Lab05实验报告

类 MIPS 单周期处理器的设计与实现

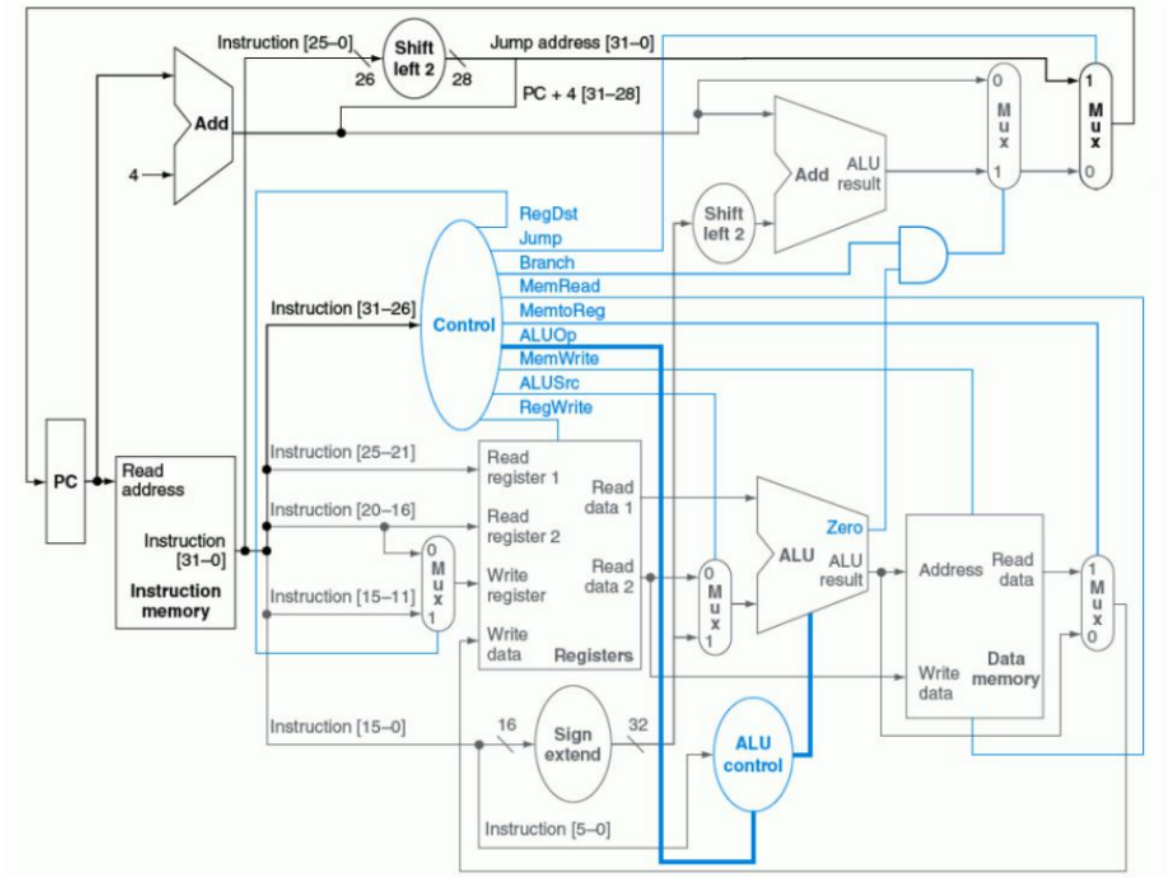
518021910489 陈沛宇

一、实验目的

1. 完成单周期的类 MIPS 处理器。
2. 设计支持 16 条 MIPS 指令（add, sub, and, or, addi, andi, ori, slt,lw, sw, beq, j, jal, jr, sll, srl,）的单周期 CPU。

二、实现步骤

1.处理器模块分析



如图所示，顶层模块需要完成**取指-译码-寄存器文件访存-ALU运算-内存访存-写回**等模块的连接，为了完成要求的16条指令需要添加的功能模块有：进行数据选择的5位和32位多路复用器模块；对立即数进行0扩展的模块；计算下一条PC的模块；取指模块。同时，为了实现这16条指令，还需要对原来在lab3和lab4中实现的模块进行部分修改。下面给出增加的指令在MIPS ISA中的格式：

操作符	31..26	25..21	20..16	15..11	10..6	5..0	示例	示例含义	操作及其解释
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt；shamt存放移位的位数，也就是指令中的立即数，其中rt=$2, rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ;(logical)，其中rt=$2, rd=$1

操作符	31..26	25..21	20..16	15..0	示例	示例含义	操作及其解释
addi	001000	rs	rt	immediate	addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ；其中rt=$1,rs=$2
andi	001100	rs	rt	immediate	andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ；其中rt=$1,rs=$2
ori	001101	rs	rt	immediate	andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ；其中rt=$1,rs=$2

操作符	31..26	25..0	示例	示例含义	操作及其解释
j	000010	address	j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0 ；address=10000/4
jal	000011	address	jal 10000	$31<-PC+4; goto 10000	$31<-PC+4；PC <- (PC+4)[31..28],address,0,0 ；address=10000/4

2.添加模块

5位/32位多路复用器模块Mux5/Mux32

多路复用器根据select信号从两个数据中选择一个输出，以5位多路复用器为例，实现方式为使用三目运算符便捷地进行选择，代码如下：（32位只需要更改数据宽度，不在此赘述）

```
module Mux5(
    input SEL,
    input [4:0] INPUT1,
    input [4:0] INPUT2,
    output [4:0] OUT
    );

    assign OUT=SEL?INPUT1:INPUT2;

endmodule
```

0扩展模块zeroext

实现0扩展模块的主要原因是，在`ori`和`andi`两条指令的实现中，需要对指令中的立即数进行0扩展后计算。该模块的实现较为简单，直接对传输给模块的立即数进行`or`操作即可，实现代码如下：

```
module zeroext(
    input [15:0] inst,
    output [31:0] data
    );
    assign data=inst | 32'h00000000;

endmodule
```

程序计数器PC

程序计数器PC也选择了使用一个单独的模块来实现而非在顶层模块文件中实现，这样可使模块的结构更清晰。观察单周期MIPS处理器示意图可知，实现PC的计算的方法为：计算PC+4；计算无条件跳转目标PC；计算条件跳转目标PC；根据ALU运算结果、译码产生的`branch`信号和`jump`信号共同决定选择上述计算出三个结果中的哪一个。该模块实现代码如下：

```
module PC(
       input zero,
       input jump,
       input branch,
       input [31:0]instruction,
       input [31:0] PC,
       output [31:0] PC_4,
       output [31:0] next_PC,
       output [31:0] jumpAddress,
       output [31:0] branchAddress,
       output [31:0] branchResult,
       output [31:0] imm
    );
        assign PC_4 = PC+4;
        assign jumpAddress ={PC_4[31:28],instruction[25:0]<<2};
        assign branchAddress = PC_4+(imm<<2);
        assign branchResult = (branch & zero)?branchAddress:PC_4;
        assign next_PC = (jump)?jumpAddress:branchResult;
endmodule
```

需要十分注意的一点是`+`和`<<`的优先级，`+`的优先级是大于`<<`的，写计算表达式时要考虑优先级防止计算出错误的跳转地址。

取指模块InstMemory

在MIPS处理器中，一条指令长度为32位，占据连续的四个字节，并采用小端序存储模式，实现时只需要注意数据宽度和数据顺序即可，实现代码如下：

```
module InstMemory(
    input [31:0] address,
    output [31:0] instruction
    );

    reg [7:0] insMem[63:0];
    assign instruction=
{insMem[address+3],insMem[address+2],insMem[address+1],insMem[address]};

endmodule
```

3.修改模块

Datapath的修改

1. `sll`与`srl`指令需要读入指令中的`shamt`部分，将其接入ALU模块。
2. `jal`指令需要将将PC+4接入dataMemory模块后的Mux32模块，将`$31`接入Register模块前的Mux32模块。
3. `jr`指令需要将Register模块的`ReadData1`接入负责跳转的Mux32模块。

指令译码模块Ctr

为Ctr模块新增了一个是否选用0扩展的信号，用于`andi`和`ori`指令的实现，共为`opCode`增加了三个`case`，用去区分`addi`、`andi`和`ori`指令，`sll`与`srl`指令为R-type，沿用原来的代码即可，修改部分实现代码如下：

```
6'b001000://addi
        begin
            regDst=0;
            aluSrc=1;
            memToReg=0;
            regWrite=1;
            memRead=0;
            memWrite=0;
            branch=0;
            aluOp=2'b00;
            jump=0;
            zeroext=0;
        end
        6'b001100://andi
        begin
            regDst=0;
            aluSrc=1;
            memToReg=0;
            regWrite=1;
            memRead=0;
            memWrite=0;
            branch=0;
            aluOp=2'b10;
            jump=0;
            zeroext=1;
        end
        6'b001101://ori
        begin
            regDst=0;
            aluSrc=1;
            memToReg=0;
            regWrite=1;
            memRead=0;
            memWrite=0;
            branch=0;
            aluOp=2'b11;
            jump=0;
            zeroext=1;
        end
```

ALUCtr模块

由于指令数目变多，ALUCtr需要区分的指令也更多了，且`sll`和`srl`指令比较特殊，其指令的`rs`段只能为`00000`，故将原来的判断语句的判断条件从8位扩展到13位，并为需要新增ALU处理的指令分配独立的`ALUCtr`信号，不需要新增的与原有处理使用同样的`ALUCtr`信号。需要注意判断条件的书写顺序，这一点已在之前的实验报告中提到，该模块的修改部分实现代码如下：

```
assign ctr_word = {aluOp,rs,Funct};
        casex(ctr_word)
            13'b00xxxxxxxxxxx:aluCtrOut = 4'b0010;//lw,sw,addi
            13'b1000000000000:aluCtrOut = 4'b0011;//sll
            13'b1000000000010:aluCtrOut = 4'b0100;//srl
```

```
            13'b10xxxxx100000:aluCtrOut = 4'b0010;//add
            13'b10xxxxx100010:aluCtrOut = 4'b0110;//sub
            13'b10xxxxx100100:aluCtrOut = 4'b0000;//and
            13'b10xxxxx100101:aluCtrOut = 4'b0001;//or
            13'b10xxxxx101010:aluCtrOut = 4'b0111;//slt
            13'b10xxxxxxxxxxx:aluCtrOut = 4'b0000;//andi
            13'b11xxxxxxxxxxx:aluCtrOut = 4'b0001;//ori
            13'bx1xxxxxxxxxxx:aluCtrOut = 4'b0110;//branch
        default:aluCtrOut = 4'b0001;
        endcase
```

ALU模块

ALU模块新增的操作主要为shift操作，该部分较为简单，只需要正确选择输入ALU的数据并计算即可，修改部分实现代码如下：

```
4'b0011: // sll
                begin
                    aluRes = input2 << input3;
                    if (aluRes == 0)
                        zero = 1;
                end
                4'b0100: // srl
                begin
                    aluRes = (input2 >> input3);
                    if (aluRes == 0)
                        zero = 1;
                end
```

寄存器文件模块Register

为该模块新增了reset功能，在上升时钟沿若收到`reset`信号则将所有寄存器文件置0，修改部分实现代码如下：

```
always @(posedge Clk)
    begin
        if(reset)
            begin
                for (regFileIndex=0;regFileIndex<32;regFileIndex=regFileIndex+1)
                    regFile[regFileIndex] <=0;      //时序逻辑<=
            end
    end
```

需注意时序逻辑赋值方式。

内存文件dataMemory

修改内存文件为按字节寻址，MIPS处理器内存的可寻址单元大小为8bit，将原来的32bit拆分成4个8bit即可，修改部分代码如下：

```
always @(address)
        begin
            if(memRead && !memWrite)
                readData={memFile[address+3],
memFile[address+2],memFile[address+1], memFile[address]};
            else
```

```
                readData=0;
        end
    always @(negedge Clk)//时序逻辑
        begin
            if(memWrite)
                begin
                    memFile[address] <= writeData[7:0];
                    memFile[address+1] <= writeData[15:8];
                    memFile[address+2] <= writeData[23:16];
                    memFile[address+3] <= writeData[31:24];
                end
        end
```

4.顶层模块

顶层模块是一个将分散的模块组合起来的模块，它的作用是通过实例化各个模块完成各个模块之间的连接，实现数据通路，使数据能够在各个模块间传输。步骤在下面详细介绍。

取指

实例化`InstMemory`模块，读入PC，输出PC所对应的 32 位指令：

```
reg [31:0] PC;
    wire [31:0] INST;

    InstMemory instMemory_init(
                            .address(PC),
                            .instruction(INST)
                            );
```

控制器译码

实例化ctr模块，读入指令的`opCode`，输出控制信号：

```
wire REG_DST,JUMP,BRANCH,MEM_READ,MEM_WRITE,ALU_SRC,REG_WRITE,ZERO_EXT;
    wire [1:0] ALU_OP;
    ctr ctr_init(
                .opCode(INST[31:26]),
                .regDst(REG_DST),
                .jump(JUMP),
                .branch(BRANCH),
                .memRead(MEM_READ),
                .memToReg(MEM_TO_REG),
                .aluOp(ALU_OP),
                .memWrite(MEM_WRITE),
                .aluSrc(ALU_SRC),
                .regWrite(REG_WRITE),
                .zeroext(ZERO_EXT)
                );
```

寄存器文件

实例化5位多路复用器Mux5模块，选择写入的目标寄存器：

```
wire [4:0] WRITE_REG;
    Mux5 mux5_init(
                .SEL(REG_DST),
                .INPUT1(INST[15:11]),
                .INPUT2(INST[20:16]),
                .OUT(WRITE_REG)
                );
```

实例化Register模块，进行寄存器文件的读写操作：

```
wire [31:0] READ_DATA1;
    wire [31:0] READ_DATA2;
    wire [31:0] WRITE_DATA;
    Register reg_init(
                .Clk(Clk),
                .readReg1(INST[25:21]),
                .readReg2(INST[20:16]),
                .regWrite(REG_WRITE),
                .writeReg(WRITE_REG),
                .writeData(WRITE_DATA),
                .readData1(READ_DATA1),
                .readData2(READ_DATA2),
                .reset(reset)
                );
```

ALU

实例化ALUCtr模块，生成指定ALU操作的控制字：

```
wire [3:0] ALU_CTR;
    wire [12:0] CTR_WORD;
    aluctr aluctr_init(
                    .Funct(INST[5:0]),
                    .aluOp(ALU_OP),
                    .rs(INST[25:21]),
                    .aluCtrOut(ALU_CTR),
                    .ctr_word(CTR_WORD)
                    );
```

实例化0扩展和符号扩展两个模块，并实例化32位多路复用器生成符合当前指令要求的立即数：

```
wire [31:0]IMMEDIATE_NUM_1;
    zeroext zeroext_init(
                        .inst(INST[15:0]),
                        .data(IMMEDIATE_NUM_1)
                        );
    wire [31:0] IMMEDIATE_NUM_2;
    signext signext_init(
                        .inst(INST[15:0]),
                        .data(IMMEDIATE_NUM_2)
                        );
    wire [31:0] IMMEDIATE_NUM;
    Mux32 mux32_imm_init(
                        .SEL(ZERO_EXT),
                        .INPUT1(IMMEDIATE_NUM_1),
```

```
                        .INPUT2(IMMEDIATE_NUM_2),
                        .OUT(IMMEDIATE_NUM)
                        );
```

实例化另一个32位多路复用器，选择ALU的第二个输入数据（第一个为从寄存器文件读取的数据）：

```
wire [31:0] ALU_SRC_B;
    Mux32 mux32_alu_init(
                    .SEL(ALU_SRC),
                    .INPUT1(IMMEDIATE_NUM),
                    .INPUT2(READ_DATA2),
                    .OUT(ALU_SRC_B)
                    );
```

实例化ALU模块，根据前面产生的信号决定做什么操作，完成运算：

```
wire ZERO;
    wire [31:0] ALU_RESULT;
    ALU alu_init(
                .input1(READ_DATA1),
                .input2(ALU_SRC_B),
                .input3(INST[10:6]),
                .aluCtr(ALU_CTR),
                .zero(ZERO),
                .aluRes(ALU_RESULT)
                );
```

内存文件

实例化dataMemory模块，完成内存文件的独写：

```
 wire [31:0] READ_DATA;   //lw
    dataMemory dataMemory_init(
                            .Clk(Clk),
                            .memWrite(MEM_WRITE),
                            .address(ALU_RESULT),
                            .writeData(READ_DATA2),
                            .memRead(MEM_READ),
                            .readData(READ_DATA)
                            );
```

实例化另一个32位多路复用器模块，选择是将ALU运算结果写回寄存器还是将从内存读取到的数据写回寄存器，或是不需要写回寄存器：

```
Mux32 mux32_memToreg_init(
                            .SEL(MEM_TO_REG),
                            .INPUT1(READ_DATA),
                            .INPUT2(ALU_RESULT),
                            .OUT(WRITE_DATA)//在寄存器部分声明
                            );
```

程序计数器

实例化PC模块，计算并选择下一条带执行指令的PC：

```
wire [31:0] PC_4;
     wire [31:0] NEXT_PC;
     wire [31:0] JAddr;
     wire [31:0] BAddr;
     wire [31:0] BResult;
     PC pc_init(
                .zero(ZERO),
                .jump(JUMP),
                .branch(BRANCH),
                .instruction(INST),
                .PC(PC),
                .PC_4(PC_4),
                .next_PC(NEXT_PC),
                .jumpAddress(JAddr),
                .branchAddress(BAddr),
                .branchResult(BResult),
                .imm(IMMEDIATE_NUM)
                );
```

根据reset信号，在上升时钟沿更新PC的值：

```
 always @ (posedge Clk)
    begin
        if (reset)
            PC <= 0;
        else
            PC <= NEXT_PC;
    end
```

三、仿真测试

测试程序

编写了如下汇编程序检测处理器是否正常工作，其中与`add`指令类似的`sub`指令，与`j`指令类似的`jr`和`jal`不在此赘述。

```
lw $1,0($0)          //$1=1
addi $2,2($0)        //$2=2
add $3,$1,$2         //$3=3
andi $5,$1,0001H     //$5=0001H
ori $6,$1,8001H      //$6=8001H
sll $5,3($1)         //$5=8
addi $2,6($2)        //$2=8
srl $6,2($2)         //$6=2
sw $3,8($0)          //memFile[2]=3
j Set4               //jump
Inc4:
addi $3,4($3)        //$3+=4
Set4:
slt $4,$1,$2         //$4=1
beq $1.$4,Inc4       //branch taken
```

中间应该有的结果数据已在汇编代码中注释，将上述指令翻译为机器指令如下，需注意使用小端序：

```
00000000 00000000 00000001 10001100
00000010 00000000 00000010 00100000
00100000 00011000 00100010 00000000
00000001 00000000 00100101 00110000
00000001 10000000 00100110 00110100
11000000 00101000 00000001 00000000
00000110 00000000 01000010 00100000
10000010 00110000 00000010 00000000
00001000 00000000 00000011 10101100
00001011 00000000 00000000 00001000
00000100 00000000 01100011 00100000
00101010 00100000 00100010 00000000
11111101 11111111 00100100 00010000
```

其中内存文件中的数据为：

```
00000001 00000000 00000000 00000000
00000010 00000000 00000000 00000000
00000001 00000000 00000000 00000000
```

激励文件

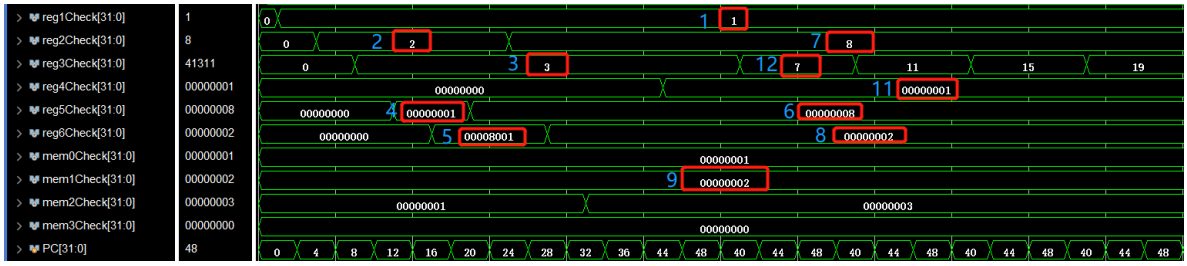
编写测试激励文件，初始化用于检测的寄存器文件和内存文件，使用`$readmemb`读取绝对路径下的数据文件data.txt和指令文件inst.txt。注意不要错误的使用`$readmemb`和`$readmemh`，前者为读取二进制后者为读取十六进制，应根据自己的文件格式合理选择。最后初始化`Clk`信号和`reset`信号。激励文件代码如下：

```
module Top_tb(

    );
    reg Clk,reset;
    always #50 Clk=!Clk;
    Top top_init(
                .Clk(Clk),
                .reset(reset)
                );
            wire [31:0] reg1Check;
            assign reg1Check = top_init.reg_init.regFile[1][31:0];
            wire [31:0] reg2Check;
            assign reg2Check = top_init.reg_init.regFile[2][31:0];
            wire [31:0] reg3Check;
            assign reg3Check = top_init.reg_init.regFile[3][31:0];
            wire [31:0] reg4Check;
            assign reg4Check = top_init.reg_init.regFile[4][31:0];
            wire [31:0] reg5Check;
            assign reg5Check = top_init.reg_init.regFile[5][31:0];
            wire [31:0] reg6Check;
            assign reg6Check = top_init.reg_init.regFile[6][31:0];
            wire [31:0] mem0Check;
            assign mem0Check=
{top_init.dataMemory_init.memFile[3],top_init.dataMemory_init.memFile[2],top_ini
t.dataMemory_init.memFile[1],top_init.dataMemory_init.memFile[0]};
            wire [31:0] mem1Check;
```

```
            assign mem1Check=
{top_init.dataMemory_init.memFile[7],top_init.dataMemory_init.memFile[6],top_ini
t.dataMemory_init.memFile[5],top_init.dataMemory_init.memFile[4]};
            wire [31:0] mem2Check;
            assign mem2Check=
{top_init.dataMemory_init.memFile[11],top_init.dataMemory_init.memFile[10],top_i
nit.dataMemory_init.memFile[9],top_init.dataMemory_init.memFile[8]};
            wire [31:0] mem3Check;
            assign mem3Check=
{top_init.dataMemory_init.memFile[15],top_init.dataMemory_init.memFile[14],top_i
nit.dataMemory_init.memFile[13],top_init.dataMemory_init.memFile[12]};
    initial begin
        $readmemb("C:/Archlabs/lab05/data.txt",
top_init.dataMemory_init.memFile);
        $readmemb("C:/Archlabs/lab05/inst.txt",
top_init.instMemory_init.insMem);

    Clk = 1;
    reset = 1;
    #25
    reset = 0;
    #2000;
end
endmodule
```

请在测试时更换绿色字体中的绝对路径，否则将找不到所需的两个文件。

仿真波形



已按执行顺序框选结果并标注步骤，可以看出仿真测试结果正确。

四、总结与心得体会

1.总结

本次实验是对前几次实验的一个融合，从一个更高的角度去组织各个功能模块，需要我们对前两次实验的功能模块烂熟于心，我也在本次实验中意识到前两次实验我的理解还是不够深，不过所幸本次实验让我加深了对他们的理解。

我的设计仍然存在不小的缺憾，如使用了较多的位数用于控制信号的传递等，这些理应都是可以优化的，在实际生产中这些都和成本挂钩，但由于时间问题没有再钻研更好的解决方法，希望未来有机会完善设计。

在此总结一下从遇到的花费了我较多时间debug的问题中学到的需要注意的事项：

1. 增加信号并修改判断条件时一定记得修改数据宽度，否则会判断出错。
2. 注意 `$readmemh` 和 `$readmemb` 的区别。
3. 注意各种数据的数据宽度。

2.心得体会

本次实验花费时间远远超过我的预期，但带给我的收获也是非常多的。

首先讲一讲我最大的收获：要注意命名规范，特别是在类似此次实验的代码编写中，时间跨度长，文件数量多，若不形成一套自己能够看懂的命名规范，在写代码时将会花费大量时间查询变量的意思，是很降低效率的。

其次，在进行模块化设计的时候也要提前想好要实现的功能，否则就会不断的修改以前的文件，添加各种信号等，到最后特别容易把脑子搞混。在较大规模的设计时一定要先统筹规划而不能做一步看一步。

本次实验让我明白我对较大规模的代码编写的掌控能力还是不足，常常不知道自己下一步应该干什么，要改进这一点只能靠训练，也希望假期开始后我能找到一些项目锻炼自己的能力。