实验二　可综合组合逻辑电路实验

一、实验目的

1．掌握可综合 Verilog 语言进行组合逻辑设计的使用。

2．学习测试模块的编写、综合和不同层次的仿真。

二、实验内容

1．设计一款可综合的 2 选 1 多路选择器，并编写 testbench 测试

代码

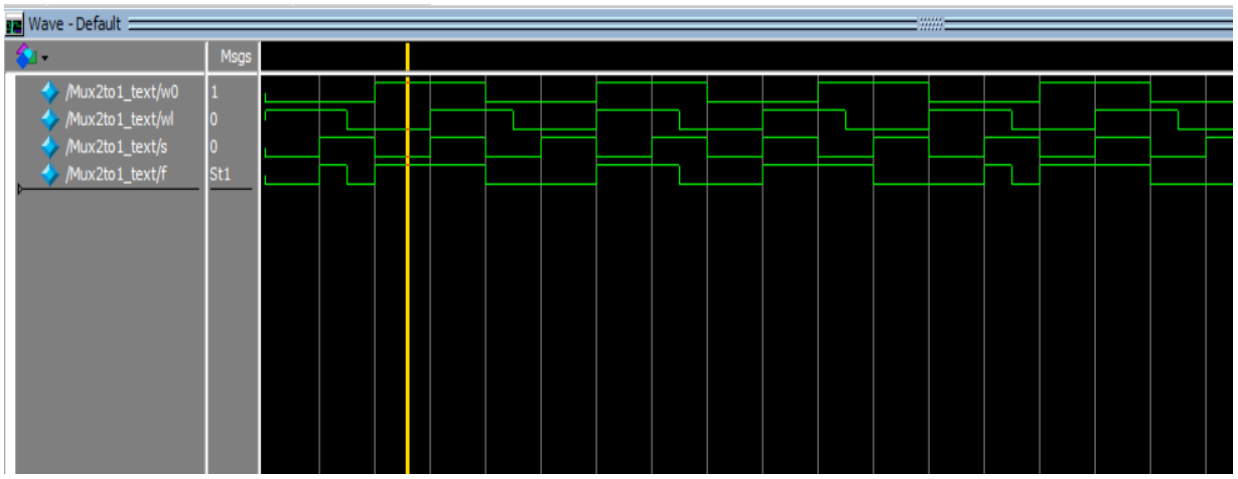
```
module Mux2to1 (w0, wl,s, f);
      input w0, wl, s;
      output f;

      assign f= s ? wl : w0;

endmodule
```

测试代码

```
`timescale 1ns / 1ps

module Mux2to1_text;
   // 输入
   reg w0, wl, s;
   // 输出
   wire f;

   // 实例化 Mux2to1 模块
   Mux2to1 uut (
      .w0(w0),
      .wl(wl),
      .s(s),
      .f(f)
   );

   always #10 s = ~s;
   always #20 w0 = ~w0;
```

```
    always #10 s = ~s;
    always #20 w0 = ~w0;
    always #15 wl = ~wl;

    // 初始块，用于初始化输入
    initial begin
        w0 = 0;
        wl = 1;
        s = 0;
        #300 $stop;
    end
endmodule
```

波形展示与分析



Mux2to1 模块定义了一个简单的 2-to-1 多路选择器，根据输入信号 s 的状态选择输出信号 f 的值是 w0 还是 wl，s 为 0 时输出 f 为 w0；s 为 1 时输出 f 为 w1。

Mux2to1_text 模块实例化了 Mux2to1 模块，并提供了刺激信号，模拟了 w0、wl、s 的变化，以及对应的输出 f。

波形分析:

波形显示了 w0、wl、s 和 f 随时间的变化。输入信号 w0 每隔 20 个时间单位翻转一次，w1 每隔 15 单位翻转，s 每隔 10 个单位反转，尽可能多的组合各种情况。我们观察图片可以发现，当 s 为 0 时，输出 f 随着 w0 的变化而变化，当 s 为 1 时，输出 f 随着 w1 的变化而变化，实现了根据 s 的值选择 w0 或者 w1 的选择器功能。

2．设计一款可综合的 2-4 译码器，并编写 testbench 测试

代码

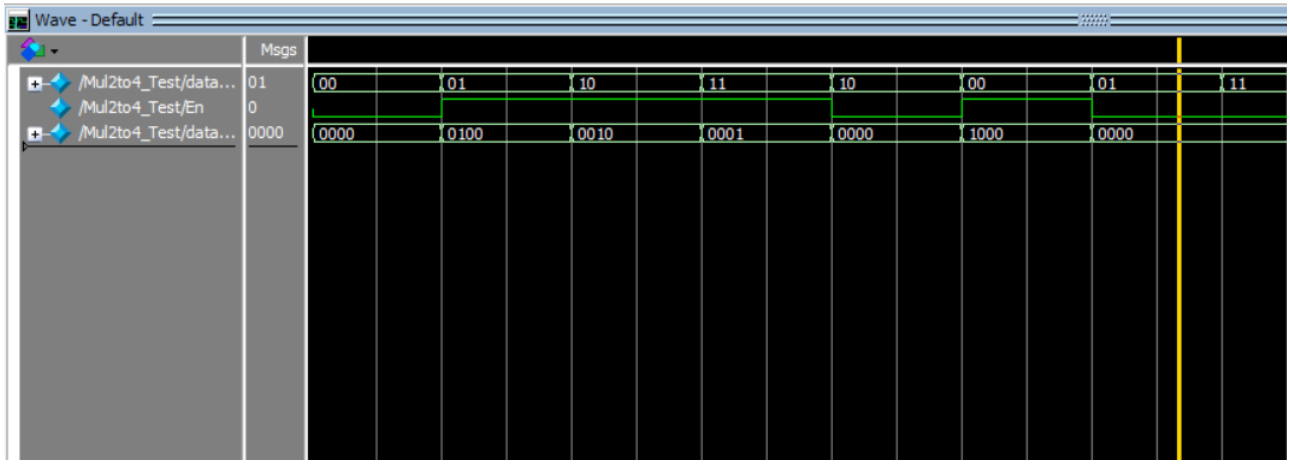
```
module Mul2to4 (data_in, En, data_out);
    input [1:0] data_in;
    input En;
    output reg [0:3] data_out;

    always @(data_in , En)
        case ({En, data_in})
            3'b100:data_out=4'b1000;
            3'b101: data_out= 4'b0100;
            3'b110:data_out=4'b0010;
            3'b111:data_out=4'b0001;
            default: data_out=4'b0000;
        endcase
endmodule
```

测试代码

```
`timescale 1ns / 1ps
module Mul2to4_Test;

  reg [1:0] data_in;   // 2 位输入数据
  reg En;                    // 使能信号

  wire [0:3] data_out; // 4 位输出数据

  // 实例化 2 到 4 译码器模块
  Mul2to4 uut (
    .data_in(data_in),
    .En(En),
    .data_out(data_out)
  );

  // 初始化块
  initial begin

    // 设置初始输入
    data_in = 2'b00;
    En = 0;
    #10; // 等待 10 个时间单元

    data_in = 2'b01;
```

```
        data_in = 2'b01;
        #10;
        data_in = 2'b11;
        En = 0;
        #10;
        En = 1;
        #10;
        data_in = 2'b10;
        En = 1;
        #10;
        data_in = 2'b11;
        En = 1;
        #10;
        data_in = 2'b10;
        En = 0;
        #10;
        data_in = 2'b00;
        En = 1;
        #10;
        data_in = 2'b01;
        En = 0;
    end

endmodule
```

波形展示与分析



Mul2to4 模块是一个简单的 2 到 4 译码器，接受两位输入数据 data_in 和一个使能信号 En，输出一个四位二进制译码结果 data_out。通过组合逻辑，根据输入数据和使能信号的组合，确定输出的二进制译码结果。上图给出了 16 种不同的组合，通过观察可以发现如果使能信号 En 为 1，且输入 data_in 分别为 2'b00、2'b01、2'b10、2'b11，则分别输出相应的二进制译码值 1000、0100、0010、0001，否则输出全零，实现了 2 到 4 译码器的功能。

3．设计一款可综合的 8-3 编码器，并编写 testbench 测试

代码

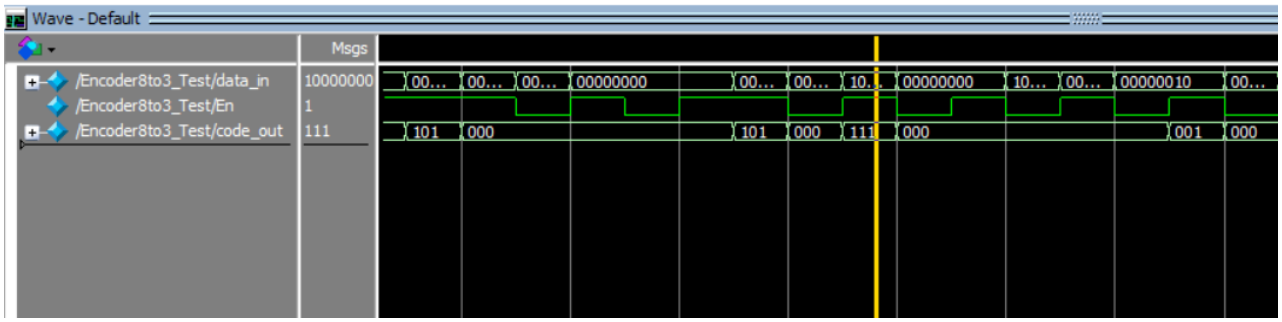
```
module Encoder8to3(data_in,En,code_out);
    input [7:0] data_in;
    input En;
    output reg [2:0] code_out;

    always @(En,data_in)
        case ({En,data_in})
            9'b100000001: code_out = 3'b000;
            9'b100000010: code_out = 3'b001;
            9'b100000100: code_out = 3'b010;
            9'b100001000: code_out = 3'b011;
            9'b100010000: code_out = 3'b100;
            9'b100100000: code_out = 3'b101;
            9'b101000000: code_out = 3'b110;
            9'b110000000: code_out = 3'b111;
            default: code_out = 3'b000; // 默认情况，全为零
        endcase

endmodule
```

测试代码

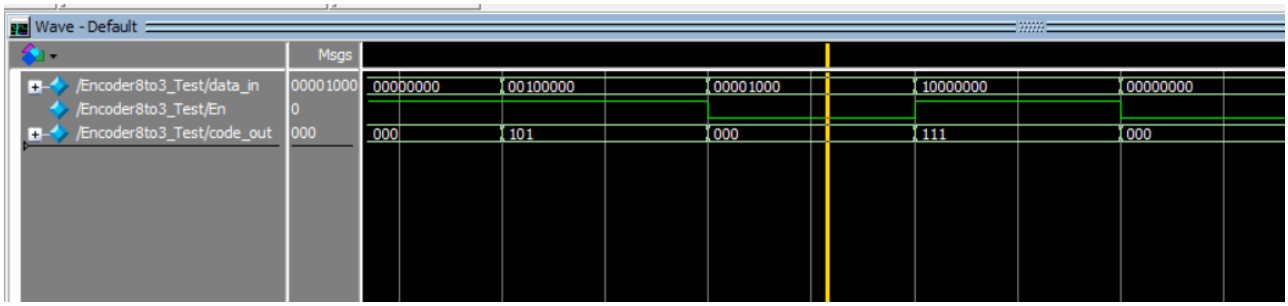
```
`timescale 1ns / 1ps

module Encoder8to3_Test;

    reg [7:0] data_in;   // 8 位输入数据
    reg En;                    // 使能信号
    wire [2:0] code_out; // 3 位输出数据

    // 实例化 8 到 3 编码器模块
    Encoder8to3 uut (
        .data_in(data_in),
        .En(En),
        .code_out(code_out)
    );

    // 声明循环计数变量
    integer i;
```

```
    // 初始化块
    initial begin

        // 循环模拟多个情况，使波形尽可能长
        for (i = 0; i < 100; i = i + 1) begin
            En = $random % 2;              // 随机选择使能信号为 0 或 1
            data_in = (1 << ($random % 8 + 1)) & ~1;
            #10;  // 等待 10 个时间单元，观察波形变化
        end

        #1000;

    end

endmodule
```

波形展示与分析



放大后的部分波形



Encoder8to3 是一个 3 到 8 编码器模块，根据输入信号和使能信号生成 3 位输出编码。测试模块通过循环生成随机输入，验证编码器在各种情况下的行为。编码器的逻辑映射确保了每个输入组合都有唯一的输出编码。测试模块通过等待时间单元观察波形变化，确保了全面而可靠的验证。当输入信号 En=0 时，默认输出 0，当 En 为 1 时，code_out 输出大小 data_in 的 1 的位数的二进制数字，由波形我们可以知道，En 为 1 时，当 data_in 第 n 位为 1 时，code_out 的大小就是 n 的二进制格式，除此之外的所有情况 code_out 的大小都为 0。

4．设计一款可综合的 1 位二进制比较器，并编写 testbench 测试

代码

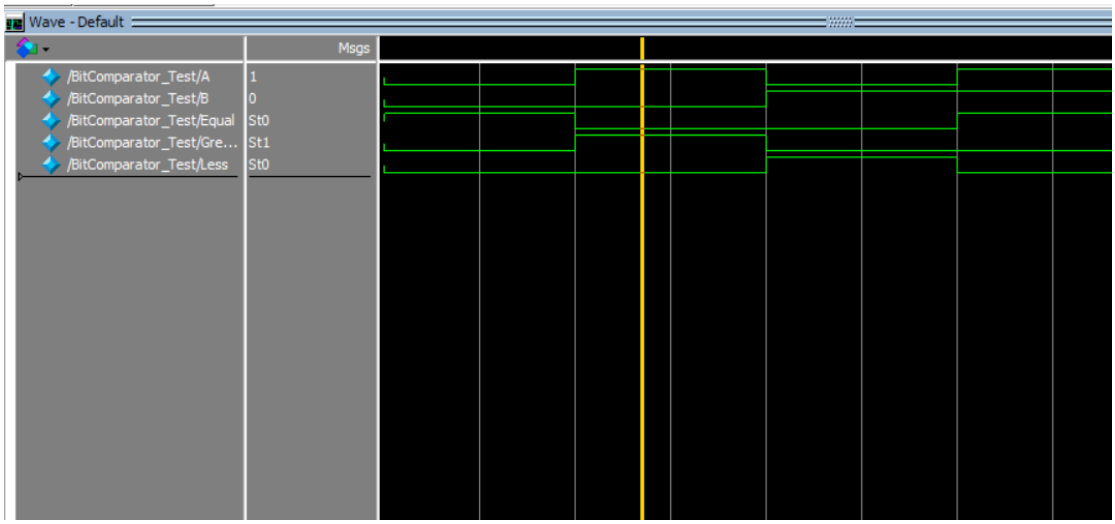
```
module BitComparator(A, B, Equal, Greater, Less);
    input A, B;
    output Equal, Greater, Less;

    assign Equal = (A == B);
    assign Greater = (A > B);
    assign Less = (A < B);

endmodule
```

测试代码

```
`timescale 1ns / 1ps
module BitComparator_Test;
    reg A, B;
    wire Equal, Greater, Less;

    // 实例化比较器模块
    BitComparator uut (
        .A(A),
        .B(B),
        .Equal(Equal),
        .Greater(Greater),
        .Less(Less)
    );

    // 初始化块
    initial begin
        // 测试案例 1: A = 0, B = 0
        A = 0;
        B = 0;
        #10;
        $display("测试案例 1: A=%b, B=%b, Equal=%b, Greater=%b, Less=%b", A, B, Equal,
Greater, Less);

        // 测试案例 2: A = 1, B = 0
        A = 1;
        B = 0;
        #10;
```

```
        $display("测试案例 2: A=%b, B=%b, Equal=%b, Greater=%b, Less=%b", A, B, Equal, Greater, Less);

        // 测试案例 3: A = 0, B = 1
        A = 0;
        B = 1;
        #10;
        $display("测试案例 3: A=%b, B=%b, Equal=%b, Greater=%b, Less=%b", A, B, Equal, Greater, Less);

        // 测试案例 4: A = 1, B = 1
        A = 1;
        B = 1;
        #10;
        $display("测试案例 4: A=%b, B=%b, Equal=%b, Greater=%b, Less=%b", A, B, Equal, Greater, Less);

        $finish;
    end

endmodule
```

波形展示与分析



BitComparator 实现了一个 1 位二进制比较器模块，通过比较输入信号 A 和 B，输出三个信号 Equal、Greater 和 Less，分别表示相等、A 大于 B 和 A 小于 B 的关系。测试模块验证了比较器在不同输入情况下的正确性，通过四个测试案例展示了其功能，通过波形图我们可以看到，根据 A 和 B 的不同组合有三种相应的不同输出，比如 A=1,B=0 时 Greater 为高位，其它两个输出信号为低位，实现了比较器功能。

5．设计一款可综合的 2+2 位简单全加器，并编写 testbench 测试

代码

```
module FullAdder2_2(carryin, data_in1, data_in2, sum, carryout);
    input carryin;                  // 进位输入
    input [1:0] data_in1;           // 输入数据位 1
    input [1:0] data_in2;           // 输入数据位 2
    output [1:0] sum;               // 输出和
    output carryout;                // 输出进位

    // 第一位的全加器
    Fulladd stageo(carryin, data_in1[0], data_in2[0], sum[0], cout1);
    // 第二位的全加器
    Fulladd stage1(cout1, data_in1[1], data_in2[1], sum[1], carryout);
endmodule

module Fulladd(cin, x_in, y_in, s, cout);
    input cin;                      // 进位输入
    input x_in;                     // 输入数据位 1
    input y_in;                     // 输入数据位 2
    output s;                       // 和输出
    output cout;                    // 进位输出

    // 计算和
    assign s = x_in ^ y_in ^ cin;
    // 计算进位
    assign cout = (x_in & y_in) | (x_in & cin) | (y_in & cin);
endmodule
```

测试代码

```
`timescale 1ns/1ps

module FullAdder2_2_Test;
    // 时钟信号
    reg clk = 0;

    // 输入信号
    reg carryin;
    reg [1:0] data_in1;
    reg [1:0] data_in2;

    // 输出信号
```

```
    wire [1:0] sum;
    wire carryout;

    // 实例化被测试的 FullAdder2_2 模块
    FullAdder2_2 DUT (
        .carryin(carryin),
        .data_in1(data_in1),
        .data_in2(data_in2),
        .sum(sum),
        .carryout(carryout)
    );

    // 时钟生成器
    always #5 clk = ~clk;

    initial begin
        carryin = 0;
        data_in1 = 2'b00;
        data_in2 = 2'b00;

        // 在模拟中运行 20 个时钟周期
        repeat (20) begin
            #1 carryin = $random % 2;
            #1 data_in1 = $random % 4;
            #1 data_in2 = $random % 4;

        end

        // 结束仿真
        $stop;
    end
endmodule
```

波形展示与分析

这段波形了一个 2+2 位全加器，用于执行两个 2 位二进制数的加法运算。，其中 carryin 是前一位的进位输入，data_in1 和 data_in2 分别表示两个二位数的输入，sum 是输出的两位二进制和，carryout 是最高位的进位输出。波形中的输入均为随机数，通过波形我们可以发现 sum 和 carryout 随着输入的变化相应变化，比如 data_in1 = 11，data_in1 = 10，carryin = 1 时，sum = 10，carryout = 1。

6．自学 verilog 编程规范和面向硬件电路的设计思想，小结与高级语言编程的主要区别。

并发执行模型： Verilog 是一种硬件描述语言（HDL），其设计理念是基于硬件电路的并发执行模型。硬件电路中的各个部分可以同时运行，因此 Verilog 中常使用 always 块来描述这种并发性。这与高级语言（如 C++或 Java）的顺序执行模型有很大不同。在 Verilog 中，多个 always 块可以同时执行，而在高级语言中，程序按照严格的顺序逐行执行。

阻塞与非阻塞赋值： Verilog 中区分了阻塞赋值和非阻塞赋值两种赋值方式。这是为了更好地模拟硬件电路中的并发行为。阻塞赋值表示在一个时间步内完成所有赋值操作，而非阻塞赋值表示在一个时间步内开始赋值，但不会阻塞后续语句的执行。这种区分在高级语言中通常是不存在的，高级语言更多地关注顺序执行。

时序逻辑与组合逻辑： Verilog 中明确了时序逻辑和组合逻辑的建模方式。时序逻辑通常使用非阻塞赋值，以模拟寄存器的行为；而组合逻辑通常使用阻塞赋值，以模拟组合逻辑电路的行为。在高级语言中，这样的区分并不明显，高级语言更注重算法和程序的逻辑。

硬件描述与算法描述： Verilog 主要用于描述硬件电路，如 FPGA 或 ASIC 中的逻辑。它更**侧重于电路的结构和行为**：相比之下，高级语言更注重算法和数据结构的描述，适用于通用计算。

时钟域和同步设计：在硬件设计中，时钟是一个关键的概念。Verilog 中明确定义了时钟域，并提供了同步设计的规范。这与高级语言中的通用编程范式有很大不同.

Verilog 更接近于底层硬件的描述语言，其设计理念和编程模型更贴近硬件电路的行为。相比之下，高级语言更适用于通用软件开发，强调算法和程序的逻辑。

三、本次实验的心得体会

1.在编写 2 选 1 多路选择器测试代码时，我用了一段以下的代码，期望 s 5ns 翻转一次，w0 20ns 翻转一次，w1 15ns 翻转一次，以尽可能多的获得各种组合

```
repeat (10) begin
   #20 w0 = ~w0; // Toggle w0 every 10 time units
   #15 wl = ~wl; // Toggle wl every 10 time units
   #10   s = ~s;     // Toggle s every 5 time units
end
```

在我仿真出波形之后，我发现虽然实现了正确的二选一判断，但是 s,w0,w1 都是 45ns 翻转一次，而且它们的起始翻转时间不同，经过多次更改数据测验后，我发现此代码的语句是依次执行的，并不能做到并行
为了实现我想要的结果，我将代码修改为以下的逻辑，终于实现了目标的翻转。这使我对高级程序语言和 verilog 的区别理解更加深刻。

```
   always #10 s = ~s;
   always #20 w0 = ~w0;
   always #15 wl = ~wl;
```

2.本次实验让我更好的掌握了 verilog 语言的编程规范，其中模块和变量的命名，代码的注释等方面和高级程序语言类似，但是需要注意的是代码逻辑和高级程序语言不同。

这次实验不仅让我熟悉了 Verilog 语言的基本语法和常用结构，还深化了我对数字电路设计原理的理解。通过编写 testbench 测试，我不仅验证了设计的正确性，还培养了对数字电路性能和功能的全面考虑能力，此次实验还深化了我对 verilog 和高级程序语言的异同的理解。