组成原理实验课程第_三_次实报告

实验名称	寄存器堆 64 位扩展实验			班级	李涛班
学生姓名	田晋宇	学号	2212039	指导老师	董前琨
实验地点	实验楼 A306		实验时间	2024.04.25	

1、 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。

2、 实验内容说明

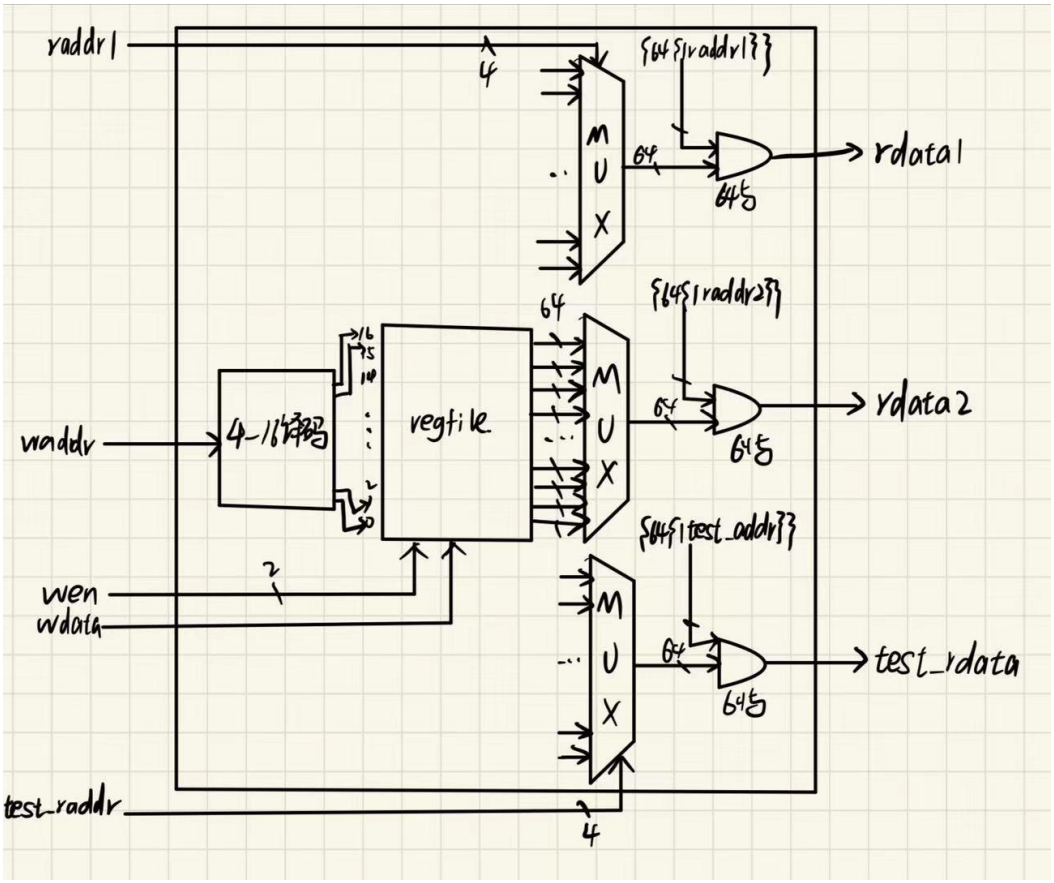
请结合实验指导手册中的实验三（寄存器堆实验）完成对寄存器堆进行 64 位位拓展的改进实验，注意以下几点：

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。

3、 实验原理图



4、 实验步骤

1. 对寄存器堆模块的修改：

本次实验实现的是 16 个 64 位寄存器，因此将所有地址信号的位宽全部改为 4 位，将读写数据的位宽改为 64 位。在对寄存器初始化时也需要修改位宽以及寄存器的个数。

```
8  module regfile(
9      input              clk,//时钟信号
10     input              wen,//写使能信号
11     input       [3 :0] raddr1,//读取端1的地址信号
12     input       [3 :0] raddr2,//读取端2的地址信号
13     input       [3 :0] waddr,//写入端口的地址信号
14     input       [63:0] wdata,//写入数据信号
15     output reg  [63:0] rdata1,//读取端1的读取数据
16     output reg  [63:0] rdata2,//读取端2的读取数据
17     input       [3 :0] test_addr,//测试端口地址信号
18     output reg  [63:0] test_data//测试端口读出的数据
19     );
20     reg [63:0] rf[15:0];//定义了16个位宽为64位的寄存器
```

在时钟信号内对数据的读写时，需要修改对地址的逻辑判断（以读取端口 1 为例）：

```
35     //读端口1
36     always @(*)
37     begin
38         case (raddr1)
39             5'd1 : rdata1 <= rf[1 ];
40             5'd2 : rdata1 <= rf[2 ];
41             5'd3 : rdata1 <= rf[3 ];
42             5'd4 : rdata1 <= rf[4 ];
43             5'd5 : rdata1 <= rf[5 ];
44             5'd6 : rdata1 <= rf[6 ];
45             5'd7 : rdata1 <= rf[7 ];
46             5'd8 : rdata1 <= rf[8 ];
47             5'd9 : rdata1 <= rf[9 ];
48             5'd10: rdata1 <= rf[10];
49             5'd11: rdata1 <= rf[11];
50             5'd12: rdata1 <= rf[12];
51             5'd13: rdata1 <= rf[13];
52             5'd14: rdata1 <= rf[14];
53             5'd15: rdata1 <= rf[15];
54             default : rdata1 <= 64'd0;
55         endcase
56     end
```

对调试端口进行修改，在顶层模块中通过调试端口将寄存器的值显示在液晶屏上，但液晶屏每次只能显示 32 位数据，因此在调试端口中每次读的数据只能是一个寄存器的

高 32 位或者低 32 位，我们将 test_addr 的位宽改为 5 位，以确保最终在液晶屏上能正确显示数据。具体的原因在顶层模块 zhongjinxingjieshi.

```
79      //调试端口，读出寄存器值显示在触摸屏上
80      always @(*)
81      begin
82          case (test_addr)
83              5'd1 : test_data <= rf[1 ][63:32];
84              5'd2 : test_data <= rf[1 ][31:0];
85              5'd3 : test_data <= rf[2 ][63:32];
86              5'd4 : test_data <= rf[2 ][31:0];
87              5'd5 : test_data <= rf[3 ][63:32];
88              5'd6 : test_data <= rf[3 ][31:0];
89              5'd7 : test_data <= rf[4 ][63:32];
90              5'd8 : test_data <= rf[4 ][31:0];
91              5'd9 : test_data <= rf[5 ][63:32];
92              5'd10: test_data <= rf[5 ][31:0];
93              5'd11: test_data <= rf[6 ][63:32];
94              5'd12: test_data <= rf[6][31:0];
95              5'd13: test_data <= rf[7][63:32];
96              5'd14: test_data <= rf[7][31:0];
97              5'd15: test_data <= rf[8][63:32];
98              5'd16 : test_data <= rf[8 ][31:0];
99              5'd17 : test_data <= rf[9 ][63:32];
100             5'd18: test_data <= rf[9 ][31:0];
101             5'd19: test_data <= rf[10][63:32];
102             5'd20: test_data <= rf[10][31:0];

103             5'd21: test_data <= rf[11][63:32];
104             5'd22: test_data <= rf[11][31:0];
105             5'd23: test_data <= rf[12][63:32];
106             5'd24: test_data <= rf[12][31:0];
107             5'd25: test_data <= rf[13][63:32];
108             5'd26: test_data <= rf[13][31:0];
109             5'd27: test_data <= rf[14][63:32];
110             5'd28: test_data <= rf[14][31:0];
111             5'd29: test_data <= rf[15][63:32];
112             5'd30: test_data <= rf[15][31:0];
113             default : test_data <= 64'd0;
114         endcase
115     end
116 endmodule
```

2. 对顶层模块的修改：

首先同样对传入寄存器堆模块参数的位宽进行修改：

```
//————{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 [31:0] 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
```

由于实验箱一次最多能写入或读取 32 位数，故将输入信号 input_sel 的位宽修改为 3：当 input_sel=0 时输入读取端 1 的地址信号；当 input_sel=1 时输入读取端 2 的地址信号；当 input_sel=2 时输入写入端的地址；当 input_sel=3 时输入写入端数据的高 32 位；当 input_sel=4 时输入输入写入端数据的低 32 位：

```
111     always @(posedge clk)
112     begin
113         if (!resetn)
114         begin
115             raddr1 <= 5'd0;
116         end
117         else if (input_valid &&  input_sel==3'd0)
118         begin
119             raddr1 <= input_value[3:0];
120         end
121     end
122
123     //当input_sel为3'b001时，表示输入数为读地址2，即raddr2
124     always @(posedge clk)
125     begin
126         if (!resetn)
127         begin
128             raddr2 <= 5'd0;
129         end
130         else if (input_valid && input_sel==3'd1)
131         begin
132             raddr2 <= input_value[3:0];
133         end
134     end
```

```
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            waddr  <= 5'd0;
        end
        else if (input_valid && input_sel==3'd2)
        begin
            waddr  <= input_value[3:0];
        end
    end

    //当input_sel为3'b011时，表示输入数为写数据H，即wdata[63:32]
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            wdata[63:32]  <= 32'd0;
        end
        else if (input_valid && input_sel==3'd3)
        begin
            wdata[63:32]  <= input_value;
        end
    end

    //当input_sel为3'b100时，表示输入数为写数据H，即wdata[31:0]
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            wdata[31:0]  <= 32'd0;
        end
        else if (input_valid && input_sel==3'd4)
        begin
            wdata[31:0]  <= input_value;
        end
    end
```

接下来我们对实验箱的显示区域进行修改，首先分别显示输入的值：读取端 1 的地址，读取端 2 的地址，读取端 1 的高 32 位，读取端 1 的低 32 位，读取端 2 的高 32 位，读取端 2 的低 32 位，写入端地址，写入端地址，写入端数据的高 32 位，写入端数据的低 32 位，接下来分别显示 16 个通用寄存器的高 32 位和低 32 位，为了显示的整齐美观，我对格子进行如下分配，最终结果在实验结果分析中展示：

```
reg [3:0] temp;
always @(posedge clk)
begin
    if (display_number > 6'd12 && display_number < 6'd45)
    begin //块号13~44显示16个通用寄存器的值
        display_valid <= 1'b1;
        display_name[39:16] <= "REG";
        temp <= test_addr[3:0]; // 保留原始地址值

        if (test_addr % 2 == 0) // 如果是偶数地址，显示高32位
        begin
            display_name[15:8] <= {4'b0011, temp}; // 高32位显示
            display_name[7:0] <= "H";
            display_value <= test_data;
        end
        else // 如果是奇数地址，显示低32位
        begin
            temp <= temp - 1; // 获取对应的偶数地址
            display_name[15:8] <= {4'b0011, temp}; // 低32位显示
            display_name[7:0] <= "L";
            display_value <= test_data;
        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读出的高32位数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDA1H";
                    display_value <= rdata1[63:32];
                end
                6'd4 : //显示读端口1的低32位数据
                begin
                    display_valid <= 1'b1;
                    display_name  <= "RDA1L";
                    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读出的高32位地址
               begin
                   display_valid <= 1'b1;
                   display_name  <= "RDA2H";
                   display_value <= rdata2[63:32];
               end
               6'd8 : //显示读端口2读出的低32位地址
               begin
                   display_valid <= 1'b1;
                   display_name  <= "RDA2L";
                   display_value <= rdata2[31:0];
               end
               6'd9 : //显示写端口写入的地址
               begin
                   display_valid <= 1'b1;
                   display_name  <= "WADDR";
                   display_value <= waddr;
               end
```

```
            6'd11 : //显示写端口写入的高32位数据
            begin
                display_valid <= 1'b1;
                display_name  <= "WDATH";
                display_value <= wdata[63:32];
            end
            6'd12 : //显示写端口写入的低32位数据
            begin
                display_valid <= 1'b1;
                display_name  <= "WDATL";
                display_value <= wdata[31:0];
            end
            default :
            begin
                display_valid <= 1'b0;
                display_name  <= 40'd0;
                display_value <= 32'd0;
            end
        endcase
    end
end
```

在 16 个通用寄存器显示的过程中，我希望每一个寄存器的 64 位数显示在左右相邻的两格里，我们分配第 13-44 格，此处我们对 display_number 进行条件判断，如果 display_number 为奇数，那就将文本显示的寄存器编号设为 temp <= (test_addr[4:0]-1)/2；如果 display_number 为偶数，那就将文本显示的寄存器编号设为 temp <= (test_addr[4:0]-2)/2。因为之前在寄存器堆模块中修改了调试端口，使得在顶层模块中根据 test_addr 的值便能自动读取高 32 位还是低 32 位，因此在顶层模块中无须再对 test_addr 进行条件判断。最终修改后的代码如下：

```
reg [3:0] temp;
always @(posedge clk)
begin
    if (display_number > 6'd12 && display_number < 6'd45)
    begin //块号13~44显示16个通用寄存器的值
        display_valid <= 1'b1;
        display_name[39:16] <= "REG";

        if (display_number % 2 == 1) // 如果是偶数地址，显示高32位
        begin
            temp <= (test_addr[4:0]-1)/2;
            display_name[15:8] <= {4'b0011, temp}; // 高32位显示
            display_name[7:0] <= "H";
            display_value <= test_data;
        end
        else // 如果是奇数地址，显示低32位
        begin
            temp <= test_addr[4:0]/2;// 获取对应的偶数地址
            display_name[15:8] <= {4'b0011, temp}; // 低32位显示
            display_name[7:0] <= "L";
            display_value <= test_data;
        end
    end
```

由于我们从第 13 号格子开始显示通用寄存器的值，因此我们对 test_addr 赋初值为 assign test_addr = display_number-5'd12;。

最后我们在约束文件中绑定 input_sel 对应的引脚，实现最左边的拨码开关为写使能信号，紧接着的三个拨码开关来表示 input_sel 的值：

```
14  set_property PACKAGE_PIN AC21 [get_ports wen]
15  set_property PACKAGE_PIN AD24 [get_ports input_sel[2]]
16  set_property PACKAGE_PIN AC22 [get_ports input_sel[1]]
17  set_property PACKAGE_PIN AC23 [get_ports input_sel[0]]
18
19  set_property IOSTANDARD LVCMOS33 [get_ports clk]
20  set_property IOSTANDARD LVCMOS33 [get_ports resetn]
21  set_property IOSTANDARD LVCMOS33 [get_ports led_wen]
22  set_property IOSTANDARD LVCMOS33 [get_ports led_raddr1]
23  set_property IOSTANDARD LVCMOS33 [get_ports led_raddr2]
24  set_property IOSTANDARD LVCMOS33 [get_ports led_waddr]
25  set_property IOSTANDARD LVCMOS33 [get_ports led_wdata]
26  set_property IOSTANDARD LVCMOS33 [get_ports wen]
27  set_property IOSTANDARD LVCMOS33 [get_ports input_sel[2]]
28  set_property IOSTANDARD LVCMOS33 [get_ports input_sel[1]]
29  set_property IOSTANDARD LVCMOS33 [get_ports input_sel[0]]
```

以上是代码的修改思路。

5、 实验结果分析

输入写入的地址与数据，按下写使能按钮，相应的寄存器成功写入：

输入读取端 1 的地址，按下时钟信号，对应寄存器的数据成功读入到读取端 1 中：

输入读取端 2 的地址，按下时钟信号，对应寄存器的数据成功读入到读取端 2 中：

6、 总结感想

本次实验完成了对原本的 32 位寄存器的扩展修改，进一步加深对寄存器工作原理的理解，本质上是通过时钟信号来控制数据读写的模块。为之后的 CPU 设计打下了良好的基础。在实验的过程中，遇到了很多困难，例如如何使 16 个通用寄存器在液晶显示屏上的显示整齐美观，开始时寄存器无法按顺序排列，且输出的编号包含乱码，原因是自己没注意参数的位宽，导致在读取地址时没有读取完整，产生了错误的现象。在此后 verilog 代码编写的过程中要明确自己的思路和目标，一步一步理清楚参数所需要的位宽，这样就能减少程序错误的发生。