dm(.clk(clkin), .Wr(memwrite),
            .reset(reset),
             .DMAdr(aluRes),
             .wd(RtData),
             .rd( memreaddata));
//.............................实例化数码管显示模块
display Smg(.clk(clkin),.sm_wei(sm_wei),.data(data),.sm_duan(sm_duan));

endmodule
```