第七章 有限状态机和可综合风格的Verilog HDL

前言

由于Verilog HDL和 VHDL 行为描述用于综合的历史还只有短短的几年，可综合风格的Verilog HDL 和VHDL的语法只是它们各自语言的一个子集。又由于HDL的可综合性研究近年来非常活跃，可综合子集的国际标准目前尚未最后形成，因此各厂商的综合器所支持的HDL子集也略有所不同。本教材中有关可综合风格的Verilog HDL的内容，我们只着重介绍RTL级、算法级和门级逻辑结构的描述，而系统级（数据流级）的综合由于还不太成熟，暂不作介绍。由于寄存器传输级（RTL）描述是以时序逻辑抽象所得到的有限状态机为依据的，所以把一个时序逻辑抽象成一个同步有限状态机是设计可综合风格的Verilog HDL模块的关键。在本章中我们将通过各种实例由浅入深地来介绍各种可综合风格的Verilog HDL模块，并把重点放在时序逻辑的可综合有限状态机的Verilog HDL设计要点。至于组合逻辑，因为比较简单，只需阅读典型的用Verilog HDL描述的可综合的组合逻辑的例子就可以掌握。为了更好地掌握可综合风格，还需要较深入地了解阻塞和非阻塞赋值的差别和在不同的情况下正确使用这两种赋值的方法。只有深入地理解阻塞和非阻塞赋值语句的细微不同，才有可能写出不仅可以仿真也可以综合的Verilog HDL模块。只要按照一定的原则来编写代码就可以保证Verilog模块综合前和综合后仿真的一致性。符合这样条件的可综合模块是我们设计的目标，因为这种代码是可移植的，可综合到不同的FPGA和不同工艺的ASIC中，是具有知识产权价值的软核。

7.1.有限状态机

有限状态机是由寄存器组和组合逻辑构成的硬件时序电路，其状态（即由寄存器组的1和0的组合状态所构成的有限个状态）只可能在同一时钟跳变沿的情况下才能从一个状态转向另一个状态，究竟转向哪一状态还是留在原状态不但取决于各个输入值，还取决于当前所在状态。（*这里指的是米里Mealy型有限状态机，而莫尔Moore型有限状态机究竟转向哪一状态只决于当前状态。*）

在Verilog HDL中可以用许多种方法来描述有限状态机，最常用的方法是用always语句和case语句。下面的状态转移图表示了一个有限状态机，例1的程序就是该有限状态机的多种Verilog HDL模型之一：

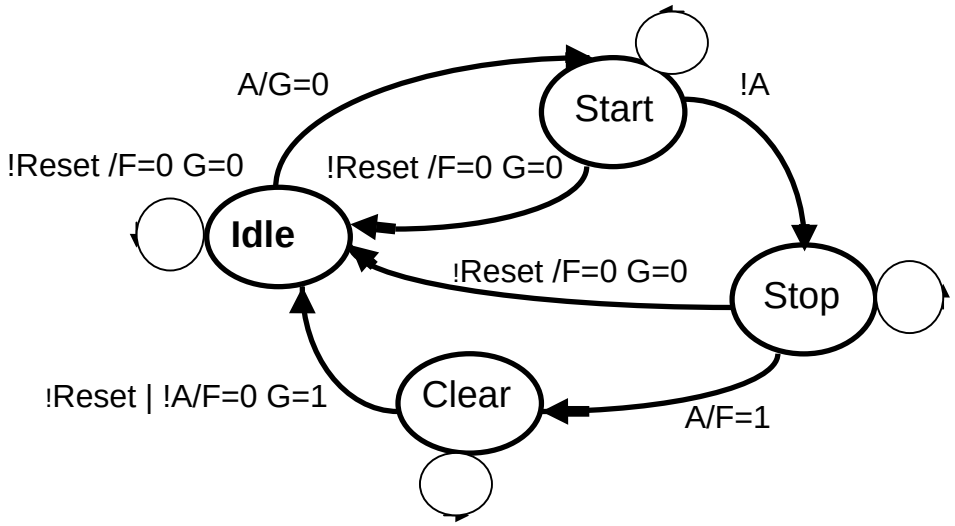


图7.1　状态转移图

上面的状态转移图表示了一个四状态的有限状态机，它的同步时钟是Clock,输入信号是 A 和 Reset,输出信号是 F 和 G。状态的转移只能在同步时钟（Clock）的上升沿时发生，往哪个状态的转移则取决于目前所在的状态和输入的信号（Reset 和 A）。下面的例子是该有限状态机的Verilog HDL模型之一：

```
       [例1]:
        module  fsm (Clock, Reset, A,  F, G);
        input Clock, Reset, A;
        output F,G;
        reg F,G;
        reg [1:0] state ;

        parameter   Idle  = 2'b00,  Start = 2'b01,
                    Stop = 2'b10, Clear = 2'b11;

          always @(posedge Clock)
            if (!Reset)
                begin
                  state <= Idle;  F<=0; G<=0;
                end
            else
              case (state)
                idle:  begin
                       if (A)  begin
                                 state <= Start;
                                 G<=0;
                               End
                       else    state <= idle;
                     end
                start: if (!A)  state <= Stop;
                        else    state <= start;
                Stop:  begin
                         if (A) begin
                                  state <= Clear;
                                  F <= 1;
                              end
                         else   state <= Stop;
                       end
                Clear: begin
                        if (!A)  begin
                                   state <=Idle;
                                   F<=0;  G<=1;
                                 End
                         else   state <= Clear;
                       end
              endcase
endmodule
```

我们还可以用另一个Verilog HDL模型来表示同一个有限状态，见下例：

```
[例2]:module  fsm (Clock, Reset, A,  F, G);
         input Clock, Reset, A;
```

```
        output F,G;
        reg F,G;
        reg [3:0] state ;

      parameter   Idle     = 4'b1000,
                  Start    = 4'b0100,
                  Stop     = 4'b0010,
                  Clear    = 4'b0001;

        always @(posedge clock)
          if (!Reset)
              begin
                state <= Idle;  F<=0; G<=0;
              end
          else
             case (state)
              Idle:  begin
                       if (A)  begin
                                 state <= Start;
                                   G<=0;
                               end
                       else state <= Idle;
                     end
              Start: if (!A)  state <= Stop;
                     else  state <= Start;
              Stop:  begin
                       if (A) begin
                                state <= Clear;
                                F <= 1;
                              End
                       Else  state <= Stop;
                     end
              Clear: begin
                       if (!A)  begin
                                  state <=Idle;
                                  F<=0;  G<=1;
                                End
                       else   state <= Clear;
                     end
              default:  state <=Idle;
             endcase
endmodule
```

[例2]与[例1]的主要不同点是状态编码，[例2]采用了独热编码，而[例1]则采用Gray码，究竟采用哪一种编码好要看具体情况而定。对于用FPGA实现的有限状态机建议采用独热码，因为虽然采用独热编码多用了两个触发器，但所用组合电路可省下许多，因而使电路的速度和可靠性有显著提高，而总的单元数并无显著增加。采用了独热编码后有了多余的状态，就有一些不可到达的状态，为此在CASE语句的最后需要增加default分支项，以确保多余状态能回到Idle状态。

我们还可以再用另一种风格的Verilog HDL模型来表示同一个有限状态，在这个模型中，我们用always语句和连续赋值语句把状态机的触发器部分和组合逻辑部分分成两部分来描述。见下例：

[例3]

```
module  fsm (Clock, Reset, A,  F, G);
input Clock, Reset, A;
output F,G;
reg [1:0] state ;
wire [1:0] Nextstate;

parameter   Idle  = 2'b00,  Start = 2'b01,
            Stop = 2'b10,   Clear = 2'b11;

  always @(posedge Clock)
    if (!Reset)
        begin
          state <= Idle;
        end
    else
     state <= Nextstate;


  assign Nextstate = ( state == Idle ) ?   (  A ?  Start  : Idle  ) :
                     ( state==Start ) ?   ( !A ?  Stop  : Start )  :
                     ( state== Stop ) ?   (  A ?  Clear : Stop )   :
                     ( state== Clear) ?   ( !A ?  Idle   : Clear)  : Idle;

  assign   F  = (( state == Stop) && A );
  assign   G =  (( state == Clear) && (!A || !Reset));

endmodule
```

我们还可以再用另一种风格的Verilog HDL模型来表示同一个有限状态，在这个模型中，我们分别用沿触发的always语句和电平敏感的always语句把状态机的触发器部分和组合逻辑部分分成两部分来描述。见下例：

[例4]

```
module  fsm (Clock, Reset, A,  F, G);
input Clock, Reset, A;
output F,G;
reg [1:0] state, Nextstate;

parameter   Idle  = 2'b00,  Start = 2'b01,
           Stop = 2'b10, Clear = 2'b11;

  always @(posedge Clock)
    if (!Reset)
        begin
          state <= Idle;
        end
    else
     state <= Nextstate;

  always @( state or A )
    begin
```

```
            F=0;
            G=0;
            if (state == Idle)
            begin
                if (A)
                Nextstate  =  Start;
                 else
                 Nextstate  =  Idle;
             G=1;
            end
            else
             if (state == Start)
              if (!A)
               Nextstate  =  Stop;
               else
                Nextstate  =  Start;

              else
               if (state == Stop)
                 if (A)
                  Nextstate  =  Clear;
                  else
                   Nextstate  =  Stop;

              else
               if (state == Clear)
              begin
                 if (!A)
                  Nextstate  =  Idle;
                  else
                     Nextstate  =  Clear;
                F=1;
              end
               else
                Nextstate= Idle;
          end

endmodule
```

上面四个例子是同一个状态机的四种不同的Verilog HDL模型，它们都是可综合的，在设计复杂程度不同的状态机时有它们各自的优势。如用不同的综合器对这四个例子进行综合，综合出的逻辑电路可能会有些不同，但逻辑功能是相同的。下面总结了有限状态机设计的一般步骤，供大家参考。

有限状态机设计的一般步骤：

1) **逻辑抽象，得出状态转换图**

就是把给出的一个实际逻辑关系表示为时序逻辑函数，可以用状态转换表来描述，也可以用状态转换图来描述。这就需要：

- 分析给定的逻辑问题，确定输入变量、输出变量以及电路的状态数。通常是取原因（或条件）作为输入变量，取结果作为输出变量。
- 定义输入、输出逻辑状态的含意，并将电路状态顺序编号。
- 按照要求列出电路的状态转换表或画出状态转换图。

这样，就把给定的逻辑问题抽象到一个时序逻辑函数了。

2) 状态化简

如果在状态转换图中出现这样两个状态，它们在相同的输入下转换到同一状态去，并得到一样的输出，则称它们为等价状态。显然等价状态是重复的，可以合并为一个。电路的状态数越少，存储电路也就越简单。状态化简的目的就在于将等价状态尽可能地合并，以得到最简的状态转换图。

3) 状态分配

状态分配又称状态编码。通常有很多编码方法，编码方案选择得当，设计的电路可以简单，反之，选得不好，则设计的电路就会复杂许多。实际设计时，需综合考虑电路复杂度与电路性能之间的折衷，在触发器资源丰富的FPGA或ASIC设计中采用独热编码（one-hot-coding）既可以使电路性能得到保证又可充分利用其触发器数量多的优势。

4) 选定触发器的类型并求出状态方程、驱动方程和输出方程。

5) 按照方程得出逻辑图

用Verilog HDL来描述有限状态机，可以充分发挥硬件描述语言的抽象建模能力，使用always块语句和case（if）等条件语句及赋值语句即可方便实现。具体的逻辑化简及逻辑电路到触发器映射均可由计算机自动完成，上述设计步骤中的第2步及4、5步不再需要很多的人为干预，使电路设计工作得到简化，效率也有很大的提高。

7.1.1用Verilog HDL语言设计可综合的状态机的指导原则:

因为大多数FPGA内部的触发器数目相当多，又加上独热码状态机（one hot state machine)的译码逻辑最为简单，所以在设计采用FPGA实现的状态机时往往采用独热码状态机(即每个状态只有一个寄存器置位的状态机)。

建议采用case,casex,或casez语句来建立状态机的模型，因为这些语句表达清晰明了，可以方便地从当前状态分支转向下一个状态并设置输出。不要忘记写上case语句的最后一个分支default，并将状态变量设为'bx，这就等于告知综合器：case语句已经指定了所有的状态，这样综合器就可以删除不需要的译码电路，使生成的电路简洁，并与设计要求一致。

如果将缺省状态设置为某一确定的状态(例如：设置default：state = state1)行不行呢？回答是这样做有一个问题需要注意。因为尽管综合器产生的逻辑和设置default：state='bx时相同，但是状态机的Verilog HDL模型综合前和综合后的仿真结果会不一致。为什么会是这样呢？因为启动仿真器时，状态机所有的输入都不确定，因此立即进入default状态，这样的设置便会将状态变量设为state1，但是实际硬件电路的状态机在通电之后，进入的状态是不确定的，很可能不是state1的状态，因此还是设置default：state='bx与实际情况相一致。但在有多余状态的情况下还是应将缺省状态设置为某一确定的有效状态，因为这样做能使状态机若偶然进入多余状态后任能在下一时钟跳变沿时返回正常工作状态，否则会引起死锁。

状态机应该有一个异步或同步复位端，以便在通电时将硬件电路复位到有效状态，也可以在操作中将硬件电路复位（大多数FPGA结构都允许使用异步复位端）。

目前大多数综合器往往不支持在一个always块中由多个事件触发的状态机（即隐含状态机，implicit state machines），**为了能综合出有效的电路，用Verilog HDL描述的状态机应明确地由唯一时钟触发。**目前大多数综合器不能综合采用Verilog HDL描述的异步状态机。异步状态机是没有确定时钟的状态机，它的状态转移不是由唯一的时钟跳变沿所触发。

千万不要使用综合工具来设计异步状态机。因为目前大多数综合工具在对异步状态机进行逻辑优化时会胡乱地简化逻辑，使综合后的异步状态机不能正常工作。**如果一定要设计异步状态机，我们建议采用电路图输入的方法，而不要用Verilog HDL输入的方法。**

Verilog HDL中，状态必须明确赋值，通常使用参数(parameters)或宏定义(define)语句加上赋值语句来实现。使用参数(parameters)语句赋状态值见下例：

```
parameter  state1 = 2 'h1,  state2 = 2 'h2;
          ...
    current_state = state2;        //把current state设置成 2'h2
          ...
```

使用宏定义(define)语句赋状态值见下例:

```
`define state1  2 'h1
`define state2  2 'h2
...
current_state = `state2; //把current state设置成 2 'h2
```

7.1.2典型的状态机实例

[例1] 宇宙飞船控制器的状态机

```
module  statmch1( launch_shuttle, land_shuttle, start_countdown,
                      start_trip_meter, clk, all_systems_go,
                       just_launched, is_landed, cnt, abort_mission);
    output   launch_shuttle, land_shuttle, start_countdown,
              start_trip_meter;
    input     clk, just_launched, is_landed, abort_mission,
               all_systems_go;
    input  [3:0]  cnt;
    reg      launch_shuttle, land_shuttle, start_countdown,
             start_trip_meter;
    //设置独热码状态的参数
    parameter  HOLD=5'h1, SEQUENCE=5'h2, LAUNCH=5'h4;
    parameter  ON_MISSION=5'h8,  LAND=5'h10;
    reg   [4:0]  present_state,  next_state;

    always @(negedge clk or posedge abort_mission)
      begin
      /****把输出设置成某个缺省值,在下面的case语句中
                    就不必再设置输出的缺省值*******/
        {launch_shuttle, land_shuttle, start_trip_meter, start_countdown} =4'b0;
        /*检查异步reset的值即abort_mission的值*/
        if(abort_mission)
            next_state=LAND;
         else
          begin   // if-else-begin
            /*如果reset为零,把next_state赋值为present_state*/
            next_state = present_state;
            /*根据 present_state 和输入信号,设置 next_state
            和输出output*/
            case ( present_state )
            HOLD: if(all_systems_go)
```

```
                    begin
                       next_state = SEQUENCE;
                       start_countdown = 1;
                    end
            SEQUENCE: if(cnt==0)
                           next_state = LAUNCH;

            LAUNCH:
                begin
                    next_state = ON_MISSION;
                    launch_shuttle = 1;
                end
            ON_MISSION:
                //取消使命前，一直留在使命状态
               if(just_launched)
                   start_trip_meter = 1;
            LAND: if(is_landed)
                        next_state = HOLD;
                     else land_shuttle = 1;
            /*把缺省状态设置为'bx(无关)或某种已知状态，使其
            在做仿真时，在复位前就与实际情况相一致*/
            default:  next_state = 'bx;
            endcase
          end          // end of if-else

      /*把当前状态变量设置为下一状态，待下一有效时钟沿来到
      时当前状态变量已设置了正确的状态值*/
        present_state = next_state;
      end         //end of always
endmodule
```

7.1.3.综合的一般原则

1) 综合之前一定要进行仿真，这是因为仿真会暴露逻辑错误,所以建议大家这样做。如果不做仿真，没有发现的逻辑错误会进入综合器，使综合的结果产生同样的逻辑错误。

2) 每一次布局布线之后都要进行仿真，在器件编程或流片之前要做最后的仿真。

3) 用Verilog HDL描述的异步状态机是不能综合的，因此应该避免用综合器来设计，如果一定要设计异步状态机则可用电路图输入的方法来设计。

4) 如果要为电平敏感的锁存器建模，使用连续赋值语句是最简单的方法。

7.1.4.语言指导原则

always块：

1) 每个always块只能有一个事件控制"@(event-expression)",而且要紧跟在always关键字后面。

2) always块可以表示时序逻辑或者组合逻辑，也可以用always块既表示电平敏感的透明锁存器又同时表示组合逻辑。但是不推荐使用这种描述方法，因为这容易产生错误和多余的电平敏感的透明锁存器。

3) **带有posedge 或 negedge 关键字的事件表达式表示沿触发的时序逻辑，没有posedge 或negedge 关键字的表示组合逻辑或电平敏感的锁存器，或者两种都表示。在表示时序和组合逻辑的事件控制表达式中如有多个沿和多个电平，其间必须用关键字 “ or ” 连接 。**

4) **每个表示时序always块只能由一个时钟跳变沿触发，置位或复位最好也由该时钟跳变沿触发。**

5) **每个在always块中赋值的信号都必需定义成reg型或整型。**整型变量缺省为32bit，使用Verilog操作符可对其进行二进制求补的算术运算。综合器还支持整型量的范围说明，这样就允许产生不是32位的整型量。句法结构：integer[<msb>:<lsb>]<identifier>。

6) always块中应该避免组合反馈回路。每次执行always块时，在生成组合逻辑的always块中赋值的所有信号必需都有明确的值；否则，需要设计者在设计中加入电平敏感的锁存器来保持赋值前的最后一个值，只有这样综合器才能正常生成电路。如果不这样做综合器会发出警告提示设计中插入了锁存器。如果在设计中存在综合器认为不是电平敏感锁存器的组合回路时，综合器会发出错误信息(例如设计中有异步状态机时)。

上面这一段不太好理解,让我们再解释一下,这也就是说,用always块设计纯组合逻辑电路时, 在生成组合逻辑的always块中参与赋值的所有信号都必需有明确的值[即在赋值表达式右端参与赋值的信号都必需在always @(敏感电平列表)中列出],如果在赋值表达式右端引用了敏感电平列表中没有列出的信号,那么在综合时，将会为该没有列出信号隐含地产生一个透明锁存器,这是因为该信号的变化不会立刻引起所赋值的变化,而必须等到敏感电平列表中某一个信号变化时,它的作用才显现出来,也就是相当于存在着一个透明锁存器把该信号的变化暂存起来，待敏感电平列表中某一个信号变化时再起作用, 纯组合逻辑电路不可能做到这一点。这样,综合后所得电路已经不是纯组合逻辑电路了,这时综合器会发出警告提示设计中插入了锁存器。见下例。

```
例：input a,b,c;
    reg e,d;
    always @(a or b or c)
      begin
          e =d & a & b;
      /* 因为d没有在敏感电平列表中,所以d变化时,
         e不能立刻变化,要等到a或b或c变化时才体现出来,
         这就是说实际上相当于存在一个电平敏感的透
         明锁存器在起作用, 把d信号的变化锁存其中 */
          d =e | c;
      end
```

赋值：

1) 对一个寄存器型(reg)和整型(integer)变量给定位的赋值只允许在一个always块内进行，如在另一always块也对其赋值，这是非法的。

2) 把某一信号值赋为'bx，综合器就把它解释成无关状态，因而综合器为其生成的硬件电路最简洁。

7.2. 可综合风格的Verilog HDL模块实例：

7.2.1.组合逻辑电路设计实例

[例1] 八位带进位端的加法器的设计实例（利用简单的算法描述）

```
module adder_8(cout,sum,a,b,cin);
  output cout;
  output [7:0] sum;
  input cin;
  input[7:0] a,b;
  assign {cout,sum}=a+b+cin;
endmodule
```

[例2]指令译码电路的设计实例

（利用电平敏感的always块来设计组合逻辑）

```
//操作码的宏定义
`define plus   3'd0
`define minus  3'd1
`define band   3'd2
`define bor    3'd3
`define unegate 3'd4

module alu(out,opcode,a,b);
output [7:0] out;
input [2:0] opcode;
input [7:0] a,b;
reg   [7:0] out;

always @(opcode or a or b)
//用电平敏感的always块描述组合逻辑
begin
  case(opcode)
  //算术运算
    `plus: out=a+b;
    `minus: out=a-b;
    //位运算
    `band: out=a&b;
    `bor:  out=a|b;
    //单目运算
    `unegate: out=~a;
    default:out=8'hx;
  endcase
end
endmodule
```

[例3].利用task和电平敏感的always块设计比较后重组信号的组合逻辑.

```
module sort4(ra,rb,rc,rd,a,b,c,d);
  parameter t=3;
  output [t:0] ra, rb, rc, rd;
  input [t:0] a, b, c, d;
  reg [t:0] ra, rb, rc, rd;

  always @(a or b or c or d)
  //用电平敏感的always块描述组合逻辑
    begin
      reg [t:0] va, vb, vc, vd;
      {va,vb,vc,vd}={a,b,c,d};
```

```
            sort2(va,vc);
            sort2(vb,vd);
            sort2(va,vb);
            sort2(vc,vd);
            sort2(vb,vc);
            {ra,rb,rc,rd}={va,vb,vc,vd};
          end

        task  sort2;
          inout [t:0] x, y;
          reg [t:0] tmp;
          if( x > y )
            begin
              tmp = x;
              x = y;
              y = tmp;
            end
        endtask

      endmodule
```

[例4]. 比较器的设计实例（利用赋值语句设计组合逻辑）

```
      module compare(equal,a,b);
      parameter size=1;
      output equal;
      input [size-1:0] a, b;
        assign  equal = (a==b) ? 1 : 0;
      endmodule
```

[例5]. 3-8译码器设计实例（利用赋值语句设计组合逻辑）

```
      module decoder(out,in);
      output [7:0] out;
      input [2:0] in;
         assign  out = 1'b1<<in;
       /**** 把最低位的1左移 in（根据从in口输入的值）位，
       并赋予out  ****/
      endmodule
```

[例6]. 8-3编码器的设计实例

编码器设计方案之一：

```
    module encoder1(none_on,out,in);
      output none_on;
      output [2:0] out;
      input [7:0] in;
      reg [2:0] out;
      reg none_on;
      always @(in)
        begin: local
          integer  i;
          out = 0;
          none_on = 1;
          /*returns the value of the highest bit
```

```
            number turned on*/
          for( i=0; i<8; i=i+1 )
            begin
              if( in[i] )
                begin
                  out = i;
                  none_on = 0;
                end
            end
        end

    endmodule
```

编码器设计方案之二：

```
    module encoder2 ( none_on, out2, out1, out0, h, g, f,
                          e, d, c, b, a);
      input h, g, f, e, d, c, b, a;
      output none_on, out2, out1, out0;
      wire [3:0] outvec;

      assign outvec= h? 4'b0111 : g? 4'b0110 : f? 4'b0101:
      e? 4'b0100 : d? 4'b0011 :c? 4'b0010 : b? 4'b0001:
      a? 4'b0000 : 4'b1000;

      assign none_on = outvec[3];
      assign    out2 = outvec[2];
      assign    out1 = outvec[1];
      assign    out0 = outvec[0];

    endmodule
```

编码器设计方案之三：

```
    module encoder3 (none_on, out2, out1, out0, h, g,
                          f, e, d, c, b, a);
        input h, g, f, e, d, c, b, a;
        output out2, out1, out0;
        output none_on;
        reg [3:0] outvec;

        assign {none_on,out2,out1,out0} = outvec;

      always @( a or b or c or d or e or f or g or h)
        begin
          if(h)      outvec=4'b0111;
          else if(g) outvec=4'b0110;
          else if(f) outvec=4'b0101;
          else if(e) outvec=4'b0100;
          else if(d) outvec=4'b0011;
          else if(c) outvec=4'b0010;
          else if(b) outvec=4'b0001;
          else if(a) outvec=4'b0000;
          else       outvec=4'b1000;
```

```
        end
    endmodule
```

[例7]. 多路器的设计实例。

使用连续赋值、case语句或if-else语句可以生成多路器电路,如果条件语句(case或if-else)中分支条件是互斥的话，综合器能自动地生成并行的多路器。

多路器设计方案之一:

```
module emux1(out, a, b, sel);
  output out;
  input a, b, sel;
     assign out = sel? A : b;
endmodule
```

多路器设计方案之二:

```
module  mux2( out, a, b, sel);
  output out;
  input a, b, sel;
  reg out;
  //用电平触发的always块来设计多路器的组合逻辑
    always @( a or b or sel )
    begin
    /*检查输入信号sel的值，如为1，输出out为a,如为0，
    输出out为b.*/
    case( sel )
         1'b1: out = a;
         1'b0: out = b;
      default: out = 'bx;
    endcase
    end
endmodule
```

多路器设计方案之三:

```
module mux3( out, a, b, sel);
    output out;
    input a, b, sel;
    reg out;
    always @( a or b or sel )
      begin
        if( sel )
           out = a;
        else
           out = b;
      end
endmodule
```

[例8]. 奇偶校验位生成器设计实例

```
module parity( even_numbits,odd_numbits,input_bus);
    output even_numbits, odd_numbits;
    input [7:0] input_bus;
      assign odd_numbits = ^input_bus;
      assign even_numbits = ~odd_numbits;
```

```
endmodule
```

[例9]. 三态输出驱动器设计实例
（用连续赋值语句建立三态门模型）
三态输出驱动器设计方案之一:

```
module trist1( out, in, enable);
  output out;
  input in, enable;
     assign  out = enable? in: 'bz;
endmodule
```

三态输出驱动器设计方案之二:

```
module trist2( out, in, enable );
  output out;
  input in, enable;
     //bufif1是 一个 Verilog门级原语 (primitive)
      bufif1 mybuf1(out, in, enable);
endmodule
```

[例10]. 三态双向驱动器设计实例

```
module bidir(tri_inout, out, in, en, b);
  inout tri_inout;
  output out;
  input in, en, b;
     assign tri_inout = en? In : 'bz;
     assign out = tri_inout ^ b;
endmodule
```

7.2.2. 时序逻辑电路设计实例

[例1]触发器设计实例

```
module dff( q, data, clk);
   output q;
   input data, clk;
   reg q;
   always @( posedge clk )
      begin
         q = data;
      end
endmodule
```

[例2]. 电平敏感型锁存器设计实例之一

```
module latch1( q, data, clk);
   output q;
   input data, clk;
      assign q = clk? data : q;
endmodule
```

[例3]. 带置位和复位端的电平敏感型锁存器设计实例之二

```
module latch2( q, data, clk, set, reset);
```

```
    output q;
    input data, clk, set, reset;
      assign q= reset? 0 : ( set? 1:(clk? data : q ) );
  endmodule
```

[例4]. 电平敏感型锁存器设计实例之三

```
  module latch3( q, data, clk);
    output q;
    input data, clk;
    reg q;
      always @(clk or data)
        begin
          if(clk)
            q=data;
        end
  endmodule
```

注意：有的综合器会产生一警告信息 告诉你产生了一个电平敏感型锁存器。因为我们设计的就是一个电平敏感型锁存器，就不用管这个警告信息。

[例5]. 移位寄存器设计实例

```
  module shifter( din, clk, clr, dout);
    input din, clk, clr;
    output [7:0] dout;
    reg [7:0] dout;
     always @(posedge clk)
      begin
       if(clr)      //清零
        dout = 8'b0;
       else
          begin
           dout = dout<<1;       //左移一位
           dout[0] = din;        //把输入信号放入寄存器的最低位
          end
      end

  endmodule
```

[例6]. 八位计数器设计实例之一

```
  module counter1( out, cout, data, load, cin, clk);
    output [7:0] out;
    output cout;
    input [7:0] data;
    input load, cin, clk;
    reg [7:0] out;
      always @(posedge clk)
        begin
          if( load )
            out = data;
          else
            out = out + cin;
        end
  assign cout= & out & cin;
```

```
        //只有当out[7:0]的所有各位都为1
        //并且进位cin也为1时才能产生进位cout
        endmodule
```

[例7]. 八位计数器设计实例之二

```
        module counter2( out, cout, data, load, cin, clk);
          output [7:0] out;
          output cout;
          input [7:0] data;
          input load, cin, clk;
          reg [7:0] out;
          reg cout;
          reg [7:0] preout;
          //创建8位寄存器
          always @(posedge clk)
          begin
          out = preout;
          end
          /****计算计数器和进位的下一个状态，
          注意：为提高性能不希望加载影响进位****/
          always @( out or data or load or cin )
            begin
              {cout, preout} = out + cin;
              if(load)
                        preout = data;
            end
        endmodule
```

7.2.3.状态机的置位与复位

7.2.3.1. 状态机的异步置位与复位

异步置位与复位是与时钟无关的.当异步置位与复位到来时它们立即分别置触发器的输出为1或0，不需要等到时钟沿到来才置位或复位。把它们列入always块的事件控制括号内就能触发always块的执行，因此，当它们到来时就能立即执行指定的操作。

状态机的异步置位与复位是用always块和事件控制实现的。先让我们来看一下事件控制的语法:

事件控制语法

```
@( <沿关键词 时钟信号
      or 沿关键词 复位信号
         or 沿关键词 置位信号> )
```

沿关键词包括 posedge（用于高电平有效的set、reset或上升沿触发的时钟）和 negedge（用于低电平有效的set、reset或下降沿触发的时钟），信号可以按任意顺序列出。

事件控制实例

1) 异步、高电平有效的置位（时钟的上升沿）

```
@(posedge clk or posedge set)
```

2) 异步低电平有效的复位（时钟的上升沿）

```
@(posedge clk or negedge reset)
```

3) 异步低电平有效的置位和高电平有效的复位（时钟的上升沿）

```
@( posedge clk or negedge set or posedge reset )
```

4) 带异步高电平有效的置位与复位的always块样板

```
always @(posedge clk or posedge set or posedge reset)
  begin
    if(reset)
      begin
        /*置输出为0*/
      end
    else
       if(set)
         begin
         /*置输出为1*/
         end
        else
          begin
          /*与时钟同步的逻辑*/
          end
  end
```

5) 带异步高电平有效的置/复位端的D触发器实例

```
module dff1( q, qb, d, clk, set, reset );
  input d, clk, set, reset;
  output q, qb;
  //声明q和qb为reg类型,因为它需要在always块内赋值
  reg q, qb;

  always @( posedge clk or posedge set or posedge reset )
    begin
      if(reset)
        begin
          q = 0;
          qb = 1;
        end
      else
        if (set)
          begin
            q = 1;
            qb = 0;
          end
        else
          begin
            q = d;
            qb = ~d;
          end
    end
endmodule
```

7.2.3.2. 状态机的同步置位与复位

同步置位与复位是指只有在时钟的有效跳变沿时刻置位或复位信号才能使触发器置位或复位(即,使触发器的输出分别转变为逻辑1或0)。

不要把set和reset信号名列入always块的事件控制表达式，因为当它们有变化时不应触发always块的执行。相反，always块的执行应只由时钟有效跳变沿触发，是否置位或复位应在always块中首先检查set和reset信号的电平。

事件控制语法:

@(<沿关键词 时钟信号>)

其中沿关键词指 posedge（正沿触发）或 negedge（负沿触发）

事件控制实例

1) 正沿触发

```
@(posedge clk)
```

2) 负沿触发

```
@(negedge clk)
```

3) 同步的具有高电平有效的置位与复位端的always块样板

```
always @(posedge clk)
     begin
        if(reset)
          begin
            /*置输出为0*/
          end
        else
             if(set)
               begin
               /*置输出为1*/
               end
             else
               begin
               /*与时钟同步的逻辑*/
               end
     end
```

4) 同步的具有高电平有效的置位/复位端的D触发器

```
module dff2( q, qb, d, clk, set, reset);
   input d, clk, set, reset;
   output q, qb;
   reg q, qb;
   always @(posedge clk)
     begin
        if(reset)
          begin
            q=0;
            qb=1;
          end
```

```
        else
          if(set)
            begin
              q=1;
              qb=0;
            end
            else
              begin
                q=d;
                qb=~d;
              end
    end
endmodule
```

7.2.4.深入理解阻塞和非阻塞赋值的不同

阻塞和非阻塞赋值的语言结构是 Verilog 语言中最难理解概念之一。甚至有些很有经验的 Verilog 设计工程师也不能完全正确地理解：何时使用非阻塞赋值何时使用阻塞赋值才能设计出符合要求的电路。他们也不完全明白在电路结构的设计中，即可综合风格的 Verilog 模块的设计中，究竟为什么还要用非阻塞赋值，以及符合 IEEE 标准的 Verilog 仿真器究竟如何来处理非阻塞赋值的仿真。本小节的目的是尽可能地把阻塞和非阻塞赋值的含义详细地解释清楚，并明确地提出可综合的 Verilog 模块编程在使用赋值操作时应注意的要点,按照这些要点来编写代码就可以避免在 Verilog 仿真时出现冒险和竞争的现象。我们在前面曾提到过下面两个要点：

- **在描述组合逻辑的 always 块中用阻塞赋值，则综合成组合逻辑的电路结构。**
- **在描述时序逻辑的 always 块中用非阻塞赋值，则综合成时序逻辑的电路结构。**

为什么一定要这样做呢？回答是，这是因为要使综合前仿真和综合后仿真一致的缘故。如果不按照上面两个要点来编写 Verilog 代码，也有可能综合出正确的逻辑，但前后仿真的结果就会不一致。

为了更好地理解上述要点，我们需要对 Verilog 语言中的阻塞赋值和非阻塞赋值的功能和执行时间上的差别有深入的了解。为了解释问题方便下面定义两个缩写字：

RHS – 方程式右手方向的表达式或变量可分别缩写为： RHS 表达式或 RHS 变量。
LHS – 方程式左手方向的表达式或变量可分别缩写为： LHS 表达式或 LHS 变量。

IEEE Verilog 标准定义了有些语句有确定的执行时间，有些语句没有确定的执行时间。若有两条或两条以上语句准备在同一时刻执行，但由于语句的排列次序不同（而这种排列次序的不同是 IEEE Verilog 标准所允许的）， 却产生了不同的输出结果。这就是造成 Verilog 模块冒险和竞争现象的原因。为了避免产生竞争，理解阻塞和非阻塞赋值在执行时间上的差别是至关重要的。

阻塞赋值

阻塞赋值操作符用等号(即 =)表示。为什么称这种赋值为阻塞赋值呢？这是因为在赋值时先计算等号右手方向（RHS）部分的值，这时赋值语句不允许任何别的 Verilog 语句的干扰，直到现行的赋值完成时刻，即把 RHS 赋值给 LHS 的时刻，它才允许别的赋值语句的执行。一般可综合的阻塞赋值操作在 RHS 不能设定有延迟，(即使是零延迟也不允许)。从理论上讲，它与后面的赋值语句只有概念上的先后，而无实质上的延迟。 若在 RHS 加上延迟，则在延迟期间会阻止赋值语句的执行, 延迟后才执行赋值，这种赋值语句是不可综合的，在需要综合的模块设计中不可使用这种风格的代码。

阻塞赋值的执行可以认为是只有一个步骤的操作：

计算 RHS 并更新 LHS，此时不能允许有来自任何其他 Verilog 语句的干扰。所谓阻塞的概念是指在同一个 always 块中，其后面的赋值语句从概念上（即使不设定延迟）是在前一句赋值语句结束后再开始赋值的。

如果在一个过程块中阻塞赋值的 RHS 变量正好是另一个过程块中阻塞赋值的 LHS 变量，这两个过程块又用同一个时钟沿触发，这时阻塞赋值操作会出现问题，即如果阻塞赋值的次序安排不好，就会出现竞争。若这两个阻塞赋值操作用同一个时钟沿触发，则执行的次序是无法确定的。下面的例子可以说明这个问题：

[例 1]. 用阻塞赋值的反馈振荡器

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

按照 IEEE Verilog 的标准，上例中两个 always 块是并行执行的，与前后次序无关。如果前一个 always 块的复位信号先到 0 时刻，则 y1 和 y2 都会取 1，而如果后一个 always 块的复位信号先到 0 时刻，则 y1 和 y2 都会取 0。这清楚地说明这个 Verilog 模块是不稳定的会产生冒险和竞争的情况。

非阻塞赋值

非阻塞赋值操作符用小于等于号（即 <= ）表示。为什么称这种赋值为非阻塞赋值？这是因为在赋值操作时刻开始时计算非阻塞赋值符的 RHS 表达式，赋值操作时刻结束时更新 LHS。在计算非阻塞赋值的 RHS 表达式和更新 LHS 期间，其他的 Verilog 语句，包括其他的 Verilog 非阻塞赋值语句都能同时计算 RHS 表达式和更新 LHS。非阻塞赋值允许其他的 Verilog 语句同时进行操作。非阻塞赋值的操作可以看作为两个步骤的过程：

1) 在赋值时刻开始时，计算非阻塞赋值 RHS 表达式。
2) 在赋值时刻结束时，更新非阻塞赋值 LHS 表达式。

非阻塞赋值操作只能用于对寄存器类型变量进行赋值，因此只能用在"initial"块和"always"块等过程块中。非阻塞赋值不允许用于连续赋值。下面的例子可以说明这个问题：

[例 2]. 用非阻塞赋值的反馈振荡器

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
```

```
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

同样，按照 IEEE Verilog 的标准，上例中两个 always 块是并行执行的，与前后次序无关。无论哪一个 always 块的复位信号先到， 两个 always 块中的非阻塞赋值都在赋值开始时刻计算 RHS 表达式,，而在结束时刻才更新 LHS 表达式。所以这两个 always 块在复位信号到来后,在 always 块结束时 y1 为 0 而 y2 为 1 是确定的。从用户的角度看这两个非阻塞赋值正好是并行执行的。

Verilog 模块编程要点：

下面我们还将对阻塞和非阻塞赋值做进一步解释并将举更多的例子来说明这个问题。在此之前，掌握可综合风格的 Verilog 模块编程的八个原则会有很大的帮助。在编写时牢记这八个要点可以为绝大多数的 Verilog 用户解决在综合后仿真中出现的 90-100% 的冒险竞争问题。

1) **时序电路建模时，用非阻塞赋值。**
2) **锁存器电路建模时，用非阻塞赋值。**
3) **用 always 块建立组合逻辑模型时，用阻塞赋值。**
4) **在同一个 always 块中建立时序和组合逻辑电路时，用非阻塞赋值。**
5) **在同一个 always 块中不要既用非阻塞赋值又用阻塞赋值。**
6) **不要在一个以上的 always 块中为同一个变量赋值。**
7) **用$strobe 系统任务来显示用非阻塞赋值的变量值**
8) **在赋值时不要使用 #0 延迟**

我们在后面还要对为什么要记住这些要点再做进一步的解释。Verilog 的新用户在彻底搞明白这两种赋值功能差别之前,一定要牢记这几条要点。照着要点来编写 Verilog 模块程序，就可省去很多麻烦。

Verilog 的层次化事件队列

详细地了解 Verilog 的层次化事件队列有助于我们理解 Verilog 的阻塞和非阻塞赋值的功能。所谓层次化事件队列指的是用于调度仿真事件的不同的 Verilog 事件队列。在 IEEE Verilog 标准中,层次化事件队列被看作是一个概念模型。设计仿真工具的厂商如何来实现事件队列,由于关系到仿真器的效率，被视为技术诀窍,不能公开发表。本节也不作详细介绍。

在 IEEE 1364-1995 Verilog 标准的 5.3 节中定义了: 层次化事件队列在逻辑上分为用于当前仿真时间的 4 个不同的队列, 和用于下一段仿真时间的若干个附加队列。

1) 动态事件队列（下列事件执行的次序可以随意安排）
   - 阻塞赋值
   - 计算非阻塞赋值语句右边的表达式
   - 连续赋值
   - 执行$display 命令
   - 计算原语的输入和输出的变化

2) 停止运行的事件队列
   - #0 延时阻塞赋值

3) 非阻塞事件队列
   - 更新非阻塞赋值语句 LHS（左边变量）的值

4) 监控事件队列
   - 执行$monitor 命令

- 执行$strobe 命令

5) 其他指定的PLI命令队列
   - (其他 PLI 命令)

以上五个队列就是 Verilog 的“层次化事件队列”

大多数 Verilog 事件是由动态事件队列调度的，这些事件包括阻塞赋值、连续赋值、$display 命令、实例和原语的输入变化以及他们的输出更新、非阻塞赋值语句 RHS 的计算等。而非阻塞赋值语句 LHS 的更新却不由动态事件队列调度。

在 IEEE 标准允许的范围内被加入到这些队列中的事件只能从动态事件队列中清除。而排列在其他队列中的事件要等到被“激活”后，即被排入动态事件队列中后，才能真正开始等待执行。IEEE 1364-1995 Verilog 标准的 5.4 节介绍了一个描述其他事件队列何时被“激活”的算法。

在当前仿真时间中，另外两个比较常用的队列是非阻塞赋值更新事件队列和监控事件队列。细节见后。

非阻塞赋值 LHS 变量的更新是按排在非阻塞赋值更新事件队列中。而 RHS 表达式的计算是在某个仿真时刻随机地开始的,与上述其他动态事件是一样的。

$strobe 和 $monitor 显示命令是排列在监控事件队列中。在仿真的每一步结束时刻，当该仿真步骤内所有的赋值都完成以后，$strobe 和 $monitor 显示出所有要求显示的变量值的变化。

在 Verilog 标准 5.3 节中描述的第四个事件队列是停止运行事件队列， 所有#0 延时的赋值都排列在该队列中。采用#0 延时赋值是因为有些对 Verilog 理解不够深入的设计人员希望在两个不同的程序块中给同一个变量赋值，他们企图在同一个仿真时刻，通过稍加延时的赋值来消除 Verilog 可能产生的竞争冒险。这样做实际上会产生问题。因为给 Verilog 模型附加完全不必要的#0 延时赋值，使得定时事件的分析变得很复杂。我们认为采用#0 延时赋值根本没有必要，完全可用其他的方式来代替，因此不推荐使用。

在下面的一些例子中，常常用上面介绍的层次化事件队列来解释 Verilog 代码的行为。时件队列的概念也常常用来说明为什么要坚持上面提到的 8 项原则。

自触发 always 块

一般而言，Verilog 的 always 块不能触发自己，见下面的例子：

[例 3] 使用阻塞赋值的非自触发振荡器

```
module osc1 (clk);
     output clk;
     reg    clk;

       initial #10 clk = 0;
       always @(clk) #10 clk = ~clk;

endmodule
```

上例描述的时钟振荡器使用了阻塞赋值。阻塞赋值时，计算 RHS 表达式并更新 LHS 的值，此时不允许其他语句的干扰。阻塞赋值必须在@(clk)边沿触发到来时刻之前完成。当触发事件到来时，阻塞赋值已经完成了，因此没有来自 always 块内部的触发事件来触发@(clk)，是一个非自触发振荡器。

而例 4 中的振荡器使用的是非阻塞赋值，它是一个自触发振荡器。

[例 4] 采用非阻塞赋值的自触发振荡器

```
module osc2 (clk);
      output clk;
      reg    clk;

      initial #10 clk = 0;
      always @(clk) #10 clk <= ~clk;

endmodule
```

@(clk)的第一次触发之后，非阻塞赋值的 RHS 表达式便计算出来，把值赋给 LHS 的事件被安排在更新事件队列中。在非阻塞赋值更新事件队列被激活之前，又遇到了@(clk)触发语句，并且 always 块再次对 clk 的值变化产生反应。当非阻塞 LHS 的值在同一时刻被更新时，@(clk)再一次触发。该例是自触发式，在编写仿真测试模块时不推荐使用这种写法的时钟信号源。

移位寄存器模型

下图表示是一个简单的移位寄存器方框图。

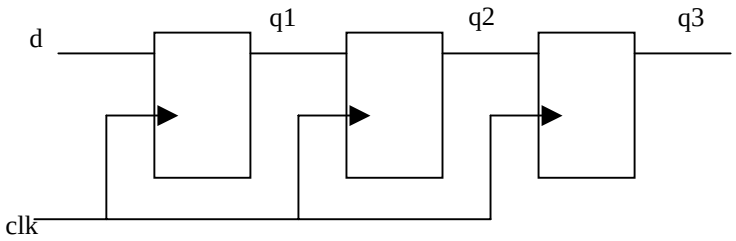


图 2 移位寄存器电路

从例 5 至例 8 介绍了四种用阻塞赋值实现图 2 移位寄存器电路的方式，有些是不正确。

[例 5]　不正确地使用的阻塞赋值来描述移位寄存器。（方式 #1）

```
module pipeb1 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk)
        begin
           q1 = d;
           q2 = q1;
           q3 = q2;
       end
    endmodule
```

在上面的模块中，按顺序进行的阻塞赋值将使得在下一个时钟上升沿时刻，所有的寄存器输出值都等于输入值 d。在每个时钟上升沿，输入值 d 将无延时地直接输出到 q3。

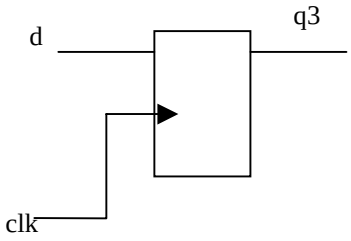


图 3　实际综合的结果

显然，上面的模块实际上被综合成只有一个寄存器的电路（见图 3），这并不是当初想要设计的移位寄存器电路。

[例 6] 用阻塞赋值来描述移位寄存器也是可行的，但这种风格并不好。（方式 #2 ）

```
module pipeb2 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk)
        begin
          q3 = q2;
          q2 = q1;
          q1 = d;
        end
    endmodule
```

在上面[例 6]的模块中，阻塞赋值的次序是经过仔细安排的，以使仿真的结果与移位寄存器相一致。虽然该模块可被综合成图 2 所示的移位寄存器，但我们不建议使用这种风格的模块来描述时序逻辑。

[例 7] 不好的用阻塞赋值来描述移位时序逻辑的风格（方式 #3）

```
 module pipeb3 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk) q1 = d;
      always @(posedge clk) q2 = q1;
      always @(posedge clk) q3 = q2;
    endmodule
```

在[例 7]中，阻塞赋值分别被放在不同的 always 块里。仿真时，这些块的先后顺序是随机的，因此可能会出现错误的结果。这是 Verilog 中的竞争冒险。按不同的顺序执行这些块将导致不同的结果。但是，这些代码的综合结果却是正确的流水线寄存器。也就是说，前仿真和后仿真的结果可能会不一致。

[例 8] 不好的用阻塞赋值来描述移位时序逻辑的风格（方式 #4）

```
    module pipeb4 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;
```

```
  always @(posedge clk) q2 = q1;
  always @(posedge clk) q3 = q2;
  always @(posedge clk) q1 = d;
endmodule
```

若在[例 8]中仅把 always 块的次序的作些变动，也可以被综合成正确的移位寄存器逻辑，但仿真结果可能不正确。

如果用非阻塞赋值语句改写以上这四个阻塞赋值的例子，每一个例子都可以正确仿真，并且综合为设计者期望的移位寄存器逻辑。

[例 9] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #1

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

[例 10] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #2

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk)
   begin
      q3 <= q2;
      q2 <= q1;
      q1 <= d;
   end
endmodule
```

[例 11] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #3

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1 <= d;
  always @(posedge clk) q2 <= q1;
```

```
    always @(posedge clk) q3 <= q2;
  endmodule
```

[例 12] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #4

```
module pipen4 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input        clk;
    reg    [7:0] q3, q2, q1;

    always @(posedge clk) q2 <= q1;
    always @(posedge clk) q3 <= q2;
    always @(posedge clk) q1 <= d;
  endmodule
```

以上移位寄存器时序逻辑电路设计的例子表明：

- 四种阻塞赋值设计方式中有一种可以保证仿真正确
- 四种阻塞赋值设计方式中有三种可以保证综合正确
- 四种非阻塞赋值设计方式全部可以保证仿真正确
- 四种非阻塞赋值设计方式全部可以保证综合正确

虽然在一个 always 块中正确的安排赋值顺序,用阻塞赋值也可以实现移位寄存器时序流水线逻辑。但是，用非阻塞赋值实现同一时序逻辑要相对简单，而且，非阻塞赋值可以保证仿真和综合的结果都是一致和正确的。**因此我们建议大家在编写 Verilog 时序逻辑时要用非阻塞赋值的方式。**

阻塞赋值及一些简单的例子

许多关于 Verilog 和 Verilog 仿真的书籍都有一些使用阻塞赋值而且成功的简单例子。例 13 就是一个在许多书上都出现过的关于触发器的例子。

[例 13]

```
module dffb (q, d, clk, rst);
     output q;
     input  d, clk, rst;
     reg    q;

    always @(posedge clk)
      if (rst) q = 1'b0;
      else     q = d;
  endmodule
```

虽然可行也很简单，但我们不建议这种用阻塞赋值来描述 D 触发器模型的风格。

如果要把所有的模块写到一个 always 块里，是可以采用阻塞赋值得到正确的建模、仿真并综合成期望的逻辑。但是，这种想法将导致使用阻塞赋值的习惯，而在较为复杂的多个 always 块的情况下可能会导致竞争冒险。

[例 14] 使用非阻塞赋值来描述 D 触发器是建议使用的风格

```
  module dffx (q, d, clk, rst);
    output q;
    input  d, clk, rst;
    reg    q;

    always @(posedge clk)
```

```
    if (rst) q  <= 1'b0;
    else     q  <= d;
endmodule
```

养成在描述时序逻辑的多个 always 块（甚至在单个 always 块）中使用非阻塞赋值的习惯比较好，见例 14 所示。

现在来看一个稍复杂的时序逻辑——线性反馈移位寄存器或 LFSR。

时序反馈移位寄存器建模

线性反馈移位寄存器（Linear Feedback Shift-Register 简称 LFSR）是带反馈回路的时序逻辑。反馈回路给习惯于用顺序阻塞赋值描述时序逻辑的设计人员带来了麻烦。见 15 所示。

[例 15] 用阻塞赋值实现的线性反馈移位寄存器，实际上并不具有 LFSR 的功能

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n)
       begin
          q3 = 1'b1;
          q2 = 1'b1;
          q1 = 1'b1;
       end
    else
        begin
         q3 = q2;
         q2 = n1;
         q1 = q3;
        end
endmodule
```

除非使用中间暂存变量，否则用例 15 所示的赋值是不可能实现反馈逻辑的。

有的人可能会想到将这些赋值语句组成单行等式（如例 16 所示），来避免使用中间变量。如果逻辑再复杂一些，单行等式是难以编写和调试的。这种方法不推荐使用。

[例 16] 用阻塞赋值描述的线性反馈移位寄存器，其功能正确，但模型的含义较难理解。

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

如果将例 15 和例 16 中的阻塞赋值用非阻塞赋值代替，如例 17 和例 18 所示，仿真结果都和 LFSR 的功能相一致。

[例 17] 用非阻塞语句描述的 LFSR，可综合其功能正确。

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

[例 18] 用非阻塞语句描述的 LFSR，可综合其功能正确。

```
  module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

从上面介绍的移位寄存器的例子以及 LFSR 的例子，建议使用非阻塞赋值实现时序逻辑。而用非阻塞赋值语句实现锁存器也是最为安全的。

原则 1 ：时序电路建模时，用非阻塞赋值。
原则 2 ：锁存器电路建模时，用非阻塞赋值。

组合逻辑建模时应使用阻塞赋值：
在 Verilog 中可以用多种方法来描述组合逻辑，但是当用 always 块来描述组合逻辑时，应该用阻塞赋值。

如果 always 块中只有一条赋值语句，使用阻塞赋值或非阻塞赋值语句都可以，但是为了养成良好的编程习惯，应该尽量使用阻塞赋值语句来描述组合逻辑。

有些设计人员提倡非阻塞赋值语句不仅可以用于时序逻辑，也可以用于组合逻辑的描述。对于简单的组合 alwasys 块是可以这样的，但是当 always 块中有多个赋值语句时，如例 19 所示的四输入与或门逻辑，使用没有延时的非阻塞赋值可能导致仿真结果不正确。有时需要在 always 块的入口附加敏感事件参数，

才能使仿真正确，因而从仿真的时间效率角度看也不合算。

[例 19] 使用非阻塞赋值语句来描述组合逻辑——不建议使用这种风格。

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d)
      begin
         tmp1 <= a & b;
         tmp2 <= c & d;
         y    <= tmp1 | tmp2;
      end
endmodule
```

例 19 中，输出 y 的值由三个时序语句计算得到。由于非阻塞赋值语句在 LHS 更新前，计算 RHS 的值，因此 tmp1 和 tmp2 仍是应进入该 always 块时的值，而不是在该步仿真结束时将更新的数值。输出 y 反映的是刚进入 always 块时的 tmp1 和 tmp2 的值，而不是在 always 块中经计算后得到的值。

[例 20]使用非阻塞赋值来描述多层组合逻辑，虽可行，但效率不高。

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;5

  always @(a or b or c or d or tmp1 or tmp2)
     begin
         tmp1 <= a & b;
         tmp2 <= c & d;
            y <= tmp1 | tmp2;
  end
endmodule
```

例 20 和例 19 的唯一区别在于，tmp1 和 temp2 加入了敏感列表中。如前所描述，当非阻塞赋值的 LHS 数值更新时，always 块将自触发并用最新计算的 tmp1 和 tmp2 的值计算更新输出 y 的值。将 tmp1 和 tmp2 加入到敏感列表中后，现在输出 y 的值是正确的。但是，一个 always 块中有多次参数传递降低了仿真器的性能，只有在没有其他合理方法的情况下才考虑这样做。

只需要在 always 块中使用阻塞赋值语句就可以实现组合逻辑，这样做既简单仿真又快是好的 Verilog 代码风格，建议大家使用。

[例 21] 使用阻塞赋值实现组合逻辑是推荐使用的编码风格。

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y    = tmp1 | tmp2;
```

```
  end
endmodule
```

例 21 和例 19 的唯一区别是，用阻塞赋值替代了非阻塞赋值。这样做可以保证仿真时经一次数据传递输出 y 的值便是正确的，仿真效率高。因此有以下原则；

原则 3 ：用 always 块描述组合逻辑时，应采用阻塞赋值语句。

时序和组合的混合逻辑——使用非阻塞赋值

有时候将简单的组合逻辑和时序逻辑写在一起很方便。当把组合逻辑和时序逻辑写到一个 always 块中时，应遵从时序逻辑建模的原则，使用非阻塞赋值，如例 22 所示。

[例 22] 在一个 always 块中同时实现组合逻辑和时序逻辑

```
module nbex2 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0; // 时序逻辑
    else        q <= a ^ b;// 异或，为组合逻辑
endmodule
```

用两个 always 块实现以上逻辑也是可以的，一个 always 块是采用阻塞赋值的纯组合部分，另一个是采用非阻塞赋值的纯时序部分。见例 23。

[例 23] 将组合和时序逻辑分别写在两个 always 块中

```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

  always @(a or b)
    y = a ^ b;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= y;
endmodule
```

原则 4：在同一个 always 块中描述时序和组合逻辑混合电路时，用非阻塞赋值。

其他将阻塞和非阻塞混合使用的原则

Verilog 语法并没有禁止将阻塞和非阻塞赋值自由地组合在一个 always 块里。虽然 Verilog 语法是允许这种写法的，但我们不建议在可综合模块的编写中采用这种风格。

[例24] 在always块中同时使用阻塞和非阻塞赋值的例子。
（应尽量避免使用这种风格的代码，在可综合模块中应严禁使用）

```
module ba_nba2 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q;

  always @(posedge clk or negedge rst_n) begin: ff
    reg tmp;
    if (!rst_n) q <= 1'b0;
    else begin
      tmp = a & b;
      q <= tmp;
    end
  end
endmodule
```

例24可以得到正确的仿真和综合结果，因为阻塞赋值和非阻塞赋值操作的不是同一个变量。虽然这种方法是可行的，但并不建议使用。

[例25] 对同一变量既进行阻塞赋值,又进行非阻塞赋值会产生综合错误。

```
module ba_nba6 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q, tmp;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q = 1'b0;  // 对 q进行阻塞赋值
    else begin
      tmp = a & b;
      q <= tmp;            // 对 q进行非阻塞赋值
    end
endmodule
```

例25在仿真时结果通常是正确的，但是综合时会出错，因为对同一变量既进行阻塞赋值,又进行了非阻塞赋值。因此，必须将其改写才能成为可综合模型。

为了养成良好的编程习惯，建议：

原则5：不要在同一个always块中同时使用阻塞和非阻塞赋值。

对同一变量进行多次赋值

在一个以上always块中对同一个变量进行多次赋值可能会导致竞争冒险,即使使用非阻塞赋值也可能产生竞争冒险。在例26中，两个always块都对输出q进行赋值。由于两个always块执行的顺序是随机的，所以仿真时会产生竞争冒险。

[例25] 使用非阻塞赋值语句，由于两个always块对同一变量q赋值
产生竞争冒险的程序：

```
module badcode1 (q, d1, d2, clk, rst_n);
```

```
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d2;
endmodule
```

当综合工具（如 Synopsys）读到[例 25]的代码时，将产生以下警告信息：

```
Warning: In design 'badcode1', there is 1 multiple-driver
         net with unknown wired-logic type.
```

如果忽略这个警告，继续编译例 26，将产生两个触发器输出到一个两输入与门。其综合级前仿真与综合后仿真的结果不完全一致。

原则 6：严禁在多个 always 块中对同一个变量赋值。

常见的对于非阻塞赋值的误解

- 非阻塞赋值和$display

  误解 1： “使用$display 命令不能用来显示非阻塞语句的赋值”

  事实是： 非阻塞语句的赋值在所有的$display 命令执行以后才更新数值

  [例]

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial
    begin
        $strobe  ("\$strobe : a = %b", a);
         a = 0;
        a <= 1;
        $display ("\$display: a = %b", a);
        #1 $finish;
    end
endmodule
```

下面是上面模块的仿真结果说明$display 命令的执行是安排在活动事件队列中，但排在非阻塞赋值数据更新事件之前。

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

- #0 延时赋值

  误解 2： “＃0 延时把赋值强制到仿真时间步的末尾”

事实是： #0 延时将赋值事件强制加入停止运行事件队列中。

[例]

```
module nb_schedule1;
  reg a, b;

  initial
    begin
       a = 0;
       b = 1;
      a <= b;
      b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

下面是上面模块的仿真结果说明#0 延时命令在非阻塞赋值事件发生前，在停止运行事件队列中执行。

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```

原则 7：用$strobe 系统任务来显示用非阻塞赋值的变量值

- 对同一变量进行多次非阻塞赋值

误解 3:“在 Verilog 语法标准中未定义可在同一个 always 块中对某同一变量进行多次非阻塞赋值”。
事实是： Verilog 标准定义了在同一个 always 块中可对某同一变量进行多次非阻塞赋值但多次赋值中，只有最后一次赋值对该变量起作用。

引用 IEEE 1364-1995 Verilog 标准【2】，第 47 页，5.4.1 节关于决定论的内容如下：

“非阻塞赋值按照语句的顺序执行,请看下例：

```
initial begin
    a <= 0;
    a <= 1;
end
```

执行该模块时，有两个非阻塞赋值更新事件加入到非阻塞赋值更新队列。以前的规则要求将非阻塞赋值更新事件按照它们在源文件的顺序加入队列，这便要求按照事件在源文件中的顺序，将事件从队列中取出并执行。因此，在仿真第一步结束的时刻，变量 a 被设置为 0，然后为 1。”
结论：最后一个非阻塞赋值决定了变量的值。

总结：
本节中所有的原则归纳如下：

- **原则 1：时序电路建模时，用非阻塞赋值。**
- **原则 2：锁存器电路建模时，用非阻塞赋值。**
- **原则 3：用 always 块写组合逻辑时，采用阻塞赋值。**
- **原则 4：在同一个 always 块中同时建立时序和组合逻辑电路时，用非阻塞赋值。**
- **原则 5：在同一个 always 块中不要同时使用非阻塞赋值和阻塞赋值。**
- **原则 6：不要在多个 always 块中为同一个变量赋值。**
- **原则 7：用$strobe 系统任务来显示用非阻塞赋值的变量值**
- **原则 8：在赋值时不要使用 #0 延迟**

结论：遵循以上原则，有助于正确的编写可综合硬件，并且可以消除 90－100％在仿真时可能产生的竞争冒险现象。

7.2.5. 复杂时序逻辑电路设计实践

[例1] 一个简单的状态机设计－－序列检测器
序列检测器是时序数字电路设计中经典的教学范例，下面我们将用Verilog HDL语言来描述、仿真、并实现它。

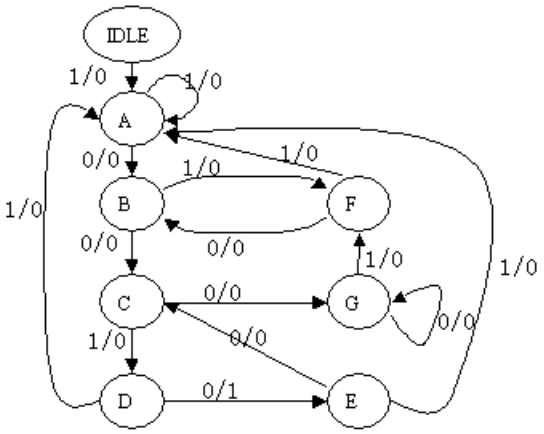
序列检测器的逻辑功能描述：

序列检测指的就是将一个指定的序列从数字码流中识别出来。本例中，我们将设计一个“10010”序列的检测器。设X为数字码流输入，Z为检出标记输出，高电平表示“发现指定序列”，低电平表示“没有发现指定序列”。考虑码流为“ 110010010000100101...” 则有下表：

时钟	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
X	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	…
Z	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	…

在时钟2－6，码流X中出现指定序列“10010”，对应输出Z在第6个时钟变为高电平－－“1”，表示“发现指定序列”。同样地，在时钟13－17码流，X中再次出现指定序列“10010”，Z输出“1”。注意，在时钟5－9还有一次检出，但它是与第一次检出的序列重叠的，即前者的前面两位同时也是后者的最后两位。

根据以上逻辑功能描述，我们可以分析得出状态转换图如下：
其中状态A－E表示5比特序列“10010”按顺序正确地出现在码流中。考虑到序列重叠的可能，转换图中



还有状态F、G。另外、电路的初始状态设为IDLE。

进一步，我们得出Verilog HDL代码。

```
//文件：sequence.v
module seqdet( x, z, clk, rst);
input x,clk, rst;
output z;

reg [2:0] state;//状态寄存器
wire z;

parameter     IDLE= 'd0,    A='d1,   B='d2,
                            C='d3,   D='d4,
                            E='d5,   F='d6,
                                     G='d7;

assign z=(state==D && x==0) ? 1 :0;
always @(posedge clk or negedge rst)
       if(!rst)
              begin
              state<=IDLE;

              end
       else
              casex( state)
                     IDLE: if(x==1)
                                          begin
                                                 state<=A;
                                          end
                     A:    if (x==0)
                                          begin
                                                 state<=B;
                                          end
                     B:     if (x==0)
                                          begin
                                                 state<=C;
                                          end
                                   else
                                          begin
                                                 state<=F;
                                          end
                     C:     if(x==1)
                                          begin
                                                 state<=D;
                                          end
                                        else
                                          begin
                                                 state<=G;
                                          end
                     D:     if(x==0)
                                          begin
                                                 state<=E;
```

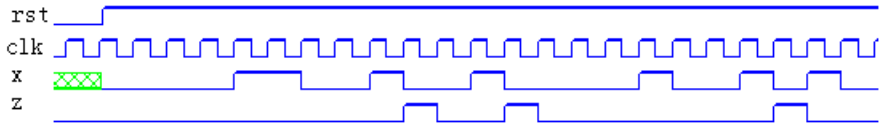
```
                                    end
                              else
                                    begin
                                          state<=A;
                                    end
                        E:      if(x==0)
                                    begin
                                          state<=C;
                                    end
                              else
                                    begin
                                          state<=A;
                                    end
                        F:       if(x==1)
                                    begin
                                          state<=A;
                                    end
                              else
                                    begin
                                          state<=B;
                                    end
                        G:       if(x==1)
                                    begin
                                          state<=F;
                                    end
                        default:      state<=IDLE;
                        endcase
endmodule
```

为了验证其正确性，我们接着编写测试用代码。

```
//文件：sequence.tf

`timescale 1ns/1ns

module t;
reg clk, rst;
reg [23:0] data;
wire z,x;
assign x=data[23];

initial
      begin
            clk<=0;
            rst<=1;
            #2 rst<=0;
            #30 rst<=1;  //复位信号
            data='b1100_1001_0000_1001_0100; //码流数据
      end

always #10 clk=~clk; //时钟信号
always @ (posedge clk) // 移位输出码流
            data={data[22:0],data[23]};
```

```
seqdet m ( .x(x), .z(z), .clk(clk), .rst(rst)); //调用序列检测器模块

// Enter fixture code here

endmodule // t
```

其中、X码流的产生，我们采用了移位寄存器的方式，以方便更改测试数据。仿真结果如下图所示：



从波形中，我们可以看到程序代码正确地完成了所要设计的逻辑功能。另外，sequence.v的编写，采用了可综合的Verilog HDL 风格，它可以通过综合器的综合最终实现到FPGA中。

说明：以上编程、仿真、综合和后仿真在PC WINDOWS NT 4.0操作系统及QuickLogic SPDE环境下通过。

[例2]EEPROM读写器件的设计

下面我们将介绍一个经过实际运行验证并可综合到各种FPGA和ASIC工艺的串行EEPROM读写器件的设计过程。列出了所有有关的Verilog HDL程序。这个器件能把并行数据和地址信号转变为串行EEPROM能识别的串行码并把数据写入相应的地址，或根据并行的地址信号从EEPROM相应的地址读取数据并把相应的串行码转换成并行的数据放到并行地址总线上。当然还需要有相应的读信号或写信号和应答信号配合才能完成以上的操作。

1. 二线制I^2C CMOS 串行EEPROM的简单介绍

二线制I^2C CMOS 串行EEPROM AT24C02/4/8/16 是一种采用CMOS 工艺制成的串行可用电擦除可编程只读存储器。串行EEPROM 一般具有两种写入方式，一种是字节写入方式，还有另一种页写入方式，允许在一个写周期内同时对一个字节到一页的若干字节进行编程写入，一页的大小取决于芯片内页寄存器的大小，不同公司的同一种型号存储器的内页寄存器可能是不一样的。为了程序的简单起见，在这里只编写串行 EEPROM 的一个字节的写入和读出方式的Verilog HDL 的行为模型代码，串行EEPROM读写器的Verilog HDL模型也只是字节读写方式的可综合模型，对于页写入和读出方式，读者可以参考有关书籍，改写串行EEPROM 的行为模型和串行EEPROM读写器的可综合模型。

2. I^2C （Inter Integrated Circuit）总线特征介绍

I^2C 双向二线制串行总线协议定义如下：

只有在总线处于“非忙”状态时，数据传输才能被初始化。在数据传输期间，只要时钟线为高电平，数据线都必须保持稳定，否则数据线上的任何变化都被当作“启动”或“停止”信号。图 1 是被定义的总线状态。

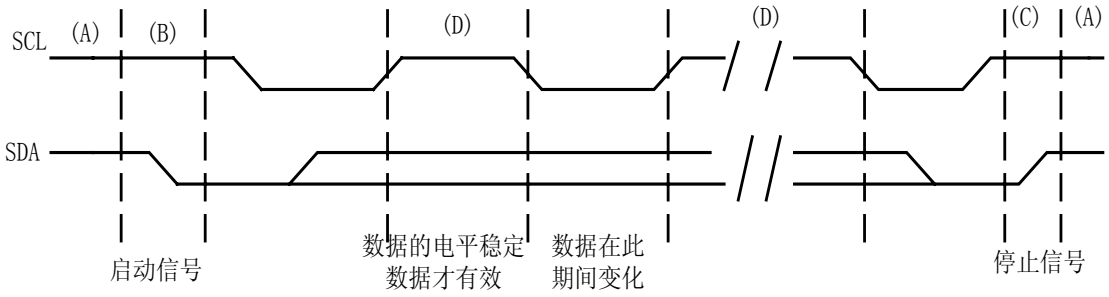


图**1**. I^2C 双向二线制串行总线特征

① 总线非忙状态（A 段）
数据线SDA 和 时钟线 SCL 都保持高电平。
② 启动数据传输（B 段）
当时钟线（SCL）为高电平状态时，数据线（SDA）由高电平变为低电平的下降沿被认为是“启动”信号。只有出现“启动”信号后，其它的命令才有效。
③ 停止数据传输（C 段）
当时钟线（SCL）为高电平状态时，数据线（SDA）由低电平变为高电平的上升沿被认为是“停止”信号。随着“停在”信号出现，所有的外部操作都结束。
④ 数据有效（D 段）
在出现“启动”信号以后，在时钟线（SCL）为高电平状态时数据线是稳定的，这时数据线的状态就要传送的数据。数据线（SDA）上的数据的改变必须在时钟线为低电平期间完成，每位数据占用一个时钟脉冲。每个数传输都是由“启动”信号开始，结束于“停止”信号。
⑤ 应答信号
每个正在接收数据的EEPROM 在接到一个字节的数据后，通常需要发出一个应答信号。而每个正在发送数据的EEPROM 在发出一个字节的数据后，通常需要接收一个应答信号。EEPROM 读写控制器必须产生一个与这个应答位相联系的额外的时钟脉冲。在EEPROM 的读操作中，EEPROM 读写控制器对EEPROM 完成的最后一个字节不产生应答位，但是应该给EEPROM 一个结束信号。

3. 二线制I^2C CMOS 串行EEPROM读写操作

1） EEPROM 的写操作（字节编程方式）

所谓EEPROM的写操作（字节编程方式）就是通过读写控制器把一个字节数据发送到EEPROM 中指定地址的存储单元。其过程如下：EEPROM 读写控制器发出“启动”信号后，紧跟着送4位I^2C总线器件特征编码1010 和3 位EEPROM 芯片地址/页地址XXX 以及写状态的R/W 位(=0)，到总线上。这一字节表示在接收到被寻址的EEPROM 产生的一个应答位后，读写控制器将跟着发送1个字节的EEPROM 存储单元地址和要写入的1个字节数据。EEPROM 在接收到存储单元地址后又一次产生应答位以后，读写控制器才发送数据字节，并把数据写入被寻址的存储单元。EEPROM 再一次发出应答信号，读写控制器收到此应答信号后，便产生“停止”信号。字节写入帧格式如图2所示：

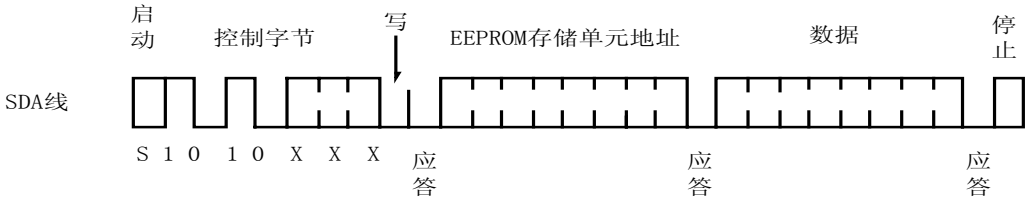


图**2**：24C02/4/8/16字节写入帧格式

2） 二线制I^2C CMOS 串行EEPROM 的读操作

所谓EEPROM的读操作即通过读写控制器读取 EEPROM 中指定地址的存储单元中的一个字节数据。串行EEPROM 的读操作分两步进行：读写器首先发送一个“启动”信号和控制字节(包括页面地址和写控制位)到EEPROM，再通过写操作设置EEPROM 存储单元地址（注意：虽然这是读操作，但需要先写入地址指针的值），在此期间EEPROM 会产生必要的应答位。接着读写器重新发送另一个“启动”信号和控制字节(包括页面地址和读控制位R/W = 1)，EEPROM收到后发出应答信号，然后，要寻址存储单元的数据就从SDA 线上输出。读操作有三种：读当前地址存储单元的数据、读指定地址存储单元的数据、读连续存储单元的数据。在这里只介绍读指定地址存储单元数据的操作。读指定地址存储单元数据的帧格式如图3：

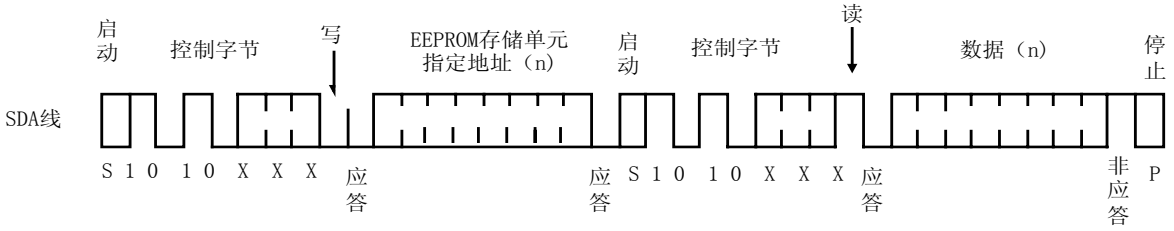


图3 ：24C02/4/8/16读指定地址存储单元的数据帧格式

4. EEPROM的Verilog HDL 程序

要设计一个串行EEPROM读写器件，不仅要编写EEPROM读写器件的可综合Verilog HDl的代码，而且要编写相应的测试代码以及EERPOM的行为模型。EEPROM的读写电路及其测试电路如图4。

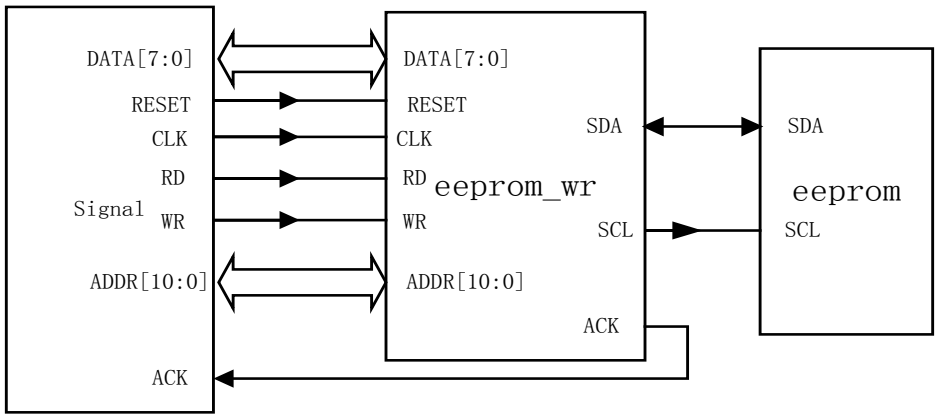


图4： EEPROM读写电路和它的测试电路

1) EEPROM的行为模型

为了设计这样一个电路我们首先要设计一个EEPROM的Verilog HDL模型,而设计这样一个模型我们需要仔细地阅读和分析EEPROM器件的说明书，因为EEPROM不是我们要设计的对象，而是我们验证设计对象所需要的器件，所以只需设计一个EEPROM的行为模型，而不需要可综合风格的模型，这就大大简化了设计过程。下面的Verilog HDL程序就是这个EEPROM（AT24C02/4/8/16） 能完成一个字节数据读写的部分行为模型,请读者查阅AT24C02/4/8/16说明书，对照下面的Verilog HDL程序理解设计的要点。因为这一程序是我们自己编写的有不完善之处敬请指正。

这里只对在操作中用到的信号线进行模拟，对于没有用到的信号线就略去了。对EEPROM用于基本总线操作的引脚SCL和SDA说明如下：SCL，串行时钟端，这个信号用于对输入和输出数据的同步，写入串行EEPROM的数据用其上升沿同步，输出数据用其下降沿同步；SDA,串行数据（/地址）输入/输出端。

```
`timescale 1ns/1ns
`define timeslice 100
module EEPROM(scl, sda);
input  scl;    //串行时钟线
inout  sda;    //串行数据线
reg out_flag;  //SDA数据输出的控制信号
reg[7:0]  memory[2047:0];
reg[10:0] address;
reg[7:0]  memory_buf;
reg[7:0]  sda_buf;   //SDA 数据输出寄存器
reg[7:0]  shift;     //SDA 数据输入寄存器
reg[7:0]  addr_byte; //EEPROM 存储单元地址寄存器
reg[7:0]  ctrl_byte; //控制字寄存器
reg[1:0]  State;     //状态寄存器
integer i;

//-------------------------------------------------------------
parameter         r7= 8'b10101111,w7= 8'b10101110,          //main7
                  r6= 8'b10101101,w6= 8'b10101100,          //main6
                 r5= 8'b10101011,w5= 8'b10101010,          //main5
                 r4= 8'b10101001,w4= 8'b10101000,          //main4
                 r3= 8'b10100111,w3= 8'b10100110,          //main3
                 r2= 8'b10100101,w2= 8'b10100100,          //main2
                 r1= 8'b10100011,w1= 8'b10100010,          //main1
                r0= 8'b10100001,w0= 8'b10100000;          //main0
//-------------------------------------------------------------
assign sda = (out_flag == 1) ? sda_buf[7] : 1'bz;
//－－－－－－－－寄存器和存储器初始化－－－－－－－－－－－－－－－－－
initial
  begin
    addr_byte     = 0;
    ctrl_byte     = 0;
    out_flag      = 0;
    sda_buf       = 0;
    State         = 2'b00;
    memory_buf    = 0;
    address       = 0;
    shift         = 0;
    for(i=0;i<=2047;i=i+1)
      memory[i]=0;
  end
//------------ 启动信号 -----------------------------
always @ (negedge sda)
  if(scl == 1 )
    begin
      State = State + 1;
      if(State == 2'b11)
```

```
          disable write_to_eeprm;
      end
  //------------ 主状态机 -------------------------
  always @(posedge sda)
    if (scl == 1 )      //停止操作
      stop_W_R;
    else
      begin
       casex(State)
          2'b01:
                    begin
                      read_in;
                     if(ctrl_byte==w7||ctrl_byte==w6||ctrl_byte==w5
                        ||ctrl_byte==w4||ctrl_byte==w3||ctrl_byte==w2
                        ||ctrl_byte==w1||ctrl_byte==w0)
                  begin
                    State = 2'b10;
                    write_to_eeprm;  //写操作
                   end
                else
                  State = 2'b00;
              end

          2'b11:
                read_from_eeprm;      //读操作

       default:
                State=2'b00;

     endcase
    end
  //------------- 操作停止------------------------------
  task stop_W_R;
    begin
      State =2'b00;  //状态返回为初始状态
      addr_byte = 0;
      ctrl_byte = 0;
      out_flag  = 0;
      sda_buf   = 0;
    end
  endtask
  //------------- 读进控制字和存储单元地址 ------------------------
  task  read_in;
    begin
      shift_in(ctrl_byte);
      shift_in(addr_byte);
    end
  endtask
  //------------EEPROM 的写操作-------------------------------------
  task write_to_eeprm;
    begin
      shift_in(memory_buf);
```

```
        address          = {ctrl_byte[3:1],addr_byte};
        memory[address]  = memory_buf;
        $display("eeprm----memory[%0h]=%0h",address,memory[address]);
        State =2'b00;              //回到0状态
      end
    endtask
//-----------EEPROM 的读操作----------------------------------------
    task read_from_eeprm;
      begin
        shift_in(ctrl_byte);
        if(ctrl_byte==r7||ctrl_byte==r6||ctrl_byte==r5||ctrl_byte==r4
           ||ctrl_byte==r3||ctrl_byte==r2||ctrl_byte==r1||ctrl_byte==r0)
          begin
            address = {ctrl_byte[3:1],addr_byte};
            sda_buf = memory[address];
            shift_out;
            State= 2'b00;
          end
      end
    endtask
//-----SDA 数据线上的数据存入寄存器，数据在SCL的高电平有效-------------
    task shift_in;
     output [7:0] shift;
      begin
         @ (posedge  scl) shift[7] = sda;
         @ (posedge  scl) shift[6] = sda;
         @ (posedge  scl) shift[5] = sda;
         @ (posedge  scl) shift[4] = sda;
         @ (posedge  scl) shift[3] = sda;
         @ (posedge  scl) shift[2] = sda;
         @ (posedge  scl) shift[1] = sda;
         @ (posedge  scl) shift[0] = sda;
         @ (negedge scl)
           begin
             #`timeslice ;
              out_flag = 1;     //应答信号输出
              sda_buf  = 0;
           end
        @(negedge scl)
           #`timeslice out_flag  = 0;
      end
    endtask
    //－－－EEPROM 存储器中的数据通过SDA 数据线输出，数据在SCL 低电平时变化
    task shift_out;
      begin
        out_flag = 1;
        for(i=6;i>=0;i=i-1)
          begin
            @ (negedge scl);
            #`timeslice;
            sda_buf = sda_buf<<1;
          end
```

```
      @(negedge scl)  #`timeslice sda_buf[7] = 1;  //非应答信号输出
      @(negedge scl)  #`timeslice out_flag  = 0;
    end
  endtask
endmodule
```

2 ） EEPROM读写器的可综合的Verilog HDL模型

下面的程序是一个串行EEPROM读写器的可综合的Verilog HDL模型，它接收来自信号源模型产生的读信号、写信号、并行地址信号、并行数据信号，并把它们转换为相应的串行信号发送到串行EEPROM（AT24C02/4/8/16） 的行为模型中去；它还发送应答信号 (ACK)到信号源模型，以便让信号源来调节发送或接收数据的速度以配合EEPROM模型的接收(写)和发送(读)数据。因为它是我们的设计对象，所以它不但要仿真正确无误，还需要可综合。

这个程序基本上由两部分组成：开关组合电路和控制时序电路，见图5。开关电路在控制时序电路的控制下按照设计的要求有节奏的打开或闭合，这样SDA可以按I^2C 数据总线的格式输出或输入，SDA和SCL一起完成EEPROM的读写操作。

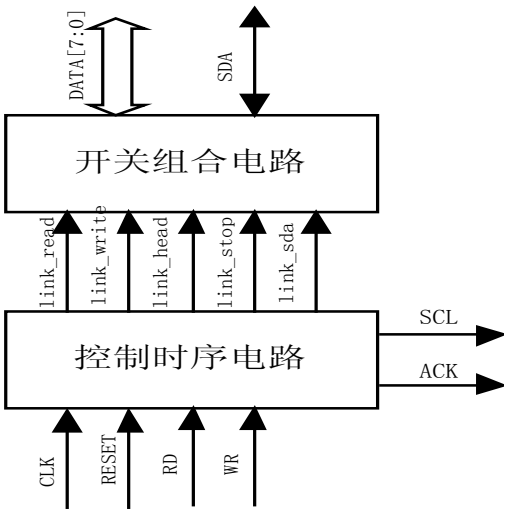


图5 ： EEPROM读写器的结构

电路最终用同步有限状态机（FSM）的设计方法实现。程序实则上是一个嵌套的状态机，由主状态机和从状态机通过由控制线启动的总线在不同的输入信号的情况下构成不同功能的较复杂的有限状态机，这个有限状态机只有唯一的驱动时钟CLK。根据串行EEPROM的读写操作时序可知，用5个状态时钟可以完成写操作，用7个状态时钟可以完成读操作，由于读写操作的状态中有几个状态是一致的，用一个嵌套的状态机即可。状态转移如图6，程序由一个读写大任务和若干个较小的任务所组成，其状态机采用独热编码，若需改变状态编码，只需改变程序中的parameter定义即可。读者可以通过模仿这一程序来编写较复杂的可综合Verilog HDL模块程序。这个设计已通过后仿真，并可在FPGA上实现布局布线。

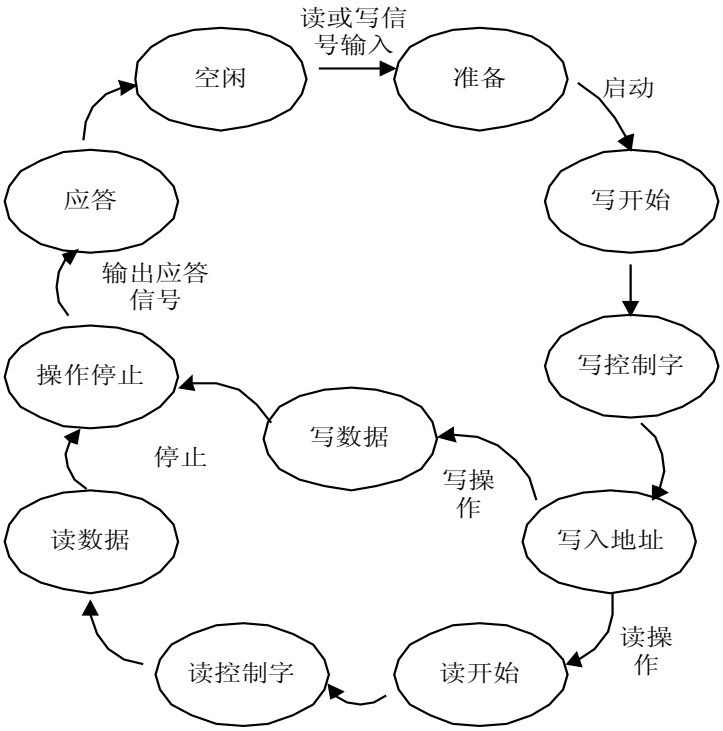


图6 ： 读写操作状态转移

```
`timescale 1ns/1ns
module EEPROM_WR(SDA,SCL,ACK,RESET,CLK,WR,RD,ADDR,DATA);
output SCL;                 //串行时钟线
output ACK;                 //读写一个周期的应答信号
input  RESET;               //复位信号
input  CLK;                 //时钟信号输入
input  WR,RD;              //读写信号
input[10:0] ADDR;           //地址线
inout SDA;                  //串行数据线
inout[7:0] DATA;            //并行数据线
reg ACK;
reg SCL;
reg WF,RF;                 //读写操作标志
reg FF;                     //标志寄存器
reg [1:0] head_buf;          //启动信号寄存器
reg[1:0] stop_buf;           //停止信号寄存器
reg [7:0] sh8out_buf;         //EEPROM写寄存器
reg [8:0] sh8out_state;        //EEPROM 写状态寄存器
reg [9:0] sh8in_state;         //EEPROM 读状态寄存器
reg [2:0] head_state;          //启动状态寄存器
reg [2:0] stop_state;          //停止状态寄存器
reg [10:0] main_state;        //主状态寄存器
reg [7:0] data_from_rm;      //EEPROM读寄存器
reg link_sda;                //SDA 数据输入EEPROM开关
reg link_read;               //EEPROM读操作开关
reg link_head;               //启动信号开关
reg link_write;              //EEPROM写操作开关
reg link_stop;               //停止信号开关
wire sda1,sda2,sda3,sda4;
```

```
//--------------串行数据在开关的控制下有次序的输出或输入--------------------
assign sda1  = (link_head)     ?   head_buf[1]    :  1'b0;
assign sda2  = (link_write)    ?   sh8out_buf[7]  :  1'b0;
assign sda3  = (link_stop)     ?   stop_buf[1]    :  1'b0;
assign sda4  = (sda1 | sda2 | sda3);
assign SDA  = (link_sda)       ?  sda4            :  1'bz;
assign DATA = (link_read)      ?  data_from_rm    :  8'hzz;

//--------------------------------主状态机状态-----------------------------------------
parameter
                 Idle        = 11'b00000000001,
                  Ready      = 11'b00000000010,
               Write_start   = 11'b00000000100,
                Ctrl_write   = 11'b00000001000,
                 Addr_write  = 11'b00000010000,
               Data_write    = 11'b00000100000,
               Read_start    = 11'b00001000000,
               Ctrl_read     = 11'b00010000000,
                Data_read    = 11'b00100000000,
                Stop         = 11'b01000000000,
                 Ackn        = 11'b10000000000,
//-------------------------并行数据串行输出状态-----------------------------
               sh8out_bit7     = 9'b000000001,
               sh8out_bit6     = 9'b000000010,
               sh8out_bit5     = 9'b000000100,
               sh8out_bit4     = 9'b000001000,
               sh8out_bit3     = 9'b000010000,
               sh8out_bit2     = 9'b000100000,
               sh8out_bit1     = 9'b001000000,
               sh8out_bit0     = 9'b010000000,
               sh8out_end     = 9'b100000000;
//-------------------------串行数据并行输出状态-----------------------------
 parameter     sh8in_begin    = 10'b0000000001,
               sh8in_bit7     = 10'b0000000010,
               sh8in_bit6     = 10'b0000000100,
               sh8in_bit5     = 10'b0000001000,
               sh8in_bit4     = 10'b0000010000,
               sh8in_bit3     = 10'b0000100000,
               sh8in_bit2     = 10'b0001000000,
               sh8in_bit1     = 10'b0010000000,
               sh8in_bit0     = 10'b0100000000,
               sh8in_end      = 10'b1000000000,
//--------------------------------启动状态-----------------------------------
               head_begin   = 3'b001,
               head_bit     = 3'b010,
               head_end     = 3'b100,
//--------------------------------停止状态-----------------------------------
               stop_begin   = 3'b001,
               stop_bit     = 3'b010,
               stop_end     = 3'b100;

 parameter       YES             = 1,
```

```
                NO              = 0;
//-------------产生串行时钟，为输入时钟的二分频-------------------
always @(negedge CLK)
  if(RESET)
     SCL <= 0;
  else
    SCL <= ~SCL;
//----------------------------主状态程序---------------------------------
always @ (posedge CLK)
  if(RESET)
    begin
     link_read  <= NO;
     link_write <= NO;
     link_head  <= NO;
     link_stop  <= NO;
     link_sda   <= NO;
     ACK        <= 0;
     RF         <= 0;
     WF         <= 0;
     FF         <= 0;
     main_state <= Idle;
    end
  else
   begin
     casex(main_state)
           Idle:
                begin
                 link_read  <= NO;
                 link_write <= NO;
                 link_head  <= NO;
                 link_stop  <= NO;
                 link_sda   <= NO;
                 if(WR)
                   begin
                    WF <= 1;
                    main_state <= Ready ;
                   end
                 else if(RD)
                   begin
                    RF <= 1;
                    main_state <= Ready ;
                   end
                else
                  begin
                   WF <= 0;
                   RF <= 0;
                   main_state <= Idle;
                  end
                end
        Ready:
             begin
                   link_read        <= NO;
```

```
                link_write       <= NO;
                link_stop        <= NO;
                link_head        <= YES;
                link_sda         <= YES;
                head_buf[1:0]    <= 2'b10;
                stop_buf[1:0]    <= 2'b01;
                head_state       <= head_begin;
                FF               <= 0;
                ACK              <= 0;
                main_state       <= Write_start;
              end
Write_start:
             if(FF == 0)
                  shift_head;
                    else
                      begin
                        sh8out_buf[7:0]  <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b0};
                        link_head        <= NO;
                        link_write       <= YES;
                        FF               <= 0;
                        sh8out_state     <= sh8out_bit6;
                        main_state       <= Ctrl_write;
                      end
Ctrl_write:
            if(FF ==0)
                  shift8_out;
                else
                  begin
                       sh8out_state    <= sh8out_bit7;
                       sh8out_buf[7:0] <= ADDR[7:0];
                       FF              <= 0;
                       main_state      <= Addr_write;
                    end
Addr_write:
            if(FF == 0)
                      shift8_out;
                    else
                      begin
                        FF <= 0;
                        if(WF)
                          begin
                            sh8out_state    <= sh8out_bit7;
                            sh8out_buf[7:0] <= DATA;
                            main_state      <= Data_write;
                          end
                        if(RF)
                          begin
                            head_buf        <= 2'b10;
                            head_state       <= head_begin;
                            main_state      <= Read_start;
                          end
                      end
```

```
Data_write:
               if(FF == 0)
                        shift8_out;
                 else
                       begin
                        stop_state       <= stop_begin;
                        main_state       <= Stop;
                        link_write       <= NO;
                        FF               <= 0;
                       end

Read_start:
              if(FF == 0)
                     shift_head;
                  else
                       begin
                         sh8out_buf       <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b1};
                         link_head        <= NO;
                         link_sda         <= YES;
                         link_write       <= YES;
                         FF               <= 0;
                         sh8out_state     <= sh8out_bit6;
                         main_state       <= Ctrl_read;
                       end
Ctrl_read:
              if(FF == 0)
                    shift8_out;
            else
                       begin
                         link_sda         <= NO;
                         link_write       <= NO;
                         FF               <= 0;
                         sh8in_state      <= sh8in_begin;
                         main_state       <= Data_read;
                    end
Data_read:
             if(FF == 0)
                   shift8in;
              else
                      begin
                         link_stop        <= YES;
                         link_sda         <= YES;
                         stop_state       <= stop_bit;
                         FF               <= 0;
                         main_state       <= Stop;
                       end
Stop:
             if(FF == 0)
                       shift_stop;
                     else
                       begin
                      ACK        <= 1;
```

```
                    FF         <= 0;
                        main_state <= Ackn;
                end
  Ackn:
              begin
                        ACK         <= 0;
                        WF          <= 0;
                        RF          <= 0;
                        main_state  <= Idle;
                      end
  default:      main_state <= Idle;
  endcase
 end
//------------------------串行数据转换为并行数据任务----------------------------------
task shift8in;
  begin
   casex(sh8in_state)
    sh8in_begin:
              sh8in_state <= sh8in_bit7;
    sh8in_bit7: if(SCL)
                  begin
                       data_from_rm[7] <= SDA;
                       sh8in_state     <= sh8in_bit6;
                      end
                else
                  sh8in_state <= sh8in_bit7;
    sh8in_bit6: if(SCL)
                  begin
                    data_from_rm[6] <= SDA;
                    sh8in_state     <= sh8in_bit5;
                  end
                else
                  sh8in_state <= sh8in_bit6;
    sh8in_bit5: if(SCL)
                  begin
                    data_from_rm[5] <= SDA;
                    sh8in_state     <= sh8in_bit4;
                  end
                else
                  sh8in_state <= sh8in_bit5;
    sh8in_bit4: if(SCL)
                  begin
                    data_from_rm[4] <= SDA;
                    sh8in_state     <= sh8in_bit3;
                  end
                else
                    sh8in_state <= sh8in_bit4;
    sh8in_bit3: if(SCL)
                  begin
                    data_from_rm[3] <= SDA;
                    sh8in_state     <= sh8in_bit2;
                  end
```

```
            else
              sh8in_state <= sh8in_bit3;
   sh8in_bit2: if(SCL)
               begin
                 data_from_rm[2] <= SDA;
                 sh8in_state     <= sh8in_bit1;
               end
             else
               sh8in_state <= sh8in_bit2;
   sh8in_bit1: if(SCL)
               begin
                 data_from_rm[1] <= SDA;
                 sh8in_state     <= sh8in_bit0;
               end
            else
              sh8in_state <= sh8in_bit1;
   sh8in_bit0: if(SCL)
               begin
                 data_from_rm[0] <= SDA;
                 sh8in_state     <= sh8in_end;
               end
            else
               sh8in_state <= sh8in_bit0;
    sh8in_end: if(SCL)
               begin
                 link_read   <= YES;
                 FF          <=  1;
                 sh8in_state <= sh8in_bit7;
               end
            else
               sh8in_state  <= sh8in_end;
     default:      begin
              link_read    <= NO;
              sh8in_state  <= sh8in_bit7;
            end
endcase
end
endtask


//------------------------------ 并行数据转换为串行数据任务 ----------------------------
task shift8_out;
 begin
  casex(sh8out_state)
        sh8out_bit7:
                    if(!SCL)
                      begin
                            link_sda     <= YES;
                            link_write   <= YES;
                         sh8out_state <= sh8out_bit6;
                       end
                     else
```

```
                sh8out_state <= sh8out_bit7;
 sh8out_bit6:
           if(!SCL)
             begin
                  link_sda      <= YES;
                  link_write    <= YES;
                sh8out_state  <= sh8out_bit5;
                sh8out_buf    <= sh8out_buf<<1;
          end
           else
                sh8out_state <= sh8out_bit6;
 sh8out_bit5:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit4;
                 sh8out_buf   <= sh8out_buf<<1;
             end
            else
              sh8out_state <= sh8out_bit5;
 sh8out_bit4:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit3;
                 sh8out_buf   <= sh8out_buf<<1;
             end
           else
                 sh8out_state <= sh8out_bit4;
 sh8out_bit3:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit2;
                 sh8out_buf   <= sh8out_buf<<1;
             end
           else
                 sh8out_state <= sh8out_bit3;
sh8out_bit2:
          if(!SCL)
            begin
                 sh8out_state <= sh8out_bit1;
                 sh8out_buf   <= sh8out_buf<<1;
            end
          else
                 sh8out_state <= sh8out_bit2;
sh8out_bit1:
          if(!SCL)
            begin
                 sh8out_state <= sh8out_bit0;
                 sh8out_buf   <= sh8out_buf<<1;
            end
          else
                 sh8out_state <= sh8out_bit1;
sh8out_bit0:
```

```
                if(!SCL)
                  begin
                       sh8out_state <= sh8out_end;
                       sh8out_buf   <= sh8out_buf<<1;
                  end
                 else
                       sh8out_state <= sh8out_bit0;
        sh8out_end:
                if(!SCL)
                  begin
                       link_sda         <= NO;
                       link_write       <= NO;
                     FF               <= 1;
                end
                 else
                       sh8out_state <= sh8out_end;
    endcase
  end
endtask
//--------------------------  输出启动信号任务  --------------------------------
task shift_head;
 begin
  casex(head_state)
        head_begin:
                if(!SCL)
                          begin
                              link_write   <= NO;
                           link_sda     <= YES;
                           link_head    <= YES;
                          head_state   <= head_bit;
                           end
                         else
                              head_state <= head_begin;
         head_bit:
                if(SCL)
                     begin
                           FF          <= 1;
                        head_buf     <= head_buf<<1;
                       head_state    <= head_end;
                          end
                else
                       head_state <= head_bit;
         head_end:
                if(!SCL)
                  begin
                           link_head    <= NO;
                            link_write   <= YES;
                     end
                  else
                           head_state <= head_end;
        endcase
 end
```

```
endtask
//---------------------------  输出停止信号任务  ------------------------------------
task shift_stop;
 begin
  casex(stop_state)
     stop_begin:  if(!SCL)
                          begin
                             link_sda      <= YES;
                              link_write   <= NO;
                             link_stop     <= YES;
                             stop_state    <= stop_bit;
                          end
                    else
                          stop_state <= stop_begin;
    stop_bit:    if(SCL)
                      begin
                            stop_buf   <= stop_buf<<1;
                           stop_state <= stop_end;
                      end
                   else
                          stop_state<= stop_bit;
    stop_end:   if(!SCL)
                  begin
                           link_head  <= NO;
                           link_stop  <= NO;
                           link_sda   <= NO;
                           FF         <= 1;
                         end
                          else
                                stop_state  <= stop_end;
    endcase
  end
endtask
endmodule
```

程序最终通过Synplify器的综合，并在Actel 3200DX 系列的FPGA上实现布局布线，通过布线后仿真 。

3 ） EEPROM的信号源模块和顶层模块

完成串行EEPROM读写器件的设计后，我们还需要做的重要一步是EEPROM读写器件的仿真。仿真可以分为前仿真和后仿真，前仿真是Verilog HDL的功能仿真，后仿真是Verilog HDL 代码经过综合、布局布线后的时序仿真。为此，我们还要编写了用于EEPROM读写器件的仿真测试的信号源程序。这个信号源能产生相应的读信号、写信号、并行地址信号、并行数据信号，并能接收串行EEPROM读写器件的应答信号 (ACK)，来调节发送或接收数据的速度。在这个程序中，我们为了保证串行EEPROM读写器件的正确性，可以进行完整的测试，写操作时输入的地址信号和数据信号的数据通过系统命令$readmemh 从addr.dat 和 data.dat 文件中取得，而在addr.dat 和data.dat文件中可以存放任意数据。读操作时从EEPROM 读出的数据存入文件eeprom.dat ,对比三个文件的数据就可以验证程序的正确性。$readmemh和$fopen等系统命令读者可以参考Verilog HDL的语法部分。最后我们把信号源、EEPROM和EEPROM读写器用顶层模块连接在一起。在下面的程序就是这个信号源的Verilog HDL模型和顶层模块。

信号源模型：

```
`timescale 1ns/1ns
`define timeslice 200
module Signal(RESET,CLK,RD,WR,ADDR,ACK,DATA);
output RESET;         //复位信号
output CLK;           //时钟信号
output RD,WR;         //读写信号
output[10:0] ADDR;     //11位地址信号
input ACK;            //读写周期的应答信号
inout[7:0] DATA;       //数据线
reg RESET;
reg CLK;
reg RD,WR;
reg W_R;             //低位：写操作；高位：读操作
reg[10:0] ADDR;
reg[7:0]  data_to_eeprom;
reg[10:0] addr_mem[0:255];
reg[7:0]  data_mem[0:255];
reg[7:0]  ROM[1:2048];
integer i,j;
integer OUTFILE;
assign DATA = (W_R) ?  8'bz : data_to_eeprom ;

//-----------------------------------时钟信号输入-----------------------------
always #(`timeslice/2)
 CLK = ~CLK;
//---------------------------------- 读写信号输入-----------------------------

initial
 begin
   RESET = 1;
   i   = 0;
   j   =0;
   W_R = 0;
   CLK = 0;
   RD  = 0;
   WR  = 0;
   #1000 ;
   RESET = 0;
    repeat(15)  //连续写15次数据
    begin
      #(5*`timeslice);
         WR = 1;
         #(`timeslice);
         WR = 0;
        @ (posedge ACK);
   end
  #(10*`timeslice);
  W_R = 1;   //开始读操作
  repeat(15)  //连续读15次数据
    begin
     #(5*`timeslice);
     RD = 1;
```

```
      #(`timeslice);
      RD = 0;
         @ (posedge ACK);
      end
  end
 //---------------------------------------写操作-----------------------------
 initial
 begin
   $display("writing-----writing-----writing-----writing");
   # (2*`timeslice);
   for(i=0;i<=15;i=i+1)
     begin
      ADDR = addr_mem[i];
      data_to_eeprom = data_mem[i];
      $fdisplay(OUTFILE,"@%0h  %0h",ADDR, data_to_eeprom);
      @(posedge ACK) ;
     end
end
 //---------------------------------------读操作-----------------------------
 initial
 @(posedge W_R)
  begin
   ADDR = addr_mem[0];
   $fclose(OUTFILE);
   $readmemh("./eeprom.dat",ROM);
   $display("Begin READING-----READING-----READING-----READING");
    for(j = 0; j <= 15; j = j+1)
    begin
       ADDR = addr_mem[j];
       @(posedge ACK);
       if(DATA == ROM[ADDR])
         $display("DATA %0h == ROM[%0h]---READ RIGHT",DATA,ADDR);
       else
         $display("DATA %0h != ROM[%0h]---READ WRONG",DATA,ADDR);
    end
 end

 initial
 begin
  OUTFILE = $fopen("./eeprom.dat");
  $readmemh("./addr.dat",addr_mem); //地址数据存入地址存储器
  $readmemh("./data.dat",data_mem); //写入EEPROM的数据存入数据存储器
 end

 endmodule
```

顶层模块：

```
`include “./Signal.v”
`include “./EEPROM.v”
`include “./EEPROM_WR.v”
`timescale 1ns/1ns
```

```
module Top;
wire RESET;
wire CLK;
wire RD,WR;
wire ACK;
wire[10:0] ADDR;
wire[7:0]  DATA;
wire SCL;
wire SDA;
Signal          signal(.RESET(RESET),.CLK(CLK),.RD(RD),
                       .WR(WR),.ADDR(ADDR),.ACK(ACK),.DATA(DATA));
EEPROM_WR   eeprom_wr(.RESET(RESET),.SDA(SDA),.SCL(SCL),.ACK(ACK),
                          .CLK(CLK),.WR(WR),.RD(RD),.ADDR(ADDR),.DATA(DATA));
EEPROM       eeprom(.sda(SDA),.scl(SCL));

endmodule
```

通过前后仿真可以验证程序的正确性。这里给出的是EEPROM读写时序的前仿真波形。后仿真波形除SCL和SDA与CLK有些延迟外,信号的逻辑关系与前仿真一致：

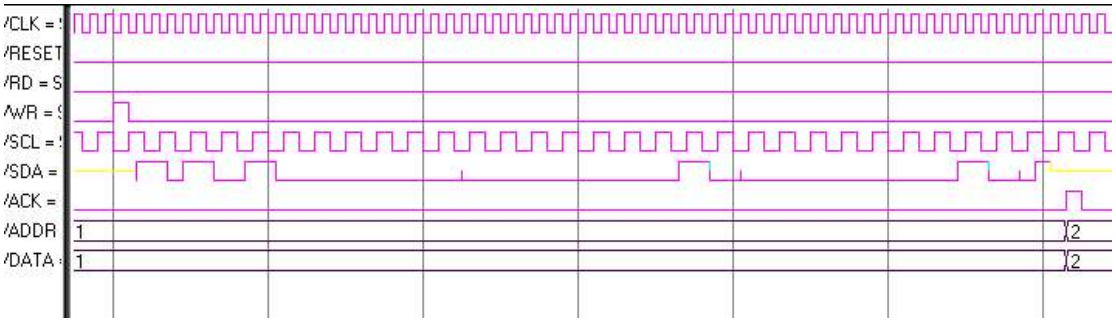


图7：EEPROM 的写时序

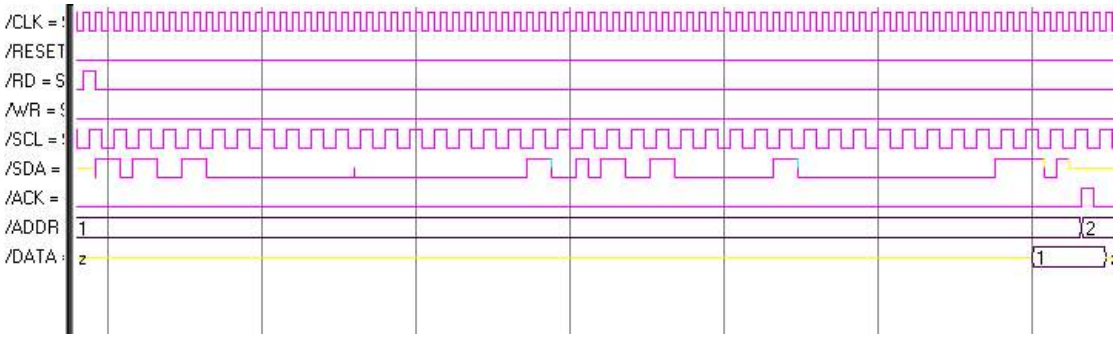


图8：EEPROM 的读时序

说明：以上编程、仿真、综合在PC WINDOWS NT 4.0操作系统、Synplify 、Actel Designer 、Altera Maxplus9.3及ModelSim Verilog环境下通过前后仿真,也在Unix Cadence Verilog-XL上通过前、后仿真（可综合到各种FPGA和ASIC工艺）。

思考题：

1) 什么是同步状态机？
2) 设计有限同步状态机的一般步骤是什么？
3) 为什么说把具体问题抽象成嵌套的状态机的思考方式可以处理极其复杂的逻辑关系？
4) 为什么要用同步状态机来产生数据流动的开关控制序列？
5) 什么是 HDL RTL级的描述方式？它与行为描述方式有什么不同？
6) 什么是综合？为什么要编写可综合模块？
7) 在设计中可综合模块和行为模块的作用分别是什么？
8) 可综合的Verilog HDL RTL级的描述方式的样板是什么？
9) 用RTL级描述方式的Verilog HDL模块是否都能综合？保证能综合的要点是什么？
10) 可综合的Verilog HDL RTL级模块的编写中用阻塞赋值和非阻塞赋值的原则是什么？
11) 保证可综合模块前后仿真一致性的关键是什么？
12) 改写7.2.5节中的例1：序列检测器的Verilog RTL级模块，使它能检测111000011序列，并进行前后仿真。
13) 读懂7.2.5节中的例2：EEPROM读写器的设计，并改写Verilog模块，使得它不只能进行随机读/写还能进行连续方式的读/写，并进行前后仿真。