设计练习进阶

前言：

在前面九章学习的基础上， 通过本章十个阶段的练习，一定能逐步掌握 Verilog HDL 设计的要点。我们可以先理解样板模块中每一条语句的作用，然后对样板模块进行综合前和综合后仿真，再独立完成每一阶段规定的练习。当十个阶段的练习做完后，便可以开始设计一些简单的逻辑电路和系统。很快我们就能过渡到设计相当复杂的数字逻辑系统。当然，复杂的数字逻辑系统的设计和验证，不但需要系统结构的知识和经验的积累，还需要了解更多的语法现象和掌握高级的 Verilog HDL 系统任务，以及与 C 语言模块接口的方法（即 PLI），这些已超出的本书的范围。有兴趣的同学可以阅读 Verilog 语法参考资料和有关文献，自己学习，我们将在下一本书中介绍 Verilog 较高级的用法。

练习一. 简单的组合逻辑设计

目的: 掌握基本组合逻辑电路的实现方法。

这是一个可综合的数据比较器，很容易看出它的功能是比较数据 a 与数据 b，如果两个数据相同,则给出结果 1,否则给出结果 0。在 Verilog HDL 中,描述组合逻辑时常使用 assign 结构。注意 equal=(a==b)?1:0,这是一种在组合逻辑实现分支判断时常使用的格式。

模块源代码：

```
//---------------- compare.v -----------------
module compare(equal,a,b);
input a,b;
output equal;
   assign  equal=(a==b)?1:0; //a 等于 b 时，equal 输出为 1；a 不等于 b 时，
                             //equal 输出为 0。
endmodule
```

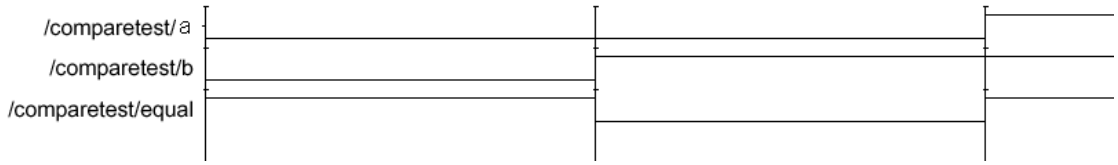
测试模块用于检测模块设计得正确与否，它给出模块的输入信号，观察模块的内部信号和输出信号，如果发现结果与预期的有所偏差，则要对设计模块进行修改。

测试模块源代码：

```
`timescale 1ns/1ns       //定义时间单位。
`include  "./compare.v" //包含模块文件。在有的仿真调试环境中并不需要此语句。
                        //而需要从调试环境的菜单中键入有关模块文件的路径和名称
module  comparetest;
   reg a,b;
   wire equal;
   initial               //initial 常用于仿真时信号的给出。
```

```
    begin
      a=0;
      b=0;
    #100   a=0; b=1;
    #100   a=1; b=1;
    #100   a=1; b=0;
    #100   $stop;       //系统任务，暂停仿真以便观察仿真波形。
    end

   compare  compare1(.equal(equal),.a(a),.b(b));    //调用模块。

endmodule
```

仿真波形（部分）：



练习：
设计一个字节（8 位）比较器。
要求：比较两个字节的大小，如 a[7:0]大于 b[7:0]输出高电平,否则输出低电平,改写测试模型,使其能进行比较全面的测试 。

练习二. 简单时序逻辑电路的设计

目的：掌握基本时序逻辑电路的实现。

在 Verilog HDL 中，相对于组合逻辑电路，时序逻辑电路也有规定的表述方式。在可综合的 Verilog HDL 模型，我们通常使用 always 块和 @(posedge clk)或 @(negedge clk)的结构来表述时序逻辑。下面是一个 1/2 分频器的可综合模型。

```
// half_clk.v:

   module half_clk(reset,clk_in,clk_out);
   input clk_in,reset;
   output clk_out;
   reg clk_out;

   always @(posedge clk_in)
```

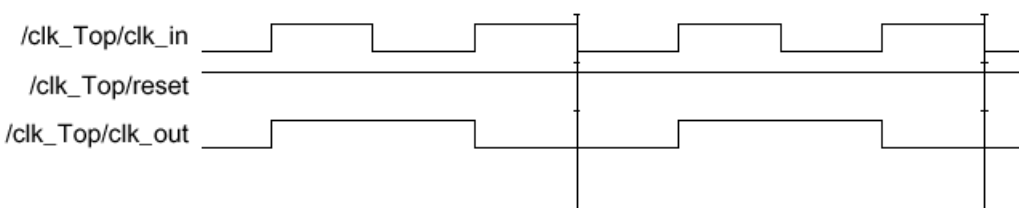
```
    begin
      if(!reset)  clk_out=0;
      else      clk_out=~clk_out;
    end
  endmodule
```

在 always 块中，被赋值的信号都必须定义为 reg 型，这是由时序逻辑电路的特点所决定的。对于 reg 型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了能正确地观察到仿真结果，在可综合风格的模块中我们通常定义一个复位信号 reset，当 reset 为低电平时，对电路中的寄存器进行复位。

测试模块的源代码：

```
//-------------------  clk_Top.v ------------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module clk_Top.v
reg clk,reset;
wire clk_out;

always  #`clk_cycle  clk = ~clk;

initial
  begin
        clk = 0;
        reset = 1;
    #100 reset = 0;
    #100 reset = 1;
    #10000 $stop;
  end

half_clk half_clk(.reset(reset),.clk_in(clk),.clk_out(clk_out));

endmodule
```

仿真波形：



练习：依然作 clk_in 的二分频 clk_out，要求输出与上例的输出正好反相。编写测试模块，给出仿真波形。

练习三. 利用条件语句实现较复杂的时序逻辑电路

目的：掌握条件语句在 Verilog HDL 中的使用。

与常用的高级程序语言一样,为了描述较为复杂的时序关系，Verilog HDL 提供了条件语句供分支判断时使用。在可综合风格的 Verilog HDL 模型中常用的条件语句有 if...else 和 case...endcase 两种结构，用法和 C 程序语言中类似。两者相较，if...else 用于不很复杂的分支关系，实际编写可综合风格的模块、特别是用状态机构成的模块时，更常用的是 case...endcase 风格的代码。这一节我们给的是有关 if...else 的范例，有关 case...endcase 结构的代码已后会经常用到。

下面给出的范例也是一个可综合风格的分频器，是将 10M 的时钟分频为 500K 的时钟。基本原理与 1/2 分频器是一样的，但是需要定义一个计数器，以便准确获得 1/20 分频

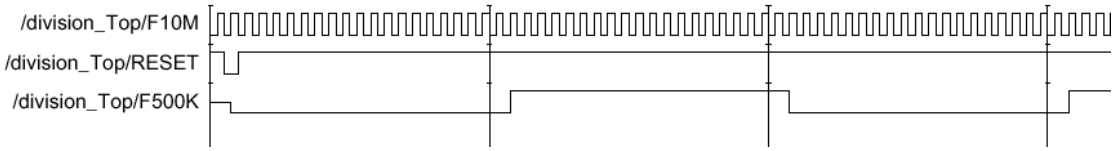
模块源代码：

```
// ---------------- fdivision.v ------------------------------
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
      always @(posedge F10M)
        if(!RESET)           //低电平复位。
          begin
            F500K <= 0;
            j <= 0;
          end
        else
          begin
            if(j==19)       //对计数器进行判断，以确定 F500K 信号是否反转。
              begin
                j <= 0;
                F500K <= ~F500K;
              end
            else
              j <= j+1;
          end
    endmodule
```

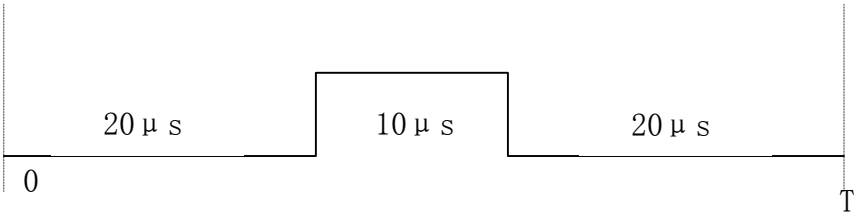
测试模块源代码：

```
//---------------  fdivision_Top.v ------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module division_Top;

reg F10M_clk,RESET;

wire  F500K_clk;

always #`clk_cycle   F10M_clk  =  ~ F10M_clk;

initial
  begin
    RESET=1;
    F10M=0;
    #100 RESET=0;
    #100 RESET=1;
    #10000 $stop;
  end

fdivision fdivision (.RESET(RESET),.F10M(F10M_clk),.F500K(F500K_clk));

endmodule
```

仿真波形：



练习：利用 10M 的时钟，设计一个单周期形状如下的周期波形。



练习四. 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

目的：1.明确掌握阻塞赋值与非阻塞赋值的概念和区别；
2.了解阻塞赋值的使用情况。

阻塞赋值与非阻塞赋值，在教材中我们已经了解了它们之间在语法上的区别以及综合后所得到的电路结构上的区别。在 always 块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。实际的时序逻辑设计中，一般的情况下非阻塞赋值语句被更多地使用，有时为了在同一周期实现相互关联的操作，也使用了阻塞赋值语句。(注意：在实现组合逻辑的 assign 结构中，无一例外地都必须采用阻塞赋值语句。

下例通过分别采用阻塞赋值语句和非阻塞赋值语句的两个看上去非常相似的两个模块 blocking.v 和 non_blocking.v 来阐明两者之间的区别。

模块源代码：

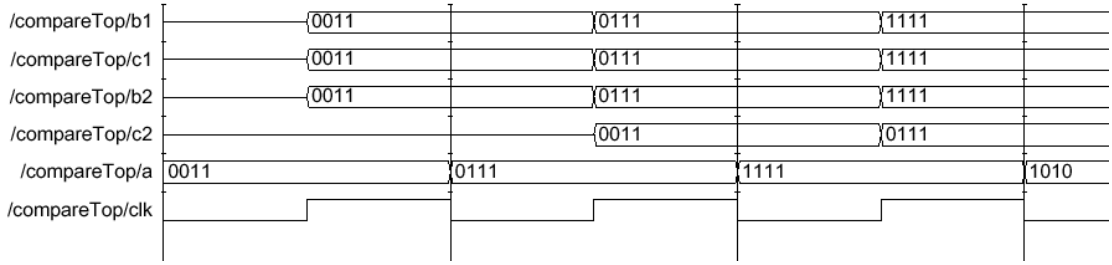
```
// -------------   blocking.v ---------------

module blocking(clk,a,b,c);
  output [3:0] b,c;
  input  [3:0] a;
  input        clk;
  reg    [3:0] b,c;
  always @(posedge clk)
   begin
    b = a;
    c = b;
    $display("Blocking: a = %d, b = %d, c = %d.",a,b,c);
   end
endmodule


//-------------  non_blocking.v --------------------
module non_blocking(clk,a,b,c);

  output [3:0] b,c;
  input  [3:0] a;
  input        clk;
  reg    [3:0] b,c;

  always @(posedge clk)
  begin
    b <= a;
    c <= b;
    $display("Non_Blocking: a = %d, b = %d, c = %d.",a,b,c);
  end
```

```
endmodule
```

测试模块源代码：

```
//-------------  compareTop.v ----------------------------

`timescale 1ns/100ps
`include "./blocking.v"
`include "./non_blocking.v"

module compareTop;

  wire [3:0] b1,c1,b2,c2;
  reg  [3:0] a;
  reg        clk;

  initial
  begin
    clk = 0;
    forever #50 clk = ~clk;
  end

  initial

  begin
    a = 4'h3;
    $display("_____________________________");
    # 100 a = 4'h7;
    $display("_____________________________");
    # 100 a = 4'hf;
    $display("_____________________________");
    # 100 a = 4'ha;
    $display("_____________________________");
    # 100 a = 4'h2;
    $display("_____________________________");
    # 100  $display("_____________________________");
    $stop;
  end
  non_blocking  non_blocking(clk,a,b2,c2);
  blocking      blocking(clk,a,b1,c1);

endmodule
```

仿真波形（部分）：



思考：在 blocking 模块中按如下写法，仿真与综合的结果会有什么样的变化？作出仿真波形，分析综合结果。

1.
```
always @(posedge clk)
  begin
    c = b;
    b = a;
  end
```

2.
```
always @(posedge clk) b=a;
always @(posedge clk) c=b;
```

练习五. 用 always 块实现较复杂的组合逻辑电路

目的: 1.掌握用 always 实现组合逻辑电路的方法;
2.了解 assign 与 always 两种组合逻辑电路实现方法之间的区别。

仅使用 assign 结构来实现组合逻辑电路,在设计中会发现很多地方会显得冗长且效率低下。而适当地采用 always 来设计组合逻辑，往往会更具实效。已进行的范例和练习中，我们仅在实现时序逻辑电路时使用 always 块。从现在开始，我们对它的看法要稍稍改变。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断，对输入数据执行相应的操作，包括加、减、与、或和求反，并且无论是指令作用的数据还是指令本身发生变化，结果都要作出及时的反应。显然，这是一个较为复杂的组合逻辑电路，如果采用 assign 语句，表达起来非常复杂。示例中使用了电平敏感的 always 块，所谓电平敏感的触发条件是指在@后的括号内电平列表中的任何一个电平发生变化，（与时序逻辑不同,它在@后的括号内没有沿敏感关键词，如 posedge 或 negedge）就能触发 always 块的动作，并且运用了 case 结构来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

```
//---------------  alu.v --------------------------
`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;
reg[7:0]    out;
input[2:0] opcode;
input[7:0] a,b;              //操作数。

always@(opcode or a or b)    //电平敏感的always块
begin
      case(opcode)
          `plus:  out = a+b;   //加操作。
          `minus: out = a-b;  //减操作。
          `band:  out = a&b;   //求与。
          `bor:   out = a|b;   //求或。
          `unegate:  out=~a; //求反。
          default:   out=8'hx;//未收到指令时，输出任意态。
      endcase
end
endmodule
```

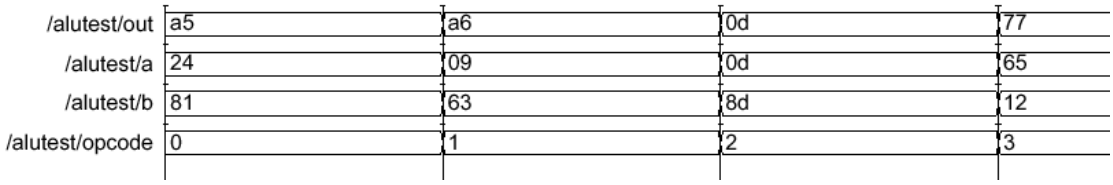
同一组合逻辑电路分别用always块和连续赋值语句assign描述时，代码的形式大相径庭，但是在always中适当运用default（在case结构中）和else（在if…else结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为 reg 型。不过，如果不使用default或else对缺省项进行说明，则易生成意想不到的锁存器，这一点一定要加以注意。

指令译码器的测试模块源代码：

```
//------------- alu_Top.v -----------------
`timescale 1ns/1ns
`include  "./alu.v"
module    alutest;
   wire[7:0] out;
   reg[7:0]  a,b;
   reg[2:0]  opcode;
   parameter   times=5;
   initial
    begin
        a={$random}%256;  //Give a radom number blongs to [0,255] .
        b={$random}%256;  //Give a radom number blongs to [0,255].
        opcode=3'h0;
        repeat(times)
          begin
          #100     a={$random}%256;  //Give a radom number.
                   b={$random}%256;  //Give a radom number.
                   opcode=opcode+1;
          end
```

```
        #100  $stop;

    end

    alu     alu1(out,opcode,a,b);

endmodule
```

仿真波形（部分）：



练习：运用 always 块设计一个八路数据选择器。要求：每路输入数据与输出数据均为 4 位 2 进制数，当选择开关(至少 3 位)或输入数据发生变化时，输出数据也相应地变化。

练习六. 在 Verilog HDL 中使用函数

目的：掌握函数在模块设计中的使用。

与一般的程序设计语言一样，Veirlog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Veirlog HDL 函数在综合时被理解成具有独立运算功能的电路，每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下例是函数调用的一个简单示范，采用同步时钟触发运算的执行，每个 clk 时钟周期都会执行一次运算。并且在测试模块中，通过调用系统任务$display 在时钟的下降沿显示每次计算的结果。

模块源代码：

```
module tryfunct(clk,n,result,reset);

  output[31:0] result;
  input[3:0]  n;
  input reset,clk;
  reg[31:0] result;

  always @(posedge clk)     //clk 的上沿触发同步运算。
    begin
         if(!reset)              //reset 为低时复位。
            result<=0;
        else
            begin
```

```
                result <= n * factorial(n)/((n*2)+1);
            end
    end

  function [31:0] factorial;      //函数定义。
    input  [3:0]  operand;
    reg    [3:0]  index;
    begin
      factorial = operand ? 1 : 0;
      for(index = 2; index <= operand; index = index + 1)
        factorial = index * factorial;
    end
  endfunction

endmodule
```

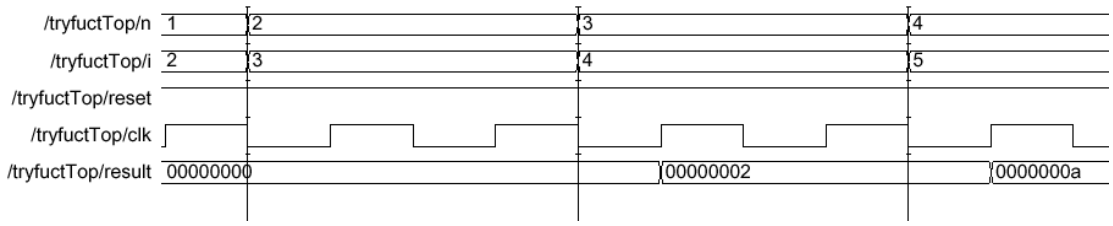
测试模块源代码：

```
`include "./step6.v"
`timescale 1ns/100ps
`define clk_cycle 50

module tryfuctTop;

reg[3:0] n,i;
reg reset,clk;

wire[31:0] result;

initial
  begin
    n=0;
    reset=1;
    clk=0;
    #100 reset=0;
    #100 reset=1;
    for(i=0;i<=15;i=i+1)
      begin
        #200 n=i;
      end
    #100 $stop;
  end

always #`clk_cycle clk=~clk;
```

```
tryfunct tryfunct(.clk(clk),.n(n),.result(result),.reset(reset));

endmodule
```

上例中函数 factorial(n)实际上就是阶乘运算。必须提醒大家注意的是，在实际的设计中，我们不希望设计中的运算过于复杂，以免在综合后带来不可预测的后果。经常的情况是，我们把复杂的运算分成几个步骤，分别在不同的时钟周期完成。

仿真波形(部分)：



练习：设计一个带控制端的逻辑运算电路，分别完成正整数的平方、立方和阶乘的运算。编写测试模块，并给出仿真波形。

练习七. 在 Verilog HDL 中使用任务（task）

目的：掌握任务在结构化 Verilog HDL 设计中的应用。

仅有函数并不能完全满足 Veirlog HDL 中的运算需求。当我们希望能够将一些信号进行运算并输出多个结果时，采用函数结构就显得非常不方便，而任务结构在这方面的优势则十分突出。任务本身并不返回计算值，但是它通过类似 C 语言中形参与实参的数据交换，非常快捷地实现运算结果的调用。此外，我们还常常利用任务来帮助我们实现结构化的模块设计，将批量的操作以任务的形式独立出来，这样设计的目的通常一眼看过去就很明了。

下面是一个利用 task 和电平敏感的 always 块设计比较后重组信号的组合逻辑的实例。可以看到，利用 task 非常方便地实现了数据之间的交换，如果要用函数实现相同的功能是非常复杂的；另外，task 也避免了直接用一般语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码：

```
//----------------- sort4.v ------------------
module sort4(ra,rb,rc,rd,a,b,c,d);
  output[3:0] ra,rb,rc,rd;
  input[3:0] a,b,c,d;
  reg[3:0] ra,rb,rc,rd;
  reg[3:0] va,vb,vc,vd;

  always @ (a or b or c or d)
    begin
```

```
      {va,vb,vc,vd}={a,b,c,d};
      sort2(va,vc);                  //va 与 vc 互换。
      sort2(vb,vd);                  //vb 与 vd 互换。
      sort2(va,vb);                  //va 与 vb 互换。
      sort2(vc,vd);                  //vc 与 vd 互换。
      sort2(vb,vc);                  //vb 与 vc 互换。
      {ra,rb,rc,rd}={va,vb,vc,vd};
    end

  task sort2;
    inout[3:0] x,y;
    reg[3:0] tmp;
    if(x>y)
      begin
        tmp=x;       //x 与 y 变量的内容互换，要求顺序执行，所以采用阻塞赋值方式。
        x=y;
        y=tmp;
      end
  endtask

endmodule
```

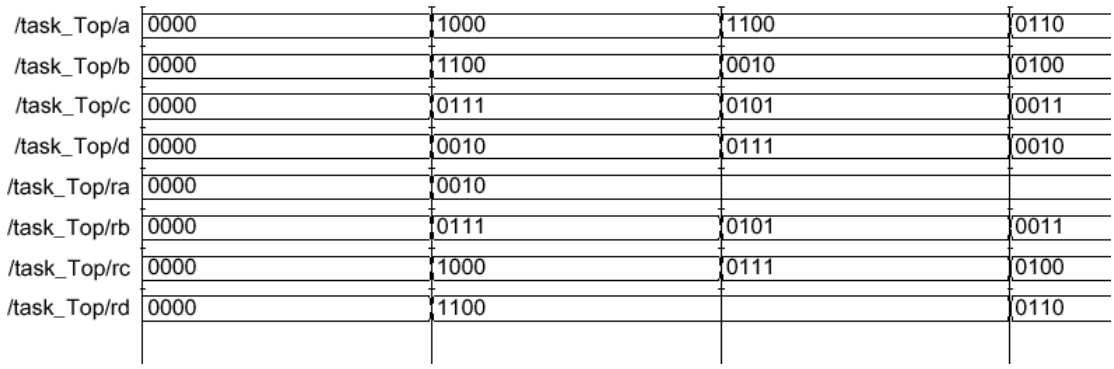
值得注意的是 task 中的变量定义与模块中的变量定义不尽相同,它们并不受输入输出类型的限制。如此例,x 与 y 对于 task sort2 来说虽然是 inout 型,但实际上它们对应的是 always 块中变量，都是 reg 型变量。

测试模块源代码：

```
`timescale 1ns/100ps
`include "sort4.v"

module task_Top;
 reg[3:0] a,b,c,d;
 wire[3:0] ra,rb,rc,rd;

 initial
  begin
    a=0;b=0;c=0;d=0;
    repeat(5)
    begin
      #100  a ={$random}%15;
            b ={$random}%15;
            c ={$random}%15;
            d ={$random}%15;
    end
```

```
    #100 $stop;

sort4 sort4 (.a(a),.b(b),.c(c),.d(d), .ra(ra),.rb(rb),.rc(rc),.rd(rd));

endmodule
```

仿真波形（部分）：



练习：设计一个模块，通过任务完成 3 个 8 位 2 进制输入数据的冒泡排序。要求：时钟触发任务的执行，每个时钟周期完成一次数据交换的操作。

练习八. 利用有限状态机进行复杂时序逻辑的设计

目的：掌握利用有限状态机实现复杂时序逻辑的方法；

在数字电路中我们已经学习过通过建立有限状态机来进行数字逻辑的设计，而在 Verilog HDL 硬件描述语言中，这种设计方法得到进一步的发展。通过 Verilog HDL 提供的语句，我们可以直观地设计出适合更为复杂的时序逻辑的电路。关于有限状态机的设计方法在教材中已经作了较为详细的阐述，在此就不赘述了。

下例是一个简单的状态机设计，功能是检测一个 5 位二进制序列“10010”。考虑到序列重叠的可能，有限状态机共提供 8 个状态（包括初始状态 IDLE）。

模块源代码：

```
seqdet.v
module seqdet(x,z,clk,rst,state);
input  x,clk,rst;
output z;
output[2:0] state;
reg[2:0] state;
wire z;
```

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

assign  z = ( state==E && x==0 )? 1 : 0;     //当 x=0 时，状态已变为 E，
                                            //状态为 D 时，x 仍为 1。因此
                      //输出为 1 的条件为 ( state==E && x==0 )。
always @(posedge clk)
   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;
```

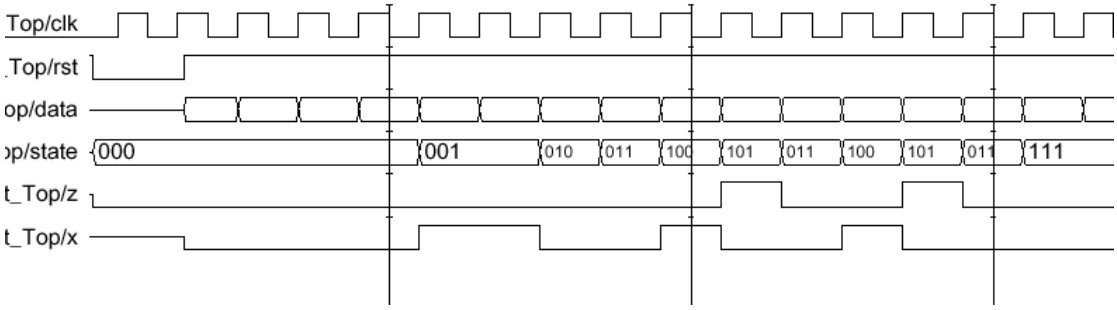
```
                    end
                else
                    begin
                        state <= A;
                    end
            E:      if(x==0)
                    begin
                        state <= C;
                    end
                else
                    begin
                        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;       //缺省状态为初始状态。
          endcase
endmodule
```

测试模块源代码：

```
//------------------- seqdet.v --------------------
`timescale 1ns/1ns
`include "./seqdet.v"
module seqdet_Top;
  reg clk,rst;
  reg[23:0] data;
  wire[2:0] state;
  wire z,x;
  assign x=data[23];
  always  #10 clk = ~clk;
  always @(posedge clk)
         data={data[22:0],data[23]};

  initial
```

```
    begin
      clk=0;
      rst=1;
      #2 rst=0;
      #30 rst=1;
      data ='b1100_1001_0000_1001_0100;
      #500 $stop;
    end

  seqdet  m(x,z,clk,rst,state);

endmodule
```

仿真波形：



练习：设计一个串行数据检测器。要求是：连续 4 个或 4 个以上的 1 时输出为 1，其他输入情况下为 0。编写测试模块并给出仿真波形。

练习九.利用状态机的嵌套实现层次结构化设计

目的： 1.运用主状态机与子状态机产生层次化的逻辑设计；
2.在结构化设计中灵活使用任务（task）结构。

在上一节，我们学习了如何使用状态机的实例。实际上，单个有限状态机控制整个逻辑电路的运转在实际设计中是不多见，往往是状态机套用状态机，从而形成树状的控制核心。这一点也与我们提倡的层次化、结构化的自顶而下的设计方法相符，下面我们就将提供一个这样的示例以供大家学习。

该例是一个简化的 EPROM 的串行写入器。事实上，它是一个 EPROM 读写器设计中实现写功能的部分经删节得到的，去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能，只具备这样一个雏形。工作的步骤是： 1.地址的串行写入； 2.数据的串行写入；3.给信号源应答，信号源给出下一个操作对象； 4.结束写操作。通过移位令并行数据得以一位一位输出。

模块源代码：

```
module writing(reset,clk,address,data,sda,ack);
  input reset,clk;
  input[7:0] data,address;

  output sda,ack;  //sda 负责串行数据输出；
                   //ack 是一个对象操作完毕后，模块给出的应答信号。
  reg link_write;  //link_write 决定何时输出。
  reg[3:0] state;  //主状态机的状态字。
  reg[4:0] sh8out_state;  //从状态机的状态字。
  reg[7:0] sh8out_buf;    //输入数据缓冲。
  reg finish_F;           //用以判断是否处理完一个操作对象。
  reg ack;

  parameter
    idle=0,addr_write=1,data_write=2,stop_ack=3;
  parameter
    bit0=1,bit1=2,bit2=3,bit3=4,bit4=5,bit5=6,bit6=7,bit7=8;

  assign   sda = link_write? sh8out_buf[7] : 1'bz;

  always @(posedge clk)
    begin
    if(!reset)               //复位。
      begin
         link_write<= 0;
         state    <= idle;
         finish_F <= 0;
         sh8out_state<=idle;
               ack<= 0;
         sh8out_buf<=0;
      end
    else
      case(state)

      idle:
        begin
        link_write  <= 0;
           state    <= idle;
           finish_F <= 0;
           sh8out_state<=idle;
                 ack<= 0;
           sh8out_buf<=address;
          state    <= addr_write;
        end
```

```
      addr_write:          //地址的输入。
        begin
        if(finish_F==0)
          begin  shift8_out; end
        else
          begin
             sh8out_state <= idle;
             sh8out_buf   <= data;
                    state <= data_write;
                finish_F <= 0;
          end
        end

      data_write:        //数据的写入。
        begin
        if(finish_F==0)
          begin  shift8_out; end
        else
          begin
              link_write <= 0;
                   state <= stop_ack;
                 finish_F <= 0;
                      ack <= 1;
          end
        end

      stop_ack:              //完成应答。
        begin
          ack <= 0;
          state <= idle;
        end

      endcase
    end

task shift8_out;                 //串行写入。
  begin
     case(sh8out_state)

     idle:
       begin
           link_write  <= 1;
          sh8out_state <= bit0;
```

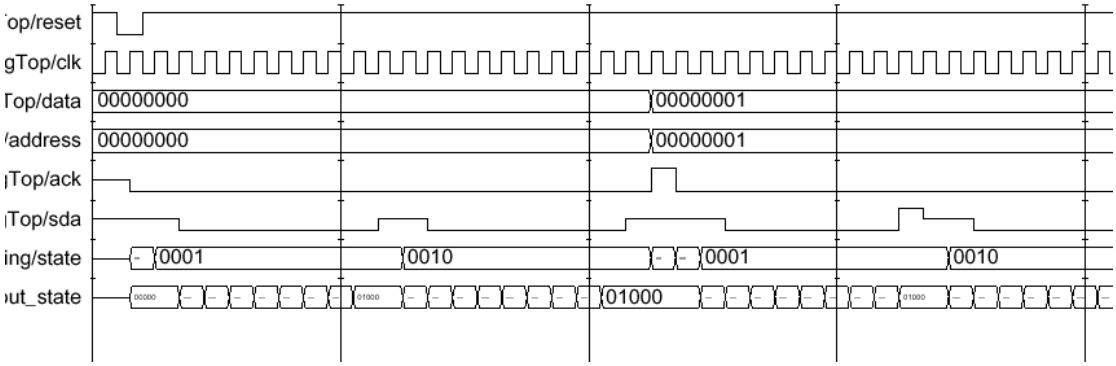
```
    end

bit0:
  begin
      link_write <= 1;
    sh8out_state <= bit1;
      sh8out_buf <= sh8out_buf<<1;
  end

bit1:
  begin
    sh8out_state<=bit2;
    sh8out_buf<=sh8out_buf<<1;
  end

bit2:
  begin
    sh8out_state<=bit3;
    sh8out_buf<=sh8out_buf<<1;
  end

bit3:
  begin
    sh8out_state<=bit4;
    sh8out_buf<=sh8out_buf<<1;
  end

bit4:
  begin
    sh8out_state<=bit5;
    sh8out_buf<=sh8out_buf<<1;
  end

bit5:
  begin
    sh8out_state<=bit6;
    sh8out_buf<=sh8out_buf<<1;
  end

bit6:
  begin
    sh8out_state<=bit7;
    sh8out_buf<=sh8out_buf<<1;
  end
```

```
     bit7:
       begin
         link_write<= 0;
         finish_F<=finish_F+1;
       end

     endcase
  end
endtask

endmodule
```

测试模块源代码：

```
`timescale 1ns/100ps
`define clk_cycle 50
module writingTop;
  reg reset,clk;
  reg[7:0] data,address;
  wire ack,sda;

  always #`clk_cycle  clk = ~clk;

  initial
    begin
        clk=0;
        reset=1;
        data=0;
        address=0;
        #(2*`clk_cycle) reset=0;
        #(2*`clk_cycle) reset=1;
        #(100*`clk_cycle) $stop;
    end

  always @(posedge ack)       //接收到应答信号后，给出下一个处理对象。
    begin
        data=data+1;
        address=address+1;
    end
  writing writing(.reset(reset),.clk(clk),.data(data),
                  .address(address),.ack(ack),.sda(sda));
endmodule
```

仿真波形：



练习：仿照上例，编写一个实现 EPROM 内数据串行读取的模块。编写测试模块，给出仿真波形。

练习十. 通过模块之间的调用实现自顶向下的设计

目的：学习状态机的嵌套使用实现层次化、结构化设计。

现代硬件系统的设计过程与软件系统的开发相似，设计一个大规模的集成电路的往往由模块多层次的引用和组合构成。 层次化、结构化的设计过程，能使复杂的系统容易控制和调试。 在 Verilog HDL 中，上层模块引用下层模块与 C 语言中程序调用有些类似，被引用的子模块在综合时作为其父模块的一部分被综合，形成相应的电路结构。在进行模块实例引用时，必须注意的是模块之间对应的端口，即子模块的端口与父模块的内部信号必须明确无误地一一对应，否则容易产生意想不到的后果。

下面给出的例子是设计中遇到的一个实例，其功能是将并行数据转化为串行数据送交外部电路编码，并将解码后得到的串行数据转化为并行数据交由 CPU 处理。显而易见，这实际上是两个独立的逻辑功能，分别设计为独立的模块，然后再合并为一个模块显得目的明确、层次清晰。

```
// -----------------    p_to_s.v ----------------------------------
module  p_to_s(D_in,T0,data,SEND,ESC,ADD_100);
    output        D_in,T0;              // D_in 是串行输出，T0 是移位时钟并给
                                        // CPU 中断，以确定何时给出下个数据。
    input   [7:0] data;                 //并行输入的数据。
    input         SEND,ESC,ADD_100;     //SEND、ESC 共同决定是否进行并到串
                                        //的数据转化。ADD_100 决定何时置数。
    wire      D_in,T0;
    reg [7:0] DATA_Q,DATA_Q_buf;

    assign        T0 = ! (SEND & ESC);      //形成移位时钟。.
    assign        D_in = DATA_Q[7];         //给出串行数据。

    always @(posedge T0 or negedge ADD_100)  //ADD_100 下沿置数，T0 上沿移位。
      begin
```

```
        if(!ADD_100)
          DATA_Q = data;
        else
          begin
          DATA_Q_buf = DATA_Q<<1;          //DATA_Q_buf 作为中介，以令综合器
          DATA_Q = DATA_Q_buf;             //能辨明。
          end
      end

endmodule
```

在 p_to_s.v 中，由于移位运算虽然可综合，但是不是简单的 RTL 级描述，直接用 DATA_Q<=DATA_Q<<1 的写法在综合时会令综合器产生误解。另外，在该设计中，由于时钟 T0 的频率较低，所以没有象以往那样采用低电平置数，而是采用 ADD_100 的下降沿置数。

```
//---------------------- s_to_p.v ---------------------------
module s_to_p(T1, data, D_out,DSC,TAKE,ADD_101);
       output       T1;                       //给 CPU 中断，以确定 CPU 何时取转化
                                              //得到的并行数据。
       output [7:0] data;
       input   D_out, DSC, TAKE, ADD_101;  //D_out 提供输入串行数据。DSC、TAKE
                                              //共同决定何时取数。
       wire   [7:0] data;
       wire         T1,clk2;
       reg    [7:0] data_latch, data_latch_buf;

       assign       clk2 = DSC  & TAKE ;   //提供移位时钟。
       assign       T1 = !clk2;

       assign       data =  (!ADD_101) ? data_latch : 8'bz;
       always@(posedge clk2)
            begin
               data_latch_buf = data_latch << 1;   //data_latch_buf 作缓冲
               data_latch     = data_latch_buf;   //，以令综合器能辨明。
                data_latch[0] = D_out;
           end
endmodule
```

将上面的两个模块合并起来的 sys.v 的源代码：

```
//------------------- sys.v ---------------------------
`include "./p_to_s.v"
`include "./s_to_p.v"
module sys(D_in,T0,T1, data, D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101);
  input         D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
  inout  [7:0]  data;
  output        D_in,T0,T1;
```

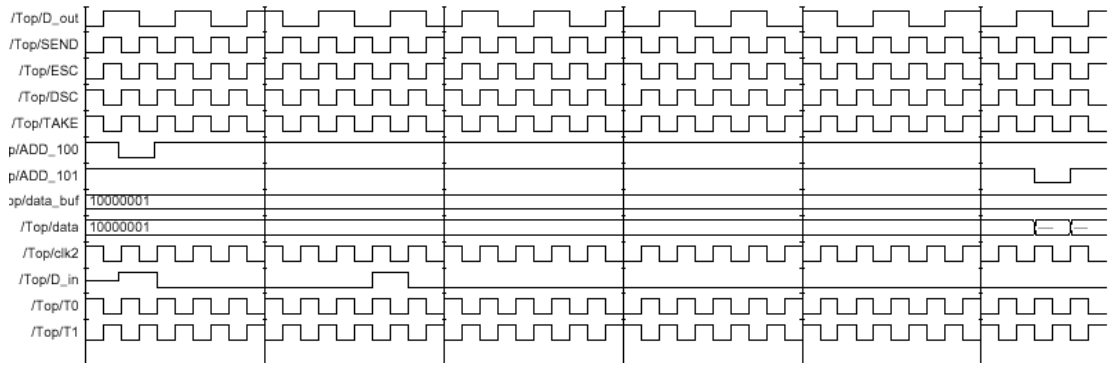
```
  p_to_s   p_to_s(.D_in(D_in),.T0(T0),.data(data),
                .SEND(SEND),.ESC(ESC),.ADD_100(ADD_100));
  s_to_p   s_to_p(.T1(T1),.data(data),.D_out(D_out),
          .DSC(DSC),.TAKE(TAKE),.ADD_101(ADD_101));

endmodule
```

测试模块源代码：

```
//-------------Top test file for sys.v ------------------
`timescale 1ns/100ps
`include "./sys.v"
module Top;
 reg D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
 reg[7:0] data_buf;
 wire [7:0] data;
 wire clk2;

 assign  data = (ADD_101) ? data_buf : 8'bz;
                                  //data 在 sys 中是 inout 型变量，ADD_101
                                  //控制 data 是作为输入还是进行输出。
assign  clk2 =DSC && TAKE;

initial
  begin
     SEND = 0;
     ESC = 0;
     DSC = 1;
     TAKE = 1;
     ADD_100 = 1;
     ADD_101 = 1;
  end

initial
  begin
    data_buf = 8'b10000001;
    #90 ADD_100 = 0;
    #100 ADD_100 = 1;
  end

always
  begin
    #50;
    SEND = ~SEND;
```

```
    ESC = ~ESC;
  end

initial
  begin
    #1500 ;
    SEND = 0;
    ESC  = 0;
    DSC  = 1;
    TAKE = 1;
    ADD_100 = 1;
    ADD_101 = 1;
    D_out = 0;
    #1150 ADD_101 = 0;
    #100 ADD_101 =1;
    #100 $stop;
  end

always
  begin
    #50 ;
    DSC = ~DSC;
    TAKE = ~TAKE;
  end

always @(negedge clk2) D_out = ~D_out;

sys    sys(.D_in(D_in),.T0(T0),.T1(T1),.data(data),.D_out(D_out),
             .ADD_101(ADD_101), .SEND(SEND),.ESC(ESC),.DSC(DSC),
                                .TAKE(TAKE),.ADD_100(ADD_100));
endmodule
```

仿真波形：



练习：设计一个序列发生器。要求根据输入的 8 位并行数据输出串行数据,如果输入数据在 0—127 之间则输出一位 0，如果输入数据在 128—255 之间则输出一位 1，同步时钟触发；并且和范例 8 的序列检测器搭接，形成一个封闭系统。编写测试模块，并给出仿真波形。