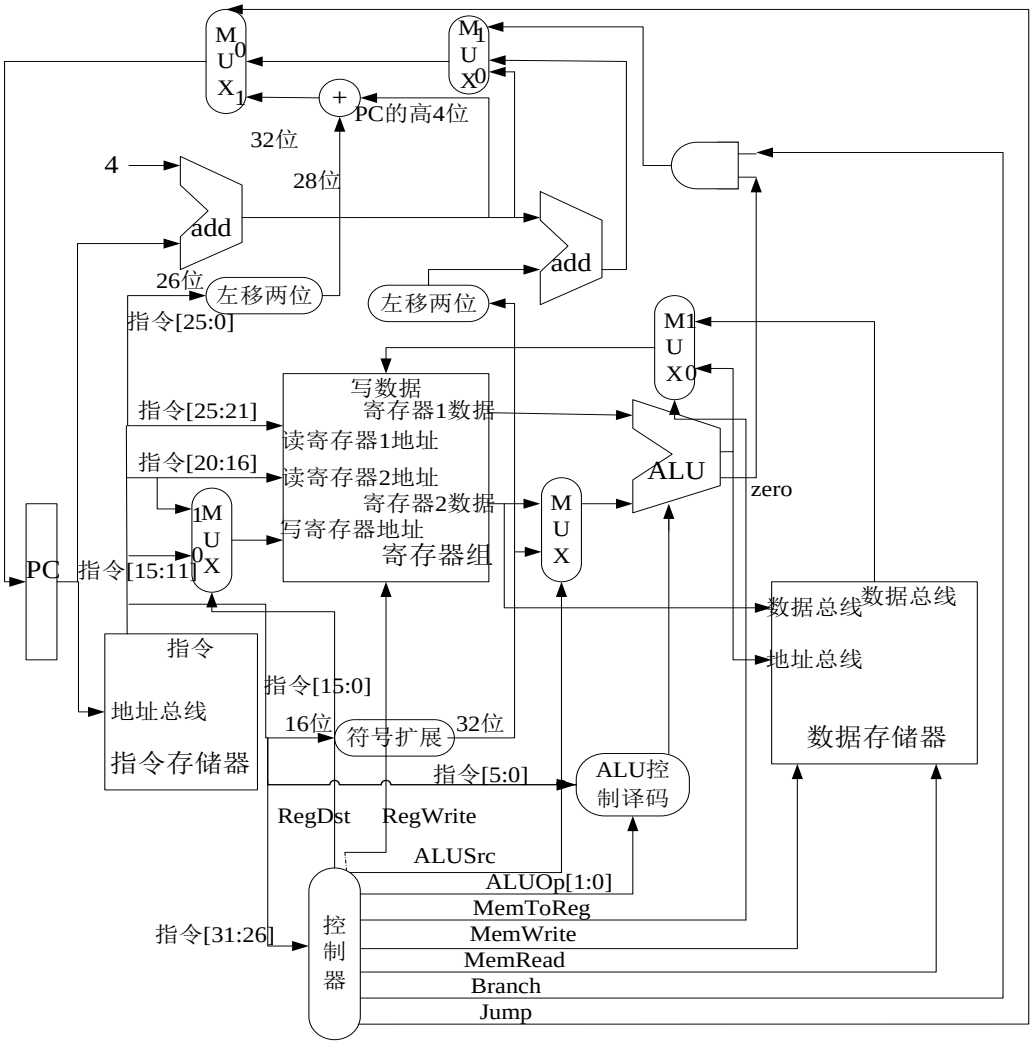
一、 实验目的

1. 了解微处理器的基本结构
2. 掌握哈佛结构的计算机工作原理
3. 学会设计简单的微处理器
4. 了解软件控制硬件工作的基本原理

二、 实验任务

- 利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，设计一个能够执行以下 MIPS 指令集的单周期类 MIPS 处理器，要求完成所有支持指令的功能仿真，验证指令执行的正确性，要求编写汇编程序将本人学号的 ASCII 码存入 RAM 的连续内存区域
  - 支持基本的内存操作如 lw，sw 指令
  - 支持基本的算术逻辑运算如 add，sub，and，or，slt，addi 指令
  - 支持基本的程序控制如 beq，j 指令

三、 基本原理

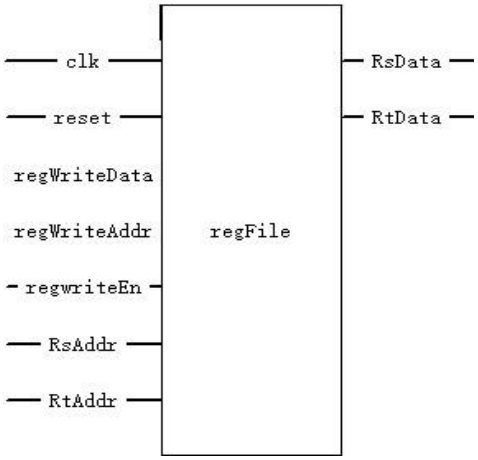

M
U
X
0
1
M1
U
X0
+
PC的高4位
32位
4
28位
add
add
26位
左移两位
左移两位
指令[25:0]
M1
U
X0
写数据
寄存器1数据
指令[25:21]
读寄存器1地址
指令[20:16]
读寄存器2地址
寄存器2数据
写寄存器地址
寄存器组
ALU
zero
M
U
X
1M
U
0X
PC
指令[15:11]
数据总线
数据总线
地址总线
指令
指令[15:0]
地址总线
指令存储器
16位
符号扩展
32位
数据存储器
指令[5:0]
ALU控
制译码
RegDst
RegWrite
ALUSrc
ALUOp[1:0]
MemToReg
MemWrite
MemRead
Branch
Jump
指令[31:26]
控
制
器

四、 各个硬件模块设计

1. 寄存器组

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，故可以声明一个包含 32 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：

input	clk
input	reset
input	RsAddr[4:0]
input	RtAddr[4:0]
input	regWriteData[31:0]
input	regWriteAddr[4:0]
input	regWriteEn
output	RsData[31:0]
output	RtData[31:0]



Verilog 代码如下，细节见注释：

```
module regFile(
    input clk,
    input reset,
    input [31:0] regWriteData,
    input [4:0] regWriteAddr,
    input regWriteEn,
    output [31:0] RsData,
    output [31:0] RtData,
    input [4:0] RsAddr,
    input [4:0] RtAddr
    );

     reg[31:0]regs[0:31]; //  寄存器组
     //  根据地址读出 Rs、Rt 寄存器数据
     assign RsData = (RsAddr == 5'b0 ) ? 32'b0 : regs[RsAddr];
     assign RtData = (RtAddr == 5'b0 ) ? 32'b0 : regs[RtAddr];
     integer i;
     always @( posedge clk ) //  时钟上升沿操作
```

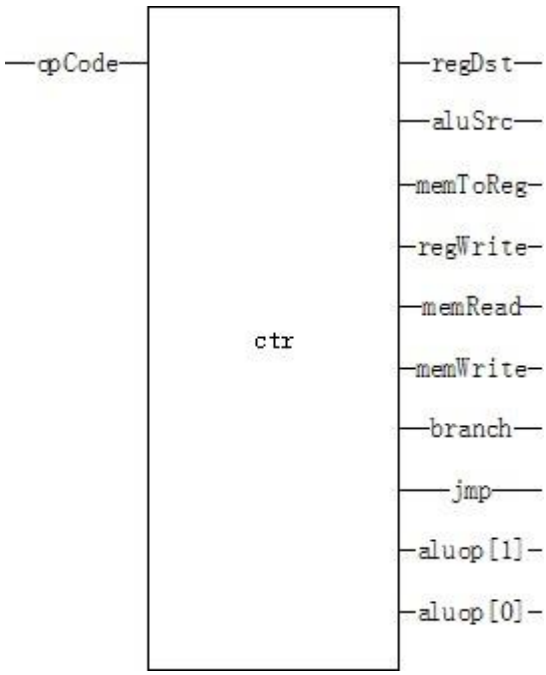
```
        begin
          if(!reset)
            begin
                if(regWriteEn == 1) //  写使能信号为 1 时写操作
                begin
                regs[regWriteAddr] = regWriteData; //  写入数据
                end
            end
            else
            begin
                for(i = 0; i < 32; i = i + 1)
                    regs[i] = 0; //  所有寄存器赋值为 0，复位
            end
        end

endmodule
```

2. 控制器模块：

控制器模块输入为指令的操作码 opCode 段，输出各个复用器、存储器读写等的信号，控制数据通路的正常进行。根据指令的不同，输出不同的信号即可。

控制信号名称	置1	清0
RegDst	表示写寄存器的地址来自指令[15:11]	来自指令[20:16]
Jump	表示PC的值来自伪直接寻址	来自另一个复用器
Branch	表示下一级复用器的输入来PC相对寻址加法器	来自PC+4
MemToReg	表示写寄存器数据来自存储器数据总线	来自ALU结果
ALUSrc	表示ALU的第二个数据源来自指令[15:0]	来自读寄存器2
RegWrite	将写寄存器数据存入写寄存器地址中	无操作
MemWrite	将写数据总线上的数据写入内存地址单元	无操作
MemRead	将内存单元的内容输出到读数据总线上	无操作

input	opcode[5:0]
output	regDst
output	aluSrc
output	memToReg
output	regWrite
output	memRead
output	memWrite
output	branch
output	aluop[1:0]
output	jmp



Verilog 代码如下，细节见注释：

```
module ctr(
    input [5:0] opCode,
    output reg regDst,
    output reg aluSrc,
    output reg memToReg,
    output reg regWrite,
    output reg memRead,
    output reg memWrite,
    output reg branch,
    output reg[1:0] aluop,
    output reg jmp
    );


    always @(opCode) //   当 opCode 变化时输出不同信号
    begin
        case(opCode)
            6'b000010: // J 型指令
            begin
                regDst = 0;
                aluSrc = 0;
                memToReg = 0;
                regWrite = 0;
                memRead = 0;
```

```
            memWrite = 0;
            branch = 0;
            aluop = 2'b00;
            jmp = 1;
        end

        6'b000000: // R 型指令
        begin
            regDst = 1;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            branch = 0;
            aluop = 2'b10;
            jmp = 0;
        end

        6'b100011: // lw
        begin
            regDst = 0;
            aluSrc = 1;
            memToReg = 1;
            regWrite = 1;
            memRead = 1;
            memWrite = 0;
            branch = 0;
            aluop = 2'b00;
            jmp = 0;
        end

        6'b101011: // sw
        begin
            regDst = 0;
            aluSrc = 1;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 1;
            branch = 0;
            aluop = 2'b00;
            jmp = 0;
        end
```

```
            6'b000100: beq 指令
            begin
                regDst = 0;
                aluSrc = 0;
                memToReg = 0;
                regWrite = 0;
                memRead = 0;
                memWrite = 0;
                branch = 1;
                aluop = 2'b01;
                jmp = 0;
            end

            6'b001000: // addi 指令
            begin
                regDst = 0;
                aluSrc = 1;
                memToReg = 0;
                regWrite = 1;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluop = 2'b00;
                jmp = 0;
            end

            default: //  缺省值
            begin
                regDst = 0;
                aluSrc = 0;
                memToReg = 0;
                regWrite = 0;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluop = 2'b00;
                jmp = 0;
            end
        endcase
    end
endmodule
```

3. ALU 控制译码

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：0000，0001，0010，01110，0111。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。

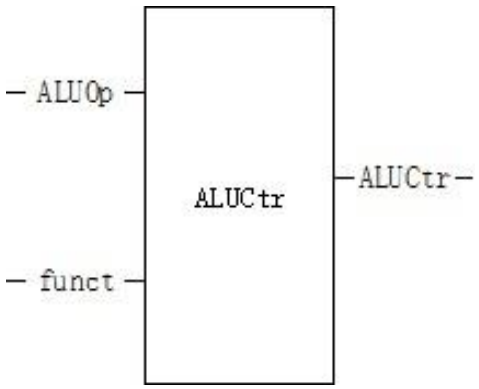
由于 lw，sw，addi 指令均要求 ALU 执行加操作，则可分为一类，编码 00；

由于 beq 指令要求 ALU 执行减操作，则分为一类，编码 01；

最后一类是 R 型指令，可以编码为 10；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 2 位操作码以及 6 位功能码输出 4 位 ALU 控制信号码。

input	ALUOp[1:0]
input	funct[5:0]
output	ALUCtr[3:0]



指令与 ALU 操作码及功能码的对应关系如下：

指令	2位操作码	指令功能	6位功能码	ALU的运算	ALU的控制信号
LW	00	取字	XXXXXX	加	0010
SW	00	存字	XXXXXX	加	0010
BEQ	01	相等跳转	XXXXXX	减	0110
R型指令	10	加	100000	加	0010
R型指令	10	减	100010	减	0110
R型指令	10	与	100100	与	0000
R型指令	10	或	100101	或	0001
R型指令	10	小于设置	101010	小于设置	0111

Verilog 代码如下，细节见注释：

```
module aluctr(
    input [1:0] ALUOp,
    input [5:0] funct,
    output reg [3:0] ALUCtr
```

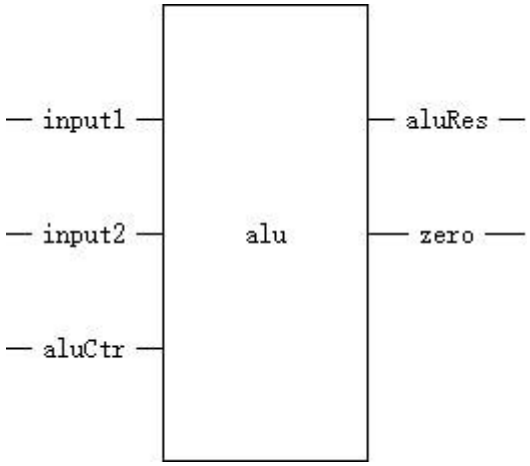
```
    );

    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
        8'b00xxxxxx: ALUCtr = 4'b0010; // lw，sw，addi
        8'b01xxxxxx: ALUCtr = 4'b0110; // beq
        8'b1xxx0000: ALUCtr = 4'b0010; // add
        8'b1xxx0010: ALUCtr = 4'b0110; // sub
        8'b1xxx0100: ALUCtr = 4'b0000; // and
        8'b1xxx0101: ALUCtr = 4'b0001; // or
        8'b1xxx1010: ALUCtr = 4'b0111; // slt
    endcase
endmodule
```

4. ALU 模块

ALU 模块主要接受两个操作数，完成各类运算操作。包括加、减、与、或、小于设置。R 型指令需要 ALU 进行运算，sw、lw 需要 ALU 进行地址运算，beq 需要 ALU 进行相等比较，以及指令地址运算。

所以需要输入 ALU 控制信号判断 ALU 进行的操作，再进一步对两个操作数进行操作，操作完输出结果。

input	input1[31:0]
input	input2[31:0]
input	aluCtr[3:0]
output	aluRes[31:0]
output	zero



ALU 输入信号与操作类型对应如下：

输入信号	操作类型
0000	与
0001	或
0010	加
0110	减
0111	小于设置

Verilog 代码如下，细节见注释：

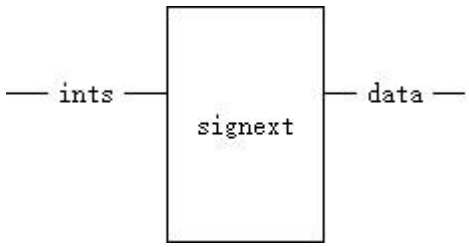
```
module alu(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output reg[31:0] aluRes,
    output reg zero
    );

     always @(input1 or input2 or aluCtr) //  运算数或控制码变化时操作
     begin
         case(aluCtr)
              4'b0110: //  减
                   begin
                        aluRes = input1 - input2;
                        if(aluRes == 0)
                             zero = 1;
                        else
                             zero = 0;
                   end
              4'b0010: //  加
                   aluRes = input1 + input2;
              4'b0000: //  与
                   aluRes = input1 & input2;
              4'b0001: //  或
                   aluRes = input1 | input2;
              4'b1100: //  异或
                   aluRes = ~(input1 | input2);
              4'b0111: //  小于设置
                   begin
                        if(input1<input2)
                             aluRes = 1;
                   end
              default:
```

```
                    aluRes = 0;
            endcase
        end
endmodule
```

5. 符号扩展模块

将 16 位有符号数扩展成 32 位有符号数，只需要在 16 位数前面补足符号即可。

input	inst[15:0]
output	data[31:0]

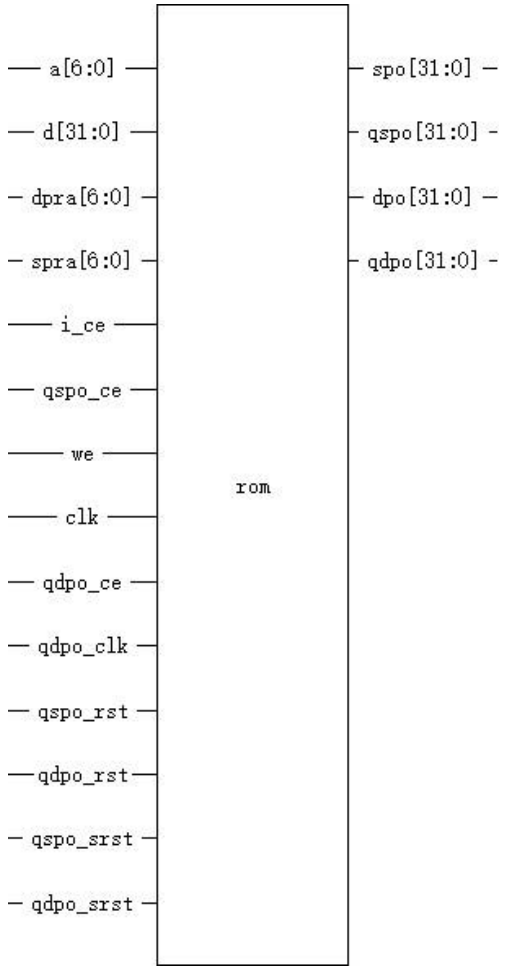
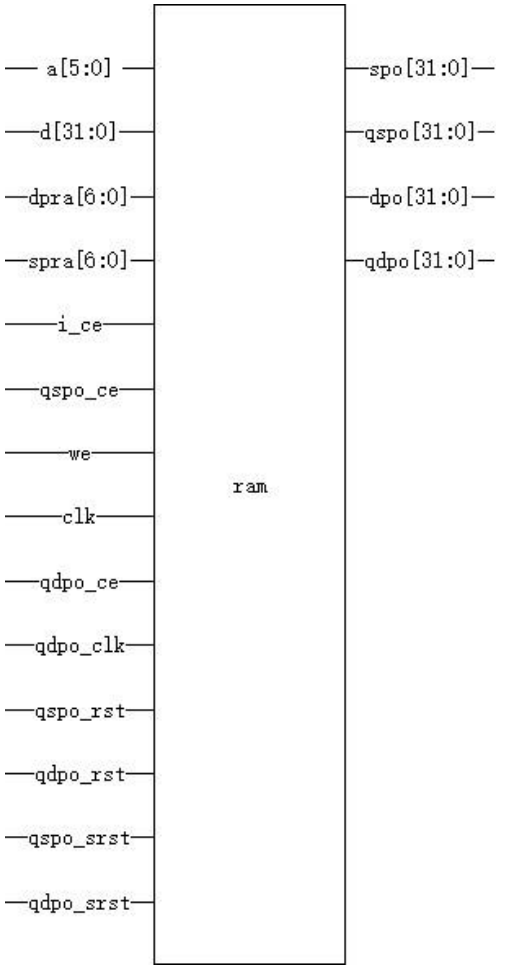


Verilog 代码如下，细节见注释：

```
module signext(
      input [15:0] inst, //  输入 16 位
      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]?{16'hffff,inst}:{16'h0000,inst};

endmodule
```

6. Rom 及 ram 模块

该模块由 xilinx 提供的 IP 核进行设计，其中 ROM 模块由地址线、数据线和时钟信号线组成。RAM 模块由时钟线，地址线，读写使能线，数据线构成。

rom	ram
	

7. 顶层模块

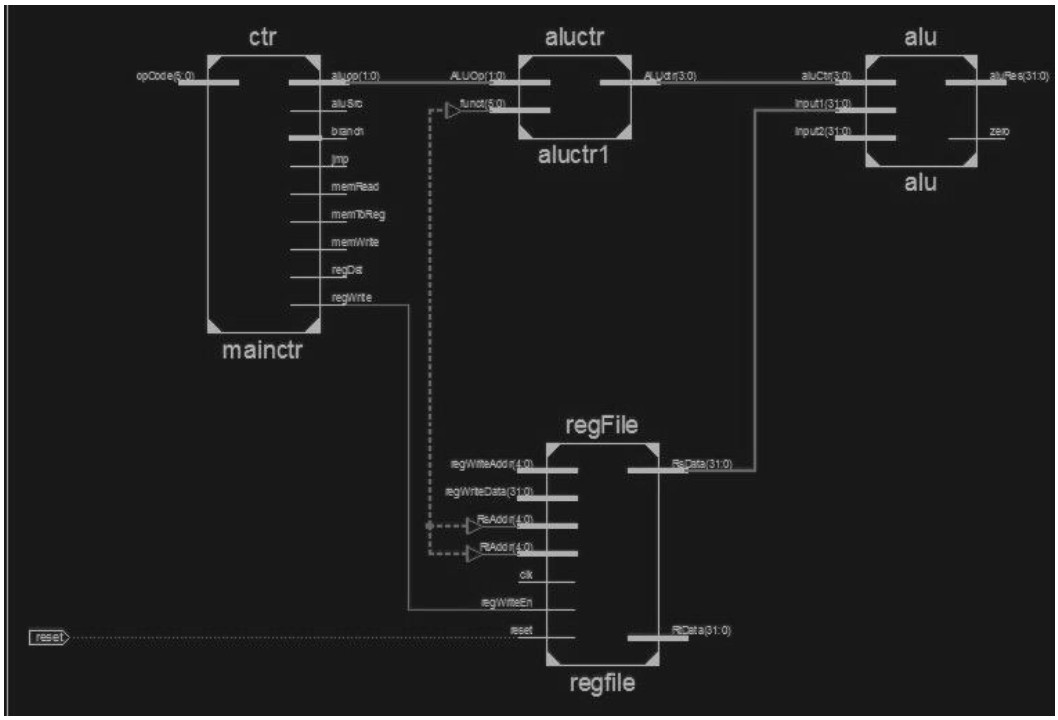
顶层模块需要将前面的多个模块实例化后，通过导线以及多路复用器将各个部件链接起来，并且在时钟的控制下修改 PC 的值，PC 是一个 32 位的寄存器，每个时钟沿自动增加 4。

多路复用器 MUX 直接通过三目运算符实现：

Assign $OUT = SEL\ ?INPUT1 : INPUT2$;

其中，OUT，SEL，INPUT1，INPUT2 都是预先定义的信号。

顶层模块需要输入时钟和复位信号，然后首先读取 rom 的机器指令，然后通过各个模块执行。示意图如下：



Veilog 代码如下，细节见注释：

```
module top(
    input clkin,
    input reset
    );

    // 指令寄存器 pc
    reg[31:0] pc, add4;
    wire choose4;
    // 复用器信号线
    wire[31:0] expand2, mux2, mux3, mux4, mux5, address, jmpaddr, inst;
    wire[4:0] mux1;

    // CPU 控制信号线
    wire reg_dst, jmp, branch, memread, memwrite, memtoreg;
    wire[1:0] aluop;
    wire alu_src, regwrite;

    // ALU 信号线
    wire zero;
    wire[31:0] aluRes;

    // ALU 控制信号线
    wire[3:0] aluCtr;
```

```
    // 内存信号线
    wire[31:0] memreaddata;

    // 寄存器信号线
    wire[31:0] RsData, RtData;

    // 扩展信号线
    wire[31:0] expand;

    always @(negedge clkin) // 时钟下降沿操作
    begin
        if(!reset)
        begin
            pc = mux5; // 计算下一条 pc，修改 pc
            add4 = pc + 4;
        end
        else
        begin
            pc = 32'b0; // 复位时 pc 写 0
            add4 = 32'h4;
        end
end

// 实例化控制器模块
ctr mainctr(
        .opCode(inst[31:26]),
        .regDst(reg_dst),
        .aluSrc(alu_src),
        .memToReg(memtoreg),
        .regWrite(regwrite),
        .memRead(memread),
        .memWrite(memwrite),
        .branch(branch),
        .aluop(aluop),
        .jmp(jmp));

// 实例化 ALU 模块
alu alu(.input1(RsData),
        .input2(mux2),
        .aluCtr(aluCtr),
        .zero(zero),
        .aluRes(aluRes));
```

```
// 实例化 ALU 控制模块
aluctr aluctr1(
        .ALUOp(aluop),
        .funct(inst[5:0]),
        .ALUCtr(aluCtr));

// 实例化 ram 模块
dram dmem(
        .a(aluRes[7:2]),
        .d(RtData),
        .clk(!clkin),
        .we(memwrite),
        .spo(memreaddata)
);

// 实例化 rom 模块
irom imem(
        .a(pc[8:2]),
        .clk(clkin),
        .spo(inst)
);

// 实例化寄存器模块
regFile regfile(
        .RsAddr(inst[25:21]),
        .RtAddr(inst[20:16]),
        .clk(!clkin),
        .reset(reset),
        .regWriteAddr(mux1),
        .regWriteData(mux3),
        .regWriteEn(regwrite),
        .RsData(RsData),
        .RtData(RtData)
);

// 实例化符号扩展模块
signext signext(.inst(inst[15:0]), .data(expand));

// 各个控制信号线，地址，符号扩展
assign mux1 = reg_dst ? inst[15:11] : inst[20:16];
assign mux2 = alu_src ? expand : RtData;
assign mux3 = memtoreg ? memreaddata : aluRes;
assign mux4 = choose4 ? address : add4;
assign mux5 = jmp ? jmpaddr : mux4;
```

```
    assign choose4 = branch & zero;
    assign expand2 = expand << 2;
    assign jmpaddr = {add4[31:28], inst[25:0], 2'b00};
    assign address = pc + expand2;

endmodule
```

五、 Rom 汇编程序设计、代码

1. 汇编代码

```
main:
addi $s1,$zero,85 // U
sw $s1,0($s0)
addi $s1,$zero,50 // 2
sw $s1,4($s0)
addi $s1,$zero,48 // 0
sw $s1,8($s0)
addi $s1,$zero,49 // 1
sw $s1,12($s0)
addi $s1,$zero,51 // 3
sw $s1,16($s0)
addi $s1,$zero,49 // 1
sw $s1,20($s0)
addi $s1,$zero,51 // 3
sw $s1,24($s0)
addi $s1,$zero,55 // 7
sw $s1,28($s0)
addi $s1,$zero,54 // 6
sw $s1,32($s0)
addi $s1,$zero,56 // 8
sw $s1,36($s0)
add $4,$2,$3 //  将$4 = $2 + $3
lw $4,4($2) //读出$2 偏移 16 个字节的内容的到$4
sw $2,8($2) //将$2 写入$2 偏移 32 字节的 RAM
sub $2,$4,$3 // $2=$4-$3
or $2,$4,$3 // $2=$4 | $3
and $2,$4,$3 // $2=$4 & $3
slt $2,$4,$3 // $4<$3，$2 = 1
beq $4,$3,exit //$4 == $3，退出
```

```
j main
exit:lw $2,0($3)
j main
```

2. 机器码

```
[00400024] 20110055  addi $17, $0, 85          ; 2: addi $s1,$zero,85
[00400028] ae110000  sw $17, 0($16)            ; 3: sw $s1,0($s0)
[0040002c] 20110032  addi $17, $0, 50          ; 4: addi $s1,$zero,50
[00400030] ae110004  sw $17, 4($16)            ; 5: sw $s1,4($s0)
[00400034] 20110030  addi $17, $0, 48          ; 6: addi $s1,$zero,48
[00400038] ae110008  sw $17, 8($16)            ; 7: sw $s1,8($s0)
[0040003c] 20110031  addi $17, $0, 49          ; 8: addi $s1,$zero,49
[00400040] ae11000c  sw $17, 12($16)           ; 9: sw $s1,12($s0)
[00400044] 20110033  addi $17, $0, 51          ; 10: addi $s1,$zero,51
[00400048] ae110010  sw $17, 16($16)           ; 11: sw $s1,16($s0)
[0040004c] 20110031  addi $17, $0, 49          ; 12: addi $s1,$zero,49
[00400050] ae110014  sw $17, 20($16)           ; 13: sw $s1,20($s0)
[00400054] 20110033  addi $17, $0, 51          ; 14: addi $s1,$zero,51
[00400058] ae110018  sw $17, 24($16)           ; 15: sw $s1,24($s0)
[0040005c] 20110037  addi $17, $0, 55          ; 16: addi $s1,$zero,55
[00400060] ae11001c  sw $17, 28($16)           ; 17: sw $s1,28($s0)
[00400064] 20110036  addi $17, $0, 54          ; 18: addi $s1,$zero,54
[00400068] ae110020  sw $17, 32($16)           ; 19: sw $s1,32($s0)
[0040006c] 20110038  addi $17, $0, 56          ; 20: addi $s1,$zero,56
[00400070] ae110024  sw $17, 36($16)           ; 21: sw $s1,36($s0)
[00400074] 00432020  add $4, $2, $3            ; 22: add $4,$2,$3
[00400078] 8c440004  lw $4, 4($2)              ; 23: lw $4,4($2)
[0040007c] ac420008  sw $2, 8($2)              ; 24: sw $2,8($2)
[00400080] 00831022  sub $2, $4, $3            ; 25: sub $2,$4,$3
[00400084] 00831025  or $2, $4, $3             ; 26: or $2,$4,$3
[00400088] 00831024  and $2, $4, $3            ; 27: and $2,$4,$3
[0040008c] 0083102a  slt $2, $4, $3            ; 28: slt $2,$4,$3
[00400090]  10830002  beq  $4,  $3,  8  [exit-0x00400090]; 29: beq
$4,$3,exit
[00400094] 08100009  j 0x00400024 [main]       ; 30: j main
[00400098] 8c620000  lw $2, 0($3)              ; 31: lw $2,0($3)
[0040009c] 08100009  j 0x00400024 [main]       ; 32: j main
```

3. 制作 coe 文件

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
20110055
ae110000
20110032
ae110004
20110030
ae110008
20110031
ae11000c
20110033
ae110010
20110031
ae110014
20110033
ae110018
20110037
ae11001c
20110036
ae110020
20110038
ae110024
00432020
8c440004
ac420008
00831022
00831025
00831024
0083102a
10830002
08000000
8c620000
08000000
```

六、 各个子模块仿真

1. ALU 控制器模块

1) 仿真模块

```
module aluctrsim;

	// Inputs
	reg [1:0] ALUOp;
	reg [5:0] funct;

	// Outputs
	wire [3:0] ALUctr;

	// Instantiate the Unit Under Test (UUT)
	aluctr uut (
		.ALUOp(ALUOp),
		.funct(funct),
		.ALUCtr(ALUctr)
	);

	initial begin
		// Initialize Inputs
		ALUOp = 0;
		funct = 0;

		// Wait 100 ns for global reset to finish
		#100;
		// Add stimulus here
		ALUOp = 2'b01;
		funct = 0;

		#100;
		ALUOp = 2'b10;
		funct = 6'b100000;

		#100;
		ALUOp = 2'b10;
		funct = 6'b101010;

	end
endmodule
```