Lab1 组合逻辑器件实验

一、实验目的

1. 掌握Verilog HDL设计数字电路的方法。
2. 掌握多路选择器和多路分配器的设计方法。
3. 掌握译码器和编码器的设计方法。
4. 掌握数据并行传输的设计方法。
5. 掌握开发板的输入输出模块的使用方法。
6. 掌握组合逻辑部件的应用方法。

二、实验环境

软件：Vivado 2020.2

硬件：Nexys A7-100T 开发板

三、实验原理

1、多路选择器和多路分配器

多路选择器是一种从多路输入端中选择某一路输入信号进行输出的组合逻辑器件，是一种多输入、单输出的标准化逻辑部件，也称多路复用器或数据选择器，它是数字系统设计中的常用元器件之一，在数字系统中有着广泛的应用。多路分配器的功能与多路选择器正好相反，它把唯一的一路输入信号输送到多路输出端中的某一路端口中国，选择从哪一路输出端送出输入信号，则取决于分配器的控制端。

1）多路选择器

假设4路选择器的输入端为d0、d1、d2、d3，输出端为y，选择端为s1、s0。其真值表如表1.1所示。

表 1.1　4 路选择器真值表

选择 s1 s0	输出 y
00	d0

01	d1
10	d2
11	d3

根据真值表得到输出y的逻辑表达式为：$y = \overline{s1} \cdot \overline{s0} \cdot d0 + \overline{s1} \cdot s0 \cdot d1 + s1 \cdot \overline{s0} \cdot d2 + s1 \cdot s0 \cdot d3$。。

根据输出逻辑表达式可以设计4路选择器的电路原理图，如图1.1所示。

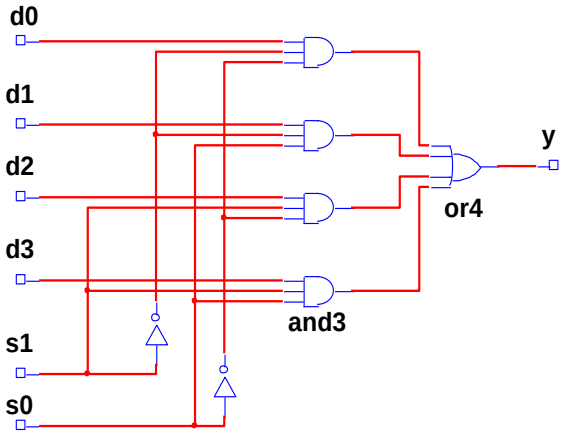


图 1.1 4 路选择器原理图

Verilog语言本质上是一门电路建模语言，对电路的描述主要有三种建模方式，分别是结构化建模、数据流建模和行为建模，分别对应电路的三种抽象视角，各有特点。下面以4路选择器的设计为例，分别介绍三种建模方式。

1、结构化建模方式

结构化建模方式将电路描述成一个分级的子模块系统，并通过逐层调用子模块来构成功能复杂的数字系统。门级结构化建模是通过调用Verilog内建的基本门级元件来对电路进行建模。常用的基本门级元件有以下8种：非门(not)、与门(and)、与非门(nand)、或门(or)、或非门(nor)、异或门(xor)、同或门 (xnor)、缓冲门(buf)。这些门级元件需要通过实例化语句来使用。

根据图1.1所示的4路选择器电路原理图，采用门级结构化建模方式，设计4路选择器的模块代码如下：

```
module mux4to1_1(                    //端口声明
        output   y,                  // 声明 1 个 wire 型输出变量 y，其宽度为 1 位。
        input    d0, d1,d2,d3,       // 声明 4 个 wire 型输入变量 d0-d3，其宽度为 1 位。
        input    s0, s1              // 声明 2 个 wire 型输入变量 s0、s1，其宽度为 1 位。
);
    //内部网线声明
    wire s0_n,s1_n;                  //两个非门的输出信号，分别对应 s0 和 s1 的非。
    wire y0,y1,y2,y3;                //四个与门的输出信号
    not(s0_n,s0);                    //调用非门，生成 s1_n 和 s0_n
    not(s1_n,s1);
    and(y0,s0_n,s1_n,d0);            //调用三输入与门，生成 y0~y3
    and(y1,s0_n,s1,d1);
```

```
    and(y2,s0,s1_n,d2);
    and(y3,s0,s1,d3);
    or(y,y0,y1,y2,y3);          //调用四输入或门
endmodule
```

结构化建模方式基本上相当于电路的门级结构图的直接翻译。其优点是，可以对电路的结构进行精确的控制，在关键路径的优化上可以起到明显的效果。不过，当电路规模增大时，使用该建模方式会变得非常烦琐，容易出错，代码维护困难。

2、数据流建模方式

数据流建模方式从数据的视角出发，通过描述数据在电路中的流动方向来描述电路的功能，因而数据流建模方式可用于对组合逻辑电路进行描述；而时序逻辑电路涉及数据的存储和更新，因而不适合采用数据流建模方式。

数据流建模方式主要通过连续赋值语句进行建模。连续赋值语句的语法为：

assign 网线 = 表达式;

其含义是用赋值号右边的表达式所描述的电路的输出来驱动赋值号左边的网线。

根据4路选择器输出y的逻辑表达式，采用数据流建模方式，实现4路选择器的模块代码如下：

```
module mux4to1_2(                  //端口声明
        output   y,                // 声明 1 个 wire 型输出变量 y，其宽度为 1 位。
        input    d0, d1,d2,d3,     // 声明 4 个 wire 型输入变量 d0-d3，其宽度为 1 位。
        input    s0, s1            // 声明 2 个 wire 型输入变量 s0、s1，其宽度为 1 位。
);
      assign y = (~s0 & ~s1 & d0) | (~s0 & s1 & d1) | (s0& ~s1 & d2) | (s0 & s1& d3);
endmodule
```

使用连续赋值语句来代替门级元件的实例化，代码更加简洁明了。若要更直接地表示电路的功能，则可采用条件运算符来描述，其可读性比布尔表达式更好，对应代码如下：

```
assign y = (s0) ? (s1 ? d3 : d2) : (s1 ? d1 : d0);
```

由此可见，数据流建模方式比门级结构化建模方式的代码更简洁。在现代数字系统的开发过程中，大部分组合逻辑电路都是采用数据流建模方式来描述的。

3．行为建模方式

行为建模方式通过一系列以高级编程语言编写的过程块来描述数字系统的功能。行为建模方式主要从整个系统的功能方面考虑，编码时无须关注电路的具体结构，而由综合器根据过程块所描述的行为综合出实现该行为的电路结构。

在Verilog中，行为建模的关键要素是always语句，其使用较多的语法为：

always @ (事件信号列表) 过程语句

其含义是，当事件信号列表中的任意一个信号发生变化时，过程语句中的信号将按照所描述的行为进行更新。过程语句中被赋值的只能是reg类型变量。

事件信号列表中可以只有单个信号，也可以有多个信号。如果事件信号列表是星号（*），表示隐式事件信号列表，是过程语句中赋值号右侧所有信号集合的一种简写方式。此时，过程语句中赋值号右侧任意一个信号发生变化，都会导致赋值号左侧的信号进行更新，其含义也可以理解为“按照过程语句中所描述的行为一直对相应信号进行驱动”。这一含义与数据流建模方式中的连续赋值语句的行为非常类似，因此可以通过隐式事件信号列表的always语句对组合逻辑电路进行建模。

采用行为建模方式实现4路选择器的模块代码如下：

```
module mux4to1_3(                  //端口声明
        output reg y,              // 注意此处 y 类型为 reg。
        input     d0, d1,d2,d3,    // 声明 4 个 wire 型输入变量 d0-d3，其宽度为 1 位。
        input     s0, s1           // 声明 2 个 wire 型输入变量 s0、s1，其宽度为 1 位。
);
     always @(*)                   //相当于 @( s0, s1, d0, d1, d2, d3)
          case ({s0,s1})
            2’b00: y=d0;
            2’b01: y=d1;
            2’b10: y=d2;
            2’b11: y=d3;
          endcase
endmodule
```

与数据流建模方式相比，行为建模方式借助高级语言的特性来对电路的功能进行描述，描述的效果更加接近人的思考方式，大大提升了开发效率，代码的可读性也更好。在现代数字系统的开发过程中，时序逻辑电路大都采用行为建模方式来开发。

采用行为建模方式编码时可以不必考虑电路的具体结构，开发者（尤其是初学者）很容易编写出有问题的代码，这些问题包括代码仿真行为正确但无法综合，代码仿真行为正确且可综合但综合后电路行为不正确等。

上述通过不同的建模方式设计了3个不同4路选择器电路设计文件，为了验证4路选择器的功能，需要创建仿真测试文件来验证电路功能。测试文件的模块参考设计如下：

```
`timescale 10 ns/ 1 ps             // 设置时间尺度和时间精度
module mux4to1_tst();              // 测试代码的端口参数列表为空
         reg d0,d1,d2,d3,s0,s1;    // 输入变量声明为 reg 型变量
         wire y;                   // 输出变量声明为 wire 型变量
// 测试的模块进行实例化
mux4to1_1 mux4to1_inst ( .d0(d0), .d1(d1), .d2(d2), .d3(d3), .s0(s0), .s1(s1), .y(y) );
initial
     begin                         //   初始化输入变量
        s0=0;s1=0;
        d0=1;d1=0;d2=0;d3=0;
        #20;
```

```
        d0=0;d1=1;d2=1;d3=1;
        #20;
        s0=0;s1=1;
        d0=0;d1=1;d2=0;d3=0;
        #20;
        d0=1;d1=0;d2=1;d3=1;
        #20;
        s0=1;s1=0;
        d0=0;d1=0;d2=1;d3=0;
        #20;
        d0=1;d1=1;d2=0;d3=1;
        #20;
        s0=1;s1=1;
        d0=0;d1=0;d2=0;d3=1;
        #20;
        d0=1;d1=1;d2=1;d3=0;
        #20;
end
endmodule
```

上述代码的逻辑仿真结果如图1.2所示。

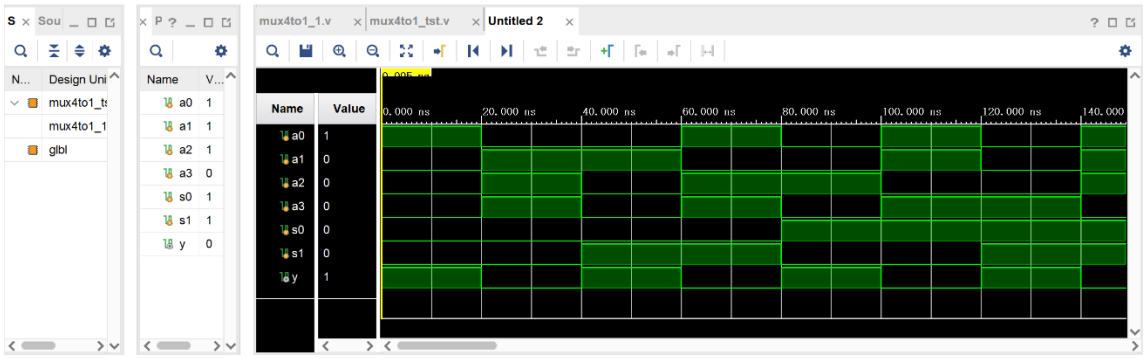


图 1.2　4 路选择器 Verilog 程序逻辑仿真波形图

实验验证：修改仿真测试文件中待测模块名称，将上述所有的编码器模块都进行仿真测试和综合实现，并上实验板可以验证；对比不同设计方式实现电路的硬件资源和性能差异，并思考原因。

提示：当工程中存在多个模块文件和多个测试仿真文件，需要分别进行测试仿真时，需要把源模块文件和测试文件都设置为顶层文件，鼠标移动到文件名上，单击右键，选择“Set as Top”，设置成功后，文件名前出现有三个小圆点的图标，表示当前文件为顶层模块。存在不同的引脚约束文件时，需要把不是当前模块对应的约束文件设置为disable。

2）多路分配器

多路分配器的功能是把唯一的输入信号发送到某个输出端口中，从哪一个输出端输出输入信号，取决

于控制端。

假设4路分配的输入端为d，输出端为d0、d1、d2、d3，选择端为s0、s1。当s1s0=00时，d0输出d；当s1s0=01时，d1输出d；当s1s0=10时，d2输出d；当s1s0=11时，d3输出d；

选择数据流的描述方法实现4路4位分配器的模块代码如下：

```
module dmux1to4(
//端口声明
      output   [3:0] d0,d1,d2,d3,      //4 路 4 位的输出信号 d0~d3。
      input    [3:0] d,                // 4 位输入信号  d。
      input    [1:0] s                 // 2 位选择控制信号 s。
);
      assign d0 = ( ~s[1] & ~s[0] ) ?   d : 4 'bz;
      assign d1 = ( ~s[1] & s[0] )   ?   d : 4'bz;
      assign d2 = ( s[1] & ~s[0] )   ?   d : 4'bz;
      assign d3 = ( s[1] & s[0] )    ?   d : 4'bz;
endmodule
```

为上述4路分配器模块建立一个仿真测试文件，对功能进行仿真。测试模块代码如下：

```
`timescale 1ns / 1ps
module dmux4to1_tst(    );
          reg [3:0] d;                          // 输入变量声明为 reg 型变量
          wire [3:0] d0,d1,d2,d3;
          reg [1:0] s;                          // 输入变量声明为 reg 型变量
          integer i;
          dmux1to4 dmux1to4_inst (          // 对要测试的模块进行实例化
            .d(d), .s(s), .d0(d0), .d1(d1), .d2(d2), .d3(d3) );
initial
      begin
         s=2'b00;
         for (i=0;i<=15;i=i+1) begin d=i; #5; end
         s=2'b01;
         for (i=0;i<=15;i=i+1) begin d=i; #5; end
         s=2'b10;
         for (i=0;i<=15;i=i+1) begin d=i; #5; end
         s=2'b11;
         for (i=0;i<=15;i=i+1) begin d=i; #5; end
         #20;
     end
endmodule
```

执行仿真，观测仿真波形图如图1.3所示。

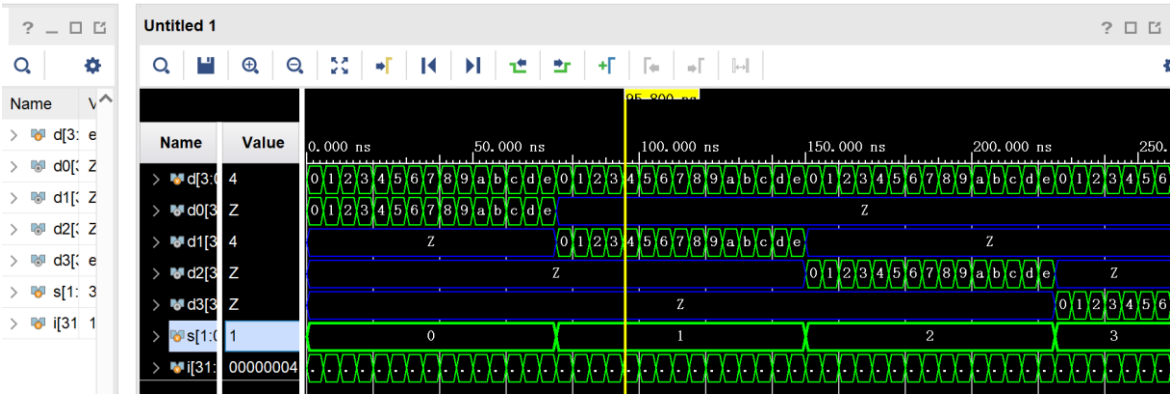


图 1.3 4 路 4 位分配器仿真测试波形图

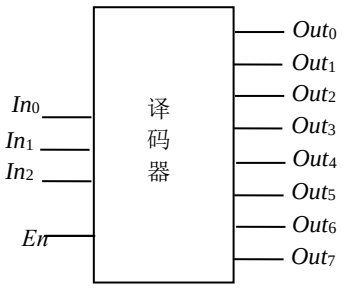
实验验证：进行仿真测试、综合和上板验证。

2、译码器和编码器

1）译码器

译码器是一种多输入、多输出组合逻辑电路，且输入端口数比输出端口数少。若译码器的输入端有n位，则输出端有2^n个，输出为2^n中取1编码，称为n-2^n译码器。译码器可用于地址译码、指令译码和实现逻辑函数等数字系统设计中。常见译码器有2-4译码器、3-8译码器等。

3-8译码器的逻辑符号图和真值表，如图1.4所示，有3个输入端In0~In2，1个使能端En，8个输出端Out0~Out7。



E_n	In_0	In_1	In_2	Out_0	Out_1	Out_2	Out_3	Out_4	Out_5	Out_6	Out_7
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

图 1.4　3-8 译码器逻辑符号和真值表

使用Verilog语言实现译码器的方法有很多种，下面分别通过门级结构化建模、数据流建模和行为化建模来介绍。

根据真值表，采用结构化建模方式实现的3-8译码器的模块代码如下：

```
module decode3to8_s (       //端口声明
    output [7:0] Out,
    input   [2:0] In,
```

```
    input     En
);
  wire NOTIn2, NOTIn1, NOTIn0;
  not U1 (NOTIn0, In[0]);
  not U2 (NOTIn1, In[1]);
  not U3 (NOTIn2, In[2]);
  and U4 (Out[0], NOTIn2, NOTIn1,NOTIn0,En);
  and U5 (Out[1], NOTIn2, NOTIn1,In[0],En);
  and U6 (Out[2], NOTIn2, In[1],NOTIn0,En);
  and U7 (Out[3], NOTIn2, In[1],In[0],En);
  and U8 (Out[4], In[2], NOTIn1,NOTIn0,En);
  and U9 (Out[5], In[2], NOTIn1,In[0],En);
  and U10 (Out[6], In[2],In[1],NOTIn0,En);
  and U11 (Out[7], In[2], In[1],In[0],En);
endmodule
```

使用结构化设计方法通常失去了使用 Verilog 创建易于理解和维护的设计目的。

采用数据流建模方式，使用连续赋值语句实现的 3-8 译码器的模块代码如下：

```
module decode3to8_d1 (       //端口声明
    output [7:0] Out,
    input  [2:0] In,
    input  En
);
  assign Out[0] =En ? (In==3'b000) : 0;
  assign Out[1] =En ? (In==3'b001) : 0;
  assign Out[2] =En ? (In==3'b010) : 0;
  assign Out[3] =En ? (In==3'b011) : 0;
  assign Out[4] =En ? (In==3'b100) : 0;
  assign Out[5] =En ? (In==3'b101) : 0;
  assign Out[6] =En ? (In==3'b110) : 0;
  assign Out[7] =En ? (In==3'b111) : 0;
endmodule
```

实际上，还可以使用移位操作来实现译码器，参考设计代码如下：

```
module decode3to8_d2 (       //端口声明
    output [7:0] Out,
    input  [2:0] In,
    input  En
);
  assign Out =En ? (1<<In) : 0;
 endmodule
```

Verilog 行为化建模采用“过程语句”来里定义逻辑行为。采用行为化建模方式实现 3-8 译码器有几种不同的方式。

使用 case 语句实现 3-8 译码器的模块代码如下：

```
module decode3to8_b1 (       //端口声明
      output reg [7:0] Out,
      input    [2:0] In,
      input    En
 );
   always @ (In,En)
      if (En==0) Out = 8'b000000000;
      else
         case (In)
            3'b000: Out = 8'b00000001;
            3'b001: Out = 8'b00000010;
            3'b010: Out = 8'b00000100;
            3'b011: Out = 8'b00001000;
            3'b100: Out = 8'b00010000;
            3'b101: Out = 8'b00100000;
            3'b110: Out = 8'b01000000;
            3'b111: Out = 8'b10000000;
         endcase
endmodule
```

上述模块中，always 语句敏感信号列表中包括译码器的所有输入，case 语句枚举所有可能的 In 二进制值，某种角度上看只是模拟了译码器的真值表，在设计更大的译码器建模时，可能会由于规模变大而容易产生错误。

使用 for 语句能更好地体现译码器的行为特性，Verilog 模块如下：

```
module decode3to8_b2 (       //端口声明
      output reg [7:0] Out,
      input    [2:0] In,
      input      En
 );
 integer i;
   always @ (In,En)
      if (En==0) Out = 8'b000000000;
      else
   begin
         for (i=0; i<=7; i=i+1)
            Out[i]=(In==i);
     end
endmodule
```

在综合时，for 循环综合出一个组合逻辑结构。将输入 In 与循环中的 i 可能取值逐一进行比较，并匹配比较的结果。

一个更为简洁的行为化建模方式实现 3-8 译码器的模块代码如下：

```
module decode3to8_b3 (       //端口声明
      output reg [7:0] Out,
      input    [2:0] In,
      input      En
 );
   always @ (In,En)
   begin
      Out = 8'b00000000;
    if(En==1) Out[In]=1;
 end
endmodule
```

在将输出初始化全 0 之后，只需要用 In 作为索引，将 Out[In]位设置为 1。

为了验证设计的正确性，编写一个 3-8 译码器的测试平台，通过对使能端 En 和输入端 In 的连续赋值，逐一检测输出结果。为上述模块建立测试模块代码如下：

```
`timescale 1ns / 1ps
module decode3to8_tb();
   reg [2:0] In;
   reg En;
   wire [7:0] Out;
   integer i, errors;
   reg [7:0] expectY;
   decode3to8_s UUT ( .Out(Out),.In(In),.En(En));   // 实例化待测单元
    initial begin
      errors = 0;
      for (i=0; i<=15; i=i+1) begin
         {En, In} = i;                                  // 应用测试输入组合
         #10 ;
         expectY = 8'b00000000;                         // 如果 En=0，则缺省输出全为 0
         if (En==1) expectY[In] = 1'b1;                 //否则，输出在 In 位上应是有效值
         if (Out !== expectY) begin
            $display("Error: En=%b, In = %3b, out = %8b",      En, In, Out);
            errors = errors + 1;
         end
      end
      $display("Test complete, %d errors",errors);
   end
endmodule
```

实验验证：修改仿真测试文件中待测模块名称，将上述所有的编码器模块都进行仿真测试和综合实现，并上实验板可以验证；对比不同设计方式实现电路的硬件资源和性能差异，并思考原因。

2）编码器

编码器是一种与译码器功能相反的组合逻辑电路，编码器的输出端口数比其输入端口数少。编码器的输入通常是多个独立信号，输出是这些独立信号中的一个有效信号的编码。最常见的编码器是 2^n-n 编码器，也称二进制编码器，它与 n-2^n 译码器功能正好相反，有 2^n 个输入端，n 个输出端。

优先权编码器是一个优先级排队电路加一个编码器。优先权编码器允许有多个输入同时有效，但只对优先级最高的输入进行编码输出。在计算机中常用作设计硬件中断控制器。

假定一个 3 位优先权编码器的优先级顺序为 I7 > I6 > I5 > I4 > I3 > I2 > I1 > I0，则该编码器的真值表如表 1.2 所示。真值表中 1 表示有效输入，x 表示任意取值。

表 1.2 优先权编码器的功能表

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
1	x	x	x	x	x	x	x	1	1	1
0	1	x	x	x	x	x	x	1	1	0
0	0	1	x	x	x	x	x	1	0	1
0	0	0	1	x	x	x	x	1	0	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	0	0	1	0	0	0

采用行为化建模方式设计优先权编码器，可以使用使用嵌套的 if 语句，从最高优先级的输入端开始判断，直至判断最低优先级的输入端；如果输入端口数较多的话，这可能会导致潜在的错误。因而一般采用 case 语句或 for 语句来实现。

采用行为化建模方式，使用 case 语句设计的 8-3 优先权编码器模块代码如下：

```
module encode8to3_1(
    output reg [2:0] Out,
    input [7:0] In
);
    always@(*) begin
        Out=1'b0;                    //缺省输出值
        casex(1'b1)
            In[7]: Out = 7;     //从最高优先级输入端开始编码
            In[6]: Out = 6;
            In[5]: Out = 5;
            In[4]: Out = 4;
            In[3]: Out = 3;
            In[2]: Out = 2;
            In[1]: Out = 1;
            In[0]: Out = 0;
```

```
            endcase
        end
    endmodule
```

采用行为化建模方式，使用 for 语句设计的 8-3 优先权编码器模块代码如下：

```
    module encode8to3_2(
        output reg [2:0] Out,
        input [7:0] In
    );
        integer i;
        always @(*) begin
           Out=1'b0;                              //缺省输出值
           begin
              for(i=0; i<=7; i=i+1)            //从最低优先级输入端开始编码
                    if(In[i]==1) Out=i;
            end
        end
    endmodule
```

编写一个 8-3 优先级编码器的测试平台来验证功能的正确性，通过对输入端 In 的连续赋值，逐一检测输出结果。一个仿真测试模块如下：

```
`timescale 1ns / 1ps
module enconde8to3_tb();
    wire [2:0] Y;
    reg [7:0] A;
    integer ii, errors;
    encode8to3_1 enc8to3_impl(.Out(Y),.In(A));
    initial begin
    errors = 0;
    for (ii=0; ii<256; ii=ii+1)
     begin
       A = ii;
       #2 ;                                          //  检测所有的错误案例
       if ( ( (A>8'b0)    && (A<2**Y ) )           // A 如果小于  2 的 Y 次幂
          || ( (A>8'b0)    && (A>=2**(Y+1)) ) )    //  或者 A 大于 2 的(Y+1)幂
          begin
             errors = errors+1;
             $display("Error: A=%b, A=%b",A,Y);
          end
       end
        $display("Test done, %d errors\n",errors);
        $stop(1);
    end
endmodule
```

实验验证：修改仿真测试文件中待测模块名称，将上述所有的编码器模块都进行仿真测试和综合实现，并上实验板可以验证；对比不同设计方式实现电路的硬件资源和性能差异，并思考原因。

3、奇偶检验器

奇偶校验是一种校验数据传输的正确性的方法，根据被传输的一组二进制代码的数位中 “1”的个数是奇数或偶数来进行校验；采用奇数的称为奇校验，反之，称为偶校验。采用何种校验是事先规定好的，通常专门设置一个奇偶校验位，用它使这组代码中 “1” 的个数为奇数或偶数；例如奇校验，当接收端收到这组代码时，校验 “1”的个数是否为奇数，从而确定传输代码的正确性。

n 位奇偶校验电路通常使用 n 个异或门级联，形成 n+1 个输入和 1 个输出电路。有两种不同结构的设计方法，一种是链式结构，一种是树状结构，如图 1.5 所示。

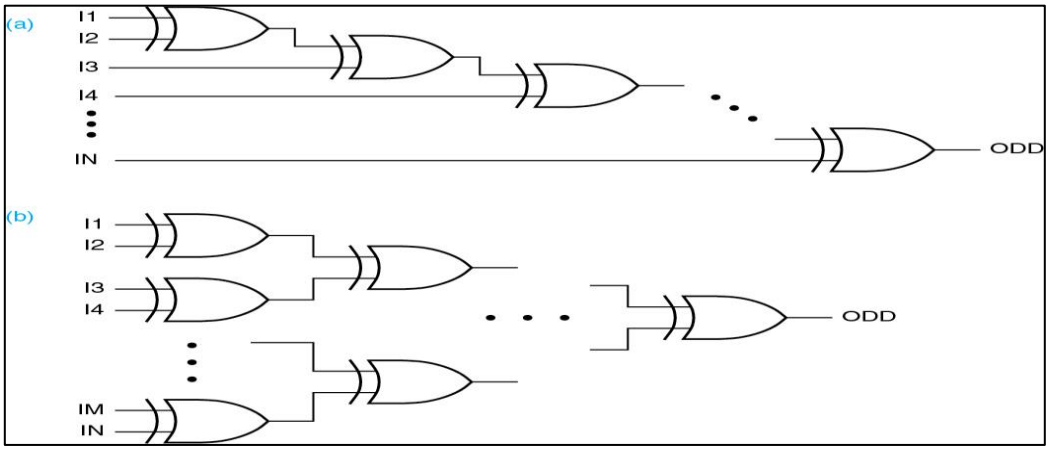


图 1.5 奇偶校验器的设计方式(a) 链式结构 (b) 树状结构

采用链式结构实现 4 位数据奇偶检验器的具体方法是：如果是奇校验，只要将该 4 位数据第一位和第二位进行异或，然后将得到的结果依次和第三位、第四位异或，这样得到的最后结果，就是奇校验位。如果是偶校验，将上面的奇校验位取反即可。

采用数据流建模方式使用异或门设计的 4 位奇偶校验器的模块代码如下：

```
module paritycheck4b(
      output odd,                          // 奇校验位
      output even，                        // 偶校验位
      input wire [3:0] In                  // 输入数据
   );
   assign odd= In [0] ^ In [1] ^ In [2] ^ In [3];       // assign odd = ^ In;
   assign even =~ odd;
endmodule
```

采用行为化建模方式使用 for 语句设计的 9 位奇偶校验器的模块代码如下：

```
module paritycheck9b(
      output reg odd,                          // 奇校验位
```

```
    output reg even，                    // 偶校验位
    input wire [8:0] In                  // 输入数据
);
    integer j;
    always @(*) begin
      odd=1'b0;                          //缺省输出值
      for(j=0; j<=8; j=j+1)
              if(In [j]) odd =~odd;      // odd= odd ^ In[j];
     end
endmodule
```

当 n 比较大的时候，采用链式结构实现奇偶校验器，会带来较大的传输延迟，而 Verilog 综合工具不能直接针对行为化建模优化出树形结构，因而需要在设计程序时就考虑实现方式。

在具体应用奇偶校验时，在发送端，奇偶校验电路计算每一组发送数据的奇偶校验位，将其与数据一起发送，在接收端，奇偶校验电路重新计算所接收数据的奇偶校验值，并将其与收到的校验值进行比较，如果二者相同，可以认为没有发生错误；如果二者不同，可以认为发生了传输错误。

实验验证：修改仿真测试文件中待测模块名称，将上述所有的编码器模块都进行仿真测试和综合实现，并上实验板可以验证，并对使用的硬件资源进行比对。

4、比较器

在计算机系统、设备接口以及许多其他应用中经常需要比较两个二进制字是否相等，实现这种功能的电路称为比较器，有些比较器将输入字解释为有符号或无符号数，还能指出字之间的算术关系（大于或小于），这种器件常称为数值比较器。

判断是否相等的比较器通常使用异或门或异或非门来实现。实现数值比较的其中一种方法就是用一个数减去另一个数，然后看结果。如果差为 0，说明两个数相等。如果无符号数，则根据借位可以表明小于或大于关系；对于带符号数的补码，则可以通过差的符号位来判断减数和被减数的大于关系。因此可以使用减法器来实现数值比较器。但是使用减法器会增加额外与数值比较器无关的逻辑电路。通常直接使用逻辑门电路来实现数值比较器，比基于减法器的比较器更小也更快。

以 8 位数值比较器为例，其原理图如图 1.6 所示，有相等 PEQQ、大于 PGTQ 和小于 PLTQ 三个输出，在实现时，只需要实现其中两个输出即可。

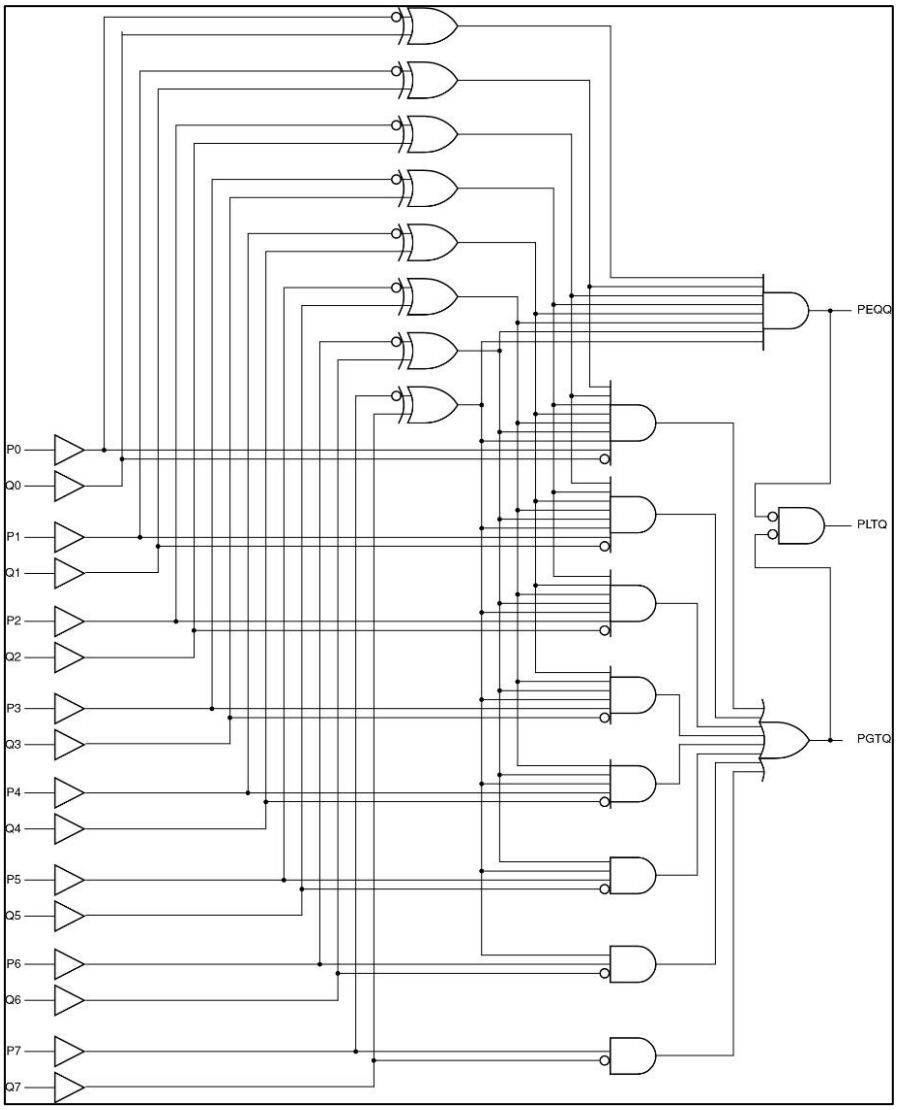


图 1.6 8 位数值比较器原理图

采用数据流建模方式实现的 8 位数值比较器的模块代码如下：

```
module compare8b_d(
    output   PGTQ, PEQQ, PLTQ;
    input [7:0] P, Q
  )
  assign PGTQ = ( (P > Q) ? 1'b1 : 1'b0 ) ;
  assign PEQQ = ( (P == Q) ? 1'b1 : 1'b0 ) ;
  assign PLTQ = ~PGTQ & ~PEQQ;          //如果写成( (P < Q) ? 1'b1 : 1'b0 )，将增加一个数值比较器
endmodule
```

采用行为化建模方式实现的 8 位数值比较器的模块代码如下：

```
module compare8b_a(
    output   PGTQ, PEQQ, PLTQ;
    input [7:0] P, Q
  )
always @ (P or Q)
```

```
    if (P == Q)
    begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (P > Q)
        begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else                                          //如果增加嵌套 if  P<Q  语句也需要增加 else 语句。
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
endmodule
```

采用伪随机数进行仿真测试的模块代码如下：

```
`timescale 1 ns / 100 ps
module compare8b_tb();
  reg [7:0] P, Q;
  wire PGTQ, PEQQ, PLTQ;
  integer ii, errors;
  compare8b_d UUT ( .P(P), .Q(Q), .PGTQ(PGTQ), .PEQQ(PEQQ), .PLTQ(PLTQ) );
  initial begin
    errors = 0;
    P = $random(1);              // 设置随机数模式
    for (ii=0; ii<10000; ii=ii+1) begin
      P = $random; Q = $random;
      #10 ;
      if ( (PGTQ) !== (P>Q) || (PLTQ) !== (P<Q) || (PEQQ) !== (P==Q) )
      begin
        errors = errors + 1;
        $display("P=%b(%0d), Q=%b(%0d), PGTQ=%b, PEQQ=%b, PLTQ=%b",
                  P, P, Q, Q, PGTQ, PEQQ, PLTQ);
      end;
    end
    $display("Test done, %0d errors", errors);
  end
endmodule
```

实验验证：修改仿真测试文件中待测模块名称，将上述所有的编码器模块都进行仿真测试和综合实现，并上实验板可以验证；对比不同设计方式实现电路的硬件资源和性能差异，并思考原因。

四、实验内容

1、4 路 3 位数据传输实验

多路选择器和多路分配器级联使用可以实现多通道数据的分时传送。在发送端通过多路选择器将各路数据分时送到总线，接收端再由多路分配器将总线上的数据适时分配到相应的输出端，从而传送到目的部件，其原理图如图 1.7 所示。

在上述多路选择器和多路分配器实验基础上实现4路3位数据传输的实验。数据传输有控制端S的编码决定，每路数据有3位，

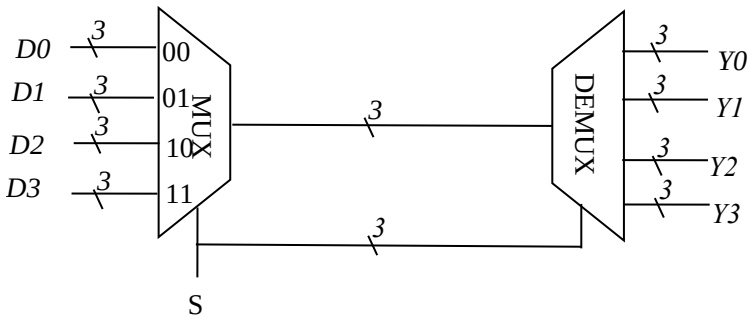


图 1.7　4 路 2 位数据传输原理图

4路3位数据传输的接口定义如下：

```
module trans4to4(
        output   [2:0] Y0,Y1,Y2,Y3,
        input    [2:0] D0,D1,D2,D3,
        input    [1:0] S
);
       // add your code here
endmodule
```

可以调用上文中的4路选择器和4路分配器模块。

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab1.zip里的trans4to4.v文件。

3、 点击添加仿真源码文件，加入lab1.zip里的trans4to4_tb.v文件。

4、 点击添加约束文件，加入lab1.zip里的trans4to4.xdc文件。

5、 根据实验要求，完成源码文件的设计。

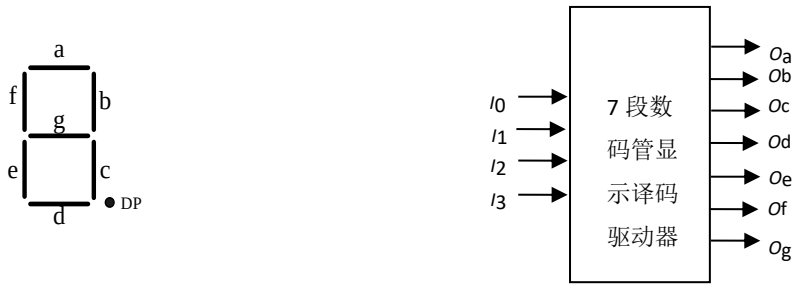
6、 对工程进行仿真测试，分析输入输出时序波形和控制台信息。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成比特流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

2、七段数码管实验

七段 LED 数码管是一种常用的显示元件，由译码驱动电路和 LED 数码管组成，主要用于显示十进制数字、小数点和部分字符。数字和字符的字形用 7 个发光二极管组成的线段来表示，这 7 个字段 a~g 分别由译码驱动器的 7 个输出端 *O*a~*O*g 控制是否发光，如图 1.8 所示。



（a）数码管编号　　（b）数码管驱动

图 1.8 七段数码管功能描述

数码管分为共阴极和共阳极两种类型，共阴极就是将八个 LED 的阴极连在一起，让其接低电平，这样给任何一个 LED 的另一端高电平，它便能点亮。而共阳极就是将八个 LED 的阳极连在一起，让其接高电平，这样，给任何一个 LED 的另一端低电平，它就能点亮。

Nexys A7-100T 实验开发板上有 8 个带小数点的七段数码管，这些数码管是共阳极连接方式，如果给数码管的某个引脚输入低电平则数码管会被点亮，若输入高电平则数码管变暗，如表 1.3 所示。每个数码管都有八个 LED 段组成，分别标识为 CA、CB、CC、CD、CE、CF 和 CG，小数点标识为 DP。这些 LED 又和 FPGA 的固定引脚相连，如 CA 和 L3 相连，DP 和 M4 相连（八个数码管的所有 CA 端同时和一个 FPGA 的引脚 L3 相连），如图 1.9 所示。如果在 FPGA 的 L3 端输出一个低电平 0，则数码管的 CA 段将被点亮，FPGA 的这些输出端默认值为 0，如果不希望某段数码管点亮，必须对引脚输入“1”。

表 1.3　共阳极数码管真值表

I_0 I_1 I_2 I_3	O_a O_b O_c O_d O_e O_f O_g	字形
0 0 0 0	0 0 0 0 0 0 1	0
0 0 0 1	1 0 0 1 1 1 1	1
0 0 1 0	0 0 1 0 0 1 0	2
0 0 1 1	0 0 0 0 1 1 0	3

0 1 0 0	1	0	0	1	1	0	0	4
0 1 0 1	0	1	0	0	1	0	0	5
0 1 1 0	0	0	0	0	0	1	0	6
0 1 1 1	0	0	0	1	1	1	1	7
1 0 0 0	0	0	0	0	0	0	0	8
1 0 0 1	0	0	0	0	1	0	0	9
1 0 1 0	0	0	0	1	0	0	0	A
1 0 1 1	1	1	0	0	0	0	0	b
1 1 0 0	0	1	1	0	0	0	1	C
1 1 0 1	1	0	0	0	0	1	0	d
1 1 1 0	0	1	1	0	0	0	0	E
1 1 1 1	0	1	1	1	0	0	0	F

每个数码管的共阳极端相当于是一个数码管的选通端，可以通过在 M1(AN7)、L1(AN6)、N4(AN5)、N2(AN4)、N5(AN3)、M3(AN2)、M6(AN1)和 N6(AN0)端输出低电平选中某个数码管，通过在 L3(CA)、N1(CB)、L5(CC)、L4(CD)、K3(CE)、M2(CF)、L6(CG)和 M4(DP)端输出低电平点亮此数码管的某段。

为了使八个数码管显示的每一个数字都能连续发光，这八个数码管应每隔 1 至 16ms 驱动一次，刷新频率约为 1 KHz 至 60Hz。例如，在 62.5Hz 的刷新方案中，整个数码管将每 16ms 刷新一次，并且每个数码管中的数字将显示 1/8 的刷新周期，即 2ms。

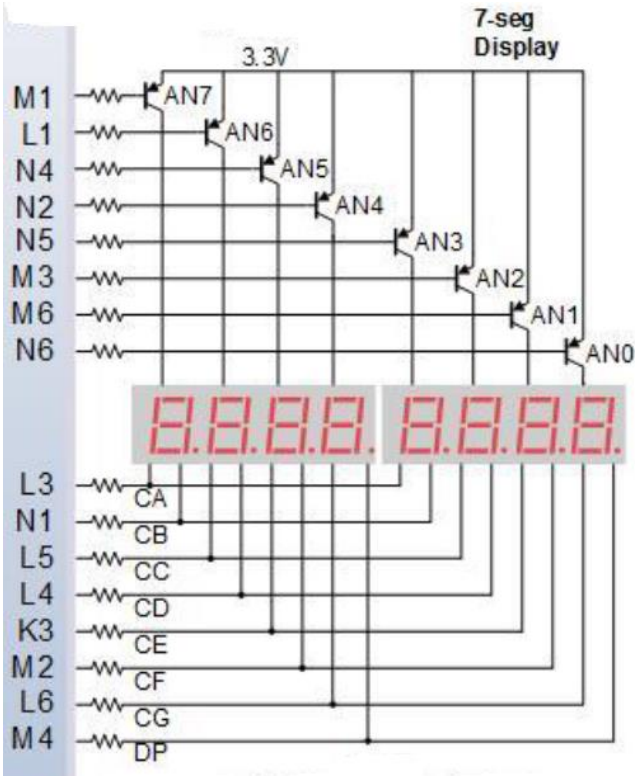


图 1.9 开发板数码管电路图

七段数码管电路的接口定义如下：

module dec7seg(

```
    //端口声明
    output   reg   [6:0] O_seg,   //7 位显示段输出
    output   reg   [7:0] O_led,   //8 个数码管输出控制
    input    [3:0] I,             //4 位数据输入，需要显示的数字
    input    [2:0] S              //3 位译码选择指定数码管显示
);
    // add your code here
endmodule
```

要求在 S 译码指定的数码管上显示数字 I。

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab1.zip里的dec7seg.v文件。

3、 点击添加仿真源码文件，加入lab1.zip里的dec7seg_tb.v文件。

4、 点击添加约束文件，加入lab1.zip里的dec7seg.xdc文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试，分析输入输出时序波形和控制台信息。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成比特流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

3、汉明码纠错实验

汉明码的主要思想是，将数据按某种规律分成若干组，对每组进行相应的奇偶检测，以提供多位校验信息，得到相应的故障字，根据故障字对发生的错误进行定位，并将其纠正。汉明校验码实质上就是一种多重奇偶校验码。

对于只能对单个位出错的情况进行定位和纠错的单纠错码（SEC），进行汉明校验的主要思想如下：将需要进行检/纠错的数据分成 i 组，每组对应 1 位校验位，共有 i 位校验位，因此，故障字为 i 位。若故障字为 0，表示无错；否则故障字的数值就是出错位在码字中的位置编号。除去 0 的情况，i 位故障字的编码个数为 2^i-1，因此构造的码字最多有 2^i-1 位，例如，当 i=3 时，码字可以有 7 位，其中 3 位为校验位，4 位为数据位。为了方便判断码字中出错的是校验位还是数据位，可将校验位的位置编号设为 2 的幂次，即校验位排在第 1（001）、2（010）、4（100）、… 的位置上，其余位置上为数据位。这样，当故障字中只有一位为 1 时，说明是校验位出错，否则就是数据位出错。例如，当 i=3 时，假设校验码为 $P_3P_2P_1$，数据信息为 $M_4M_3M_2M_1$，则码字排列为 $P_1P_2M_1P_3M_2M_3M_4$。通常把上述由数据位和校验位构成的码字称为汉明码。表 1.4 给出了 7 位汉明码的故障字和出错情况的对应关系。第 1 组的故障位 S1 由校验位 P1 和数据

位 M1、M2、M4 生成，第 2 组的故障位 S2 由校验位 P2 和数据位 M1、M3、M4 生成、第 3 组的故障位 S3 由校验位 P3 和数据位 M2、M3、M4 生成。

表 1.4　7 位汉明码的故障字和出错情况对应关系

序号 / 分组 \ 含义	1	2	3	4	5	6	7	故障字	正确	出错位 1 2 3 4 5 6 7
	P_1	P_2	M_1	P_3	M_2	M_3	M_4			
第3组				√	√	√	√	S_3	0	0 0 0 1 1 1 1
第2组		√	√			√	√	S_2	0	0 1 1 0 0 1 1
第1组	√		√		√		√	S_1	0	1 0 1 0 1 0 1

假设在终部件得到的数据位 M′为 M4M3M2M1，校验位 P′′为 P3P2P1，每组采用偶校验，则根据 M′得到 P′的每一位如下：

$P_1' = M_1 \oplus M_2 \oplus M_4$

$P_2' = M_1 \oplus M_3 \oplus M_4$

$P_3' = M_2 \oplus M_3 \oplus M_4$

因为故障字 S=P′⊕P′′，因此，根据 P′和 P′′得到故障字的每一位如下：

$S_1 = M_1 \oplus M_2 \oplus M_4 \oplus P_1$

$S_2 = M_1 \oplus M_3 \oplus M_4 \oplus P_2$

$S_3 = M_2 \oplus M_3 \oplus M_4 \oplus P_3$

因此，在终部件的汉明码检/纠错电路只要根据在终部件得到的数据位 M4M3M2M1 和校验位 P3P2P1 形成的码字，按表 1.4 所示的方式划分成 3 组，每组按照上述偶校验方式，得到每一组的故障位 Si，由故障位构成的故障字 S3S2S1 的值就能确定码字中哪一位发生了错误。图 1.10 给出了 7 位汉明码检/纠错电路原理图，由偶校验器、译码器和异或门构成。

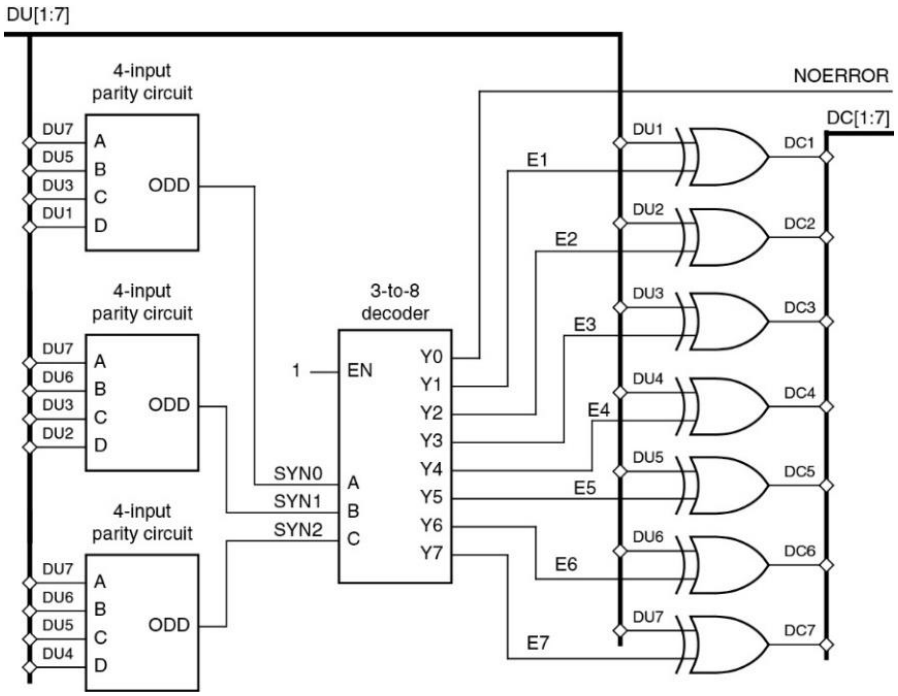


图 1.10　7 位汉明码检/纠错电路原理图

在图 1.10 中 DU[1:7]是 4 位数据位 M4M3M2M1 和 3 位校验位 P3P2P1 构成的 7 位汉明码，码字 DU[1:7]=P1P2M1P3M2M3M4。例如，当输入 DU[1:7]为 1000001 时，说明数据位 M4M3M2M1 为 1000，检验位 P3P2P1 为 001，根据上述公式得到故障字各位如下：

S1 = M1⊕M2⊕M4⊕P1= 0⊕0⊕1⊕1=0

S2 = M1⊕M3⊕M4⊕P2= 0⊕0⊕1⊕0 =1

S3 = M2⊕M3⊕M4⊕P3= 0⊕0⊕1⊕0 =1

因此，故障字 S3S2S1 为 110，说明码字中第 6 位（对应 DU[6]，即 M3）发生错误。在图 1.10 中， A 组偶校验结果为 S1=0，B 组偶校验结果为 S2=1，C 组偶校验结果为 S3=1，对应位置编号为 S3S2S1=110，3-8 译码器输入端 C、B、A 分别为 1、1、0，其输出 Y6 为 1，其余输出为 0，Y6 与 DU[6]异或生成 DU[6]的相反值，使得出错位得到纠正。

请设计一个能够对上述 7 位汉明码进行纠错的电路，通过拨动开关输入 7 位数据，纠错后的正确结果输出到 led1-led7 指示灯；错误位显示在某个七段数码管上，由错误位指定，没有错误则在 0 号数码管上显示 0；校验正确标志位输出到三色 led 数码管的绿色显示灯。（改成 8 位信息位，在七段数码管上显示错误地址，用三色指示灯表示是否正确）

7 位汉明码纠错电路的接口定义如下：

```
module hamming7check(
    output reg [7:1] DC,              //纠错输出 7 位正确的结果
    output reg    NOERROR,            //校验结果正确标志位
    output [2:0] ERR_adr,             //错误位地址
    output reg[6:0] O_seg,            //7 段数据
    output reg [7:0] an,              //数码管选择
    input    [7:1] DU                 //输入 7 位汉明码
);
        // add your code here
endmodule
```

可以调用 4 位奇偶校验器和 3-8 译码器模块。

请根据上述描述，按照下列步骤完成实验。

1、 使用Vivado创建一个新工程。

2、 点击添加设计源码文件，加入lab1.zip里的hamming7check.v文件。

3、 点击添加仿真源码文件，加入lab1.zip里的hamming7check_tb.v文件。

4、 点击添加约束文件，加入lab1.zip里的hamming7check.xdc文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试，分析输入输出时序波形和控制台信息。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成比特流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1. 设计 32 位比较器，综合后分析资源占用情况。
2. 设计 32 位译码器，综合后分析资源占用情况。
3. 利用 8 个数码管来展示你的学号，每秒移动 1 位，实现滚动显示。
4. 如何设计 8 位信息位的汉明码纠错电路。