组成原理实验课程第三次实验报告

实验名称：寄存器堆实现　班级：李涛老师 学生姓名：陆皓喆 学号：2211044

指导老师：董前琨　实验地点：实验楼A306 实验时间：2024.04.18

一、实验目的

1.熟悉并掌握$MIPS$计算机中寄存器堆的原理和设计方法。

2.初步了解$MIPS$指令结构和源操作数/目的操作数的概念。

3.熟悉并运用$verilog$语言进行电路设计。

4.为后续设计cpu的实验打下基础。

二、实验内容说明

请结合实验指导手册中的实验三（寄存器堆实验）完成对寄存器堆进行64位位拓展的改进实验，注意以下几点：

1、原始的32个32位寄存器堆，需要修改成16个64位的寄存器堆，注意地址和位宽变化。

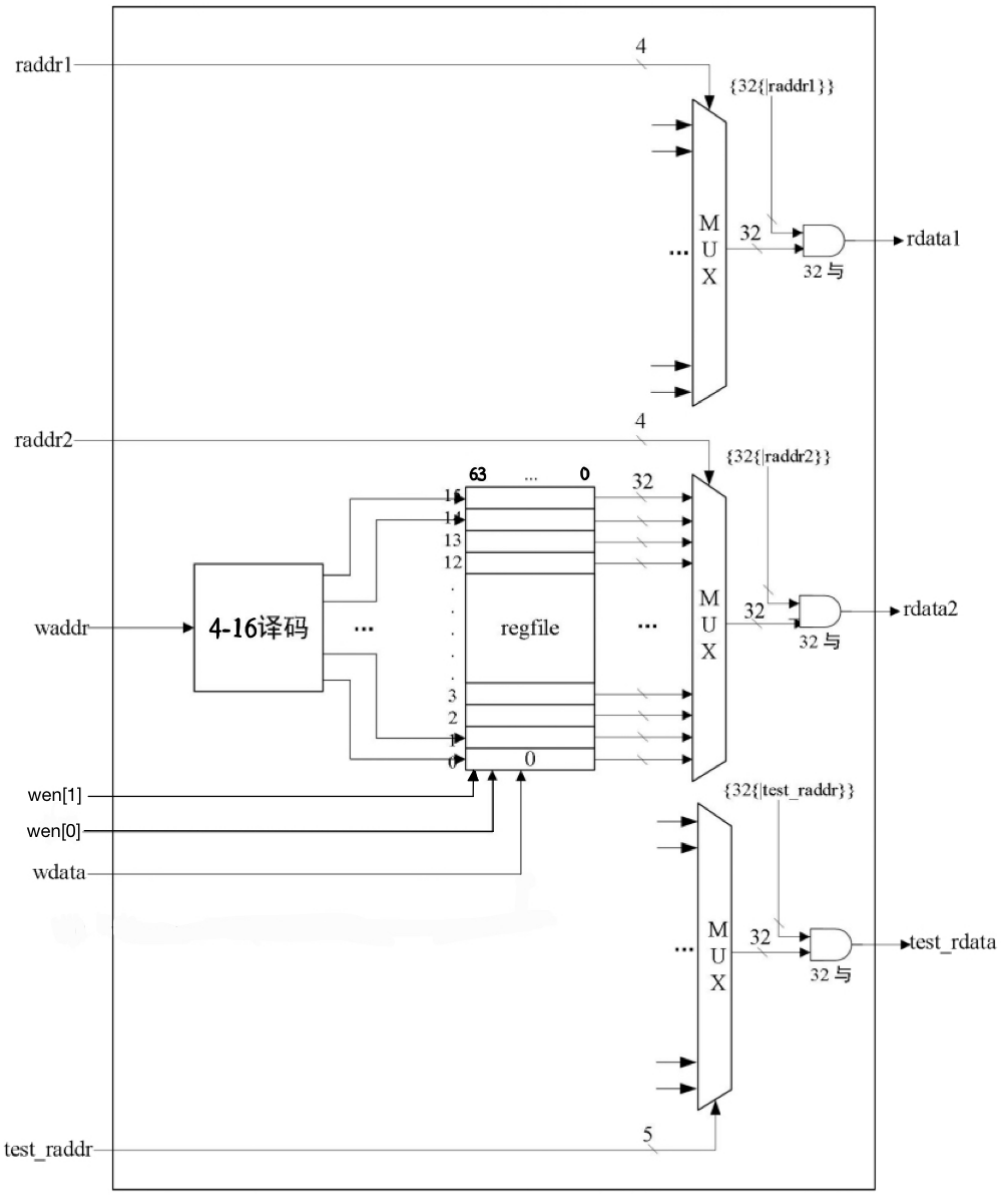
2、在$display$模块，注意读出数据和写入的数据都应是64位，lcd屏上的格子需要调整分配。此外，`input_sel`等相关信号注意位宽是否调整。

3、本次实验没有仿真，直接上试验箱验证，实验报告中注意对实验结果的介绍，分析和总结需要详细说明。

三、实验原理图

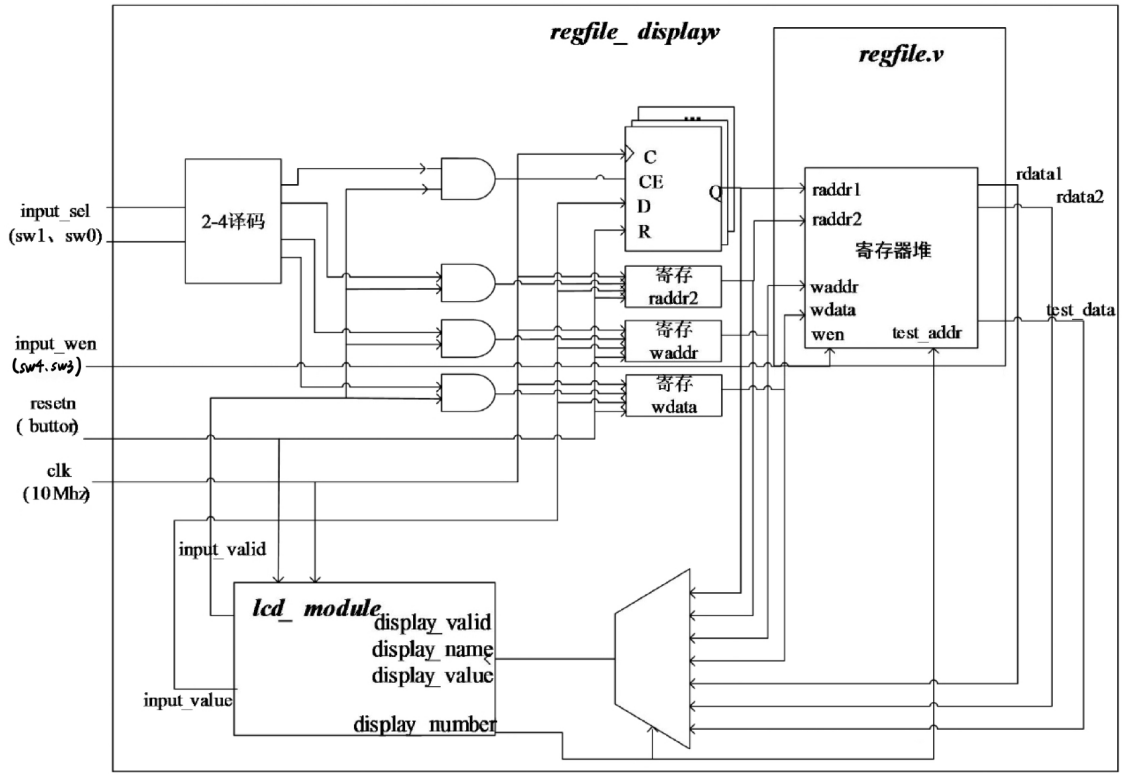
3.1 底层原理图

具体修改内容：将`raddr1`等输入的位数进行了修改；对5-32译码修改为了4-16译码，并将寄存器的位数进行了对应的修改，从32-32来到了64-16。输入的`wen`的位数也从1来到了2。



3.2 顶层原理图

具体修改内容：将输入的`wen`的位数修改为两位，并且对应的xdc约束文件也做一定的修改。



3.3 具体解释

相比于原来的实验原理图，我们做了以下优化。我们将原来的**5-32译码**更换成了**4-16译码器**，这样我们就可以通过输入四位二进制数来判定修改哪个寄存器，另外，我们扩展了原先的一位二进制输入`wen`，扩展为两位二进制输入的`wen[1]`，用来确定输入的是寄存器的高16位还是低16位，如果输入的是`wen`的第一位，那么修改高位；反之如果输入的是`wen`的第二位，那么我们就修改低位。另外，对`radder2`等数组的输入位数也做了一定的调整。我们使用前四个按钮：后两个按钮置1，前两个按钮在输入的情况下，表示输入到哪一个寄存器的高位还是低位：

> - 如果输入10，代表写入后面的数的前四位；
> - 如果输入11，代表写入第二个数的后四位；
> - 如果输入00，代表写入第一个数的前四位；
> - 如果输入01，代表输入第一个数的后四位。

当改变前两个按钮的时候就可以实现寄存器的读取与赋值。

在顶层模块图中，我们通过修改输入的`wen`的位数，设置两个按钮`sw4`和`sw3`，实现了二位二进制输入4种不同情况。

> - 当`input_sel`为2'b00时，表示输入数为读地址1，即`raddr1`；
> - 当`input_sel`为2'b01时，表示输入数为读地址2，即`raddr2`；
> - 当`input_sel`为2'b10时，表示输入数为写地址，即`waddr`；
> - 当`input_sel`为2'b11时，表示输入数为写数据，即`wdata`。

四、实验步骤

4.1 修改regfile.v文件

4.1.1 具体修改内容

首先，修改对应的寄存器位数，将原来的`wen`**修改为2位**，将原来的`raddr1`**和**`raddr2`**和**`waddr`**修改为4位**，将`wdata`、`rdata1`、`rdata2`**修改为64位**，并且调整寄存器的**位宽与个数**，**从32-32调整到64-16。**

```
    input             clk,
    input      [1 :0] wen,
    input      [3 :0] raddr1,
    input      [3 :0] raddr2,
    input      [3 :0] waddr,
    input      [63:0] wdata,
    output reg [63:0] rdata1,
    output reg [63:0] rdata2,
    input      [3 :0] test_addr,
    output reg [63:0] test_data
    );
    reg [63:0] rf[15:0];
```

然后我们展示部分逻辑的修改：我们通过对`wen`的限制，实现了**4种不同的拨码对应不同的四种输入**，下面的情况是，如果`wen`输入10，那么我们就修改`waddr`中的低16位。

```
    always @(posedge clk)
    begin
        if (wen == 2'd3)
        begin
            rf[waddr][15:0] <= wdata[15:0];
        end
    end
```

然后在后面，我们只需要将对应的寄存器数量从**32改为16**就可以了，由于代码过于繁杂，所以不在此处展示，可以参考后面的修改后的源代码。

4.1.2 修改完的源代码

```
module regfile(
    input             clk,
    input      [1 :0] wen,
    input      [3 :0] raddr1,
    input      [3 :0] raddr2,
    input      [3 :0] waddr,
    input      [63:0] wdata,
    output reg [63:0] rdata1,
    output reg [63:0] rdata2,
    input      [3 :0] test_addr,
    output reg [63:0] test_data
    );
```

```
    reg [63:0] rf[15:0];

    // three ported register file
    // read two ports combinationally
    // write third port on rising edge of clock
    // register 0 hardwired to 0

    always @(posedge clk)
    begin
        if (wen == 2'd3)
        begin
            rf[waddr][15:0] <= wdata[15:0];
        end
    end

    always @(posedge clk)
    begin
        if (wen == 2'd2)
        begin
            rf[waddr][31:16] <= wdata[31:16];
        end
    end

    always @(posedge clk)
    begin
        if (wen == 2'd1)
        begin
            rf[waddr][47:32] <= wdata[47:32];
        end
    end

    always @(posedge clk)
    begin
        if (wen == 2'd0)
        begin
            rf[waddr][63:48] <= wdata[63:48];
        end
    end
    //读端口1
    always @(*)
    begin
        case (raddr1)
            4'd1 : rdata1 <= rf[1 ];
            4'd2 : rdata1 <= rf[2 ];
            4'd3 : rdata1 <= rf[3 ];
            4'd4 : rdata1 <= rf[4 ];
            4'd5 : rdata1 <= rf[5 ];
            4'd6 : rdata1 <= rf[6 ];
            4'd7 : rdata1 <= rf[7 ];
            4'd8 : rdata1 <= rf[8 ];
            4'd9 : rdata1 <= rf[9 ];
            4'd10: rdata1 <= rf[10];
            4'd11: rdata1 <= rf[11];
            4'd12: rdata1 <= rf[12];
            4'd13: rdata1 <= rf[13];
            4'd14: rdata1 <= rf[14];
```

```
            4'd15: rdata1 <= rf[15];
            default : rdata1 <= 64'd0;
        endcase
    end
    //读端口2
    always @(*)
    begin
        case (raddr2)
            4'd1 : rdata2 <= rf[1 ];
            4'd2 : rdata2 <= rf[2 ];
            4'd3 : rdata2 <= rf[3 ];
            4'd4 : rdata2 <= rf[4 ];
            4'd5 : rdata2 <= rf[5 ];
            4'd6 : rdata2 <= rf[6 ];
            4'd7 : rdata2 <= rf[7 ];
            4'd8 : rdata2 <= rf[8 ];
            4'd9 : rdata2 <= rf[9 ];
            4'd10: rdata2 <= rf[10];
            4'd11: rdata2 <= rf[11];
            4'd12: rdata2 <= rf[12];
            4'd13: rdata2 <= rf[13];
            4'd14: rdata2 <= rf[14];
            4'd15: rdata2 <= rf[15];
            default : rdata2 <= 64'd0;
        endcase
    end
     //调试端口，读出寄存器值显示在触摸屏上
    always @(*)
    begin
        case (test_addr)
            4'd1 : test_data <= rf[1 ];
            4'd2 : test_data <= rf[2 ];
            4'd3 : test_data <= rf[3 ];
            4'd4 : test_data <= rf[4 ];
            4'd5 : test_data <= rf[5 ];
            4'd6 : test_data <= rf[6 ];
            4'd7 : test_data <= rf[7 ];
            4'd8 : test_data <= rf[8 ];
            4'd9 : test_data <= rf[9 ];
            4'd10: test_data <= rf[10];
            4'd11: test_data <= rf[11];
            4'd12: test_data <= rf[12];
            4'd13: test_data <= rf[13];
            4'd14: test_data <= rf[14];
            4'd15: test_data <= rf[15];
            default : test_data <= 64'd0;
        endcase
    end
endmodule
```

4.2 修改regfile_display.v文件

4.2.1 具体修改内容

我们修改了输入的拨码开关的数量，将其修改为了各两个输入。

```
    input [1:0] wen,
    input [1:0] input_sel
```

然后我们添加了对应输入时的led灯的亮灭情况，并且将灯的数量扩展到了4个，其他都不变。后面还是跟上面那个文件一样，对应的做位数上的修改，其他地方都不变。

下面的逻辑部分我们做了修改，我们举部分例子进行说明。

```
    //当input_sel为2'b00时，表示输入数为读地址1，即raddr1
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            raddr1 <= 4'd0;
        end
        else if (input_valid &&  input_sel==2'd0)
        begin
            raddr1 <= input_value[3:0];
        end
    end
```

这部分完成了逻辑的判断，我们判断输入的`sel`的值，如果是00的话，我们就表示输入读地址1，并且输入4位有效地址，下面的部分也是一样的，完成了对于输入的具体判断与响应。

后面我们也完成了对于$display$具体输出位置的重新编写，将输出的位置做了一些变化。

4.2.2 修改完的源代码

```
module regfile_display(
    //时钟与复位信号
    input clk,
    input resetn,    //后缀"n"代表低电平有效

    //拨码开关，用于产生写使能和选择输入数
    input [1:0] wen,
    input [1:0] input_sel,

    //led灯，用于指示写使能信号，和正在输入什么数据
    output led_wen1,
    output led_wen2,
    output led_wen3,
    output led_wen4,
    output led_waddr,    //指示输入写地址
    output led_wdata,    //指示输入写数据
    output led_raddr1,   //指示输入读地址1
    output led_raddr2,   //指示输入读地址2
```

```
    //触摸屏相关接口，不需要更改
    output lcd_rst,
    output lcd_cs,
    output lcd_rs,
    output lcd_wr,
    output lcd_rd,
    inout[15:0] lcd_data_io,
    output lcd_bl_ctr,
    inout ct_int,
    inout ct_sda,
    output ct_scl,
    output ct_rstn
);
//-----{LED显示}begin
    assign led_wen1   = (wen==2'd0);
    assign led_wen2   = (wen==2'd1);
    assign led_wen3   = (wen==2'd2);
    assign led_wen4   = (wen==2'd3);
    assign led_raddr1 = (input_sel==2'd0);
    assign led_raddr2 = (input_sel==2'd1);
    assign led_waddr  = (input_sel==2'd2);
    assign led_wdata  = (input_sel==2'd3);
//-----{LED显示}end

//-----{调用寄存器堆模块}begin
    //寄存器堆多增加一个读端口，用于在触摸屏上显示32个寄存器值
    wire [63:0] test_data;
    wire [3 :0] test_addr;
    reg  [3 :0] raddr1;
    reg  [3 :0] raddr2;
    reg  [3 :0] waddr;
    reg  [63:0] wdata;
    wire [63:0] rdata1;
    wire [63:0] rdata2;
    regfile rf_module(
        .clk   (clk   ),
        .wen   (wen   ),
        .raddr1(raddr1),
        .raddr2(raddr2),
        .waddr (waddr ),
        .wdata (wdata ),
        .rdata1(rdata1),
        .rdata2(rdata2),
        .test_addr(test_addr),
        .test_data(test_data)
    );
//-----{调用寄存器堆模块}end

//--------------------{调用触摸屏模块}begin--------------------//
//-----{实例化触摸屏}begin
//此小节不需要更改
    reg         display_valid;
    reg  [39:0] display_name;
    reg  [31:0] display_value;
    wire [5 :0] display_number;
```

```
    wire        input_valid;
    wire [31:0] input_value;

    lcd_module lcd_module(
        .clk            (clk           ),   //10Mhz
        .resetn         (resetn        ),

        //调用触摸屏的接口
        .display_valid  (display_valid ),
        .display_name   (display_name  ),
        .display_value  (display_value ),
        .display_number (display_number),
        .input_valid    (input_valid   ),
        .input_value    (input_value   ),

        //lcd触摸屏相关接口，不需要更改
        .lcd_rst        (lcd_rst       ),
        .lcd_cs         (lcd_cs        ),
        .lcd_rs         (lcd_rs        ),
        .lcd_wr         (lcd_wr        ),
        .lcd_rd         (lcd_rd        ),
        .lcd_data_io    (lcd_data_io   ),
        .lcd_bl_ctr     (lcd_bl_ctr    ),
        .ct_int         (ct_int        ),
        .ct_sda         (ct_sda        ),
        .ct_scl         (ct_scl        ),
        .ct_rstn        (ct_rstn       )
    );
//-----{实例化触摸屏}end

//-----{从触摸屏获取输入}begin
//根据实际需要输入的数修改此小节，
//建议对每一个数的输入，编写单独一个always块
    //32个寄存器显示在7~38号的显示块，故读地址为（display_number-1）
    assign test_addr = (display_number-5'd13)/2;
    //当input_sel为2'b00时，表示输入数为读地址1，即raddr1
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            raddr1 <= 4'd0;
        end
        else if (input_valid &&  input_sel==2'd0)
        begin
            raddr1 <= input_value[3:0];
        end
    end

    //当input_sel为2'b01时，表示输入数为读地址2，即raddr2
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            raddr2 <= 4'd0;
        end
        else if (input_valid && input_sel==2'd1)
```

```
        begin
            raddr2 <= input_value[3:0];
        end
    end

    //当input_sel为2'b10时，表示输入数为写地址，即waddr
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            waddr  <= 4'd0;
        end
        else if (input_valid && input_sel==2'd2)
        begin
            waddr  <= input_value[3:0];
        end
    end

    //当input_sel为2'b11时，表示输入数为写数据，即wdata
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            wdata  <= 63'd0;
        end

        else if (input_valid && input_sel==2'd3)
        begin
            if(wen==2'd0)
            begin
                wdata[63:48]  <= input_value[15:0];
            end
            if(wen==2'd1)
            begin
                wdata[47:32]  <= input_value[15:0];
            end
            if(wen==2'd2)
            begin
                wdata[31:16]  <= input_value[15:0];
            end
            if(wen==2'd3)
            begin
                wdata[15:0]  <= input_value[15:0];
            end
        end
    end
//-----{从触摸屏获取输入}end

//-----{输出到触摸屏显示}begin
//根据需要显示的数修改此小节，
//触摸屏上共有44块显示区域，可显示44组32位数据
//44块显示区域从1开始编号，编号为1~44，
    always @(posedge clk)
    begin
        if (display_number >6'd12 && display_number <6'd45)
        begin //块号7~38显示32个通用寄存器的值
```

```
            if(display_number%2==1)
            begin
                display_valid <= 1'b1;
                display_name[39:16] <= "REG";
                display_name[15: 8] <= {4'b0011,4'b0000};
                display_name[7 : 0] <= {4'b0011,test_addr[3:0]};
                display_value       <= test_data[63:32];
             end

            if (display_number%2==0)
            begin //块号7~38显示32个通用寄存器的值
                display_valid <= 1'b1;
                display_name[39:16] <= "REG";
                display_name[15: 8] <= {4'b0011,4'b0000};
                display_name[7 : 0] <= {4'b0011,test_addr[3:0]};
                display_value       <= test_data[31:0];
              end
        end

        else
        begin
            case(display_number)
                6'd1 : //显示读端口1的地址
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RADD1";
                    display_value <= raddr1;
                end
                6'd3 : //显示读端口1读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDAT1";
                    display_value <= rdata1[63:32];
                end
                6'd4 : //显示读端口1读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDAT1";
                    display_value <= rdata1[31:0];
                end
                6'd5 : //显示读端口2的地址
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RADD2";
                    display_value <= raddr2;
                end
                6'd7 : //显示读端口2读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDAT2";
                    display_value <= rdata2[63:32];
                end
                6'd8 : //显示读端口2读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDAT2";
```

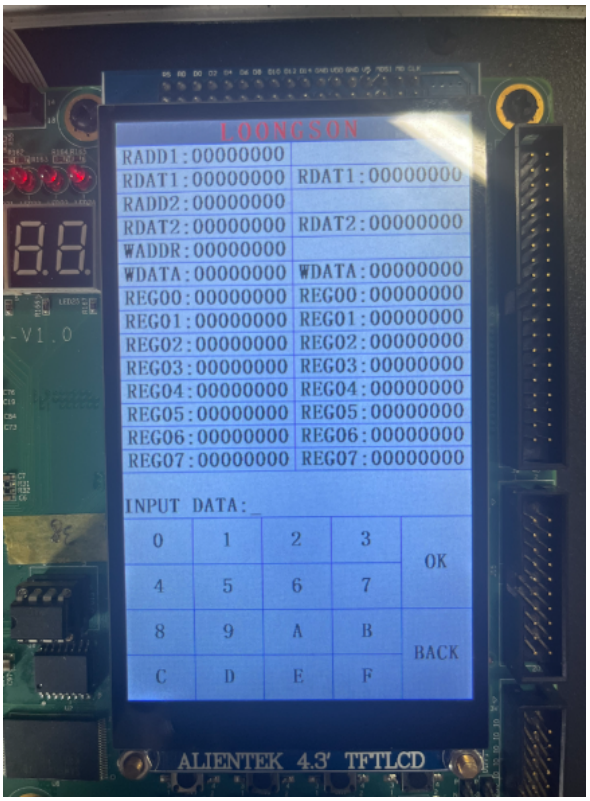
```
                    display_value <= rdata2[31:0];
                end
                6'd9 : //显示写端口的地址
                begin
                    display_valid <= 1'b1;
                    display_name  <= "WADDR";
                    display_value <= waddr;
                end
                6'd11 : //显示写端口写入的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "WDATA";
                    display_value <= wdata[63:32];
                end
                6'd12 : //显示写端口写入的数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "WDATA";
                    display_value <= wdata[31:0];
                end
                default :
                begin
                    display_valid <= 1'b0;
                    display_name  <= 40'd0;
                    display_value <= 32'd0;
                end
            endcase
        end
    end

endmodule
```

五、实验结果分析

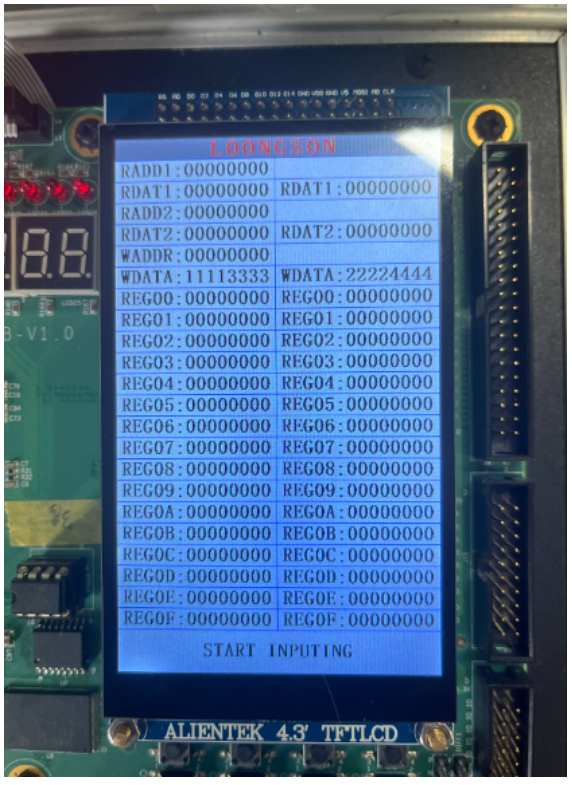
5.1 上箱验证

由于本次实验没有要求实现仿真模拟，因此我们直接上实验箱进行验证。

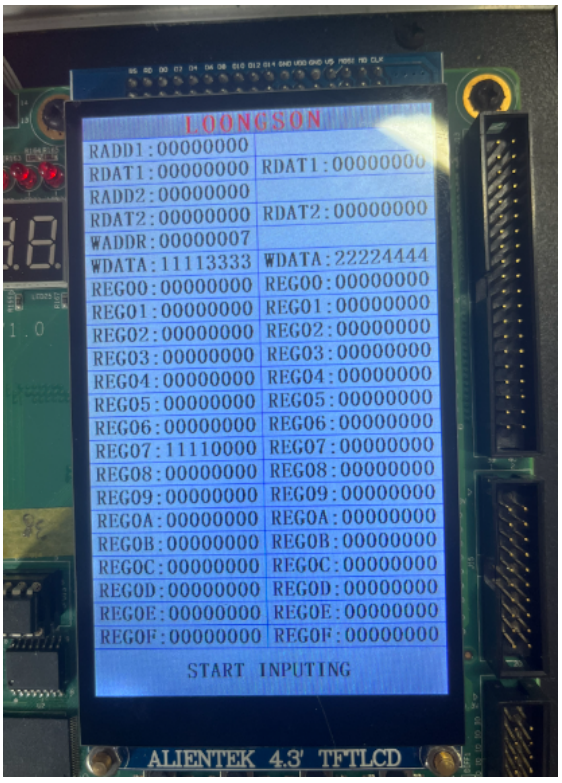
首先启动实验箱，初始化面板。



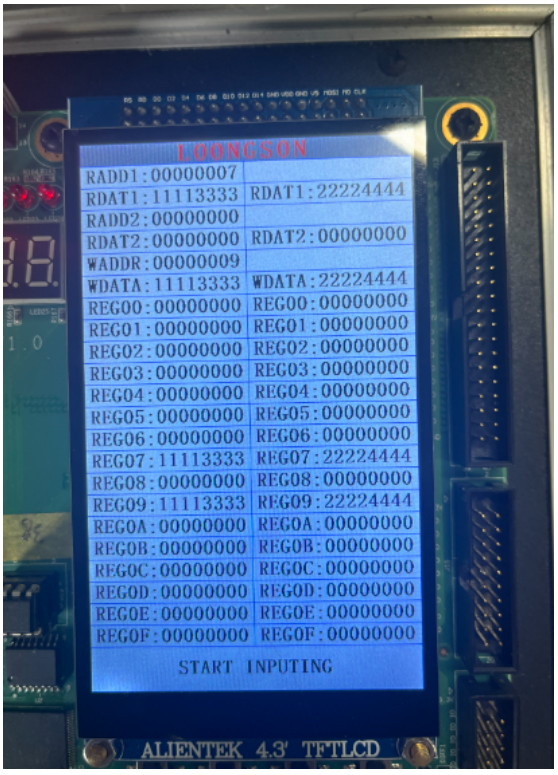
然后我们先写入数据，使用拨码开关，调整到对应的输入模式，写入数据，分别使用四种输入模式，输入两个数的高低位即可。我们在写数据寄存器中输入1111333322224444，注意该数字是由四次输入组成的。



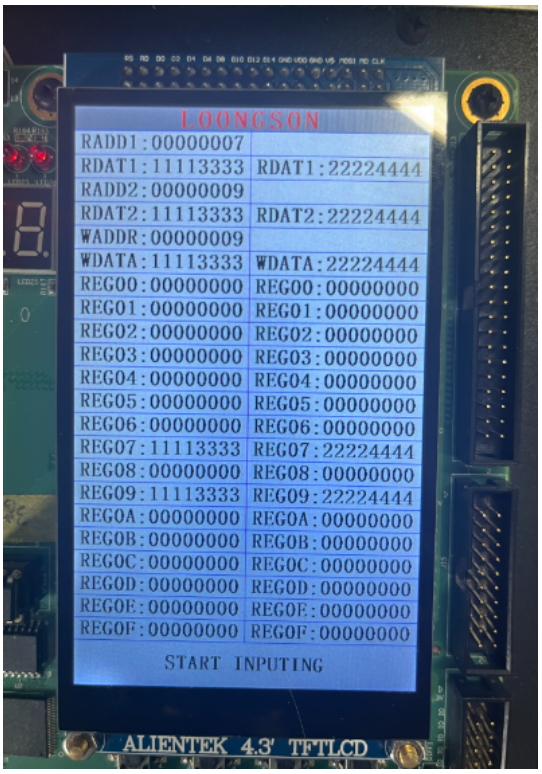
然后，我们调整到写地址的功能，写入7，这样我们就完成了往 `REG07` 中存储数据的任务。可以发现，`REG07` 中确实存储进了写数据寄存器中的数据。



上图展示了我们往寄存器7中写入第一个数的高位的过程，同理，我们完成剩下的位数的输入，再选择写入地址9，再将写入的数据存入寄存器9中，再选择读寄存器1功能，得到以下的结果：



然后我们完成后面的读取，验证完毕，程序无误。



六、总结感想

在本次的实验中，我学到了对于顶层模块以及底层模块的修改与调整，进一步掌握了寄存器的存储与读取原理，对于verilog语言的理解也更深了。

在本次实验中，遇到了不少问题，比如说对于位宽的调整失误导致代码一直报错、对按钮的对应约束不熟悉导致在复现时也出现了不少问题。经过一周的奋战，最终解决了这个问题，成功完成了实验。

我也收获了一些经验，比如说在修改源代码的时候，对于位宽的设置以及其必要性也更加了解了，对于verilog中的各类代码的逻辑与规范也有了更深的理解。希望在后续实验中，我也可以顺利的完成！