第11章 验 证

本章介绍了如何编写测试验证程序 (test bench)。测试验证程序用于测试和验证设计的正确性。Verilog HDL提供强有力的结构来说明测试验证程序。

11.1 编写测试验证程序

测试验证程序有三个主要目的：

1) 产生模拟激励(波形)。

2) 将输入激励加入到测试模块并收集其输出响应；

3) 将响应输出与期望值进行比较。

Verilog HDL提供了大量的方法以编写测试验证程序。在本章中，我们将对其中的某些方法进行探讨。典型的测试验证程序形式如下：

```
module Test_Bench;
  //通常测试验证程序没有输入和输出端口。
  Local_reg_and_net_declarations
  Generate_waveforms_using_initial_&_always_statements
  Instantiate_module_under_test
  Monitor_output_and_compare_with_expected_values
endmodule
```

测试中，通过在测试验证程序中进行实例化，激励自动加载于测试模块。

11.2 波形产生

有两种产生激励值的主要方法：

1) 产生波形，并在确定的离散时间间隔加载激励。

2) 根据模块状态产生激励，即根据模块的输出响应产生激励。

通常需要两类波形。一类是具有重复模式的波形，例如时钟波形，另一类是一组指定的值确定的波形。

11.2.1 值序列

产生值序列的最佳方法是使用initial语句。例如:

```
initial
  begin
    Reset = 0;
    #100 Reset = 1;
    #80 Reset = 0;
    #30 Reset = 1;
end
```

产生的波形如图11-1所示。Initial语句中的赋值语句用时延控制产生波形。此外，语句内时延也能够按如下实例所示产生波形。

```
initial
  begin
    Reset = 0;
    Reset = #100 1;
    Reset = #80 0;
    Reset = #30 1;
end
```

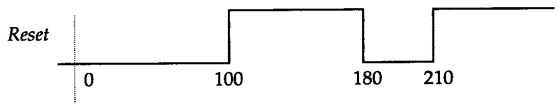


图11-1 使用initial语句产生的波形

因为使用的是阻塞性过程赋值，上面语句中的时延是相对时延。如果使用绝对时延，可用带有语句内时延的非阻塞性过程性赋值，例如，

```
initial
  begin
    Reset <= 0;
    Reset <= #100 1;
    Reset <= #180 0;
    Reset <= #210 1;
end
```

这三个initial语句产生的波形与图11-1中所示的波形一致。

为重复产生一个值序列，可以使用always语句替代initial语句，这是因为initial语句只执行一次而always语句会重复执行。下例的always语句所产生的波形如图11-2所示。

```
parameter REPEAT_DELAY= 35;
integer CoinValue;

always
  begin
    CoinValue = 0;
    #7 CoinValue = 25;
    #2 CoinValue = 5;
    #8 CoinValue = 10;
    #6 CoinValue = 5;
    #REPEAT_DELAY;
end
```

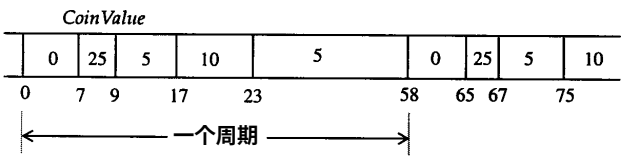


图11-2 使用always语句产生的重复序列

11.2.2 重复模式

重复模式的生成通过使用如下形式的连续赋值形式加以简化：

```
assign # (PERIOD/2)   Clock = ~ Clock;
```

但是这种做法并不完全正确。问题在于 *Clock*是一个线网(只有线网能够在连续赋值中被赋值)，它的初始值是z，并且，z等于x，~x等于x。因此*Clock*的值永远固定为值x。

现在需要一种初始化*Clock*的方法。可以用initial语句实现。

```
initial
  Clock = 0;
```

但是现在Clock必须是寄存器数据类型(因为只有寄存器数据类型能够在 initial语句中被赋值)，因此连续赋值语句需要被变换为 always语句。下面是一个完整的时钟产生器模块。

```
module Gen_Clk_A (Clk_A);
  output Clk_A;
  reg Clk_A;
  parameter tPERIOD = 10;

  initial
    Clk_A = 0;

  always
    # (tPERIOD/2)  Clk_A = ~ Clk_A;
endmodule
```

图11-3显示了该模块产生的时钟波形。

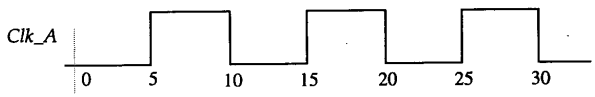


图11-3 周期性的时钟波形

下面给出了产生周期性时钟波形的另一种可选方式。

```
module Gen_Clk_B (Clk_B);
  output Clk_B;
  reg Start;

  initial
    begin
      Start = 1;
      #5 Start = 0;
    end

nor #2 (Clk_B, Start, Clk_B);
endmodule
//产生一个高、低电平宽度均为2的时钟。
```

initial语句将*Start*置为1，这促使或非门的输出为 0(从x值中获得)。5个时间单位后，在*Start*变为0时，或非门反转产生带有周期为 4个时间单位的时钟波形。产生的波形如图 11-4所示。

如果要产生高低电平持续时间不同的时钟波形，可用 always语句建立模型，如下所示：

```
module Gen_Clk_C (Clk_C);
  parameter tON = 5, tOFF = 10;
  output Clk_C;
```

```
  reg Clk_C;

  always
    begin
      # tON    Clk_C= 0;
      # tOFF   Clk_C= 1;
    end
endmodule
```

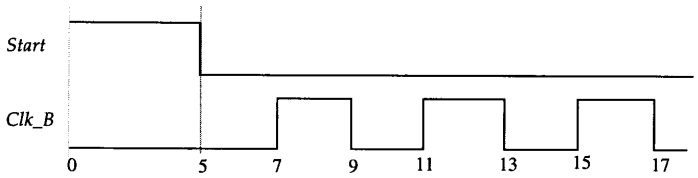


图11-4 受控时钟

因为值0和1被显式地赋值，在这种情况下不必使用 initial语句。图11-5显示了这一模块生成的波形。

为在初始时延后产生高低电平持续时间不同的时钟，可以在 initial语句中使用forever循环语句。

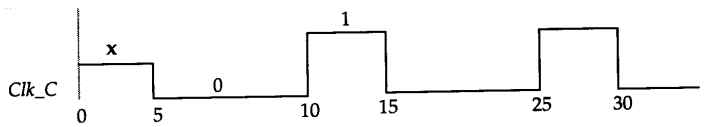


图11-5 高低电平持续时间不同的时钟

```
module Gen_Clk_D (Clk_D);
  output Clk_D;
  reg Clk_D;
  parameter START_DELAY = 5, LOW_TIME = 2, HIGH_TIME = 3;

  initial
    begin
      Clk_D = 0;
      # START_DELAY ;

      forever
        begin
          # LOW_TIME ;
          Clk_D = 1;
          # HIGH_TIME;
          Clk_D = 0;
        end
    end
endmodule
```

上面模块所产生的波形如图11-6所示。

为产生确定数目的时钟脉冲，可以使用 repeat循环语句。下面带参数的时钟模块产生一定数目的时钟脉冲数列，时钟脉冲的高低电平持续时间也是用参数表示的。

```
module Gen_Clk_E(Clk_E);
  output Clk_E;
  reg Clk_E;
  parameter Tburst = 10, Ton = 2, Toff = 5;

  initial
    begin
      Clk_E = 1'b0;

      repeat(Tburst)
        begin
        # Toff Clk_E= 1'b1;
        # Ton Clk_E= 1'b0;
        end
    end
endmodule
```

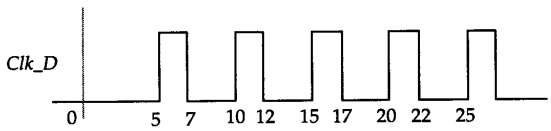


图11-6 带有初始时延的时钟

模块*Gen_Clk_E*在具体应用时，参数*Tburst*、*Ton*和*Toff*可带不同的值。

```
module Test;
  wire Clk_Ea, Clk_Eb, Clk_Ec;

  Gen_Clk_E G1(Clk_Ea);
    //产生10个时钟脉冲，高、低电平持续时间分别为2个和5个时间单位。

  Gen_Clk_E  # (5, 1, 3)  C2(Clk_Eb);
    //产生5个时钟脉冲，高、低电平持续时间分别为1个和3个时间单位。

  Gen_Clk_E  # (25, 8, 10)  C3(Clk_Ec);
    //产生25个时钟脉冲，高、低电平持续时间分别为8个和10个时间单位。
endmodule
```

*Clk_Eb*的波形如图11-7所示。

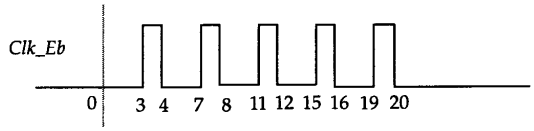


图11-7 确定数目的时钟脉冲

可用连续赋值产生一个时钟的相移时钟。下述模块产生的两个时钟波形如图11-8所示。一个时钟是另一个时钟的相移时钟。

```
module Phase (Master_Clk, Slave_Clk);
  output Master_Clk, Slave_Clk;
  reg Master_Clk;
```

```
  wire Slave_Clk;
  parameter tON = 2, tOFF = 3, tPHASE_DELAY= 1;

  always
    begin
      #tON Master_Clk= 0;
      #tOFF Master_Clk= 1;
  end

  assign #tPHASE_DELAY Slave_Clk= Master_Clk;
endmodule
```

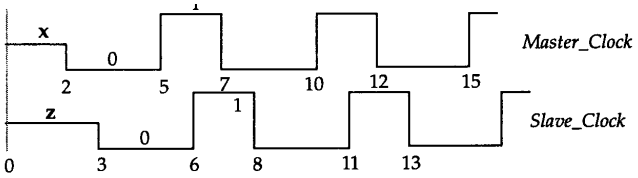


图11-8 相移时钟

11.3 测试验证程序实例

11.3.1 解码器

下面是2-4解码器和它的测试验证程序。任何时候只要输入或输出信号的值发生变化，输出信号的值都会被显示输出。

```
`timescale 1ns / 1ns
module Dec2x4 (A, B, Enable, Z);
  input A, B, Enable;
  output [0:3] Z;
  wire Abar, Bbar;

  not # (1, 2)
    V0 (Abar, A),
    V1 (Bar, B);

  nand # (4, 3)
    N0 (Z [0],Enable, Abar, Bbar),
    N1 (Z [1],Enable, Abar, B),
    N2 (Z [2],Enable, A, Bbar),
    N3 (Z [3],Enable, A, B);
endmodule

module Dec_Test;
  reg Da, Db, Dena;
  wire [0:3] Dz;

  //被测试的模块：
  Dec2x4 D1(Da, Db, Dena, Dz);
```

```
  //产生输入激励：
  initial
    begin
      Dena = 0;
      Da = 0;
      Db = 0;
      #10 Dena = 1;
      #10 Da = 1;
      #10 Db = 1;
      #10 Da = 0;
      #10 Db = 0;
      #10 $stop;
    end

  //输出模拟结果：
  always
    @ (Dena or Da or Db or Dz)
      $display ("At time %t, input is %b%b%b, output is %b"
            $time, Da, Db, Dena, Dz);
endmodule
```

下面是测试模块执行时产生的输出。

```
At time                    4, input is 000, output is 1111
At time                   10, input is 001, output is 1111
At time                   13, input is 001, output is 0111
At time                   20, input is 101, output is 0111
At time                   23, input is 101, output is 0101
At time                   26, input is 101, output is 1101
At time                   30, input is 111, output is 1101
At time                   33, input is 111, output is 1100
At time                   36, input is 111, output is 1110
At time                   40, input is 011, output is 1110
At time                   44, input is 011, output is 1011
At time                   50, input is 001, output is 1011
At time                   54, input is 001, output is 0111
```

11.3.2 触发器

下例是主从D触发器及其测试模块。

```
module MSDFF (D, C, Q, Qbar);
  input D, C;
  output Q, Qbar;

  not
    NT1 (NotD, D),
    NT2 (NotC, C),
    NT3 (NotY, Y);
  nand
    ND1 (D1, D, C),
    ND2 (D2, C, NotD),
    ND3 (Y, D1, Ybar),
```

```
    ND4 (Ybar, Y, D2),
    ND5 (Y1, Y, NotQ),
    ND6 (Y2, NotY, NotQ),
    ND7 (Q, Qbar, Y1),
    ND8 (Qbar, Y2, Q);
endmodule

module Test;
  reg D, C;
  wire Q, Qb;

  MSDFF M1(D, C, Q, Qb);

  always
    #5 C = ~C;

  initial
    begin
      D = 0;
      C = 0;
      #40 D = 1;
      #40 D = 0;
      #40 D = 1;
      #40 D = 0;
      $stop;
    end

  initial
    $ monitor ("Time = %t ::", $time,"C=%b, D=%b, Q=%b,
            Qb=%b", C,D, Q, Qb );
endmodule
```

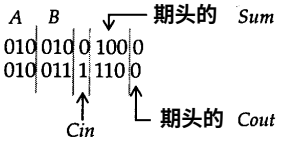
在此测试验证模块中，触发器的两个输入和两个输出结果均设置了监控，故只要其中任何值发生变化就输出指定变量的值。下面是执行产生的输出结果。

```
Time=                  0:: C=0, D=0, Q=x, Qb=x
Time=                  5:: C=1, D=0, Q=x, Qb=x
Time=                 10:: C=0, D=0, Q=0, Qb=1
Time=                 15:: C=1, D=0, Q=0, Qb=1
Time=                 20:: C=0, D=0, Q=0, Qb=1
Time=                 25:: C=1, D=0, Q=0, Qb=1
Time=                 30:: C=0, D=0, Q=0, Qb=1
Time=                 35:: C=1, D=0, Q=0, Qb=1
Time=                 40:: C=0, D=1, Q=0, Qb=1
Time=                 45:: C=1, D=1, Q=0, Qb=1
Time=                 50:: C=0, D=1, Q=1, Qb=0
Time=                 55:: C=1, D=1, Q=1, Qb=0
Time=                 60:: C=0, D=1, Q=1, Qb=0
Time=                 65:: C=1, D=1, Q=1, Qb=0
Time=                 70:: C=0, D=1, Q=1, Qb=0
Time=                 75:: C=1, D=1, Q=1, Qb=0
Time=                 80:: C=0, D=0, Q=1, Qb=0
```

```
Time=                 85:: C=1, D=0, Q=1, Qb=0
Time=                 90:: C=0, D=0, Q=0, Qb=1
Time=                 95:: C=1, D=0, Q=0, Qb=1
Time=                100:: C=0, D=0, Q=0, Qb=1
Time=                105:: C=1, D=0, Q=0, Qb=1
Time=                110:: C=0, D=0, Q=0, Qb=1
Time=                115:: C=1, D=0, Q=0, Qb=1
Time=                120:: C=0, D=1, Q=0, Qb=1
Time=                125:: C=1, D=1, Q=0, Qb=1
Time=                130:: C=0, D=1, Q=1, Qb=0
Time=                135:: C=1, D=1, Q=1, Qb=0
Time=                140:: C=0, D=1, Q=1, Qb=0
Time=                145:: C=1, D=1, Q=1, Qb=0
Time=                150:: C=0, D=1, Q=1, Qb=0
Time=                155:: C=1, D=1, Q=1, Qb=0
```

11.4 从文本文件中读取向量

可用$readmemb系统任务从文本文件中读取向量(可能包含输入激励和输出期望值)。下面为测试3位全加器电路的例子。假定文件“test.vec”包含如下两个向量。



向量的前三位对应于输入A，接下来的三位对应于输入B，再接下来的位是进位，八到十位是期望的求和结果，最后一位是期望进位值的输出结果。下面是全加器模块和相应的测试验证程序。

```
module Adder1Bit (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;

  assign Sum = (A ^ B ) ^ Cin;
  assign Cout = (A ^ B )| (A & Cin) | (B & Cin);
endmodule

module Adder3Bit(First, Second, Carry_In,Sum_Out, Carry_Out);
  input [0:2] First, Second;
  input Carry_In;
  output [0:2] Sum_Out;
  output Carry_Out;
  wire [0:1]Car;

  Adder1Bit
    A1 (First[2], Second[2], Carry_In,Sum_Out[2], Car[1]),
    A2 (First[1], Second[1], Car[1], Sum_Out [1],Car[0]),
    A3 (First[0], Second[0], Car[0], Sum_Out [0], Carry_Out);
endmodule

module TestBench;
```

```
    parameter BITS = 11, WORDS= 2;
    reg [1:BITS]Vmem [1:WORDS];
    reg[0:2]A,B,Sum_Ex;
    reg Cin, Cout_Ex;
    integer J;
    wire [0:2] Sum;
    wire Cout;

    //被测试验证的模块实例。
    Adder3Bit   F1 (A, B, Cin, Sum, Cout);

    initial
      begin
        $readmemb ("test.vec", Vmem);

        for (J = 1; J <= WORDS; J = J + 1)
          begin
            {A, B, Cin, Sum_Ex, Cout_Ex} = Vmem [J];
            #5;  //延迟5个时间单位等待电路稳定。

            if ((Sum ! = = Sum_Ex) | | (Cout ! = = Cout_Ex))
              $display ("****Mismatch on vector %b ****", Vmem [J]);
            else
              $display ("No mismatch on vector %b", Vmem [J]);
          end
      end
  endmodule
```

测试模块中首先定义存储器 *Vmem*，字长对应于每个向量的位数，存储器字数对应于文件中的向量数。系统任务 $**readmemb**从文件“ test.vec ”中将向量读入存储器 *Vmem*中。for循环通过存储器中的每个字，即每个向量，将这些向量应用于待测试的模块，等待模块稳定并探测模块输出。条件语句用于比较期望输出值和监测到的输出值。如果发生不匹配的情况，则输出不匹配消息。下面是以上测试验证模块模拟执行时产生的输出。因为模型中不存在错误，因此没有报告不匹配情形。

```
No mismatch on vector 01001001000
No mismatch on vector 01001111100
```

11.5 向文本文件中写入向量

在上节的模拟验证模块实例中，我们看到值如何被打印输出。设计中的信号值也能通过如$**fdisplay**、$fmonitor和$fstrobe等具有写文件功能的系统任务输出到文件中。下面是与前一节中相同的测试验证模块实例，本例中的验证模块将所有输入向量和观察到的输出结果输出到文件“ mon.Out ”中。

```
  module F_Test_Bench;
    parameter BITS = 11, WORDS= 2;
    reg [1:BITS] Vmem [1:WORDS];
    reg [0:2] A, B, Sum_Ex;
    reg Cin,Cout_Ex
```

```
  integer J;
  wire [0:2] Sum;
  wire Cout;

  //待测试验证模块的实例。
  Adder3Bit   F1 (A, B, Cin, Sum, Cout);

  initial
    begin: INIT_LABLE
      integer Mon_Out_File;

      Mon_Out_File= $fopen ("mon.out");
      $readmemb ("test.vec", Vmem);

      for (J = 1; J <= WORDS; J = J + 1)
        begin
          {A, B, Cin, Sum_Ex, Cout_Ex} = Vmem [J];
          #5;  //延迟5个时间单位，等待电路稳定。

          if ((Sum ! = = Sum_Ex) | | (Cout ! = = Cout_Ex))
            $display ("****Mismatch on vector %b ****", Vmem [J]);
          else
            $display ("No mismatch on vector %b", Vmem [J]);

          //将输入向量和输出结果输入到文件：
          $fdisplay (Mon_Out_File,"Input = %b%b%b, Output = %b%b",
                  A, B, Cin, Sum, Cout);
        end
      $fclose (Mon_Out_File);
    end
endmodule
```

下面是模拟执行后文件“ mon.out ”包含的内容。

```
Input = 0100100, Output = 1000
Input = 0100111, Output = 1100
```

11.6 其他实例

11.6.1 时钟分频器

下面是应用波形方法的完整测试验证程序。待测试的模块名为 *Div*。输出响应写入文件以便于以后进行比较。

```
module Div (Ck, Reset, TestN, Ena);
  input Ck, Reset, TestN;
  output Ena;
  reg [0:3] Counter;

  always
    @ (posedge Ck) begin
      if (~Reset)
```

```
        Counter = 0;
      else
        begin
          if (~ TestN)
            Counter = 15;
          else
            Counter = Counter + 1
        end
      end
  assign Ena = (Counter = = 15) ? 1: 0;
endmodule

module Div_TB;
  integer Out_File;
  reg Clock, Reset, TestN;
  wire Enable;

  initial
    Out_File = $fopen ("out.vec");

  always
    begin
      #5 Clock = 0;
      #3 Clock = 1;
    end

  Div D1 (Clock, Reset, TestN, Enable);

  initial
    begin
      Reset = 0;
      #50 Reset = 1;
    end

initial
  begin
    TestN = 0;
    #100 TestN = 1;
    #50 TestN = 0;
    #50 $fclose (Out_File);
    $finish;          /模拟结束。
  end
//将使能输出信号上的每个事件写入文件。

  initial
    $fmonitor (Out_File,"Enable changed to %b at time %t", Enable, $time);
endmodule
```

模拟执行后，文件“ out.vec **”所包含的输出结果如下：**

```
Enable changed to x at time            0
Enable changed to 0 at time            8
```

```
Enable changed to 1 at time          56
Enable changed to 0 at time         104
Enable changed to 1 at time         152
```

11.6.2 阶乘设计

本例介绍产生输入激励的另一种方式，在该方式中，根据待测模块的状态产生相应的输出激励。该输出激励产生方式对有穷状态自动机（ FSM）的模拟验证非常有效，因为状态机的模拟验证需根据各个不同的状态产生不同的输入激励。设想一个用于计算输入数据阶乘(factorial)的设计。待测试模块与测试验证模块之间的握手机制如图 11-9所示。

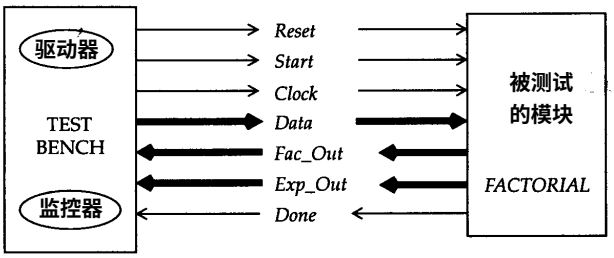


图11-9 测试验证模块与待测试模块间的握手机制

模块的输入信号 *Reset*将阶乘模型复位到初始状态，在加载输入数据 *Data*后，*Start*信号被置位；计算完成时，对输出 *Done*置位，表明计算结果出现在输出 *Fac_Out*和*Exp_Out*上。阶乘结果值为*Fat_Out* $*2^{Exp\_Out}$，测试验证模块在 *Data*上提供从值1开始递增到20的输入数据。测试验证模块加载数据，对 *Start*信号置位并等待*Done*信号有效，然后加载于下一输入数据。若输出结果不正确，即打印错误信息。阶乘模块及其测试验证模块描述如下：

```
`timescale 1ns / 1ns
module FACTORIAL(Reset, StartSig, Clk, Data, Done,
                 FacOut, ExpOut);
  input Reset, StartSig, Clk;
  input [4:0] Data;
  output Done;
  output [7:0] FacOut, ExpOut;

  reg Stop;
  reg [4 : 0] InLatch;
  reg [7:0] Exponent, Result;
  integer I;

  initial Stop = 1;

  always
    @ (posedge Clk) begin
      if (( StartSig = = 1) && (Stop = = 1) && (Reset = = 1))
        begin
          Result = 1;
          Exponent = 0;
          InLatch = Data;
```

```
            Stop = 0;
          end
        else
          begin
            if (( InLatch > 1) && (Stop = = 0)
              begin
                Result = Result * InLatch;
                InLatch = InLatch- 1;
              end

            if (InLatch < 1)
              Stop = 1;
        //标准化：
          for (I = 1; I <= 5; I = I + 1)
            if (Result >256)
              begin
                Result = Result / 2;
                Exponent = Exponent+ 1;
              end
          end
    end

    assign Done = Stop;
    assign FacOut = Result;
    assign ExpOut = Exponent;
  endmodule

  module FAC_TB;
    parameter IN_MAX = 5, OUT_MAX = 8;
    parameter RESET_ST = 0, START_ST = 1, APPL_DATA_ST = 2,
              WAIT_RESULT_ST = 3;
    reg Clk, Reset, Start;
    wire Done;
    reg [IN_MAX-1 : 0] Fac_Out, Exp_Out;
    integer Next_State;
    parameter MAX_APPLY = 20;
    integer Num_Applied;

    initial
      Num_Applied= 1;

    always
      begin: CLK_P
        #6 Clk = 1;
        #4 Clk = 0;
      end

    always
      @ (negedge Clk)   //时钟下跳边沿触发
        case (Next_State)
```

```
        RESET_ST:
          begin
            Reset = 1;
            Start = 0;
          Next_State = APPL_DATA_ST;
          end
        APPL_DATA_ST:
          begin
            Data = Num_Applied;
            Next_State = START_ST;
          end
        START_ST:
          begin
            Start = 1;
            Next_State = WAIT_RESULT_ST;
          end
        WAIT_RESULT_ST:
          begin
            Reset = 0;
            Start = 0;
            wait (Done = = 1);

            if (Num_Applied = =
                        Fac_Out *('h0001 <<Exp_Out))
              $display ("Incorrect result from factorial",
                        "model for input value %d", Data);

            Num_Applied = Num_Applied + 1;

            if (Num_Applied < MAX_APPLY)
              Next_State = APPL_DATA_ST;
            else
              begin
                $display ("Test completed successfully");
                $finish;  //模拟结束。
              end
            end
          default:
            Next_State = START_ST;
        endcase

      //将输入激励加载到待测试模块：
      FACRORIAL F1(Reset, Start, Clk, Data, Done,
                   Fac_Out, Exp_Out);
    endmodule
```

11.6.3 时序检测器

下面是时序检测器的模型。模型用于检测数据线上连续三个 1的序列。在时钟的每个下沿检查数据。图11-10列出了相应的状态图。带有测试验证模块的模型描述如下：

```
module Count3_ls(Data, Clock, Detect3_ls);
  input Data, Clock;
  output Detect3_ls;
  integer Count;
  reg Detect3_1s;

  initial
    begin
      Count = 0;
      Detect3_ls= 0;
    end

  always
    @ (negedge Clock) begin
      if ( Data = = 1)
        Count = Count + 1;
      else
        Count = 0;

      if (Count >= 3)
        Detect3_ls = 1;
      else
        Detect3_ls= 0;
    end
  endmodule

  module Top;
    reg Data, Clock;
    integer Out_File;

    //待测试模块的应用实例。
    Count3_ls   F1(Data, Clock, Detect);

    initial
      begin
    Clock = 0;

      forever
        #5 Clock = ~ Clock;
    end

  initial
    begin
     Data = 0;
      #5 Data = 1;
      #40 Data = 0;
      #10 Data = 1;
      #40 Data = 0;
      #20 $stop;   // 模拟结束。
    end
```

```
    initial
      begin
        //在文件中保存监控信息。
        Out_File = $fopen ("results.vectors");
        $fmonitor (Out_File,"Clock = %b, Data = %b, Detect = %b",
                  Clock, Data, Detect);
      end
  endmodule
```

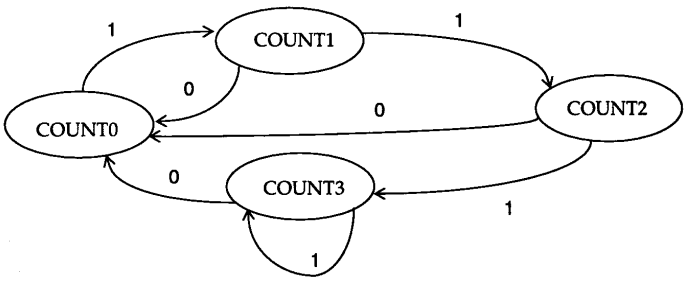


图11-10 时序检测器

习题

1. 产生一个高电平持续时间和低电平持续时间分别为 3 ns和10 ns的时钟。
2. 编写一个产生图11-11所示波形的Verilog HDL模型。

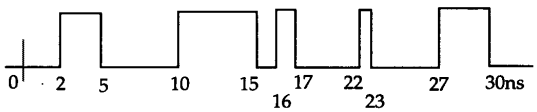


图11-11 波形

3. 产生一个时钟*ClockV*，该时钟是模块*Gen_Clk_D*中描述的时钟*Clk_D*(如图11-6所示)的相移时钟，相位延迟为15 ns。[提示：用连续赋值语句可能会不合适。]
4. 编写测试时序检测器的测试验证程序。时序列检测器按模式 10010在每个时钟正沿检查输入数据流。如果找到该模式，将输出置为 1；否则输出置为0。
5. 编写一个模块生成两个时钟， *ClockA*和*ClockB*。*ClockA* 延迟10 ns后有效，*ClockB*延迟40 ns后有效。两个时钟有相同的高、低电平持续时间，高电平持续时间为 1 ns，低电平持续时间为2ns。*ClockB*与时钟*ClockA*边沿同步，但极性相反。
6. 描述4位加法/减法器的行为模型。用测试验证模块测试该模型。在测试验证模块内描述所有输入激励及其期望的输出值。将输入激励、期望的输出结果和监控输出结果转储到文本文件中。
7. 描述在两个4位操作数上执行所有关系操作(<, <=, >, >=)的ALU。编写一个从文本文件中读取测试模式和期望结果的测试验证模块。
8. 编写一个对输入向量作算术移位操作的模块。指定输入长度用参数表示，缺省值为 32。同时指定移位次数用参数表示，缺省值为 1。编写一个模拟、测试模块以验证对 12位向量进行

8次移位算术操作的正确性。

9. 编写*N*倍时钟倍频器模型。输入是频率未知的参考时钟。输出时钟的倍数与参考时钟的每个正沿同步。[提示：确定参考时钟的时钟周期。
10. 编写一个模型，显示输入时钟每次由0转换到1的时间。
11. 编写一个计数器模型，该计数器在*Count_Flag*为1期间对时钟脉冲(正沿)计数。如果计数超过*MAX_COUNT*，*OverFlow*被置位，并且计数值停留在*MAX_COUNT*界限上。*Count_Flag*的上沿促使计数器复位到0，并重新开始计数。编写测试验证模块，并测试该模型的正确性。
12. 编写参数化的格雷（Gray）码计数器，其缺省长度是3。当变量*Reset*为0时，计数器被异步复位。计数器在每个时钟负沿计数。然后在模拟验证模块中对 4位格雷码计数器进行测试验证。
13. 编写一个带异步复位的T触发器的行为模型。如果开关为1，输出在0和1之间反复。如果开关为0，输出停留在以前状态。接下来用*specify*块指定T触发器的数据建立时间为2 ns，保持时间为3 ns。编写模拟测试模块以测试该模型。