数字电子技术实验指导书

一、常见组合逻辑电路Verilog实现

1.1 加法器

1.1.1 半加器

不考虑有来自低位的进位将两个1位二进制数相加称为半加。实现半加运算的电路称为半加器。

半加器真值表如下，其中A、B是两个加数，S是相加的和，CO是向高位的进位。

A	B	S	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

半加器逻辑表达式如下：

$$\begin{cases} S = A \oplus B \\ CO = AB \end{cases}$$

半加器verilog实现代码如下：

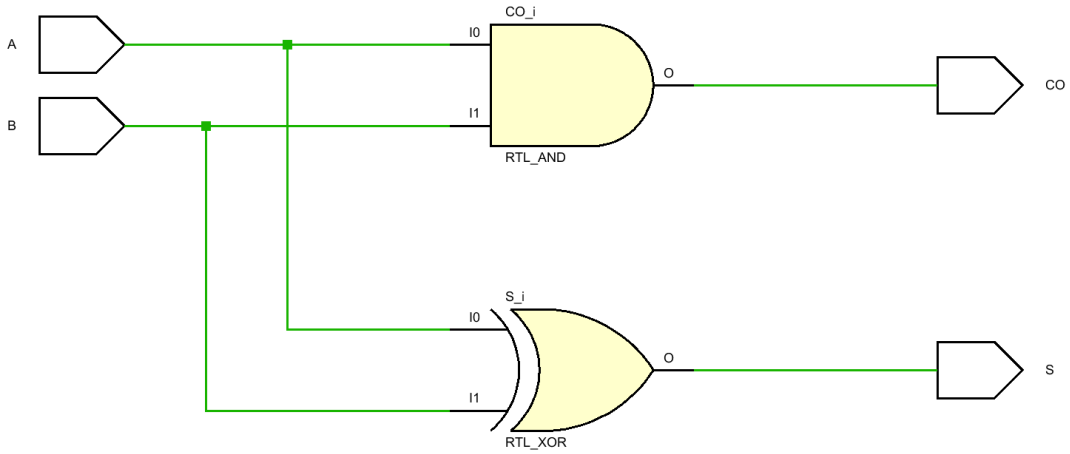
```
module half_adder(
      input A,
      input B,
      output S,
      output CO
);

    assign S = A ^ B;
    assign CO = A  & B;

endmodule
```

半加器仿真代码如下：

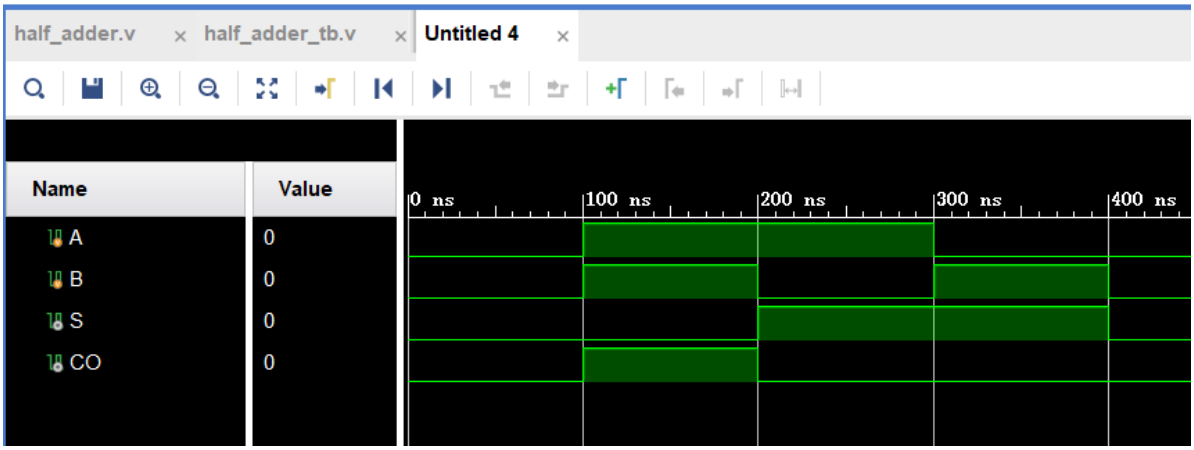
```
module half_adder_tb();
reg    A;
reg    B;
wire   S;
wire   CO;
```

```
initial begin
    A = 0;
    B = 0;
    #100
    A = 1;
    B = 1;
    #100
    A = 1;
    B = 0;
    #100
    A = 0;
    B = 1;
    #100
    A = 0;
    B = 0;
end

half_adder  INSTANCE_half_adder(
    .S  ( S  ),
    .CO ( CO ),
    .A  ( A  ),
    .B  ( B  )
);
```

RTL逻辑电路如下：



行为级仿真结果如下：



1.1.2 全加器

将两个多位二进制相加时，除了最低位以外，每一位都应该考虑来自低位的进位，即，将两个对应位的加数和来自低位的进位3个数相加，这种运算称为全加，所用的电路称为全加器。

全加器真值表如下，其中A、B是两个加数，CI是输入端的进位，S是相加的和，CO是输出端的进位。

CI	A	B	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器逻辑表达式如下：

$$\begin{cases} S = (A'B'CI' + AB'CI + A'BCI + ABCI')' \\ CO = A'B' + B'CI' + A'CI'' \end{cases}$$

全加器verilog实现代码如下：

```
module full_adder(
    input    A,
    input    B,
    input    CI,
    output   S,
```

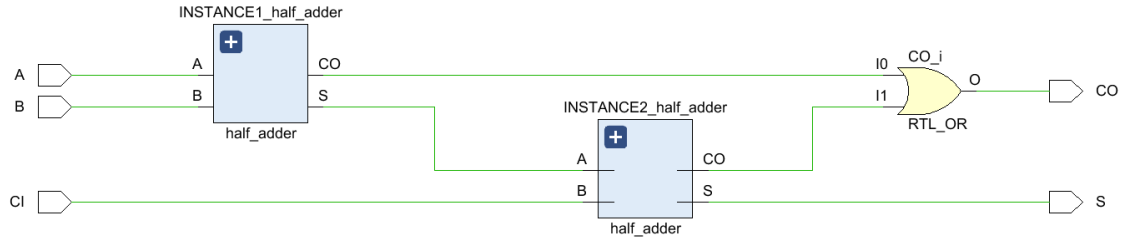
```
    output   CO
    );
wire c1,s1,c2;

half_adder INSTANCE1_half_adder(
    .A(A),
    .B(B),
    .S(s1),
    .CO(c1)
    );

half_adder INSTANCE2_half_adder(
    .A(s1),
    .B(CI),
    .S(S),
    .CO(c2)
    );
assign CO = c1 || c2;

endmodule
```

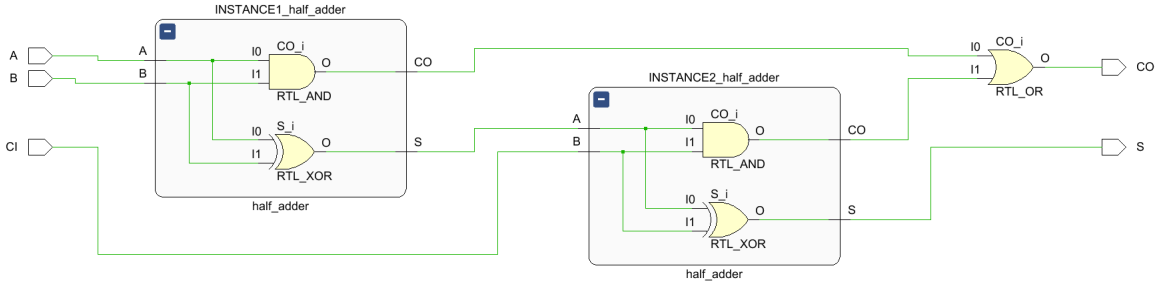
全加器仿真代码如下：

```
module full_adder_tb();
reg    A;
reg    B;
reg    CI;
wire   S;
wire   CO;

initial begin
    CI = 0;
    A = 0;
    B = 0;
    #100
    CI = 0;
    A = 0;
    B = 1;
    #100
    CI = 0;
    A = 1;
    B = 0;
    #100
    CI = 0;
    A = 1;
    B = 1;
    #100
    CI = 1;
    A = 0;
    B = 0;
    #100
    CI = 1;
    A = 0;
    B = 1;
```

```
    #100
    CI = 1;
    A = 1;
    B = 0;
    #100
    CI = 1;
    A = 1;
    B = 1;
end

full_adder  INSTANCE_full_adder(
    .A  (A ),
    .B  (B ),
    .CI (CI),
    .S  (S ),
    .CO (CO)
);

endmodule
```

全加器RTL逻辑电路如下：





全加器行为级仿真结果如下：

1.2 编码器

1.2.1 引入

假设你正在家里，有多件事情需要你同时处理，比如接听电话、煮咖啡、和看电视。这些活动都可以视为输入信号。
优先编码器在这种情况下会发挥作用。假设接听电话的优先级最高，因为电话铃声最响，且可能带来紧急情况。煮咖啡次之，咖啡机在自动煮咖啡时也会发出声音，但不会像电话那么紧急。看电视最低，电视机的声音通常比较轻。当你同时听到电话铃声、咖啡机声音和电视机声音时，你会优先选择接听电话，因为它的优先级最高。这就是优先编码器的工作原理，它根据输入信号的优先级进行编码和输出。

1.2.2 逻辑表达

我们以8-3优先编码器为例，下面给出其真值表。X0、X1、X2、X3、X4、X5、X6、X7为输入，Y0、Y1、Y2为输出。

X7	X6	X5	X4	X3	X2	X1	X0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

1.2.3 具体实现

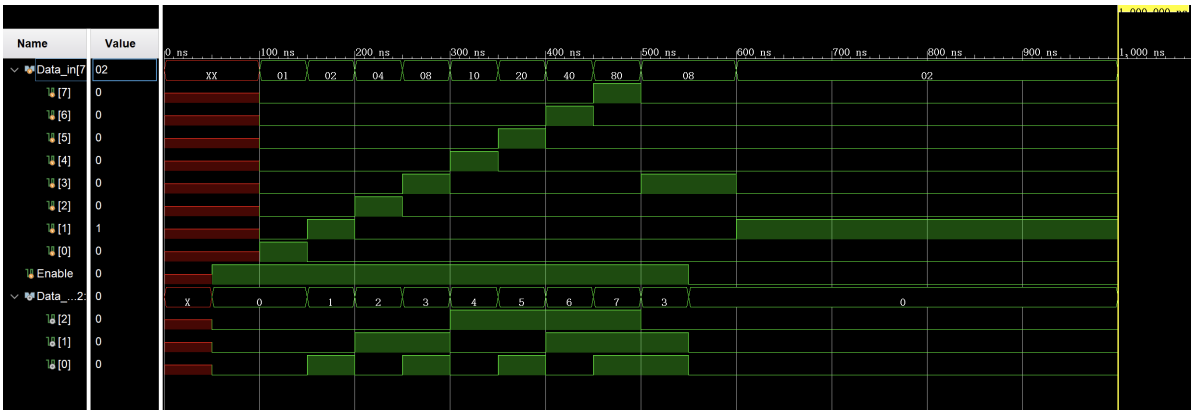
verilog代码实现如下：

```
module encoder(
   input      [7:0] Data_in,
   input            Enable,
   output reg [2:0] Data_out
   );

always@(*) begin
    if(Enable==0)
        Data_out=3'b000;
    else begin
        case(Data_in)
          8'b00000001 :Data_out=3'b000;
          8'b00000010 :Data_out=3'b001;
          8'b00000100 :Data_out=3'b010;
          8'b00001000 :Data_out=3'b011;
          8'b00010000 :Data_out=3'b100;
          8'b00100000 :Data_out=3'b101;
          8'b01000000 :Data_out=3'b110;
          8'b10000000 :Data_out=3'b111;
```

```
            default:Data_out=3'b000;
        endcase
    end
end
endmodule
```

编写testbench文件并且仿真，得到波形。

```
module tb_encoder();
reg  [7:0] Data_in;
reg        Enable;
wire [2:0] Data_out;

initial begin
   #50 Enable=1'b1;
   #50 Data_in=8'b00000001;
   #50 Data_in=8'b00000010;
   #50 Data_in=8'b00000100;
   #50 Data_in=8'b00001000;
   #50 Data_in=8'b00010000;
   #50 Data_in=8'b00100000;
   #50 Data_in=8'b01000000;
   #50 Data_in=8'b10000000;
   #50 Data_in=8'b00001000;
   #50 Enable=1'b0;
   #50 Data_in=8'b00000010;
end

encoder u_tb_encoder(
    .Data_in   (Data_in ),
    .Enable    (Enable  ),
    .Data_out  (Data_out)
     );

endmodule
```



1.3 解码器

3-8译码器

真值表

A	B	C	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

3-8译码器Verilog实现如下：

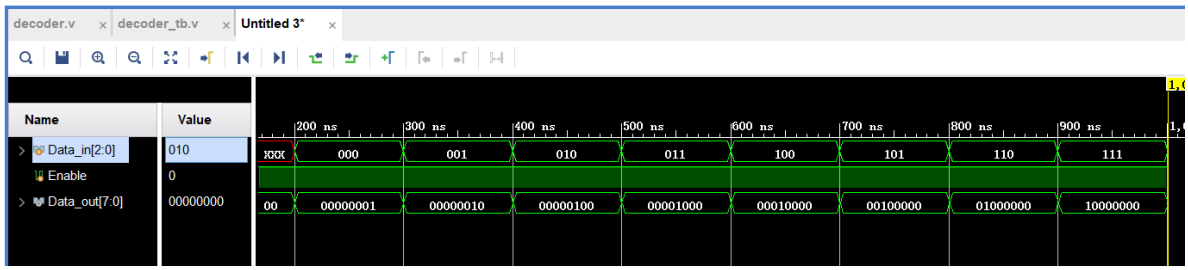
```
`timescale 1ns / 1ps

module decoder(
   input      [2:0] Data_in,
   input            Enable,
   output reg [7:0] Data_out
   );

always@(*) begin
    if(Enable==0)
        Data_out=8'b00000000;
    else begin
        case(Data_in)
          3'b000:Data_out=8'b00000001;
          3'b001:Data_out=8'b00000010;
          3'b010:Data_out=8'b00000100;
          3'b011:Data_out=8'b00001000;
          3'b100:Data_out=8'b00010000;
          3'b101:Data_out=8'b00100000;
          3'b110:Data_out=8'b01000000;
          3'b111:Data_out=8'b10000000;
          default:Data_out=8'b00000000;
        endcase
    end
end
endmodule
```

3-8译码器仿真代码如下：

```
`timescale 1ns / 1ps
module decoder_tb();
reg  [2:0] Data_in;
reg        Enable;
```

```
wire [7:0] Data_out;

initial begin
   #100 Enable=1'b1;
   #100 Data_in=3'b000;
   #100 Data_in=3'b001;
   #100 Data_in=3'b010;
   #100 Data_in=3'b011;
   #100 Data_in=3'b100;
   #100 Data_in=3'b101;
   #100 Data_in=3'b110;
   #100 Data_in=3'b111;
   #100 Data_in=3'b010;
   Enable=0;
   #100 Data_in=3'b111;
end

decoder u_decoder_tb(
    .Data_in   (Data_in ),
    .Enable    (Enable  ),
    .Data_out  (Data_out)
     );

endmodule
```

3-8译码器仿真结果如下：



1.4 数据选择器

1.4.1 引入

现在我们举例说明数据选择器的作用。假设你的面前有一个自动售货机，它可以向你提供可乐、雪碧、芬达、七喜四种饮料。此时，你需要做出一次选择，购买哪一种饮料，并且把你的想法告诉自动售货机。假设自动售货机内部采用二进制的形式表示这四种饮料，具体表示方法如下表所示：

种类	代码
可乐	00
雪碧	01
芬达	10
七喜	11

由表格可知，如果你想要购买雪碧，则需要输入01。这便是数据选择器的功能：在选择信号的控制下，从多个输入中选择一个输出。

1.4.2 逻辑表达

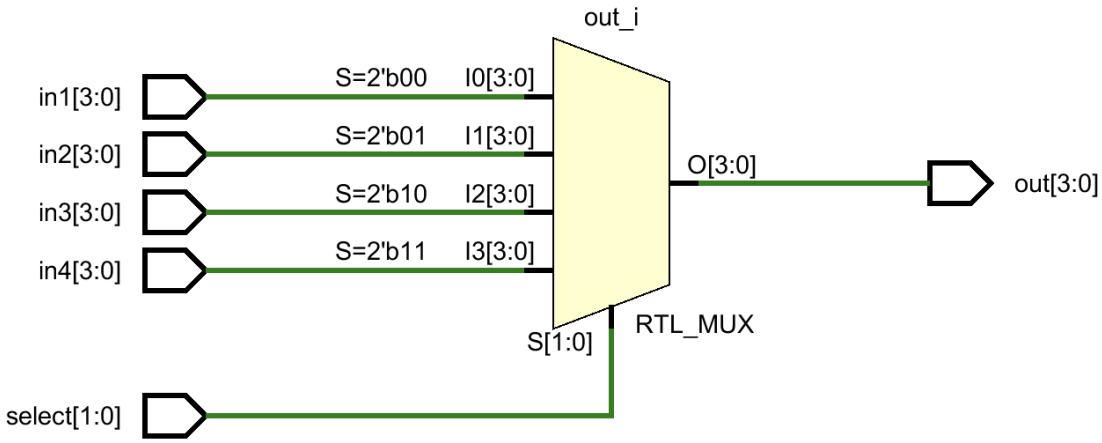
数据选择器在逻辑功能上相当于多个输入的单刀多掷开关。它的工作原理是，根据选择输入端（S）的状态，确定输出端（Y）的值。具体来说，如果S端有n个选择输入端，那么可以选择2^n个输入通道中的一个进行输出。我们将引入中的例子进行抽象化。引入中的例子实际上是一个四选一的数据选择器，即有4个输入通道A、B、C、D和2个输入端，保证每一个输入通道都可以被选择。根据输入端S0、S1的状态，便可以确定输出值Y。真值表如下表所示：

S1	S0	Y
0	0	A
0	1	B
1	0	C
1	1	D

1.4.3 具体实现

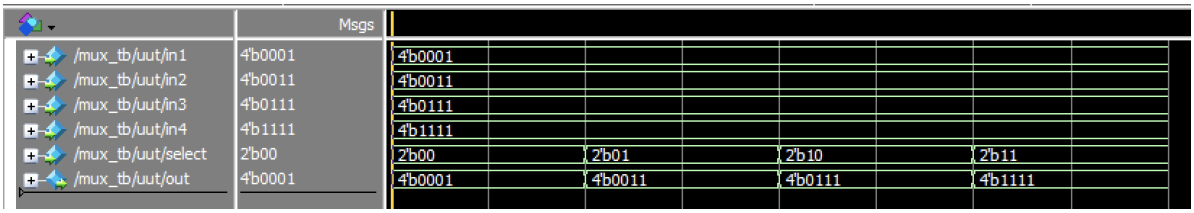
```
module mux (
    input wire [3:0] in1, in2, in3, in4,//输入信号
    input wire [1:0] select,//控制信号
    output reg [3:0] out//输出信号
);
    always@* begin
        case (select)
            2'b00: out = in1;
            2'b01: out = in2;
            2'b10: out = in3;
            2'b11: out = in4;
        default: out = 4'bx;
        endcase
    end
endmodule
```

这里给出的示例代码使用的是case语句进行描述，希望同学们可以举一反三，采用assign语句或条件语句实现四选一数据选择器。
四选一数据选择器的RTL级电路如下图所示。



同时，给出Testbench代码与仿真波形。

```
module mux_tb;

    reg [3:0] in1, in2, in3, in4;
    reg [1:0] select;
    wire [3:0] out;

    initial begin
    in1 = 4'b0001;
    in2 = 4'b0011;
    in3 = 4'b0111;
    in4 = 4'b1111;
    select = 2'b00;

    #10 select = 2'b01;
    #10 select = 2'b10;
    #10 select = 2'b11;
    #10 $stop;
    end
    mux uut(
        .in1 (in1) , .in2 (in2) , .in3 (in3), .in4 (in4), .select(select),.out
(out)
    );
endmodule
```



根据波形图可以清楚的看出选择信号select与输出信号out的一一对应关系。

1.5 数值比较器

1.5.1 引入

数值比较器，顾名思义，用于比较两个或多个数值的大小。让我们设想一个场景，学生们在参加一个比赛，这个比赛需要他们比较两队的得分。现在，得分已经计算出来了，分别是5分和10分。我们可以通过简单的减法操作，将两个数字相减，得到一个结果。根据这个结果的正负，我们可以判断哪个队伍得分更高。在这个例子中，我们使用了一个简单的数值比较器。数值比较器是一种电子设备，它可以根据两个输入值的大小来判断输出什么值。在这个例子中，我们输入了两个整数5和10，然后通过减法操作得到一个结果。根据这个结果的正负，我们可以判断哪个队伍得分更高。
数值比较器是一种非常重要的电子设备，它在很多领域都有广泛的应用。例如，在计算机中，它被用于比较寄存器中的值；在工业控制中，它被用于比较传感器读数；在电子游戏中，它被用于比较得分等等。因此，学习和理解数值比较器的原理和应用是非常重要的。

1.5.2 逻辑表达

我们以两个一位二进制数值为输入的情况举例，给出真值表，如下表所示：

A	B	A>B	A=B	A<B
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

上述真值表只为了表达简单的逻辑关系，不用于数值比较器的通用真值表。

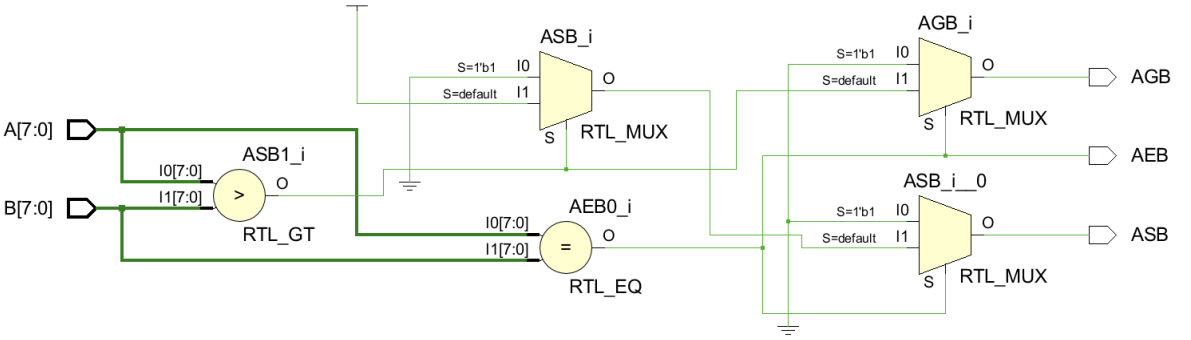
1.5.3 具体实现

本次实验编写一个程序，输入为两个八位二进制数，输出为这两个数的大小关系。

```
module compare(A,B,AEB,ASB,AGB);
input [7:0] A,B;
output reg AEB,ASB,AGB;//AEB为A=B标识位，ASB为A<B标识位，AGB为A>B的标识位
always@(A,B)
begin
    if(A==B)
        begin
        AEB=1'b1;
        ASB=1'b0;
        AGB=1'b0;
        end
     else
     if(A>B)
        begin
        AEB=1'b0;
        ASB=1'b0;
        AGB=1'b1;
        end
     else
        begin
        AEB=1'b0;
        ASB=1'b1;
        AGB=1'b0;
        end
end
```

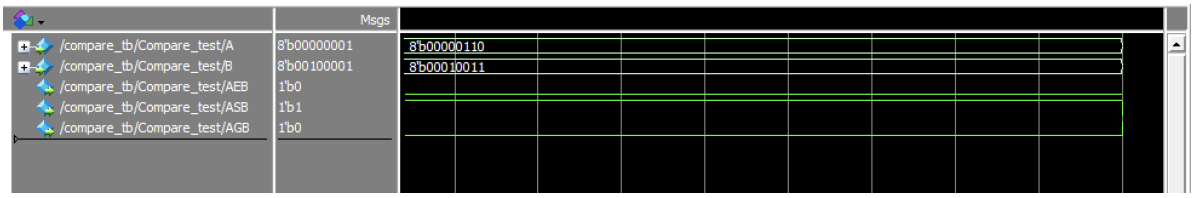
```
endmodule
```

RTL级电路如下图所示。



接下来编写testbench代码对程序进行验证，并且生成波形图。

```
module compare_tb;
    reg [7:0] A,B;
    wire AEB,ASB,AGB;
        initial begin
        A=8'b0;
        B=8'b0;
        end
    always #1 A={$random}%(50);//生成0-49间的随机数
    always #1 B={$random}%(49);//生成0-48间的随机数
    compare Compare_test(.A(A),.B(B),.AEB(AEB),.ASB(ASB),.AGB(AGB));
endmodule
```



从波形图中可以看出此时随机生成的A是比B小的，所以ASB标识位是置1的，AEB和AGB标识位是置0的。

二、常见时序逻辑电路Verilog实现

2.1 计数器

计数器是数字设备的基本逻辑部件，其主要功能是记录输入脉冲的个数。

按照二进制计数器规则，本实验设计为时钟10ns为一周期，初始化时令rst端口置1，持续15ns后归0，并不再变化。计数器的计数最大值为8，每当计数器满载，下一次时钟的上升沿会使计数器从0开始计数。

计数器verilog实现

```
`timescale 1ns / 1ns

module counter(
    clk,
    rst,
    data_out
    );

parameter N = 9;
```

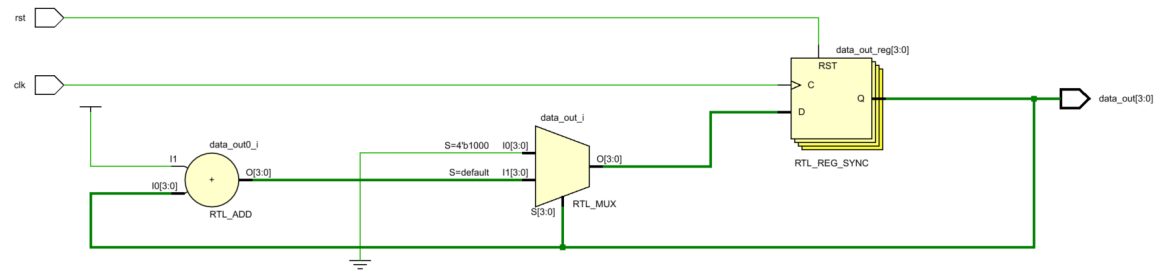
```
parameter DWIDTH = 4;
input clk;
input rst;
output reg [DWIDTH-1:0] data_out;

always @ ( posedge clk )
begin
    if( rst ) begin
        data_out <= 'b0;
    end
    else if( data_out == (N-1) )begin
        data_out <= 'b0;
    end
    else begin
        data_out <= data_out + 1'b1;
    end
end

endmodule
```

计数器仿真

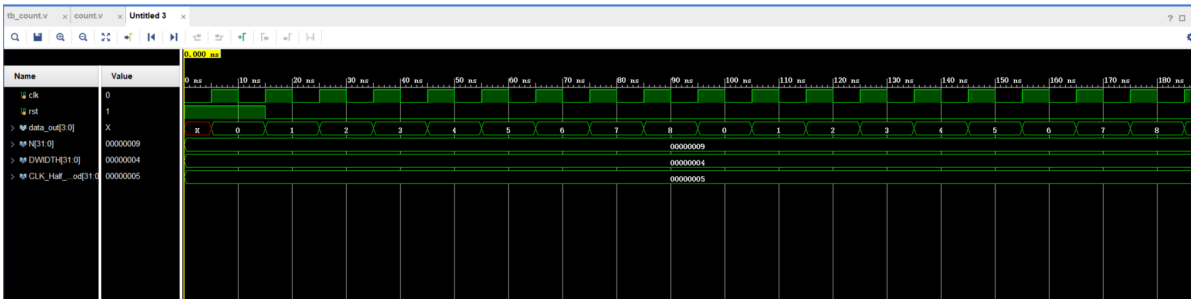
```
`timescale 1ns / 1ns

module counter_tb();

parameter N = 9;
parameter DWIDTH = 4;
parameter CLK_Half_Period = 5;

reg clk;
reg rst;
wire [DWIDTH-1:0] data_out;

counter #(
    .N(N),
    .DWIDTH(DWIDTH)
)
U1(
    .clk(clk),
    .rst(rst),
    .data_out(data_out)
    );

initial begin
    rst = 1;
    clk = 0;
    $display("TestBench is start!");
    #15;
    rst = 0;
    #200;
    $stop;
end

always #CLK_Half_Period clk = ~clk;
```

```
endmodule
```

RTL逻辑电路如下：

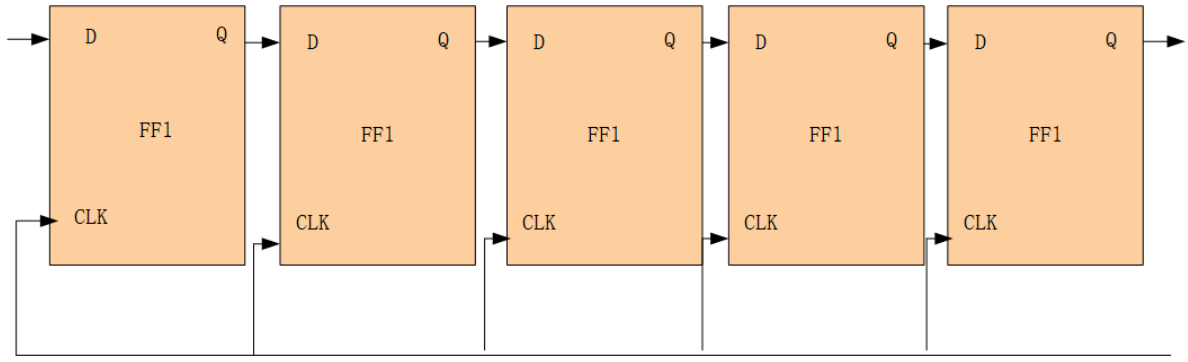


行为级仿真结果如下：



2.2 移位寄存器

移位寄存器是级联的触发器，其中一个触发器的输出引脚Q连接到下一个触发器的数据输入引脚D。因为所有触发器都在同一时钟工作，所以存储在移位寄存器中的位阵列将移动一个位置。本次实验设计的是左移位寄存器。



初始化完成后，即时钟开始置位完成后的第一个周期，此时加载变量由0变为1，移位寄存器开始工作，开始依次输出十六进制下的B、7、E、D，并不断循环。

移位寄存器十六进制输出对应的二进制数：

十六进制	二进制
B	1011
7	0111
E	1110
D	1101

移位寄存器

移位寄存器verilog实现：

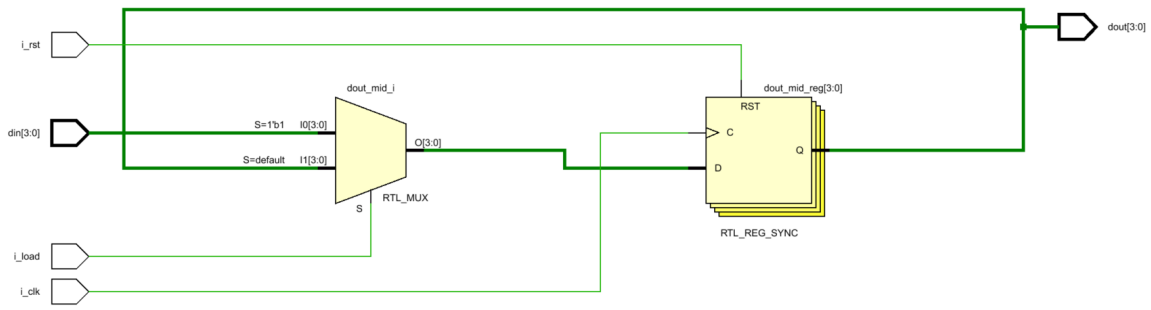
```
`timescale 1ns / 1ps

module cycle_left_register #(parameter MSB = 4)(
 input [MSB - 1 : 0] din,
 input i_rst,
 input i_load,
 input i_clk,
 output  [MSB - 1 : 0] dout
    );

 reg [MSB - 1 : 0] dout_mid;
 always@(posedge i_clk) begin
  if(i_rst) begin
   dout_mid <= 'd0;
  end
  else if(i_load) begin
   dout_mid <= din;
  end
  else begin
   dout_mid <= {dout_mid[MSB - 2 : 0], dout_mid[MSB - 1]};
  end
 end
 assign dout = dout_mid;


endmodule
```

移位寄存器仿真代码：

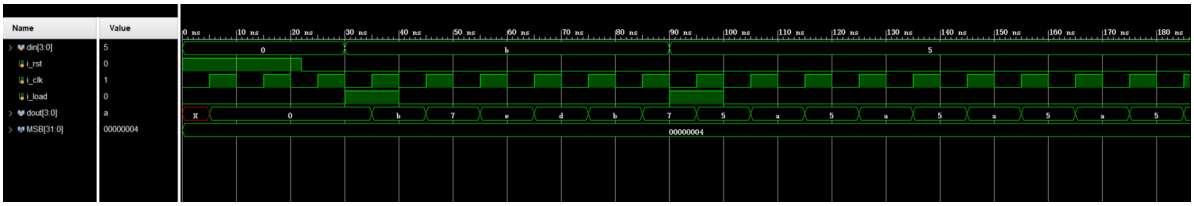
```
`timescale 1ns / 1ps

module cycle_left_register_tb(

    );
 parameter MSB = 4;

 reg [MSB - 1 : 0] din;
 reg i_rst;
 reg i_clk;
 reg i_load;
 wire [MSB - 1 : 0] dout;
```

```
 //generate clock
 initial begin
  i_clk = 0;
  forever begin
   #5 i_clk = ~i_clk;
  end
 end

 //generate rst and input data
 initial begin
  i_rst = 1;
  din = 0;
  i_load = 0;

  # 22

  i_rst = 0;
  @(negedge i_clk) begin
  din = 'b1011;
  i_load = 1;
  end

  @(negedge i_clk) begin
   i_load = 0;
  end

  repeat(5) @(posedge i_clk);

  @(negedge i_clk) begin
   din = 'd0101;
   i_load = 1;
  end

  @(negedge i_clk) i_load = 0;

  repeat(4) @(posedge i_clk);

  #10 $finish;

 end

 initial
      $monitor (" i_rst = %0b, i_load = %0b, din = %b, dout = %b", i_rst,
i_load, din, dout);

 cycle_left_register #(.MSB(MSB))inst_cycle_left_register(
  .i_clk(i_clk),
  .i_rst(i_rst),
  .i_load(i_load),
  .din(din),
  .dout(dout)
  );



endmodule
```

RTL逻辑电路如下：



行为级仿真结果如下：



2.3 顺序脉冲发生器

产生4路顺序脉冲

顺序脉冲发生器verilog代码：

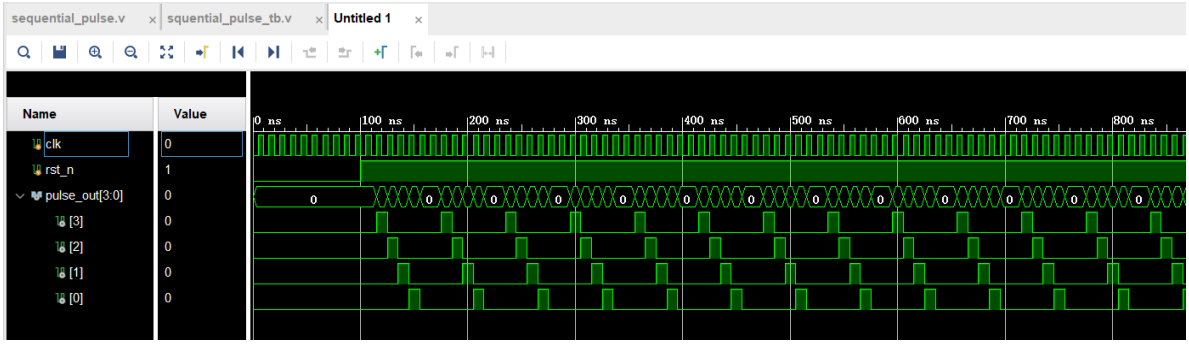
```
`timescale 1ns / 1ps

module sequential_pulse(
    input       clk,
    input       rst_n,

    output reg [5:0] pulse_out
    );

    //reg [3:0] pulse_out;
    reg [2:0] pulse_state;
    reg [2:0] next_pulse_state;

always@(pulse_state)begin
    case (pulse_state)
        3'b000: begin
        pulse_out<=6'b100000;
        next_pulse_state<=3'b001;
        end

        3'b001: begin
        pulse_out<=6'b010000;
        next_pulse_state<=3'b010;
        end

        3'b010: begin
```

```
        pulse_out<=6'b001000;
        next_pulse_state<=3'b011;
        end

        3'b011: begin
        pulse_out<=6'b000100;
        next_pulse_state<=3'b100;
        end

        3'b100: begin
        pulse_out<=6'b000010;
        next_pulse_state<=3'b101;
        end

        3'b101: begin
        pulse_out<=6'b000001;
        next_pulse_state<=3'b000;
        end
     endcase
end

always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
        pulse_state<=3'b000;
    else
        pulse_state<=next_pulse_state;
end

endmodule
```

顺序脉冲发生器仿真代码：

```
`timescale 1ns / 1ps


module sequential_pulse_tb();
reg clk;
reg rst_n;
wire [5:0] pulse_out;

//=== for clk ====
initial begin
    clk = 1'b0;
    forever begin
        #5 clk =~clk; //100MHz
    end
end
//==== main ===
initial begin
    rst_n = 1'b0;
    #100;
    rst_n = 1'b1;
    #100;
end
```

```
sequential_pulse u_sequential_pulse(
      .clk              (clk  ),
      .rst_n            (rst_n),
      .pulse_out        (pulse_out)
       );

endmodule
```



2.4 序列信号发生器

有限状态机方式实现01011010序列信号产生器

序列信号发生器verilog实现代码如下：

```
//有限状态机方式实现01011010序列信号产生器
module sequence_signal(
    input clk,
    input rst_n,
    output reg dout
     );
//    reg dout;
    reg [3:0] pre_state, next_state;


    always @(posedge clk or negedge rst_n)
        begin
            if(rst_n == 0)
                pre_state <= 3'b000;
            else
                pre_state <= next_state;
        end

    always @(pre_state)
        begin
            case(pre_state)
                 3'b000:
                    begin
```

```
                        dout = 1'b0;
                        next_state <=  3'b001;
                    end
                3'b001:
                    begin
                        dout = 1'b1;
                        next_state =  3'b010;
                    end
                 3'b010:
                    begin
                        dout = 1'b0;
                        next_state = 3'b011;
                    end
                3'b011:
                    begin
                        dout = 1'b1;
                        next_state = 3'b100;
                    end
                3'b100:
                    begin
                        dout = 1'b1;
                        next_state = 3'b101;
                    end
                3'b101:
                    begin
                    dout = 1'b0;
                    next_state =3'b110;
                    end
                3'b110:
                    begin
                    dout = 1'b1;
                    next_state =3'b111;
                    end
                3'b111:
                    begin
                    dout = 1'b0;
                    next_state = 3'b000;
                    end
                default: next_state = 3'b000;
            endcase
        end

endmodule
```

序列信号发生器仿真代码：

```
`timescale 1ns / 1ps

module sequence_signal_tb;
    reg clk;
    reg rst_n;

    wire dout;
```

```
    always
        #10
         clk = ~clk;

    initial begin
            clk = 1'b0;
            rst_n = 1'b1;
            #10
            rst_n = 1'b0;
            #10
            rst_n = 1'b1;
        end
    sequence_signal U1(.clk(clk), .rst_n(rst_n), .dout(dout));

endmodule
```