实验一 组合逻辑设计

一、实验目的

1 掌握 Verilog 语言和 Vivado、Logisim 开发平台的使用；

2 掌握基础组合逻辑电路的设计和测试方法。

二、实验内容（用 Logisim 或 Vivado 实现）

1 基础门电路（多输入门电路、复用器等）的设计和测试；

2 基础功能模块（编码器、译码器等）的设计与测试。

三、实验要求

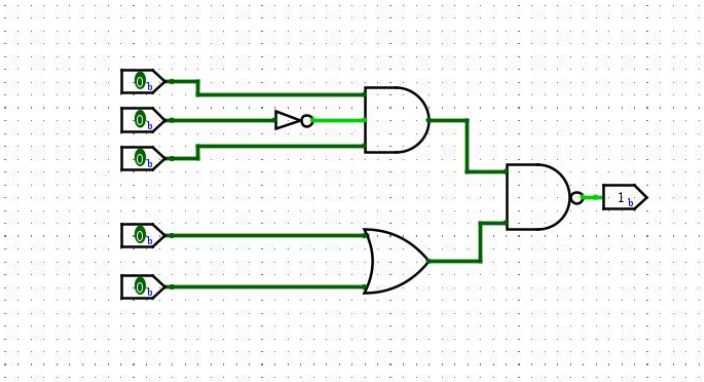
1 掌握 Vivado 与 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

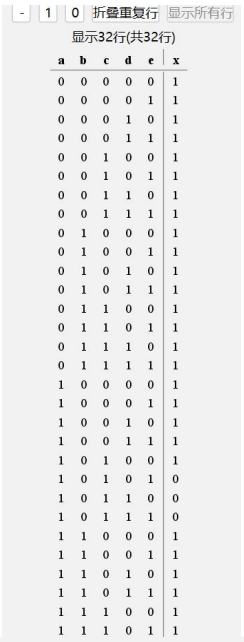
2 记录设计和调试过程（Verilog 代码/电路图/表达式/真值表，Vivado 仿真结果，Logisim 验证结果等）；

3 分析 Vivado 仿真波形/Logism 验证结果，注重输入输出之间的对应关系。

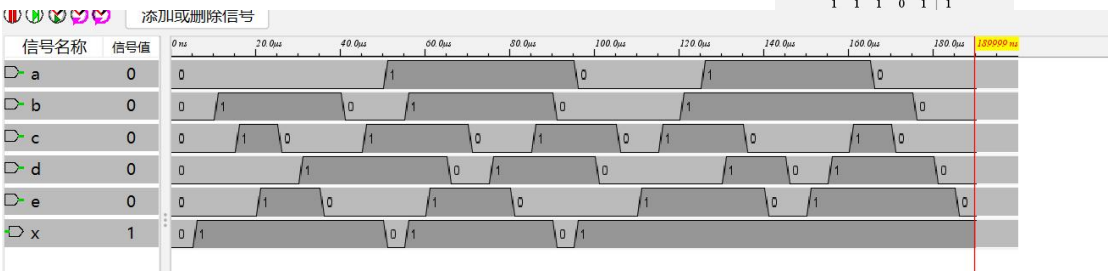
四、实验过程及分析

多输入门电路电路图：





a	b	c	d	e	x
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1



以上是用 logisim 实现的多输入门电路，根据这个表达式 x=~(a· ~b· c· (d+e))所做。

我们可以根据表达式来构建电路：根据表达式，我们可以看出这是一个由四个输入变量 a、b、c、d 和 e 组成的布尔表达式，其中 ~ 表示取反操作，· 表示逻辑与操作，+表示逻

辑或操作。我们可以使用一个与门来实现 a· ~b· c 这个部分，再通过一个非门来实现 ~b。最后，我们可以使用一个或门来实现（d+e）这个部分，然后将其与（a· ~b· c）通过一个与非门进行逻辑与操作，即可得到最终的结果 x。

也可以通过输入表达式，直接构建电路。

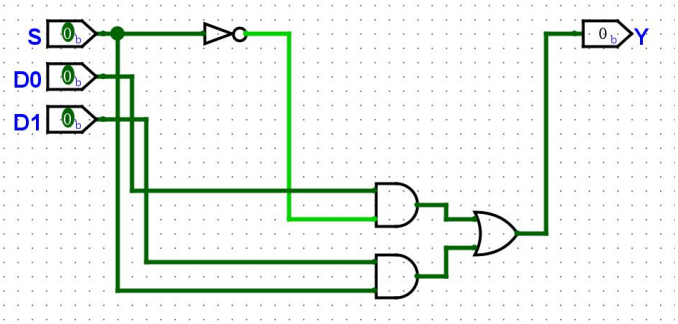
通过真值表可以看到，只有 a=1，b=0，c=1，d+e=1 时最后的输出 x 才是为 0.

复用器：

Logisim:

输出表达式(双击可编辑)：

$Y = D0 \cdot \overline{S} + D1 \cdot S$



显示8行(共8行)

S	D0	D1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

当 S 为逻辑 0（低电平）时，~S 为逻辑 1（高电平），所以第一项 D0·~S 为 D0·1=D0，相应地，第二项 D1·S 为 D1·0=0。因此，当 S 为逻辑 0 时，Y 的取值为 D0。

当 S 为逻辑 1（高电平）时，~S 为逻辑 0（低电平），所以第一项 D0·~S 为 D0·0=0，相应地，第二项 D1·S 为 D1·1=D1。因此，当 S 为逻辑 1 时，Y 的取值为 D1。

首先，使用非门（或称为反向器）将输入信号 S 进行反相得到~S。

然后，将 D0 和~S 作为输入连接到一个与门中。

同时，将 D1 和 S 作为输入连接到一个与门中。

将两个与门的输出连接到一个或门中，作为输入信号。

最后，将或门的输出作为 Y 的输出信号。

这样，当输入 D0 为逻辑 0 且 S 为逻辑 0 时，由于~S 为逻辑 1，与门 1 输出为逻辑 0；当输入 D1 为逻辑 0 且 S 为逻辑 1 时，与门 2 输出为逻辑 0；当输入 D0 为逻辑 1 且 S 为逻辑 0 时，与门 1 输出为逻辑 1；当输入 D1 为逻辑 1 且 S 为逻辑 1 时，与门 2 输出为逻辑 1。最终，或门将两个与门的输出进行逻辑或操作，也就是 Y = D0·~S + D1·S 的结果。

也可以直接通过表达式来构建电路；

Vivado:

mux41.v × sim_mux41.v × Untitled 1 ×

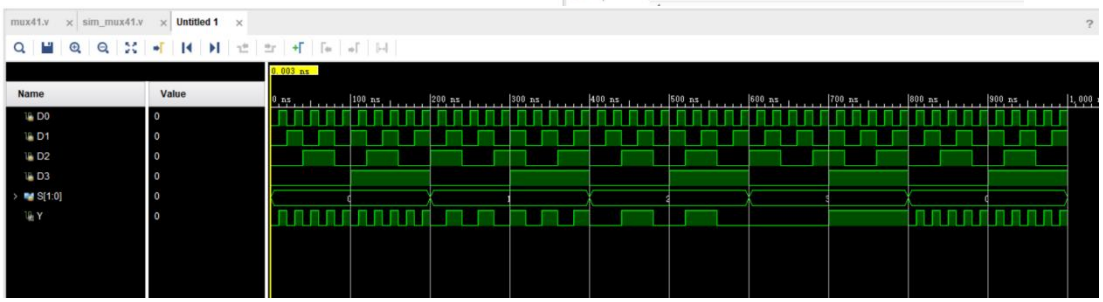
C:/Users/Administrator/mux41/mux41.srcs/sources_1/new/mux41.v

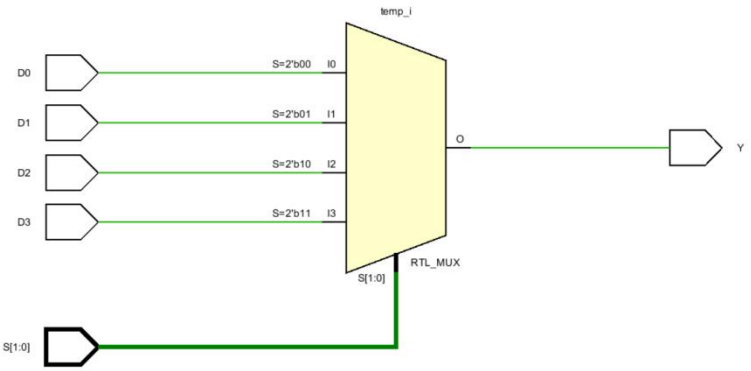
```
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////


module mux41(
input D0, input D1, input D2, input D3,
input [1:0] S,
output Y);
    reg temp;
    always@(*)
    begin
        case(S)
            2'b00 : temp = D0;
            2'b01 : temp = D1;
            2'b10 : temp = D2;
            2'b11 : temp = D3;
            default : temp = D0;
        endcase
    end
    assign Y =  temp;

endmodule
```

mux41.v × sim_mux41.v × Untitled 1 ×

C:/Users/Administrator/mux41/mux41.srcs/sim_1/new/sim_mux41.v

```
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////


module sim_mux41;
    reg D0,D1, D2, D3;
    reg [1:0] S;
    wire Y;
    mux41 mux(D0, D1, D2, D3, S, Y);
    initial
    begin
        D0=0; D1=0; D2=0; D3=0;S=2'b00;
        fork
            repeat (100) #10 D0=~D0;
            repeat (50) #20 D1=~D1;
            repeat (25) #40 D2 =~D2;
            repeat (10) #100 D3= ~D3;
            repeat(5) #200 S=S+1;
        join
    end

endmodule
```





有 4 个数据输入端口 D0、D1、D2 和 D3，2 个选择输入端口 S[1] 和 S[0]，以及 1 个数据输出端口 Y。

在 always 块中，使用 case 语句根据输入的选择信号 S 来选择相应的数据输入端口，将其赋值给临时变量 temp。如果选择信号 S 不在 00、01、10、11 这 4 个值中，则默认选择 D0。然后，使用 assign 语句将临时变量 temp 赋值给输出端口 Y。

在模块 sim_mux41 中，我们定义了 4 个输入信号 D0、D1、D2 和 D3，以及 2 位选择信号 S，以及一个输出信号 Y。在 initial 块中，我们对输入信号进行了一些简单的测试。

首先，我们将 D0、D1、D2 和 D3 初始化为 0，并将选择信号 S 初始化为 00。然后，我们使用 fork-join 结构并行地执行了几个 repeat 语句，来改变输入信号的值。

repeat (100) #10 D0=~D0;：每 10 个时间单位，反转 D0 的值一次。这意味着 D0 的值会在 100 个时间单位内反转 10 次。
repeat (50) #20 D1=~D1;：每 20 个时间单位，反转 D1 的值一次。这意味着 D1 的值会在 50 个时间单位内反转 2.5 次，即反转 2 次。
repeat (25) #40 D2 =~D2;：每 40 个时间单位，反转 D2 的值一次。这意味着 D2 的值会在 25 个时间单位内反转 0.625 次，即反转 0 次。
repeat (10) #100 D3= ~D3;：每 100 个时间单位，反转 D3 的值一次。这意味着 D3 的值会在 10 个时间单位内反转 1 次。
此外，我们还使用了 repeat(5) #200 S=S+1; 语句，每 200 个时间单位，选择信号 S 的值加 1。这意味着选择信号 S 的值会在 5 个时间单位内从 00 变为 01，然后变为 10，最后变为 11。

根据以上的输入变化，我们可以观察到输出信号 Y 的变化。
根据 mux41 模块的实现，当选择信号 S 的值为 00 时，输出信号 Y 的值等于输入信号 D0 的值；当选择信号 S 的值为 01 时，输出信号 Y 的值等于输入信号 D1 的值；当选择信号 S 的值为 10 时，输出信号 Y 的值等于输入信号 D2 的值；当选择信号 S 的值为 11 时，输出信号 Y 的值等于输入信号 D3 的值。如果选择信号 S 不在上述 4 个值中，则输出信号 Y 的值等于输入信号 D0 的值。

根据以上的输入变化和选择信号的变化，我们可以计算出对应的输出信号 Y 的值。在每个时间单位结束时，我们可以记录下输入信号 D0、D1、D2、D3 和选择信号 S 的值，以及输出信号 Y 的值。

编码器：

8-3 优先编码器：

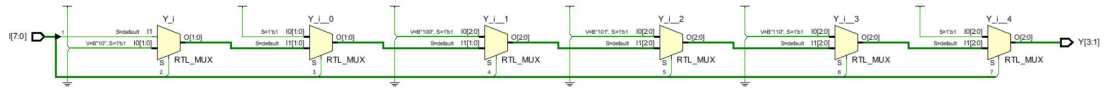
pre_encoder83_case.v *

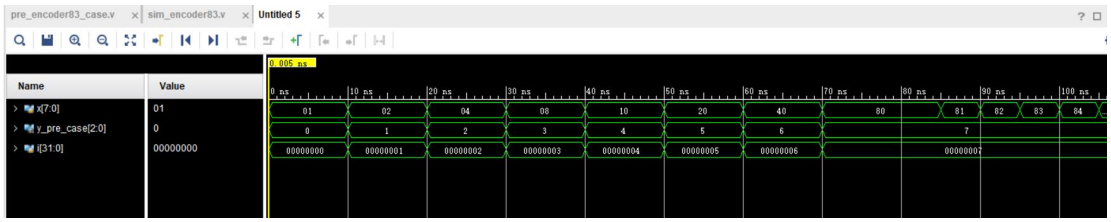
C:/Users/Administrator/encoder83/encoder83.srcs/sources_1/new/pre_encoder83_case.v

```
module pre_encoder83_case(I, Y);
    input I;
    output Y;
    wire[7:0] I;
    reg[3:1] Y;

    always @(I) begin
        casex(I)
            8'b0000_0001 : Y = 3'b000;
            8'b0000_001X : Y = 3'b001;
            8'b0000_01XX : Y = 3'b010;
            8'b0000_1XXX: Y = 3'b011;
            8'b0001_XXXX: Y = 3'b100;
            8'b001X_XXXX: Y = 3'b101;
            8'b01XX_XXXX: Y = 3'b110;
            8'b1XXX_XXXX:Y = 3'b111;
            default : Y =3'b000;
        endcase
    end
endmodule
```

```
module pre_encoder83_if(I, Y);
    input I;
    output Y;
    wire[7:0] I;
    reg[3:1] Y;

    always @(*) begin
        if(I[7] ==1) Y=3'b111;
        else if(I[6] ==1) Y=3'b110;
        else if(I[5] ==1) Y=3'b101;
        else if(I[4] ==1) Y=3'b100;
        else if(I[3] ==1) Y=3'b011;
        else if(I[2] ==1) Y=3'b010;
        else if(I[1] ==1) Y=3'b001;
        else if(I[0] ==1) Y=3'b000;
        else Y=3'b000;
    end
endmodule
```

```
module sim_encoder83();
    reg[7:0] x;
    wire[2:0] y_assign, y_case, y_pre_case, y_pre_if;
    integer i;

    initial begin
        x=1;
        for( i =0; i <7; i=i +1)# 10 x=x * 2;
        #10 x= 128;
        while(x >0)#5 x=x+1;
    end

    pre_encoder83_case pre_encoder83_case_1(.I(x),.Y(y_pre_case));

endmodule
```





第一份程序的模块是一个 3 位预编码器的模块。它有一个 8 位的输入 I 和一个 3 位的输出 Y。I 被定义为一个 8 位的 wire 类型，而 Y 被定义为一个 3 位的 reg 类型。

在 always 块中，使用 casex 语句对输入 I 进行匹配。casex 语句是一种在 Verilog 中用于多路选择的语句。它根据输入 I 的值来选择相应的输出 Y 的值。
casex 语句中的每个匹配模式都是一个 8 位的二进制值，其中 X 表示可以是 0 或 1 的任意值。匹配模式的顺序是从上到下的，所以第一个匹配到的模式将被执行，而其他的模式将被忽略。根据匹配模式，给输出 Y 赋予不同的值。例如，如果输入 I 的值为 8'b0000_0001，则输出 Y 的值将被赋为 3'b000。如果输入 I 的值为 8'b0000_001X，则输出 Y 的值将被赋为 3'b001。如果输入 I 的值不匹配任何模式，则执行 default 语句，将输出 Y 的值赋为 3'b000。
这个预编码器的作用是将输入 I 的 8 位值编码成一个 3 位的预编码值，用于后续的处理或传输。根据不同的输入，它将输出不同的预编码值。

第二份的模块也是一个 3 位预编码器的模块。它有一个 8 位的输入 I 和一个 3 位的输出 Y。I 被定义为一个 8 位的 wire 类型，而 Y 被定义为一个 3 位的 reg 类型。
在 always 块中，使用 if 语句对输入 I 的每个位进行判断。if 语句根据输入 I 的每个位的值来选择相应的输出 Y 的值。
首先，判断输入 I 的最高位 I[7]的值是否为 1。如果是 1，则输出 Y 的值被赋为 3'b111。如果不是 1，则继续判断下一个位。
按照相同的方式，依次判断 I 的每个位的值。如果某个位的值为 1，则输出 Y 的值将被赋为相应的 3 位预编码值，如 3'b110、3'b101 等。如果某个位的值不为 1，则继续判断下一个位。如果所有位的值都不为 1，则执行最后的 else 语句，将输出 Y 的值赋为 3'b000。

sim_encoder83 是一个预编码器的测试模块。它实例化了一个待测的预编码器模块，并为其提供了输入信号 x 和输出信号 y_pre_case。
在 initial 块中，首先给输入信号 x 赋初值为 1。然后使用一个 for 循环，每延迟 10ns，将 x 信号左移一位，共执行 7 次。这样可以模拟输入信号 x 的变化。接着，延迟 10ns 后，将 x 信号赋值为 128，这样可以模拟输入信号 x 的一个特定值。然后使用一个 while 循环，每延迟 5ns，将 x 信号减 1，直到 x 信号的值为 0。这样可以模拟输入信号 x 的递减过程。
在测试模块中，实例化了一个待测的预编码器模块 pre_encoder83_case，并将输入信号 x 连接到该模块的输入端口 I，将输出信号 y_pre_case 连接到该模块的输出端口 Y。通过这样的实例化，可以对待测的预编码器模块进行功能验证。
它可以对预编码器模块进行仿真测试，验证其在不同输入信号 x 下的输出信号 y_pre_case 是否符合预期的预编码规则。

译码器：

24 译码器：

输出表达式(双击可编辑)：

$Y3 = A0 \cdot A1$

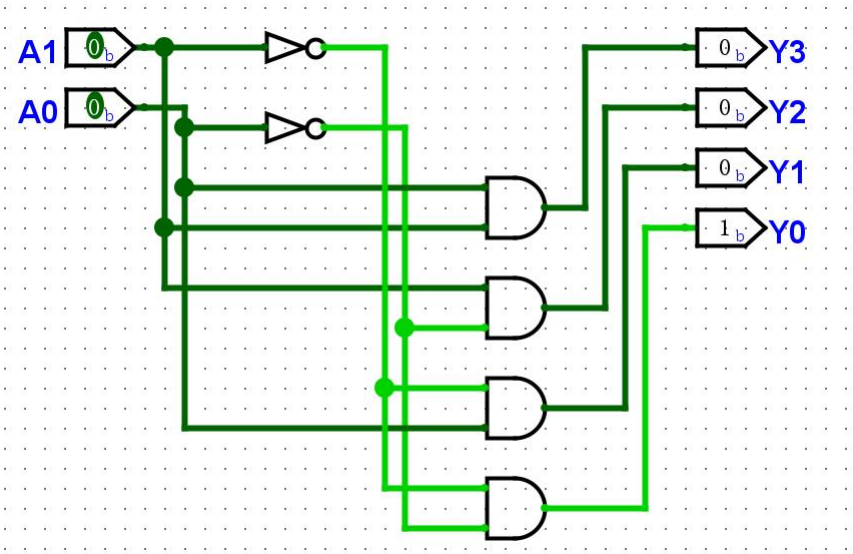
$Y2 = A1 \cdot \overline{A0}$

$Y1 = \overline{A1} \cdot A0$

$Y0 = \overline{A1} \cdot \overline{A0}$

显示4行(共4行)

A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



将两个输入线分别标记为 A1 和 A0。
使用一个非门将 A1 进行反相得到~A1。
使用一个非门将 A0 进行反相得到~A0。
使用一个与门将 A1 和~A0 作为输入，并将其输出连接到输出线 Y2 上。
使用一个与门将 A1 和 A0 作为输入，并将其输出连接到输出线 Y3 上。
使用一个与门将~A1 和 A0 作为输入，并将其输出连接到输出线 Y1 上。
使用一个与门将~A1 和~A0 作为输入，并将其输出连接到输出线 Y0 上。
实现了简易的 2-4 译码器的功能

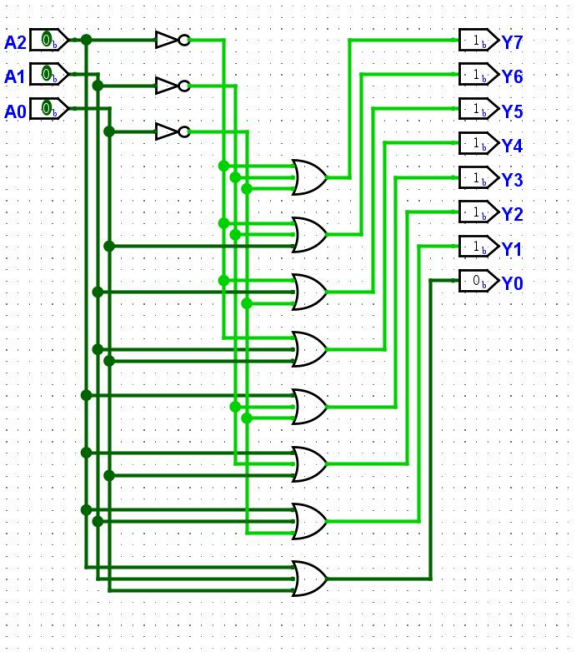
也可以直接通过四个表达式来构建电路，便可以直接得到上图所示电路；

38 译码器：

A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

输出表达式(双击可编辑)：

$Y7 = \overline{A2}+\overline{A1}+\overline{A0}$
$Y6 = \overline{A2}+\overline{A1}+A0$
$Y5 = \overline{A2}+A1+\overline{A0}$
$Y4 = \overline{A2}+A1+A0$
$Y3 = A2+\overline{A1}+\overline{A0}$
$Y2 = A2+\overline{A1}+A0$
$Y1 = A2+A1+\overline{A0}$
$Y0 = A2+A1+A0$



38 译码器是一种数字电路，用于将 3 位二进制输入转换成 8 位二进制输出。输入和输出之间的关系是一一对应的,即每一种输入组合都对应着唯一的输出值。具体的关系如上真值表。输入 A2A1A0 三位二进制位，将其转化为对应十进制位输出 0，其他都输出 1.Y7 为 1 时需要 000 才行，借此我们可以轻易写出表达式三个取反相加即可。最后根据表达式构建电路图如上所示。

五、调试和心得体会

首先，我使用 logisim 实现了多输入门电路。通过这个实验，我深入理解了多输入门的工作原理，以及如何使用逻辑门实现复杂的逻辑功能。
接下来，我使用 logisim 和 vivado 实现了复用器即多路选择器。通过这个实验，学会了如何使用复用器将多个输入信号选择输出，并且实现了多种复杂的逻辑功能。

然后，我使用 vivado 实现了编码器和 83 优先编码器。通过这个实验，我学习了编码器的工作原理以及如何将多个输入信号编码成少量的输出信号。
最后，我使用 logisim 实现了 24 译码器和 38 译码器。通过这个实验，我学习了译码器的工作原理以及如何将少量的输入信号解码成多个输出信号。在调试过程中，我们需要仔细检查输入信号和输出信号的正确连接，以及输出信号的正确性。

通过以上实验，我们不仅掌握了多个重要电路的实现方法，还学会了使用 logisim 和 vivado 进行电路的调试和仿真。在实验过程中，我们遇到了一些问题，但通过仔细分析和调试，最终成功解决了这些问题。通过实验，我们深入理解了计算机组成原理中的重要概念和原理，提高了我们的实践能力和解决问题的能力。

本次实验复习了上学期数电实验的内容，重新对 vivado 编程语言有了新的认识，并且还学习了 logisim 软件来实现一些逻辑电路。

总的来说，计算机组成原理的实验课让我们在实践中学习了多个重要的电路的实现和调试方法，提高了我们的实践能力和解决问题的能力。通过实验，我们更加深入地理解了计算机组成原理中的重要概念和原理，为我们进一步学习和研究计算机组成原理打下了坚实的基础。