| 成绩 |
| --- |
| |

# 中国农业大学

# 课程论文

## (2019 -2020 学年夏季学期)

论文题目：基于单周期 CPU 的 32 位模型计算机

课程名称：计算机组成与体系结构课程设计

任课教师：黄岚　冯子腾

班　　级：计算机 172

学　　号：2017304010413

姓　　名：张靖祥

# 目录

**摘　要：** 本实验为基于单周期 CPU 的 32 位模型计算机的设计，以实验指导书为模板，vivado 为开发工具，Verilog 语言为程序设计语言，minisys 实验板为最终的成果。在实验过程中，我们小组重点设计了控制器模块、取指模块、存储器模块和整体的顶部模块的封装，参考借鉴了数据通路的设计、译码器和运算器的设计、IO 接口的设计以及整个实验的设计框架。最终，通过 minisys 的汇编器将汇编指令转化为机器指令，并将整个 CPU 下载到 minisys 实验板上进行实验，总计完成了输入输出、节日彩灯、原码一位乘法和综合应用四项内容.
**关键字：** 模型计算机 设计

# 一、课程设计任务与实验设备、开发工具链

课程设计任务：完成单周期 CPU 的 32 位模型计算机的设计，重点为设计控制器，取指部件，存储器。
实验设备：minisys 实验板
开发工具链：vivado 2015.4、Minisys1Av2.2 汇编器

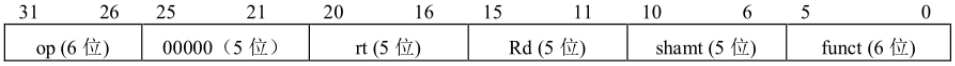
# 二、总体设计思路

## 1. 指令系统

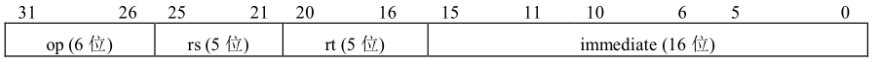
| 助记符 | 指　　令　　格　　式 | | | | | | 示　例 | 示例含义 | 操作及解释 |
|---|---|---|---|---|---|---|---|---|---|
| BIT # | 31..26 | 25..21 | 20..16 | 15..11 | 10..6 | 5..0 | | | |
| **R-类型** | **op** | **rs** | **rt** | **rd** | **shamt** | **func** | | | |
| add | 000000 | rs | rt | rd | 00000 | 100000 | add $1,$2,$3 | $1=$2+S3 | (rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1 |
| addu | 000000 | rs | rt | rd | 00000 | 100001 | addu $1,$2,$3 | $1=$2+S3 | (rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1 |
| sub | 000000 | rs | rt | rd | 00000 | 100010 | sub $1,$2,$3 | $1=$2-S3 | (rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1 |
| subu | 000000 | rs | rt | rd | 00000 | 100011 | subu $1,$2,$3 | $1=$2-S3 | (rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1 |
| and | 000000 | rs | rt | rd | 00000 | 100100 | and $1,$2,$3 | $1=$2&S3 | (rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1 |
| or | 000000 | rs | rt | rd | 00000 | 100101 | or $1,$2,$3 | $1=$2\|S3 | (rd)←(rs) \| (rt); rs=$2,rt=$3,rd=$1 |
| xor | 000000 | rs | rt | rd | 00000 | 100110 | xor $1,$2,$3 | $1=$2^S3 | (rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1 |
| nor | 000000 | rs | rt | rd | 00000 | 100111 | nor $1,$2,$3 | $1= ~($2 \| S3) | (rd)←~((rs) \| (rt)); rs=$2,rt=$3,rd=$1 |
| slt | 000000 | rs | rt | rd | 00000 | 101010 | slt $1,$2,$3 | if($2<$3) $1=1 else $1=0 | if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1 |
| sltu | 000000 | rs | rt | rd | 00000 | 101011 | sltu $1,$2,$3 | if($2<$3) $1=1 else $1=0 | if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1, 无符号数 |
| sll | 000000 | 00000 | rt | rd | shamt | 000000 | sll $1,$2,10 | $1=$2<<10 | (rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10 |
| srl | 000000 | 00000 | rt | rd | shamt | 000010 | srl $1,$2,10 | $1=$2>>10 | (rd) ← (rt)>>shamt, rt=$2, rd=$1, shamt=10, (逻辑右移) |
| sra | 000000 | 00000 | rt | rd | shamt | 000011 | sra $1,$2,10 | $1=$2>>10 | (rd) ← (rt)>>shamt, rt=$2, rd=$1, shamt=10, (算术右移，注意符号位保留) |
| sllv | 000000 | rs | rt | rd | 00000 | 000100 | sllv $1,$2,$3 | $1=$2<<$3 | (rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1 |
| srlv | 000000 | rs | rt | rd | 00000 | 000110 | srlv $1,$2,$3 | $1=$2>>$3 | (rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移) |
| srav | 000000 | rs | rt | rd | 00000 | 000111 | srav $1,$2,$3 | $1=$2>>$3 | (rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留) |
| jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 | jr $31 | goto $31 | (PC)←(rs) |
| **I-类型** | **op** | **rs** | **rt** | **immediate** | | | | | |
| addi | 001000 | rs | rt | immediate | | | addi $1,$2,10 | $1=$2+10 | (rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2 |
| addiu | 001001 | rs | rt | immediate | | | addiu $1,$2,10 | $1=$2+10 | (rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2 |
| andi | 001100 | rs | rt | immediate | | | andi $1,$2,10 | $1=$2&10 | (rt)←(rs)&(zero-extend)immediate,rt=$1,rs=$2 |
| ori | 001101 | rs | rt | immediate | | | ori $1,$2,10 | $1=$2\|10 | (rt)←(rs)\|(zero-extend)immediate,rt=$1,rs=$2 |
| xori | 001110 | rs | rt | immediate | | | xori $1,$2,10 | $1=$2^10 | (rt)←(rs)^(zero-extend)immediate,rt=$1,rs=$2 |
| lui | 001111 | 00000 | rt | immediate | | | lui $1,10 | $1=10*65536 | (rt)←immediate<<16 & 0FFFF0000H，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 |
| lw | 100011 | rs | rt | offset | | | lw $1,10($2) | $1=Memory[ $2+10] | (rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2 |
| sw | 101011 | rs | rt | offset | | | sw $1,10($2) | Memory[ $2+10] =$1 | Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2 |
| beq | 000100 | rs | rt | offset | | | beq $1,$2,40 | if($1=$2) goto PC+4+40 | if ((rt)=(rs)) then (PC)←(PC)+4+( (Sign-Extend) offset<<2), rs=$1, rt=$2 |
| bne | 000101 | rs | rt | offset | | | bne $1,$2,40 | if($1≠$2) goto PC+4+40 | if ((rt) ≠(rs)) then (PC)←(PC)+4+( (Sign-Extend) offset<<2) , rs=$1, rt=$2 |
| slti | 001010 | rs | rt | immediate | | | slti $1,$2,10 | if($2<10) $1=1 else $1=0 | if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1 |
| sltiu | 001011 | rs | rt | immediate | | | sltiu $1,$2,10 | if($2<10) $1=1 else $1=0 | if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1 |
| **J-类型** | **op** | **address** | | | | | | | |
| j | 000010 | address | | | | | j 10000 | goto 10000 | (PC)←( (Zero-Extend) address<<2), address=10000/4 |
| jal | 000011 | address | | | | | jal 10000 | $31=PC+4 goto 10000 | ($31)←(PC)+4; (PC)←( (Zero-Extend) address<<2), address=10000/4 |

本实验中的单周期 CPU 有 3 个种类的指令，分别是 R 型指令（寄存器型指令）、I 型指令（立即数型指令）和 J 型指令（跳转型指令）。

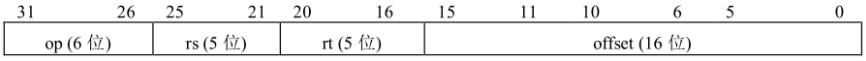
R 型指令的指令格式如下。其中 op 字段为操作码，R 型指令的 op 字段一律为 000000，rt 和 rd 为操作数的寄存器编号，rs 为要写回的寄存器编号，shamt 为位移指令的位移位数，funct 为 R 型指令的操作方法。



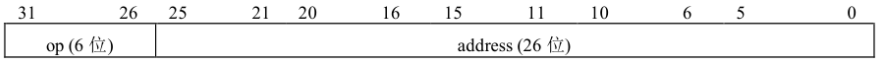

I 型指令的指令格式如下。其中 op 为操作码，rt 寄存器和 immediate 立即数为 ALU 的操作数，最后的结果写回到 rs 寄存器中



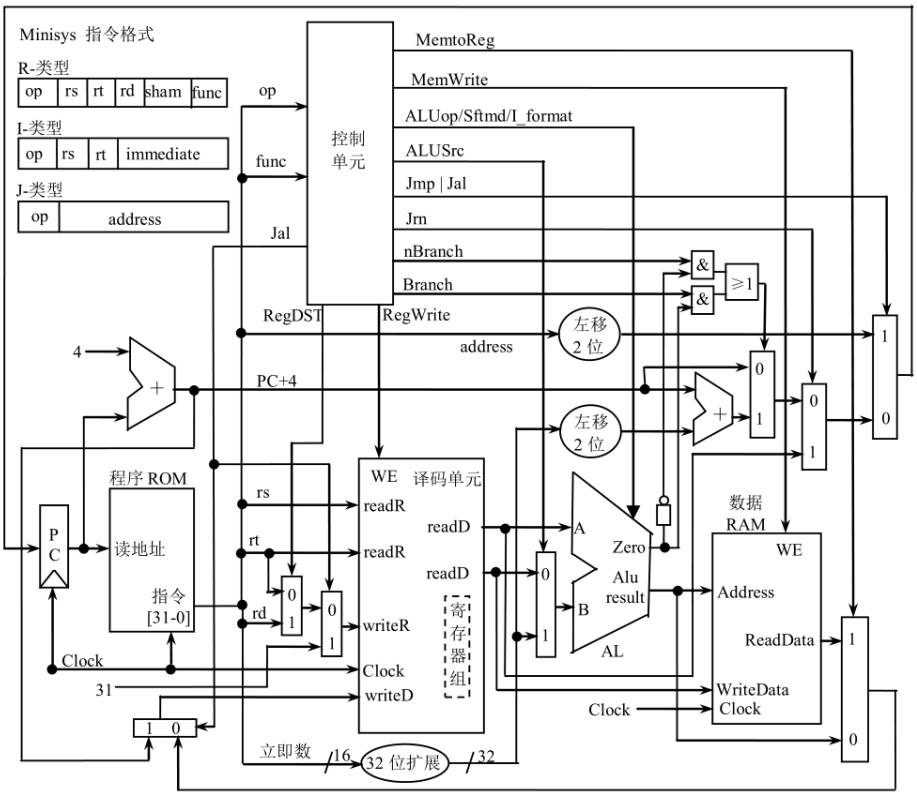

其中 I 型指令的 sw 和 lw 指令的立即数部分为偏移量，sw 指令中的 rs 目标寄存器为输出，而选中的内存单元的内容为输入。beq 和 bne 指令中为 rs 和 rt 进行比较，判断是否转移到 pc+4+偏移量的地址




J 型指令的指令格式如下，address 为直接跳转的地址




## 2. 数据通路



# 三、每个模块的设计与实现（共同完成的部分）

## 1. *带有 IO 操作的控制单元的设计

### （1）控制单元的输入输出

```
input[5:0]   Opcode;            // 来自取指单元instruction[31..26]
input[21:0]  Alu_resultHigh;    // *来自执行单元Alu_Result[31:10]
input[5:0]   Function_opcode;   // 来自取指单元r-类型 instructions[5..0]
output       Jrn;               // 为1表明当前指令是jr
output       RegDST;            // 为1表明目的寄存器是rd，否则目的寄存器是rt
output       ALUSrc;            // 为1表明第二个操作数是立即数（beq，bne除外）
output       MemorIOtoReg;  // *为1表明需要从存储器或IO读数据到寄存器
output       RegWrite;      // 为1表明该指令需要写寄存器
output       MemRead;       // *为1表明从存储器读
output       MemWrite;      // *为1表明该指令需要写存储器
output       IORead;        // *为1表明是IO读
output       IOWrite;       // *为1表明是IO写
output       Branch;        // 为1表明是Beq指令
output       nBranch;       // 为1表明是Bne指令
output       Jmp;           // 为1表明是J指令
output       Jal;           // 为1表明是Jal指令
output       I_format;      // 为1表明该指令是除beq，bne，LW，SW之外的其他I-类型指令
output       Sftmd;         // 为1表明是移位指令
output[1:0]  ALUOp;         // 是R-类型或I_format=1时位1为1，beq、bne指令则位0为1
```

### （2）RegDST 的控制电路设计

| op→ | 001101 | 001001 | 100011 | 101011 | 000100 | 000010 | 000000 |
|---|---|---|---|---|---|---|---|
| 指令操作码 | ori | addiu | lw | sw | beq | j | R-format |
| | 0 | 0 | 0 | x | x | x | 1 |

RegDST 信号的分析

```
assign R_format = (Opcode==6'b000000)? 1'b1:1'b0;        //--00h
assign RegDST = R_format;                                //说明目标是rd，否则是rt
```

### （3）RegWrite 的控制电路设计

| op→ | 001xxx | 000000 | 100011 | 101011 | 000011 | 000010 | 000000 |
|---|---|---|---|---|---|---|---|
| 指令操作码 | I-format | jr | lw | sw | jal | j | R-format |
| | 1 | 0 | 1 | x | 1 | x | 1 |

RegWrite 信号的分析

```
assign Sw = (Opcode==6'b101011)? 1'b1:1'b0;
assign Jal = (Opcode==6'b000011)? 1'b1:1'b0;
assign Jrn = (Opcode==6'b000000 && Function_opcode == 6'b001000)? 1'b1:1'b0;
assign Lw = (Opcode==6'b100011)? 1'b1:1'b0;
assign RegWrite = ( Jrn || Sw || Branch || nBranch || Jmp  )? 1'b0:1'b1;
```

### （4）ALU 的控制电路设计

| 指令操作码 | ALUop |
|---|---|
| LW | 00 |
| SW | 00 |
| BEQ,BNE | 01 |
| R-format | 10 |
| I-format | 10 |

```
assign ALUOp = {(R_format || I_format),(Branch || nBranch)};
 // 是R－type或需要立即数作32位扩展的指令1位为1,beq、bne指令则0位为1
```

### （5）其他部分

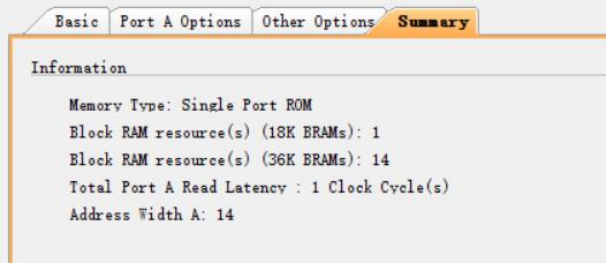
```
assign Branch = (Opcode==6'b000100)? 1'b1:1'b0;
assign nBranch = (Opcode==6'b000101)? 1'b1:1'b0;
assign Jmp = (Opcode==6'b000010)? 1'b1:1'b0;
assign Sw = (Opcode==6'b101011)? 1'b1:1'b0;
assign ALUSrc = (Opcode[5:3]==3'b001 || Lw || Sw)? 1'b1:1'b0;
assign MemWrite = ((Sw == 1) && (Alu_resultHigh != 22'b1111111111111111111111)  )? 1'b1:1'b0;
assign MemRead = ((Lw == 1) && (Alu_resultHigh != 22'b1111111111111111111111)  )? 1'b1:1'b0;
assign IOWrite = ((Sw == 1) && (Alu_resultHigh == 22'b1111111111111111111111)  )? 1'b1:1'b0;
assign IORead = ((Lw == 1) && (Alu_resultHigh == 22'b1111111111111111111111)  )? 1'b1:1'b0;
assign MemorIOtoReg = (Opcode==6'b100011)? 1'b1:1'b0;
assign Sftmd = (Opcode==6'b000000 && Function_opcode[5:3] == 3'b000 )? 1'b1:1'b0;
```

## 2. *取指单元的设计

### （1）取指单元的输入输出

```
output[31:0] Instruction;          // 输出指令到其他模块
output[31:0] PC_plus_4_out;        // (pc+4)送执行单元
input[31:0]  Add_result;           // 来自执行单元,算出的跳转地址
input[31:0]  Read_data_1;          // 来自译码单元，jr指令用的地址
input        Branch;               // 来自控制单元
input        nBranch;              // 来自控制单元
input        Jmp;                  // 来自控制单元
input        Jal;                  // 来自控制单元
input        Jrn;                  // 来自控制单元
input        Zero;                 //来自执行单元
input        clock,reset;          //时钟与复位
output[31:0] opcplus4;             // JAL指令专用的PC+4
```

（2）定义 ROM




```
prgrom instmem(
    .clka(clock),          // input wire clka
    .addra(PC[15:2]),      // input wire [13 : 0] addra
    .douta(Instruction)          // output wire [31 : 0] douta
);
```

（3）PC+4 的处理

```
assign PC_plus_4[31:2] = PC[31:2] + 1;
assign PC_plus_4[1:0] = 2'b00;
assign PC_plus_4_out = PC_plus_4;
```

## （4）beq、bne、jr 跳转对 PC 的修改

```
always @* begin  // beq $n ,$m if $n=$m branch   bne if $n /=$m branch jr
// 请考虑以上三条指令的判断条件，
// 以及三条指令的执行该给next_PC赋什么值
if (Jrn == 1)
    begin
    next_PC[1:0] = 2'b00;
    next_PC[31:2] = Read_data_1[29:0];
    end
else if ((Branch == 1 && Zero == 1)||(nBranch == 1 && Zero != 1))
    begin
    next_PC[1:0] = 2'b00;
    next_PC[31:2] = Add_result[29:0];
    end
else
    begin
    next_PC = PC_plus_4;
    end
end
```

## （5）最终修改 PC 值

```
always @(negedge clock) begin  //（含J，Jal指令和reset的处理）
   if (reset == 1)
        begin
        PC = 32'h00000000;
        end
   else if (Jmp == 1)
        begin
        PC[27:2] = Instruction[25:0];
        PC[31:28] = 4'b0000;
        PC[1:0] = 2'b00;
        end
    else if (Jal == 1)
        begin
        opcplus4[29:0] = PC_plus_4[31:2];
        opcplus4[31:30] = 2'b00;
        PC[27:2] = Instruction[25:0];
        PC[31:28] = 4'b0000;
        PC[1:0] = 2'b00;
        end
    else
        begin
        PC = next_PC;
        end
end
```

## 3. 译码器的设计

### （1）译码单元的输入和输出

```
output[31:0] read_data_1;                // 输出的第一操作数
output[31:0] read_data_2;                // 输出的第二操作数
input[31:0]  Instruction;                // 取指单元来的指令
input[31:0]  read_data;                  //  从DATA RAM or I/O port取出的数据
input[31:0]  ALU_result;                 // 从执行单元来的运算的结果，需要扩展立即数到32位
input        Jal;                        //  来自控制单元，说明是JAL指令
input        RegWrite;                   // 来自控制单元
input        MemorIOtoReg;               // 来自控制单元
input        RegDst;                     //  来自控制单元
output[31:0] Sign_extend;                // 扩展后的32位立即数
input        clock,reset;                 // 时钟和复位
input[31:0]  opcplus4;                  // 来自取指单元，JAL中用

wire[4:0] read_register_1_address;     // 要读的第一个寄存器的号（rs）
wire[4:0] read_register_2_address;     // 要读的第二个寄存器的号（rt）
wire[4:0] write_register_address_1;    // r-form指令要写的寄存器的号（rd）
wire[4:0] write_register_address_0;    // i-form指令要写的寄存器的号(rt)
wire[15:0] Instruction_immediate_value;  // 指令中的立即数
wire[5:0] opcode;                        // 指令码
```

### （2）指令中各分量提取

```
assign opcode = Instruction[31:26];                      //OP
assign read_register_1_address = Instruction[25:21];    //rs
assign read_register_2_address = Instruction[20:16];    //rt
assign write_register_address_1 = Instruction[15:11];   //rd(r-form)
assign write_register_address_0 = Instruction[20:16];   //rt(i-form)
assign Instruction_immediate_value = Instruction[15:0]; //data,rladr(i-form)
```

### （3）定义寄存器组和对寄存器进行读写操作

```
assign read_data_1 = register[read_register_1_address];
assign read_data_2 = register[read_register_2_address];
```

（4）目标寄存器的指定

```
always @* begin
//这个进程指定不同指令下的目标寄存器
    if((RegDst==1) && (Jal==0)) //r-form jal (pc+1 > sp[$Rlast])
        write_register_address = write_register_address_1; //RD(15-11)  R-form指令
    else if((RegDst==0) && (Jal==1))
        write_register_address = 5'b11111;
        //JAL 指令需要将下个指令的地址给最后一个寄存器
    else  write_register_address  = write_register_address_0;//i-form rt(20-16)
end
```

（5）准备要写的数据

```
always @* begin  //这个进程基本上是实现结构图中右下的多路选择器，lw $5,$3(100)
    if((MemorIOtoReg==0) && (Jal== 0)) begin              //不是LW and IO, 也不是JAL指令
        write_data = ALU_result[31:0];
    end else if((MemorIOtoReg==0) && (Jal== 1)) begin     //不是LW，但是是jal,下个地址存$15
        write_data = opcplus4;
    end else begin
        write_data = read_data;                           //是LW指令
    end
end
```

（6）完成对寄存器的写操作

```
 integer i;
 always @(posedge clock) begin       // 本进程写目标寄存器
     if(reset==1) begin              // 初始化寄存器组
         for(i=0;i<32;i=i+1) register[i] <= i;
     end else if(RegWrite==1) begin  // 注意寄存器0恒等于0
         case(write_register_address[4:0])
             5'd0:register[0] <= 32'd0;
             5'd1:register[1] <= write_data;
             5'd2:register[2] <= write_data;
                       • • • • • •
             5'd30:register[30] <= write_data;
             5'd31:register[31] <= write_data;
             default:register[0] <= 32'd0;
         endcase
     end
 end
```

### （7）立即数扩展

```
wire sign;                          // 取符号位的值
assign sign = Instruction_immediate_value[15];
assign Sign_extend[31:0] = ((opcode==6'b001100)      // andi
                  ||(opcode==6'b001101)              // ori
                  ||(opcode==6'b001110)              // xori
                  ||(opcode==6'b001011))             // sltiu
                  ? {16'h0000,Instruction_immediate_value[15:0]}   // 立即数0扩展
                  :{sign,sign,sign,sign,sign,sign,sign,sign,
                  sign,sign,sign,sign,sign,sign,sign,sign,
                  Instruction_immediate_value[15:0]};   //立即数符号扩展
```

## 4. 执行模块的设计

### （1）执行模块的输入和输出

```
input[31:0]  Read_data_1;       // 从译码单元的Read_data_1中来
input[31:0]  Read_data_2;       // 从译码单元的Read_data_2中来
input[31:0]  Sign_extend;       // 从译码单元来的扩展后的立即数
input[5:0]   Function_opcode;   // 取指单元来的r-类型指令功能码,r-form instructions[5:0]
input[5:0]   Exe_opcode;        // 取指单元来的操作码
input[1:0]   ALUOp;             // 来自控制单元的运算指令控制编码
input[4:0]   Shamt;             // 来自取指单元的instruction[10:6]，指定移位次数
input        Sftmd;             // 来自控制单元的，表明是移位指令
input        ALUSrc;            // 来自控制单元，表明第二个操作数是立即数（beq，bne除外）
input        I_format;          // 来自控制单元，表明是除beq，bne，LW，SW之外的I-类型指令
input        Jrn;               // 来自控制单元，书名是JR指令
output       Zero;              // 为1表明计算值为0
output[31:0] ALU_Result;        // 计算的数据结果
output[31:0] Add_Result;        // 计算的地址结果
input[31:0]  PC_plus_4;         // 来自取指单元的PC+4

reg[31:0] ALU_Result;
wire[31:0] Ainput,Binput;
reg[31:0] Cinput,Dinput;
reg[31:0] Einput,Finput;
reg[31:0] Ginput,Hinput;
reg[31:0] Sinput;
reg[31:0] ALU_output_mux;
wire[32:0] Branch_Add;
wire[2:0] ALU_ctl;
wire[5:0] Exe_code;
wire[2:0] Sftm;
wire Sftmd;
reg s;
```

## （2）运算数据的选择

```
assign Ainput = Read_data_1;
assign Binput = (ALUSrc == 0) ? Read_data_2 : Sign_extend[31:0]; //R/LW,SW  sft  else的时候含LW和SW
```

## （3）完成算术逻辑运算

执行码与指令的对应关系：

| Exe_code[3..0] | ALUOp[1..0] | ALU_ctl[2..1] | 指令助记符 |
|---|---|---|---|
| 0100 | 10 | 000 | and,andi |
| 0101 | 10 | 001 | or,ori |
| 0000 | 10 | 010 | add,addi |
| xxxx | 00 | 010 | lw, sw |
| 0001 | 10 | 011 | addu, addiu |
| 0110 | 10 | 100 | xor,xori |
| 0111 | 10 | 101 | nor,lui |
| 0010 | 10 | 110 | sub, slti |
| xxxx | 01 | 110 | beq，bne |
| 0011 | 10 | 111 | subu, sltiu |
| 1010 | 10 | 111 | slt |
| 1011 | 10 | 111 | sltu |

```
assign Sftm = Function_opcode[2:0];    // 实际有用的只有低三位(移位指令)
assign Exe_code = (I_format==0) ? Function_opcode : {3'b000,Exe_opcode[2:0]};
assign ALU_ctl[0] = (Exe_code[0] | Exe_code[3]) & ALUOp[1];
assign ALU_ctl[1] = ((!Exe_code[2]) | (!ALUOp[1]));
assign ALU_ctl[2] = (Exe_code[1] & ALUOp[1]) | ALUOp[0];
assign Zero = (ALU_output_mux[31:0]== 32'h00000000) ? 1'b1 : 1'b0;
```

```
assign Branch_Add = PC_plus_4[31:2] + Sign_extend[31:0];
assign Add_Result = Branch_Add[31:0];   //算出的下一个PC值已经做了除4处理，所以不需左i
always @(ALU_ctl or Ainput or Binput) begin
    case(ALU_ctl)
        3'b000:ALU_output_mux = Ainput & Binput; //36=24h=>0100(P2=1,OP1=1)
        //CT0=(P0 OR P3) AND OP1=0,CT1=!P2 OR !OP1=0,CT2=(P1 AND OP1)OR OP0=0
        3'b001:ALU_output_mux = Ainput | Binput; //37=25h=>0101(P2=1,P0=1,OP1=1)
        //CT0=(P0 OR P3) AND OP1=[1],CT1=!P2 OR !OP1=0,CT2=(P1 AND OP1)OR OP0=0
        3'b010:ALU_output_mux = Ainput +  Binput;//32=20h=>0000(OP1=1) ADD
        //35=23h=>0011(LW) ct0=0,ct1=1,ct2=0,43=2bh=>1011(SW),ct0=0,ct1=1,ct2=0
        //CT0=(P0 OR P3) AND OP1=0,CT1=!P2 OR !OP1=[1],CT2=(P1 AND OP1)OR OP0=0
        3'b011:ALU_output_mux = Ainput +  Binput;//33=21h=>0001(P0=1,OP1=1)ADDU
        //CT0=(P0 OR P3) AND OP1=1,CT1=!P2 OR !OP1=[1],CT2=(P1 AND OP1)OR OP0=0
        3'b100:ALU_output_mux = Ainput ^ Binput; //38=26H=10 0110(P1,P2=1,OP1=1)XOR
        //CT0=(P0 OR P3) AND OP1=0,CT1=!P2 OR !OP1=[0],CT2=(P1 AND OP1)OR OP0=1
        3'b101:ALU_output_mux = ~(Ainput | Binput);//39=27H=10 0111(P0-P2=1 OP1=1)NOR
        //CT0=(P0 OR P3) AND OP1=1,CT1=!P2 OR !OP1=[0],CT2=(P1 AND OP1)OR OP0=1
        3'b110:ALU_output_mux = Ainput -  Binput;//34=22H=>0010(P1=1,OP1=1)SUB,BEQ
        //CT0=(P0 OR P3) AND OP1=0,CT1=!P2 OR !OP1=[1],CT2=(P1 AND OP1)OR OP0(BEQ)=1
        3'b111:ALU_output_mux = Ainput -  Binput;//42=2AH=>1010(p3,p1,op1) SLT
        //CT0=(P0 OR P3) AND OP1=1,CT1=!P2 OR !OP1=[1],CT2=(P1 AND OP1)OR OP0=1 [SLT]
        default:ALU_output_mux = 32'h00000000;
    endcase
```

（4）完成移位运算

```
always @* begin  // 6种移位指令
   if(Sftmd)
    case(Sftm[2:0])
        3'b000:Sinput = Binput << Shamt;                    //Sll rd,rt,shamt  00000
        3'b010:Sinput = Binput >> Shamt;                    //Srl rd,rt,shamt  00010
        3'b100:Sinput = Binput << Ainput;                   //Sllv rd,rt,rs 000100
        3'b110:Sinput = Binput >> Ainput;                   //Srlv rd,rt,rs 000110
        3'b011:Sinput = $signed(Binput) >>> Shamt;          //Sra rd,rt,shamt 00011
        3'b111:Sinput = $signed(Binput) >>> Ainput;         //Srav rd,rt,rs 00111
        default:Sinput = Binput;
    endcase
   else Sinput = Binput;
end
```

（5）完成比较转移的 PC 值计算

```
assign Zero = (ALU_output_mux[31:0]== 32'h00000000) ? 1'b1 : 1'b0;
assign Branch_Add = PC_plus_4[31:2] + Sign_extend[31:0];
assign Add_Result = Branch_Add[31:0];   //算出的下一个PC值已经做了除4处理，所以不需左移16位
```

（6）完成最终运算结果的输出

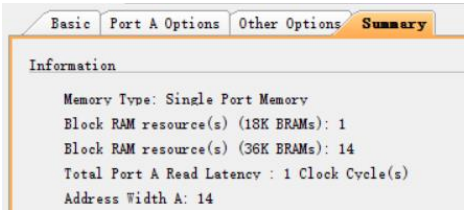
```
always @* begin
    if(((ALU_ctl==3'b111) && (Exe_code[3]==1))||((ALU_ctl[2:1]==2'b11) && (I_format==1)))
    //slti(sub)  处理所有SLT类的问题
        ALU_Result = {30'b000000000000000000000000000000,ALU_output_mux[31]};
        //D31=1 IS neg (LESS)
    else if((ALU_ctl==3'b101) && (I_format==1)) ALU_Result[31:0] =
                    {Binput[15:0],16'b0000000000000000};//lui data
    else if(Sftmd==1) ALU_Result = Sinput;
    else  ALU_Result = ALU_output_mux[31:0]; //otherwise
end
```

## 5. *存储单元的设计

### （1）存储单元的输入和输出

```
output[31:0] read_data;  // 从存储器中获得的数据
input[31:0] address;     //来自memorio模块，源头是来自执行单元算出的alu_result
input[31:0] write_data;  //来自译码单元的read_data2
input  Memwrite;         //来自控制单元
input  clock;
```

### （2）定义 RAM




```
//分配128KB RAM，编译器实际只用 64KB RAM
ram ram (
    .clka(clk),             // input wire clka
    .wea(Memwrite),         // input wire [0 : 0] wea
    .addra(address[15:2]),  // input wire [13 : 0] addra
    .dina(write_data),      // input wire [31 : 0] dina
    .douta(read_data)       // output wire [31 : 0] douta
);
```

## 6. memorio 模块的设计

该模块用于完成地址译码、数据转换和输入数据选择。

### （1） memorio 模块的输入和输出

```
input[31:0] caddress;      // from alu_result in executs32
input memread;             // read memory, from control32
input memwrite;            // write memory, from control32
input ioread;              // read IO, from control32
input iowrite;             // write IO, from control32
input[31:0] mread_data;    // data from memory
input[15:0] ioread_data;   // data from io,16 bits
input[31:0] wdata;         // the data from idecode32,that want to write memory or io
output[31:0] rdata;        // data from memory or IO that want to read into register
output[31:0] write_data;   // data to memory or I/O
output[31:0] address;      // address to mAddress and I/O
output LEDCtrl;            // LED CS
output SwitchCtrl;         // Switch CS

reg[31:0] write_data;
wire iorw;
```

### （2） memorio 模块的功能实现

```
assign  address = caddress;
assign  rdata = (memread==1) ? mread_data : {16'h00,ioread_data[15:0]};
assign  iorw = (iowrite||ioread);

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
assign  LEDCtrl = ((iorw==1) && (caddress[31:4] == 28'hFFFFFC6)) ? 1'b1:1'b0;
assign  SwitchCtrl = ((iorw==1) && (caddress[31:4] == 28'hFFFFFC7)) ? 1'b1:1'b0;

always @* begin
    if((memwrite==1)||(iowrite==1)) begin
        write_data = wdata;
    end else begin
        write_data = 32'hZZZZZZZZ;
    end
end
```

## 7. 多路选择器，LED 灯和 24 位拨码开关的设计

### （1）多路选择器

```
input reset;            // 复位信号
input ior;              //  从控制器来的I/O读，
input switchctrl;       //  从memorio经过地址高端线获得的拨码开关模块片选
input[15:0] ioread_data_switch;  //从外设来的读数据，此处来自拨码开关
output[15:0] ioread_data;   // 将外设来的数据送给memorio

reg[15:0] ioread_data;

always @* begin
    if(reset == 1)
        ioread_data = 16'b0000000000000000;
    else if(ior == 1) begin
        if(switchctrl == 1)
            ioread_data = ioread_data_switch;
        else   ioread_data = ioread_data;
    end
end
```

## （2）LED 灯

```
input led_clk;              // 时钟信号
input ledrst;               // 复位信号
input ledwrite;            // 写信号
input ledcs;              // 从memorio来的，由低至高位形成的LED片选信号
input[1:0] ledaddr;         //  到LED模块的地址低端  !!!!!!!!!!!!!!!!!!
input[15:0] ledwdata;     //  写到LED模块的数据，注意数据线只有16根
output[23:0] ledout;    //  向板子上输出的24位LED信号

reg [23:0] ledout;

always@(posedge led_clk or posedge ledrst) begin
    if(ledrst) begin
        ledout <= 24'h000000;
    end
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    else if(ledcs && ledwrite) begin
        if(ledaddr == 2'b00)
            ledout[23:0] <= { ledout[23:16], ledwdata[15:0] }
        else if(ledaddr == 2'b10 )
            ledout[23:0] <= { ledwdata[7:0], ledout[15:0] };
        else
            ledout <= ledout;
    end
    else begin
        ledout <= ledout;
    end
end
```

## （3）24 位拨码开关

```
input switclk;                 //  时钟信号
input switrst;                 //  复位信号
input switchcs;               //从memorio来的，由低至高位形成的switch片选信号  !
input[1:0] switchaddr;          //  到switch模块的地址低端  !!!!!!!!!!!!!!!
input switchread;               //  读信号
output [15:0] switchrdata;       //  送到CPU的拨码开关值注意数据总线只有16根
input [23:0] switch_i;          //  从板上读的24位开关数据

reg [23:0] switchrdata;
always@(negedge switclk or posedge switrst) begin
    if(switrst) begin
        switchrdata <= 0;
    end
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    else if(switchcs && switchread) begin
        if(switchaddr==2'b00)
            switchrdata[15:0] <= switch_i[15:0];   // data output,lower 16 bits
        else if(switchaddr==2'b10)
            switchrdata[15:0] <= { 8'h00, switch_i[23:16] }; //data output, uppe
        else
            switchrdata <= switchrdata;
    end
    else begin
        switchrdata <= switchrdata;
    end
end
```

# 四、具体设计与调试具体方案

本实验的设计的流程为：

1. 创建 xc7a100tfgg484-1 项目
2. 导入源文件，仿真文件，引脚约束文件
3. 根据提示完成控制器代码的编写并进行仿真
4. 添加时钟的 ip 核并进行仿真
5. 定义 ROM 指令存储器
6. 根据提示完成取指单元代码的编写并进行仿真
7. 定义 RAM 指令存储器并进行仿真
8. 更改控制器模块并添加 IO 功能
9. 设计顶部文件，将控制、取指、译码、执行、存储器、led 灯、拨码开关等模块进行连接
10. 通过 Minisys1Av2.2 汇编器对汇编指令转化为机器指令，并将其导入到 ROM 和 RAM 中，进行顶部文件的仿真
11. 用 vivado 对项目进行合成、实施、引脚分配、转化比特流文件最后下载到实验板上，进行测试

调试方案：对源文件进行仿真，将内部的变量添加到波形图中进行显示，找出错误的地方并分析改正

# 五、设计的主要特色（独自完成部分）

## 1. 添加了数码管的 IO 控制

（1） 在 memorio 中添加了数码管的控制指令

```
assign  LeddataCtrl = ((iorw==1) && (
                        caddress[31:4] == 28'hFFFFFD0
                      ||caddress[31:4] == 28'hFFFFFD1
                      ||caddress[31:4] == 28'hFFFFFD2
                      ||caddress[31:4] == 28'hFFFFFD3
                      ||caddress[31:4] == 28'hFFFFFD4
                      ||caddress[31:4] == 28'hFFFFFD5
                      ||caddress[31:4] == 28'hFFFFFD6
                      ||caddress[31:4] == 28'hFFFFFD7)) ? 1'b1:1'b0;
```

（2） 在顶层文件中添加了部分的线路

```
memorio memio(
    .caddress(alu_result),
    .address(address),
    .memread(memread),
    .memwrite(memwrite),
    .ioread(ioread),
    .iowrite(iowrite),
    .mread_data(read_data),
    .ioread_data(ioread_data),
    .wdata(read_data_2),
    .rdata(rdata),
    .write_data(write_data),
    .LEDCtrl(ledctrl),
    .SwitchCtrl(switchctrl),
    .LeddataCtrl(LeddataCtrl)
);
```

```
digital_tube digital_tube5(
    .tube_clk(clock),
    .tuberst(!prst),
    .tubewrite(LeddataCtrl),
    .tubeaddr(address[7:0]),
    .tubedata_in(write_data[31:0]),
    .tubedata_out(tube_data),
    .tubeaddr_out(tube_addr)
);
```

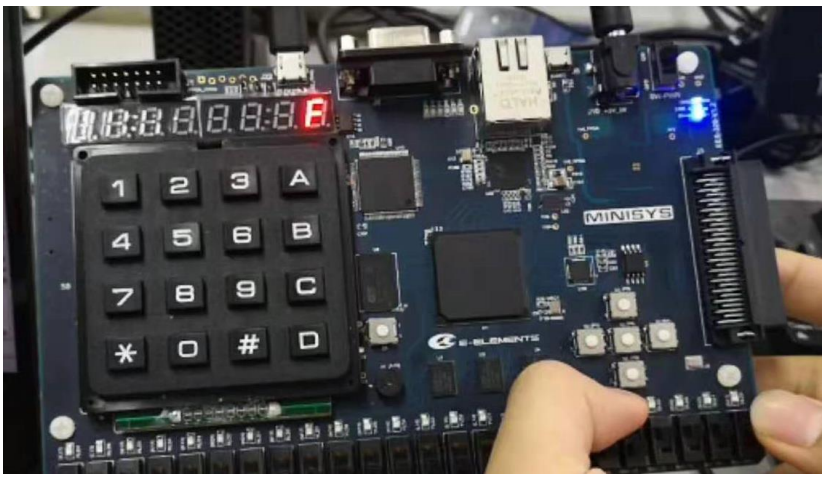
（3） 添加了数码管模块

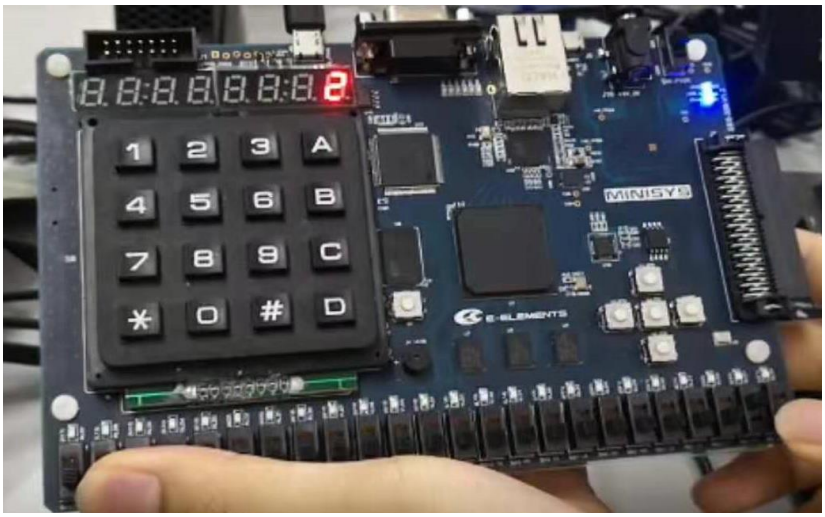
```
module digital_tube(tube_clk, tuberst, tubewrite, tubeaddr,tubedata_in,tubedata_out,tubeaddr_out);
    input tube_clk;             // 时钟信号
    input tuberst;              // 复位信号
    input tubewrite;            // 写信号
    input[7:0] tubeaddr;        // 从memorio来的，由低至高位形成片选信号
    input[31:0] tubedata_in;        //从register里来的写入的数据

    output[7:0] tubedata_out;   //传出的数据
    output[7:0] tubeaddr_out;   //传出数据的地址

    reg [7:0] tubedata_out;
    reg [3:0] data;
    reg [7:0] tubeaddr_out;

    always@(posedge tube_clk or posedge tuberst) begin
        if(tuberst || tubewrite==0) begin
            tubedata_out <= 8'hFF;
        end
        else if(tubewrite) begin
            case(tubeaddr[7:0])
                8'h00: tubeaddr_out[7:0] = 8'b11111110;
                8'h10: tubeaddr_out[7:0] = 8'b11111101;
                8'h20: tubeaddr_out[7:0] = 8'b11111011;
                8'h30: tubeaddr_out[7:0] = 8'b11110111;
                8'h40: tubeaddr_out[7:0] = 8'b11101111;
                8'h50: tubeaddr_out[7:0] = 8'b11011111;
                8'h60: tubeaddr_out[7:0] = 8'b10111111;
                8'h70: tubeaddr_out[7:0] = 8'b01111111;
            endcase
            case(tubeaddr[7:0])
                8'h00: data[3:0] = tubedata_in[3:0];
                8'h10: data[3:0] = tubedata_in[7:4];
                8'h20: data[3:0] = tubedata_in[11:8];
                8'h30: data[3:0] = tubedata_in[15:12];
                8'h40: data[3:0] = tubedata_in[19:16];
                8'h50: data[3:0] = tubedata_in[23:20];
                8'h60: data[3:0] = tubedata_in[27:24];
                8'h70: data[3:0] = tubedata_in[31:28];
            endcase
            case(data[3:0])
                4'b0000: tubedata_out[7:0] = 8'hC0;
                4'b0001: tubedata_out[7:0] = 8'hF9;
                4'b0010: tubedata_out[7:0] = 8'hA4;
                4'b0011: tubedata_out[7:0] = 8'hB0;
                4'b0100: tubedata_out[7:0] = 8'h99;
                4'b0101: tubedata_out[7:0] = 8'h92;
                4'b0110: tubedata_out[7:0] = 8'h82;
                4'b0111: tubedata_out[7:0] = 8'hF8;
                4'b1000: tubedata_out[7:0] = 8'h80;
                4'b1001: tubedata_out[7:0] = 8'h90;
                4'b1010: tubedata_out[7:0] = 8'h88;
                4'b1011: tubedata_out[7:0] = 8'h83;
                4'b1100: tubedata_out[7:0] = 8'hC6;
                4'b1101: tubedata_out[7:0] = 8'hA1;
                4'b1110: tubedata_out[7:0] = 8'h86;
                4'b1111: tubedata_out[7:0] = 8'h8E;
            endcase
        end
        else begin
            tubedata_out <= tubedata_out;
            data <= data;
        end
    end

endmodule
```

（4） 添加引脚约束文件（部分截图）

```
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {tube_addr[0]}]
set_property PACKAGE_PIN A18 [get_ports {tube_addr[7]}]
set_property PACKAGE_PIN A20 [get_ports {tube_addr[6]}]
set_property PACKAGE_PIN B20 [get_ports {tube_addr[5]}]
```

（5） 图片展示




（6）产生的问题及错误，详见：六、调试过程中的 7

# 六、调试过程

调试过程出现的问题如下

1. 取指单元 Add_result 为 ALU 计算结果返回的值，在设计该模块的时候我一直让

next_PC=next_PC+Add_result 导致仿真结果出错。

2. 取值单元跳转是因为取指令后跳转到的指令后 26 位所表示的地址，而不是 Read_data_1，而指令需要预先导入到 ROM 中才能仿真成功。
3. 译码单元中寄存器赋初值的时候为 register[i] <= i，而在看到这里的时候我把 i 改成了 0，让寄存器初值都为 0，在仿真的时候没有问题，但是下载到实验板上后就会出现问题，因为 lw $1,0XC70($28)这条汇编代码需要把一号寄存器（初值为 1）乘（$2^{16}$-1），而如果没有赋初值则结果为 0，导致无法选中拨码开关和 LED 灯。
4. 顶部文件中 Idecode32 单元中的 read_data，需要读入的输入为 rdata，即为 memorio 中的输出的数据而不是 memory 中输出的 read_data，因为 memory 中输出的 read_data 要和拨码的结果在 memorio 中进行汇总并转化为统一的 rdata 输入到寄存器中。
5. 第四道课后题的汇编代码中的 lw $1,0X0C70($28)应该改为 lw $1,0X0C72($28)，因为要从高 8 位读入数据，如果是 70 则选择的地址为低 16 位。同理下一行的 srl $2,$1,13，位移 13 位应该改为 5 位，恰好能读出的是 SW23~SW21 的数据并与初值进行比较并跳转。
6. 加载 ROM 的时候，突然就显示红色叹号，无法加载也无法继续，重启计算机后也不行。最后由于不明原因第二天就恢复正常了。
7. LED 的引脚，代码没有问题的情况下，执行汇编指令的时候 LW 加载到寄存器中总是显示 xxxx，于是我改成了使用 ori 进行寄存器加载。汇编指令中每次进行 LED 的信号变化都维持 100 个周期，但是最后显示到 LED 数码管上的时候，每次对其中一个数码管的值进行修改都会影响所有的数码管显示，此问题尚未解决，原因不明，所以最后的截图中只显示了一个数码管。

# 七、本次设计的总结

本实验中，在既定框架中完成对于 32 位单周期 CPU 的设计，LED 数码管的设计，调试以及汇编指令的编写的过程中，让我们学习到了 vivado 的基本使用，设计 CPU 的流程和理念并对 CPU 有了更加系统和深刻的理解，汇编指令的分析和编写。