组成原理实验课程第_三_次实报告

实验名称	寄存器堆位拓展			班级	李涛老师
学生姓名	王昱	学号	2212046	指导老师	董前琨
实验地点	A306		实验时间	2024.3.21	

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

1. 做好预习：

1） 掌握寄存器堆的工作原理；

2） 确定寄存器堆的输入输出端口设计；

3） 在课前画好寄存器堆的设计框图或实验原理图；

4） 如果对 FPGA 板了解的话，可确定设计中与 FPGA 板上交互的接口，画出包含外围模块的整体设计框图，即补充完善图 4.1。

2. 实验实施：

1） 确认寄存器堆的设计框图的正确性；

2） 编写 verilog 代码；

3） 对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料；

4） 完成调用寄存器堆模块的外围模块的设计，并编写代码；

5） 对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。

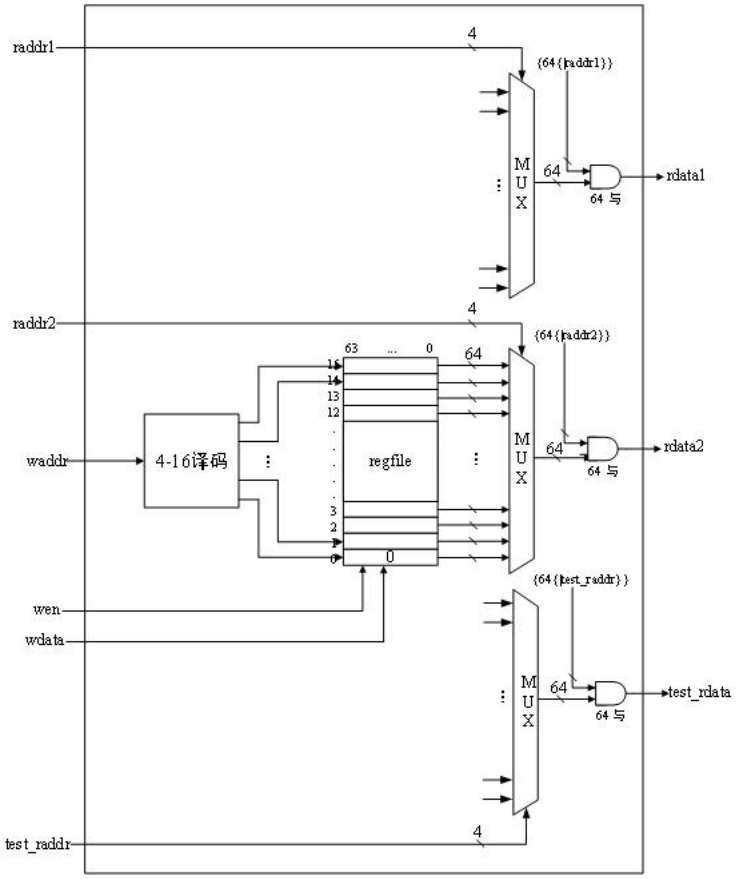
3. 实验检查：

1） 完成上板验证后，让指导老师或助教进行检查，进行现场演示，按照检查人员的要求，对特定寄存器读写，可对演示结果进行拍照作为实验报告结果一项的材料。

4. 实验报告的撰写：

1） 实验结束后，需按照规定的格式完成实验报告的撰写。

3、 实验原理图



实验原理图说明：

将 32 个 32 位寄存器修改位 16 个 64 位寄存器，需要以下修改：

- waddr 是要写入的寄存器号，只需要十六个寄存器，因此用四位即可储存 16 个寄存器号，所以把 5-32 译码器修改为 4-16 译码器；
- raddr1 和 raddr2 表示读取目标寄存器的寄存器号，而修改后只需要四位即可表示所有的寄存器，所以把 raddr1 和 raddr2 的位宽修改为四位，读取的数据改为 64 位；
- wen 是写使能信号，只有当 wen 有效的适合才可以输入数据；
- wdata 是要写入的数据，这里修改为输入 64 位数据。

4、 实验步骤

修改关键代码：

（1）修改 register.v:

```
module regfile(
    input           clk,
    input           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];
```

修改说明：

- 这里将各个寄存器的位宽修改为从 5 修改为 4，用于指示十六位寄存器，输入和输出的数据拓展为六十四位。
- reg [63:0] rf[15:0];定义了一个寄存器数组，其中包含 16 个 64 位宽的寄存器。这些寄存器可以通过上述的读写地址进行访问和修改。
- 整个模块的功能是允许用户通过指定的地址读取或写入数据到寄存器中。当写使能 wen 激活时，可以将数据 wdata 写入到地址 waddr 指定的寄存器。同时，可以随时从地址 raddr1 和 raddr2 指定的寄存器读取数据。这种设计在 CPU 的寄存器组中非常常见，用于快速存取 CPU 内部的状态或临时数据。

```
//读端口 1
  always @(*)
  begin
      case (raddr1)
          5'd1 : rdata1 <= rf[1 ];
          5'd2 : rdata1 <= rf[2 ];
          5'd3 : rdata1 <= rf[3 ];
          5'd4 : rdata1 <= rf[4 ];
          5'd5 : rdata1 <= rf[5 ];
          5'd6 : rdata1 <= rf[6 ];
          5'd7 : rdata1 <= rf[7 ];
          5'd8 : rdata1 <= rf[8 ];
          5'd9 : rdata1 <= rf[9 ];
          5'd10: rdata1 <= rf[10];
          5'd11: rdata1 <= rf[11];
          5'd12: rdata1 <= rf[12];
          5'd13: rdata1 <= rf[13];
          5'd14: rdata1 <= rf[14];
          5'd15: rdata1 <= rf[15];
          default : rdata1 <= 64'd0;
      endcase
  end
  //读端口 2
  always @(*)
  begin
      case (raddr2)
          5'd1 : rdata2 <= rf[1 ];
          5'd2 : rdata2 <= rf[2 ];
          5'd3 : rdata2 <= rf[3 ];
          5'd4 : rdata2 <= rf[4 ];
          5'd5 : rdata2 <= rf[5 ];
          5'd6 : rdata2 <= rf[6 ];
          5'd7 : rdata2 <= rf[7 ];
          5'd8 : rdata2 <= rf[8 ];
```

```
5'd9 : test_data <= rf[9 ];
                5'd10: test_data <= rf[10];
                5'd11: test_data <= rf[11];
                5'd12: test_data <= rf[12];
                5'd13: test_data <= rf[13];
                5'd14: test_data <= rf[14];
                5'd15: test_data <= rf[15];
                default : test_data <= 64'd0;
            endcase
    end
endmodule
```

修改说明：

- 这里修改读端口 1、读端口 2 和调试端口，把寄存器改为 16 个。
- 读端口(rdata): 这部分代码使用一个 always 块和 case 语句来根据读地址 raddr1 的值，从寄存器数组 rf 中选择相应的寄存器数据输出到 rdata。如果 raddr 的值不在 1 到 15 之间，则 rdata 输出为 0。
- 整个模块的作用是允许通过不同的地址读取寄存器的值。

（2）修改 register.display

```
    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 wen,
    input [2:0] input_sel,
```

修改说明：

- input clk：这是一个时钟信号输入，通常用于同步电路中的操作。
- input resetn：这是一个复位信号输入，后缀“n”表示这是一个低电平有效的信号。input wen：当 wen 为高电平时，允许数据写入。
- input [2:0] input_sel：这是一个 3 位宽的输入选择信号，用于根据不同的值选择不同的操作或数据输入。这里把 input_sel 的位宽改为三位，用来指示不同的操作，输入数据高三十二位和低三十二位的不同需求。

```
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)
    );
```

```
assign test_addr = display_number-5'd10;
always @(posedge clk)
    begin
        if (!resetn)
        begin
            raddr1 <= 4'd0;
        end
        else if (input_valid &&    input_sel==3'd0)
        begin
            raddr1 <= input_value[3:0];
        end
    end

    always @(posedge clk)
    begin
        if (!resetn)
        begin
            raddr2 <= 4'd0;
        end
        else if (input_valid && input_sel==3'd1)
        begin
            raddr2 <= input_value[3:0];
        end
    end

    always @(posedge clk)
    begin
```

```
if (!resetn)
        begin
            waddr   <= 4'd0;
        end
        else if (input_valid && input_sel==3'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高 32 位，即 wdata
    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
```

修改说明：

- assign test_addr = display_number-5'd10;：这行代码将 display_number 减去 10（5 位二进制数），并将结果赋值给 test_addr。把显示块的个数由 6 个变为 9 个，用于显示寄存器的显示块改为从 10 开始。
- 读地址 1 (raddr1): 当 input_sel 为 3'd0（即二进制 00）时，表示当前的输入值（input_value）是读地址 1。如果复位信号为低电平，则 raddr1 被设置为 0。如果复位信号不激活且输入有效，则 input_value 的低 4 位被赋值给 raddr1。
- 读地址 2 (raddr2): 类似于读地址 1，但是这里是针对读地址 2。当 input_sel 为 3'd1（时，根据 input_value 更新 raddr2。
- 写地址 (waddr): 当 input_sel 为 3'd2（即二进制 10）时，表示当前的输入是写地址。复位和输入逻辑与读地址相同，但是这里更新的是写地址 waddr。
- 写数据高 32 位 (wdata[63:32]): 当 input_sel 为 3'd3 时，表示当前的输入是写数据的高 32 位。如果有有效输入，input_value 会被赋值给 wdata 的高 32 位。
- 写数据低 32 位 (wdata[31:0]): 当 input_sel 为 3'd4 时，表示当前的输入是写数据的低 32 位。如果有有效输入，input_value 会被赋值给 wdata 的低 32 位。

综上所述，当拨码开关为 000 时，输入 raddr1 寄存器的地址；拨码开关为 001 时，输入 raddr2 寄存器的地址；拨码开关为 010 时，输 waddr 寄存器的地址；拨码开关为 011 时，输入写入数据的高 32 位；拨码开关为 100 时，输入写入数据的低 32 位。

```
always @(posedge clk)
    begin
        if (display_number >6'd9 && display_number <6'd26 )
        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,test_addr[3:0]};
            display_name[7:0]<="H";
            display_value<=test_data[63:32];

          end
        else if (display_number >6'd25 && display_number <6'd42 )
        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,test_addr[3:0]};
                    display_name[7:0]<="L";
                    display_value<=test_data[31:0];
        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'd2 : //显示读端口 1 读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name    <= "RDAT1H";
                    display_value <= rdata1[63:32];
                end
                6'd3 : //显示读端口 1 读出的数据
                begin
                    display_valid <= 1'b1;
                    display_name    <= "RDAT1L";
                    display_value <= rdata1[31:0];
                end
                6'd4 : //显示读端口 2 的地址
                begin
                    display_valid <= 1'b1;
                    display_name    <= "RADD2";
```

```
            display_value <= raddr2;
                        end
                        6'd5 : //显示读端口 2 读出的数据
                        begin
                            display_valid <= 1'b1;
                            display_name    <= "RDAT2H";
                            display_value <= rdata2[63:32];
                        end
                        6'd6 : //显示读端口 2 读出的数据
                        begin
                            display_valid <= 1'b1;
                            display_name    <= "RDAT2L";
                            display_value <= rdata2[31:0];
                        end
                        6'd7 : //显示写端口的地址
                        begin
                            display_valid <= 1'b1;
                            display_name    <= "WADDR";
                            display_value <= waddr;
                        end
                        6'd8 : //显示写端口写入的数据
                        begin
                            display_valid <= 1'b1;
                            display_name    <= "WDATAH";
                            display_value <= wdata[63:32];
                        end
                        6'd9 : //显示写端口写入的数据
                        begin
                            display_valid <= 1'b1;
                            display_name    <= "WDATAL";
                            display_value <= wdata[31:0];
                        end
                        default :
                        begin
                            display_valid <= 1'b0;
                            display_name    <= 40'd0;
                            display_value <= 32'd0;
                        end
                    endcase
                end
            end
```

修改说明:

- 修改显示块，当 display_number 的值在 10 到 25 之间时（对应块号 7 到 38），代码会

显示 32 个通用寄存器的高 32 位值。display_name 被设置为"REG"加上寄存器的编号和"H"，表示高 32 位。display_value 被设置为 test_data 的高 32 位。

- 当 display_number 的值在 26 到 41 之间时，代码会显示 32 个通用寄存器的低 32 位值。显示逻辑与高 32 位相似，但 display_name 的最后一个字符是"L"，表示低 32 位。

这里修改读写数据的显示块，以下是每个编号显示块的含义：

① 1 号显示块：显示读端口 1 的地址。
② 2 号显示块：显示从读端口 1 地址读取的数据的高 32 位。
③ 3 号显示块：显示从读端口 1 地址读取的数据的低 32 位。
④ 4 号显示块：显示读端口 2 的地址。
⑤ 5 号显示块：显示从读端口 2 地址读取的数据的高 32 位。
⑥ 6 号显示块：显示从读端口 2 地址读取的数据的低 32 位。
⑦ 7 号显示块：显示写端口的地址。
⑧ 8 号显示块：显示到写端口写入的数据的高 32 位。
⑨ 9 号显示块：显示到写端口写入的数据的低 32 位。

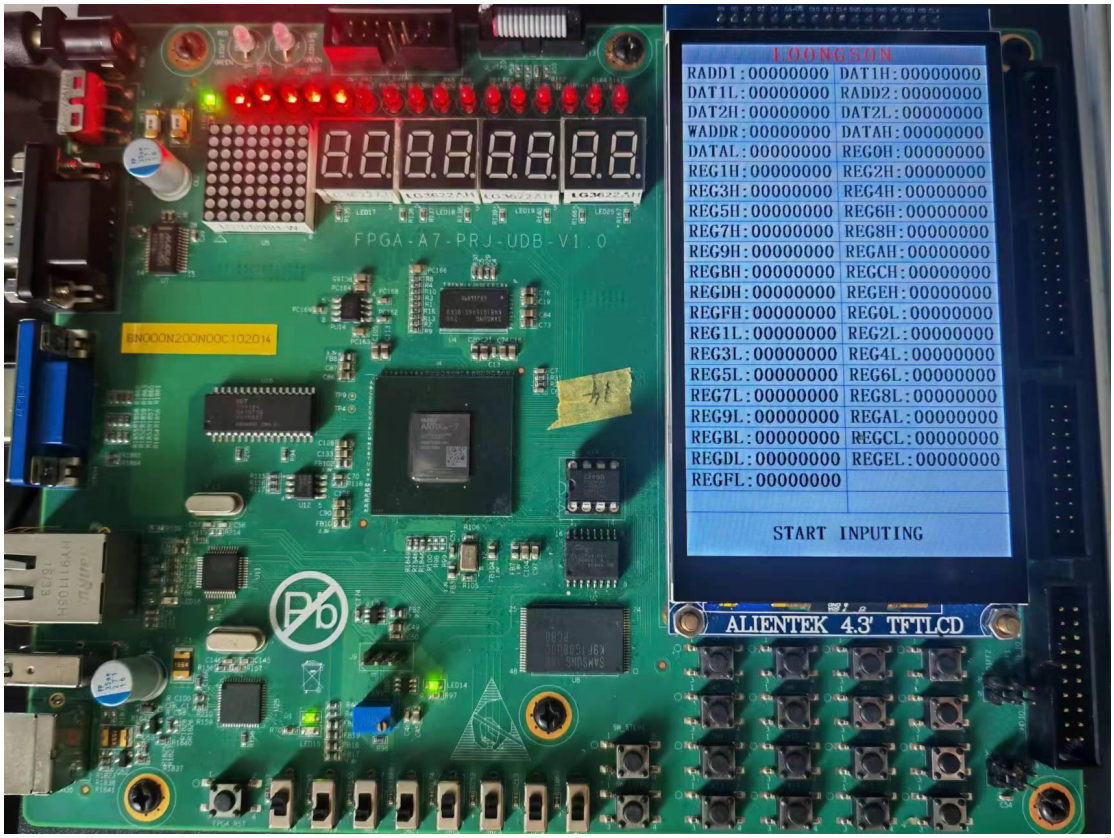
（3）修改约束文件：

```
set_property PACKAGE_PIN AC19 [get_ports clk]
set_property PACKAGE_PIN Y3 [get_ports resetn]
set_property PACKAGE_PIN H7 [get_ports led_wen]
set_property PACKAGE_PIN D5 [get_ports led_waddr]
set_property PACKAGE_PIN A3 [get_ports led_wdata]
set_property PACKAGE_PIN A5 [get_ports led_raddr1]
set_property PACKAGE_PIN A4 [get_ports led_raddr2]

set_property PACKAGE_PIN AC21 [get_ports wen]
set_property PACKAGE_PIN AD24 [get_ports input_sel[2]]
set_property PACKAGE_PIN AD22 [get_ports input_sel[1]]
set_property PACKAGE_PIN AC23 [get_ports input_sel[0]]

set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports resetn]
set_property IOSTANDARD LVCMOS33 [get_ports led_wen]
set_property IOSTANDARD LVCMOS33 [get_ports led_raddr1]
set_property IOSTANDARD LVCMOS33 [get_ports led_raddr2]
set_property IOSTANDARD LVCMOS33 [get_ports led_waddr]
set_property IOSTANDARD LVCMOS33 [get_ports led_wdata]
set_property IOSTANDARD LVCMOS33 [get_ports wen]
set_property IOSTANDARD LVCMOS33 [get_ports input_sel[2]]
set_property IOSTANDARD LVCMOS33 [get_ports input_sel[1]]
set_property IOSTANDARD LVCMOS33 [get_ports input_sel[0]]
```

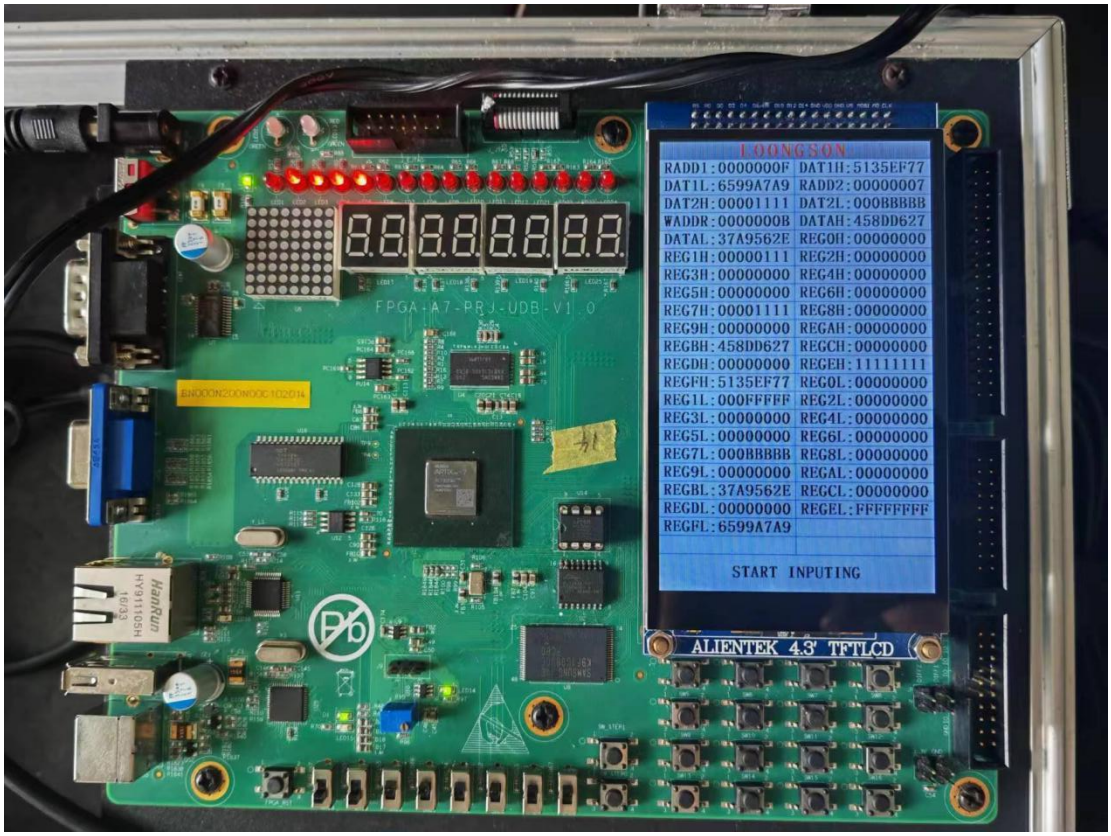
修改说明：

命令指定了哪些物理引脚与设计中的哪些端口相连。这里增加了 input_sel[2]，对应了拨码开关新增的一位。

5.实验结果分析



如图所示，上机箱初始为图中的状态



写入操作:

- 拨码开关调为 1：设置最左边的拨码开关为 1，通常用于激活写入模式。

- 控制拨码开关调为 010：将拨码开关设置为 010，然后将地址 0000000E 写入写寄存器的地址。
- 拨码开关调为 011：设置拨码开关为 011，将数据 458DD627 写入 DATAH，这是数据的高 32 位。
- 拨码开关调为 100：设置拨码开关为 100，将数据 37A8452E 写入 DATAL，这是数据的低 32 位。
- 拨码开关调为 0：将最左边的拨码开关调回 0，通常用于激活存储模式，此时数据被存入地址 0000000E，REGEH 和 REGEL 分别存入数据的高 32 位和低 32 位。

接下来，重复上述步骤，改变地址并输入不同的数据，这样可以在多个地址位置存储数据。

读取操作：

- 拨码开关调为 000：设置拨码开关为 000，对应读地址 1，输入地址 0000000F，读取该地址的数据。DAT1H 和 DAT1L 显示的数据与之前写入 REGEH 和 REGEL 的数据相同，说明数据写入成功。
- 拨码开关调为 001：设置拨码开关为 001，对应读地址 2，输入地址 00000005，读取该地址的数据。DAT2H 和 DAT2L 显示的数据与 REG7H 和 REG7L 的数据相同，说明数据写入成功。

6.总结感想

（1）通过老师的一步步指导以及自己课后的学习，继续了解了 verilog 语言，学会了 vivado 软件和实验箱的基本使用方法与技巧，收获很多，为之后的实验打下了牢固的基础；

（2）深刻体会了用 vivado 如何实现一个寄存器堆，自己进行了改进，将其改进为 64 位；

（3）通过本次实验，更加理解了整个程序的流程以及实验箱的结构，进一步熟悉了 verilog 代码的编写；

（4）学习了 Visio 的基本用法，为以后作图打下了基础。