第12章 建模实例

本章给出了一些用Verilog HDL编写的硬件建模实例。

12.1 简单元件建模

连线是一种最基本的硬件单元。连线在 Verilog HDL中可被建模为线网数据类型。考虑 4 位与门，其行为描述如下：

```
`timescale 1ns/1ns
module And4 (A, B, C);
  input [3:0] B, C;
  output [3:0] A;

  assign #5 A = B & C;
endmodule
```

&(与)逻辑的时延定义为5 ns。这个模型代表的硬件如图 12-1所示。

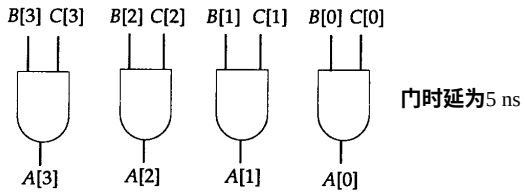

图12-1 一个4位与门

本实例和下面的实例表明布尔等式如何在连续赋值语句中被建模为表达式。连线单元能被建模为线网数据类型。例如，在下面的描述中， F表示将~ (非)操作符的输出连接到^ (异或)操作符输入的线网。该模块表示的电路如图 12-2所示。

```
module Boolean_Ex (D, G, E, );
  input G, E;
  output D;
  wire F;

  assign F = ~E;
  assign D = F ^ G;
endmodule
```

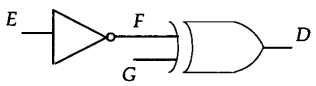

图12-2 组合电路

考虑下列行为和如图 12-3所示相应硬件的表示。

```
module Asynchronous;
  wire A, B, C, D;

  assign C = A | D
  assign A = ~ (B & C);
endmodule
```

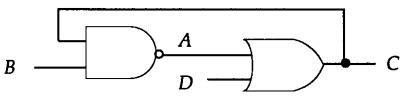

图12-3 异步反馈环路

该电路带有一个异步反馈环路。如果模型用特定的值集 (*B*=1, *D*=0)仿真，仿真时间将由于仿真器总在两个赋值语句间迭代而永远停滞不前。迭代时间是两个零时延。因此，在使用带有零时延的连续赋值语句对线网赋值，以及在表达式中使用相同的线网值时，必须格外小心。

在特定的情况下，有时需要使用这样的异步反馈。下面将演示一个这样的异步反馈；语句代表一个周期为20 ns的周期性波形。其硬件表示如图 12-4所示。注意这样的always 语句需要一个initial 语句将寄存器初始化为0或1，否则寄存器的值将固定在值x上。

```
reg Ace;

initial
  Ace = 0;

always
  #10 Ace = ~ Ace;
```

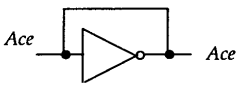

图12-4 时钟发生器

向量线网或向量寄存器型元件能被访问，既可以访问称为位选择的单个元素，也可以访问称为部分选择的片段。例如，

```
reg A;
reg [0:4] C;
reg [5:0] B, D;

always
  begin
    . . .
    D [4:0] = B [5:1] | C; //D [4:0]和B[5:1]都是部分选择。
    D [5] = A & B [5];  //D [5]和B[5]都是位选择。
  end
```

第一个过程性赋值语句暗示着：

```
D [4] = B [5] | C [0];
D [3] = B [4] | C [1];
. . .
```

位选择、部分选择和向量可以并置，形成更大的向量。如，

```
wire [7:0] C, CC;
wire CX;
. . .
assign C = {CX, CC [6:0]};
```

也可以引用索引值在运行时才可计算的向量元素。如：

```
Adf = Plb[K];
```

意味着解码器的输出为*Adf*，并且*K*指定选择地址。*Plb*是向量；本语句对解码器的行为建模。

可以使用预定义的移位操作符执行移位操作。移位操作可以用合并操作符建模。例如，

```
wire [0:7] A, Z;
. . .
assign Z = {A [1:7], A [0]};          /循环左移。
assign Z = {A [7], A [0:6]};          /循环右移。
assign Z = {A [1:7], 1'b0};           左移操作。
```

向量的子域称为部分选择，也能够在表达式中使用。例如， 32位指令寄存器*Instr_Reg*中前16位表示地址，接下来8位表示操作码，余下的8位表示索引。给出如下说明：

```
reg [31:0] Memory [0: 1023 ];
wire [31:0] Instr_Reg;
wire [15:0] Address;
wire [7:0] Op_Code, Index;
wire [0:9] Prog_Ctr;
wire Read_Ctl;
```

从*Instr_Reg*读取子域信息的一种方法是使用三个连续赋值语句。指令寄存器的部份选择被赋值给指定的线网。

```
assign INstr_Reg =  Memory[Prog_Ctr];

assign Address = Instr_Reg[31:16];
assign Op_Code = Instr_Reg[15:8];
assign Index = Instr_Reg[7:0];
...
always
  @(posedge Read_Ctl)
     Task_Call( Address, Op_Code, Index);
```

可用连续赋值语句为三态门的行为建模，如：

```
wire Triout = Enable ? TriIN 1'bz;
```

当*Enable*为1时，*TriOut*获得*TriIn*的值。当*Enable*为0时，*TriOut*为高阻值。

12.2 建模的不同方式

本节给出了Verilog HDL语言提供的三种不同建模方式的实例：数据流方式、行为方式和结构方式。参看图12-5所示的电路，该电路将输入*A*的值存入寄存器，然后与输入*C*相乘。

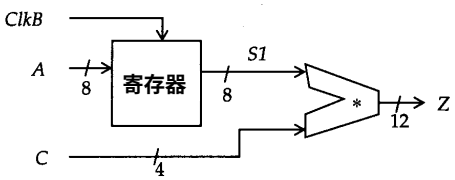

图12-5 带缓冲的乘法器

第一种建模方式是数据流方式，它使用连续赋值语句对电路建模。

```
module Save_Mult_Df(A,C,ClkB,Z);
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
  output [0:11] Z;
  wire S1;

  assign Z = S1 * C;
  assign S1 = ClkB ? A : S1;
endmodule
```

这种表示方法不直接蕴含任何结构，却隐式地描述了结构。但是，它的功能性非常清晰。寄存器已使用时钟控制方式建模。

第二种描述电路的方法是使用带有顺序程序块的 always 语句，将电路建模为顺序程序描述。

```
module Save_Mult_Seq (A,C,ClkB,Z);
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
```

```
  output [0:11] Z;
  reg [0:11] Z;

  always
    @(A or C or ClkB)
      begin: SEQ
        //由于标记了程序块，可以说明一个局部寄存器变量S1。
        reg [0:7] S1;

        if (ClkB)
           S1 =  A;

        Z = S1 * C;
      end
endmodule
```

这种模型也描述了电路的行为，但是没有蕴含任何结构，无论是隐式还是显示的。在这种方式中，寄存器用*if*语句建模。

第三种描述*Save_Mult*电路的方法是假定已存在8位寄存器和8位乘法器，将电路建模为线网表。

```
module Save_Mult_Netlist (A,C,ClkB,Z)
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
  output [0:11] Z;
  wire [0:7] S1,S3;
  Wire [0:15] S2;

  Reg8   R1 (.Din(A),.Clk(ClkB),.Dout(S1));
  Mult8  M1 (.A(S1),.B({4'1b0000,C}),.Z(Z));
endmodule
```

这种描述方式显式地描述了电路结构，但其行为是未知的。这是因为 *Reg8*和*Mult8*的模块名是任意的，并且它们可以有与其相关的各种行为。

12.3 时延建模

考虑3输入或非门。它的行为可以使用连续赋值语句建模，如

```
assign #12 Gate_Out = ~(A | B | C);
```

这条语句对带有12个时间单位时延的或非门建模。这一时延表示从信号A、B或C上事件发生到结果值出现在信号*Gate_Out*上的时间。一个事件可以是任何值的变化，如x -> z、 x - > 0, 或者1- > 0。

如果要显式地对上升和下降时间建模，则在连续赋值语句中使用两个时延，例如：

```
assign #(12,14) Zoom = ~(A | B | C);
  /*12是上升时延，14是下降时延，min(12,14) = 1是转换到x的时延*/
```

在能够对高阻值Z赋值的逻辑中，也能够定义第三种时延，即关断时延。例如：

```
assign #(12,14,10)Zoom = A > B ? C : 1bz;
  //上升时延为12，14是下降时延，min(12,14，10) = 1是转换到x的时延，关断时延为10。
```

每个时延值都能够用*min:typ:max*表示，例如，

```
assign #(9:10:11,11:12:13,13:14:15) Zoom = A > B ? C : 1bz;
```

时延值通常可以是表达式。

基本门和UDP中的时延可以通过在实例中指定时延值建模。下面是5输入基本与门。

```
and #(2,3) A1(Ot,In1,In2,In3,In4,In5);
```

已指定输出的上升时延为2个时间单位，下降时延为3个时间单位。

模型中位于端口边界的时延可使用指定程序块来定义。如下面的半加器模块的例子。

```
module  Half_Adder(A,B,S,C);
  input A,B;
  output S,C;

  specify
    (A => S) = (1.2,0.8);
    (B => S) = (1.0,0.6);
    (A => C) = (1.2,1.0);
    (B => C) = (1.2,0.6);
  endspecify

  assign S = A ^ B;
  assign C = A | B;
endmodule
```

除在连续赋值语句中对时延建模外，时延还可以用指定程序块建模。是否存在一种从模块外部指定时延的方法？一种选择是使用SDF[⊖]（标准时延格式）和Verilog仿真器可能提供的反标机制。如果需要在Verilog HDL模型中显式地指定这一信息，一种方法是在Half-Adder模块上创建两个虚模块，每个模块带有不同的时延集。

```
module  Half_Adder(A,B,S,C);
  input A,B;
  output S,C;

  assign S = A ^ B;
  assign C = A | B;
endmodule

module  Ha_Opt(A,B,S,C);
  input A,B;
  output S,C;

  specify
    (A => S) = (1.2,0.8);
    (B => S) = (1.0,0.6);
    (A => C) = (1.2,1.0);
    (B => C) = (1.2,0.6);
  endspecify

  Half_Adder  H1(A,B,S,C);
endmodule

module  Half_Pess(A,B,S,C);
```

⊖ 见参考文献

```
  input A,B;
  output S,C;

  specify
    (A => S) = (0.6,0.4);
    (B => S) = (0.5,0.3);
    (A => C) = (0.6,0.5);
    (B => C) = (0.6,0.3);
  endspecify

  Half_Adder  H1(A,B,S,C);
endmodule
```

有了这两个模块，*Half_Adder*模块独立于任何时延，并且依赖于你所选用的时延方式，即模拟相应的高层模块*Ha_Opt*或*Ha_Pess*。

传输时延

在连续赋值语句和门级原语模型中指定的时延为惯性时延。传输时延能够用带有语句内时延的非阻塞性赋值语句建模。实例如下，

```
module Transport (WaveA,DelayedWave);
  parameter TRANSPORT_DELAY = 500;
  input WaveA;
  output DelayedWave;
  reg DelayedWave;

  always
    @(WaveA) DelayedWave <= #TRANSPORT_DELAY WaveA;
endmodule
```

always语句中包含带有语句内时延的非阻塞性赋值。 *WaveA*上的任何变化以后都会使*DelayedWave*在延迟*TRANSPORT_DELAY*时获得调度。结果在*WaveA*上出现的波形在被延迟*TRANSPORT_DELAY*后出现在*DelayedWave*上；这种时延波形的实例如图12-6所示。

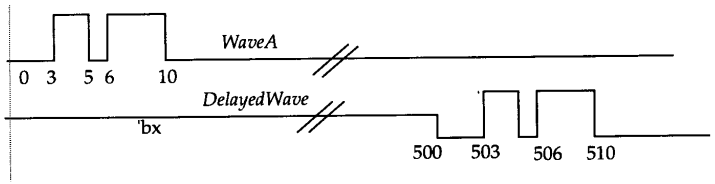

图12-6 传输时延实例

12.4 条件操作建模

在特定条件下发生的操作可以使用带有条件操作符的连续赋值语句，或在 always语句中用if语句或case语句建模。请看一个算术逻辑电路。它的行为可用如下所示的连续赋值语句建模。

```
module  Simple_ALU(A,B,C, PM, ALU);
  input [0:3] A,B, C;
  input PM;
```

```
  output [0:3] ALU;

  assign ALU = PM ? A + B: A - B;
endmodule
```

多路选择开关也能够使用 always 语句建模。首先确定选线的值，然后， case语句根据这个值选择相应的输入赋值到输出。

```
`timescale 1ns/1ns
module  Multiplexer(Sel, A,B,C,D, Mux_Out);
  input [0:1] Sel;
  input A, B,C, D;
  output Mux_Out;
  reg Mux_Out;
  reg Temp;
  parameter MUX_DELAY= 15;

  always
    @ (Sel or A or B or C or D)
      begin: P1
        case (Sel)
          0: Temp = A;
          1: Temp = B;
          2: Temp = C;
          3: Temp = D;
         endcase
        Mux_Out = # MUX_DELAY Temp;
    end
endmodule
```

多路选择开关也可以用如下形式的连续赋值语句建模：

```
assign #MUX_DELAY Mux_Out= (Sel == 0)? A : (Sel == 1)? B :
                 (Sel == 2)? C : (Sel == 3)? D : 1'bx;
```

12.5 同步时序逻辑建模

到目前为止，本章中的绝大多数实例都是组合逻辑。对于同步时序逻辑建模，语言中已提供寄存器数据类型对寄存器和存储器建模。但是并不意味着每种寄存器数据类型都可对同步时序逻辑建模。通过控制赋值方式对同步时序逻辑建模是一种常见的方法。

参见如下实例，它表明了如何通过控制寄存器对同步边沿触发的 D型触发器建模。

```
`timescale 1ns/1ns
module  D_Flip_Flop (D, Clock, Q);
  input D, Clock;
  output Q;
  reg Q;

always
  @ (posedge Clock)
     Q = #5 D;
endmodule
```

always语句表明当 *Clock*上出现上升沿时， *Q*会在5 ns后被赋值为 *D*；否则*Q*不发生变化（寄存器在被赋新值前保持原值）。always语句中的行为表达了*D*型触发器的语义。提供这一模

型后，8位寄存器可按如下方式进行建模。

```
module Register8(D, Q, Clock);
  parameter START = 0, STOP = 7;
  input [START : STOP] D;
  input Clock;
  output [START : STOP] Q;
  wire [START : STOP] Cak;

  D_Flip_Flop  DFF0
              [START : STOP] (. D (D), . Clock (Cak), .Q (Q));

  buf  B1      (Cak [0], Cak [1], Cak [2], Cak [3], Cak [4],
                Cak [5], Cak [6], Cak [7], Clock);
endmodule
```

考虑如图12-7所示的门级交叉耦合锁存器电路及其数据流模型。

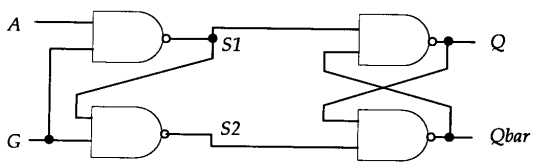

图12-7 门级锁存器

```
module Gated_FF (A, G, Q, Qbar);
  input A, G;
  output Q, Qbar;
  wire S1, S2;

  assign S1 = ~ (A & G);
  assign S2 = ~ (S1 & G);
  assign Q = ~ (Qbar & S1);
  assign Qbar = ~ (Q & S2);
endmodule
```

在此例中，连续赋值语句的语义蕴含了锁存器结构。

存储器可用寄存器数组建模。实例如下，其中*ASIZE*是地址端口的位数，*DSIZE*是RAM数据端口的位数。

```
module RAM_Generic(Address, Data_In, Data_out, RW);
  parameter ASIZE = 6, DSIZE = 4;
  input [ASIZE-1 :0] Address;
  input [DSIZE-1 :0] Data_In;
  input RW;
  output [DSIZE-1 :0] Data_Out;
  reg [DSIZE－1:0] Data_Out;
  reg [0: DSIZE－1] Mem_FF [0:63];

  always
    @ (RW)
      if (RW)          //从RAM中读取数据。
        Data_Out = Mem_FF [Address];
```

```
        else          //向RAM写入数据。
          Mem_FF [Address] = Data_In;
    endmodule
```

同步时序逻辑也可以用电平敏感或边沿触发控制方式建模。电平敏感类型锁存器建模实例如下。

```
    module Level_Sens_FF(Strobe, D, Q, Qbar);
      input Strobe, D;
      output Q, Qbar;
      reg Q, Qbar;

      always
        begin
          wait (Strobe = = 1);
            Q = D;
            Qbar = ~ D;
        end
    endmodule
```

当*strobe*为1时，*D*上的任何事件都被传输到*Q*；当*Strobe*变成0时，*Q*和*Qbar*中的值保持不变，并且输入*D*上的任何变化都不再影响*Q*和*Qbar*上的值。

理解过程性赋值语句的语义对决定同步时序逻辑的行为功能非常重要。考虑模块 *Body1*和*Body2*之间的不同之处。

```
    module Body1;
      reg A;

    initial A = 0;

      always A = ~ A;
    endmodule

    module Body2;
      wire Clock;
      reg A;

      initial A = 0;

      always
        @ (Clock )
          if (~ Clock)
            A = ~ A;
    endmodule
```

模块*Body1*蕴含的电路结构如图12-8所示，而模块*Body2*蕴含的电路结构如图12-9所示。

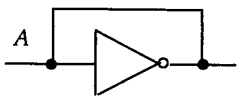

图12-8 不蕴含触发器

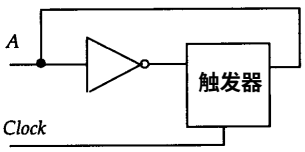

图12-9 蕴含触发器

如果*Body1*按如上所述进行模拟，模拟将由于零时延异步反馈而陷入死循环（模拟时间停止不前）。在模块*Body 2*中，*A*的值只有在*Clock*信号下降沿时锁存，在其他情形下（*Clock*不处在上升沿时），*A*上的任何变化（触发器的输入）都不会影响触发器的输出。

12.6 通用移位寄存器

通用串行输入、串行输出移位寄存器能够使用 always语句块内的for循环语句建模。寄存器的数量被定义为参数，这样通用移位寄存器的数量在其他设计中被引用时，可以修改。

```
module Shift_Reg (D, Clock, Z);
  input D, Clock;
  output Z;
  parameter NUM_REG = 6,
  reg [1: NUM_REG ] Q;
  integer P;

  always
    @ (negedge Clock) begin
      //寄存器右移一位：
      for (P = 1; P< NUM_REG; P = P + 1)
        Q[P+1] = Q[P];

      //加载串行数据：
      Q[1] = D;
    end

  //从最右端寄存器获取输出：
  assign Z = Q [NUM_REG];
endmodule
```

可以通过用不同的参数值引用模块 *Shift_Reg*获取不同长度的移位寄存器。

```
module Dummy;
  wire Data, Clk, Za, Zb, Zc;

  //6位移位寄存器：
  Shift_Reg  SRA(Data, Clk, Za);

  //4位移位寄存器：
  Shift_Reg #4 SRB (Data, Clk, Zb);

  //10位移位寄存器：
  Shift_Reg #10 SRC (Data, Clk, Zc);
endmodule
```

12.7 状态机建模

状态机通常可使用带有 always语句的case语句建模。状态信息存储在寄存器中。 case语句的多个分支包含每个状态的行为。下面是表示状态机简单乘法算法的实例。当 *Reset*信号为高时，累加器*Acc*和计数器 *Count*被初始化。当 *Reset*变为低时，乘法开始运算。如果乘数 *Mplr*在*Count*位的值为1，被乘数加到累加器上。然后，被乘数左移1位且计数器加1。如果*Count*是16，乘法运算完成，并且 *Done*信号被置为高。如若不然，检查乘数 *Mplr*的*Count*位，并重复 always语句。状态图如图12-10所示，其后是相应的状态机模型。

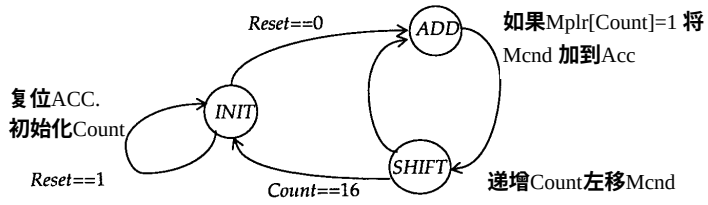

图12-10 乘法器的状态图

```
module Multiply (Mplr, Mcnd, Clock, Reset, Done, Acc);
       //Mplr是乘数，Mcnd是被乘数。
  input [15:0] Mplr, Mcnd;
  input Clock, Reset;
  output Done;
  reg Done;
  output [31:0] Acc;
  reg [31:0] Acc;
  parameter INIT = 0, ADD = 1, SHIFT = 2;
  reg [0:1] Mpy_State;
  reg [31:0] Mcnd_Temp;

  initial Mpy_State = INIT;      //初始状态为INIT。

  always
    @ (negedge Clock) begin: PROCESS
      integer Count;

      case (Mpy_State)
        INIT:
          if (Reset)
           Mpy_State = INIT;
           /*由于Mpy_State将保持原值，上面的语句并不需要*/。
          else
            begin
              Acc = 0;
              Count = 0;
              Mpy_State = ADD;
              Done = 0;
              Mcnd_Temp [15:0] =Mcnd;
              Mcnd_Temp [31:16] = 16'd0;
            end

        ADD:
          begin
            if (Mplr [Count])
              Acc = Acc + Mcnd_Temp;

              Mpy_State = SHIFT;
          end

        SHIFT:
```

```
        begin
          //Mcnd_Temp左移:
          Mcnd_Temp = {Mcnd_Temp[30:0], 1'b0};
          Count = Count+ 1;

          if (Count == 16)
            begin
              Mpy_State = INIT;
              Done = 1;
            end
          else
            Mpy_State = ADD;
        end
      endcase  //对Mpy_State的case语句结束。
    end  //顺序程序块PROCESS。
  endmodule
```

寄存器*Mpy_State*保存状态机模型的状态。最初，状态机模型处于 *INIT*状态，并且只要*Reset*为真，模型就停留在这一状态。当*Reset*为假时，累加器*Acc*被清空。计数器*Count*被复位，被乘数*Mcnd*被加载到临时变量 *Mcnd_Temp*中，然后模型状态前进到状态 *ADD*。当模型处于*ADD*状态时，只有当乘数在*Count*位置的位为1时，*Mcnd_Temp*中的被乘数才被加到*Acc*上，然后模型状态前进到 *SHIFT*状态。在这一状态，乘法器再一次左移，计数器加 1；并且如果计数器值为16，*Done*置为真，模型返回到*INIT*状态。此时，*acc*包含乘法运算的结果。如果计数器值小于16，模型本身反复通过*ADD*和*SHIFT*状态直到计数器值变为16。

状态转换发生在时钟的各个下跳沿；这通过使用@ (**negedge** *Clock*)时序控制来指定。

12.8 交互状态机

交互状态机能够使用通过公共寄存器通信的独立的 always语句进行描述。考虑图 12-11所示的两个交互进程的状态图， *TX*是一个发送器， *MP*是一个微处理器。如果进程 *TX*不忙，进程*MP*将要发送的数据放置在数据总线上，然后向进程 *TX*发送信号*Load_TX*，通知其装载数据并开始发送数据。进程 *TX*在数据传送期间设置*TX_Busy*表明其处于忙状态，不能从进程 *MP*接收任何进一步的数据。

下面显示了这两个交互进程的框架模型，图中仅显示了控制信号和状态转换，没有描述操作数据的代码。

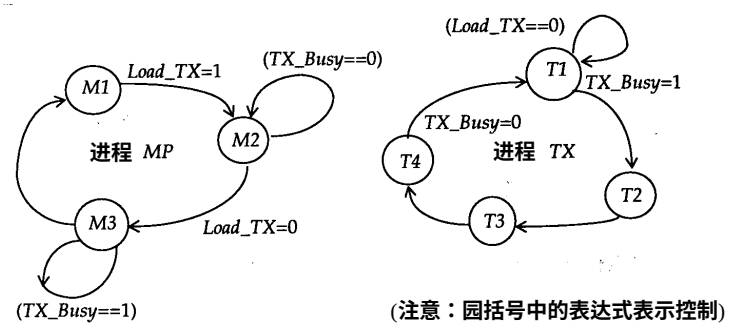

图12-11 两个交互进程的状态图

```
module Interacting_FSM(Clock);
  input Clock;

  parameter M1 = 0, M2 = 1, M3 = 2;
  parameter T1 = 0, T2 = 1; T3 = 2, T4 = 3;
  reg [0:1] MP_State;
  reg [0:1] TX_State;
  reg Load_TX, TX_Busy

  always
    @ (negedge Clock) begin: MP
      case (MP_State)
        M1:   //向数据总线装载数据。
          begin
            Load_TX = 1;
            MP_State = M2;
          end

        M2:   //等待确认信号。
          if (TX_Busy)
            begin
              MP_State = M3;
              Load_MX = 0;
            end

        M3:           /等待进程TX结束。
            if (~TX_Busy)
              MP_State = M1;
        endcase
  end   //顺序块MP结束。

  always
    @ (negedge Clock) begin: TX
      case (TX_State)
        T1:             /等待装载的数据。
          if (Load_TX)
            begin
              TX_State = T2;
              TX_Busy = 1;//从数据总线中读取数据。
            end

        T2:          /发送开始标志。
          TX_State = T3;

        T3:         //传送数据。
          TX_State = T4;

        T4:   //发送跟踪标志以结束传送。
          begin
            TX_Busy = 0;
            TX_State = T1;
```

```
        end
      endcase
    end                    //顺序块TX结束。
endmodule
```

此交互有限状态机的时序行为关系如图 12-12所示。

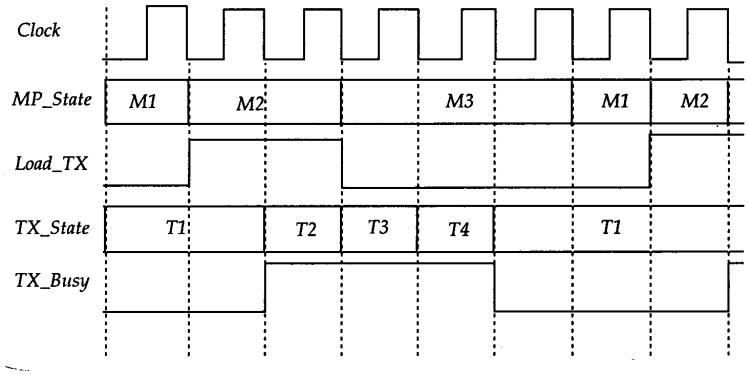

图12-12 两个交互进程的时序行为

考虑两个交互进程的另外一个实例，时钟分频器 *DIV*和接收器*RX*。在这种情况下，进程*DIV*产生一个新时钟，并且进程状态变换（转换）序列与新时钟同步。状态图如图 12-13所示。

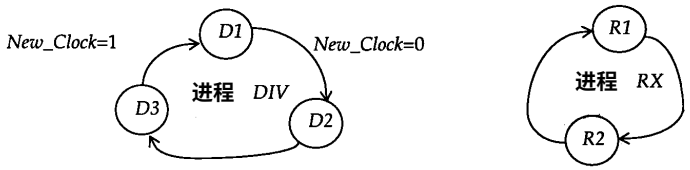

图12-13 *DIV*产生*RX*的时钟

```
module Another_Example_FSM2 (Clock);
  input Clock;

  parameter D1 = 1, D2 = 2, D3 = 3;
  parameter R1 = 1, R2 = 2;
  reg [0:1] Div_State, RX_State;
  reg New_Clock;

  always
    @ (posedge Clock) begin: DIV
      case (Div_State)
        D1:
          begin
            Div_State = D2;
            New_Clock = 0;
          end

        D2:
          Div_State = D3;
```

```
        D3:
          begin
            New_Clock = 1;
            Div_State = D1;
          end
        endcase
      end          //顺序块DIV结束。

    always
      @ (negedge New_Clock) begin: RX
        case (RX_State)
          R1: RX_State= R2;
          R2: RX_State= R1;
        endcase
    end    //顺序块结束。
  endmodule
```

顺序块*DIV*在其状态序列转换过程中产生新时钟。这一进程的状态转换发生在时钟的上升沿。顺序块在*New_Clock*的每个下降沿执行。图12-14显示了这些交互状态机的波形序列。

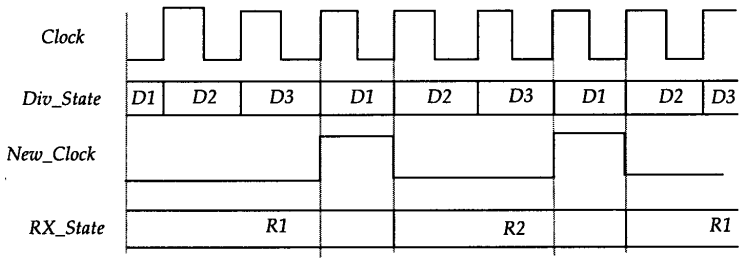

图12-14 进程*RX*和*DIV*之间的交互

12.9 Moore有限状态机建模

Moore有限状态机（FSM）的输出只依赖于状态而不依赖其输入。这种类型有限状态机的行为能够通过使用带有在状态值上转换的case语句的always语句建模。图12-15显示了Moore有限状态机的状态转换图实例，接着是Moore有限状态机对应的行为模型。

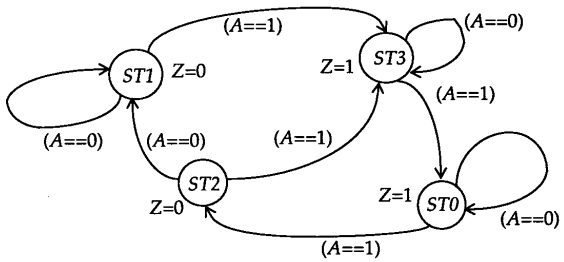

图12-15 Moore机的状态图

```
module Moore_FSM(A, Clock, Z);
  input A, Clock;
```

```
    output Z;
    reg Z;

    parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
    reg [0:1] Moore_State;

    always
      @ (negedge  Clock)
        case (Moore_State)
          ST0:
          begin
            Z = 1;
            if (A)
              Moore_State = ST2;
          end

        ST1:
          begin
            Z = 0;
            if (A)
              Moore_State = ST3;
           end

          ST2:
            begin
              Z = 0;
              if (~A)
                Moore_State = ST1;
              else
                Moore_State= ST3;
            end

          ST3:
            begin
              Z = 1;
              if (A)
                Moore_State = ST0;
            end
        endcase
endmodule
```

12.10 Mealy型有限状态机建模

在Mealy型有限状态机中，输出不仅依赖机器的状态而且依赖于它的输入。这种类型的有限状态机能够使用与 Moore FSM相似的形式建模。即使用 always语句。为了说明语言的多样性，使用不同的方式描述 Mealy机。这一次，我们用两条 always语句，一条对有限状态机的同步时序行为建模，一条对有限状态机的组合部分建模。图 12-16给出了状态转换表的一个实例，接着是相应的行为模型。

当前状态 \ 输入A	0	1
ST0	ST0 / 0	ST3 / 1
ST1	ST1 / 1	ST0 / 0
ST2	ST2 / 0	ST1 / 1
ST3	ST2 / 0	ST1 / 0

表中的项为次态和输出Z

图12-16 Mealy机状态转换表

```
module Mealy_FSM (A, Clock, Z);
  input A, Clock;
  output Z;
  reg Z;

  parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
  reg [1:2] P_State, N_State;

  always
    @ (negedge Clock)   //同步时序逻辑部分。
      P_State = N_State;
  always
    @ (P_State or A) begin: COMB_PART
      case (P_State)
        ST0:
          if (A)
            begin
              Z = 1;
              N_State = ST3;
             end
            else
              Z = 0;

        ST1:
          if (A)
            begin
              Z = 0;
              N_State = ST0;
             end
            else
              Z = 1;

        ST2:
          if (~A)
              Z = 0;
            else
              begin
                Z = 1;
                N_State = ST1;
```

```
                end

          ST3:
            begin
              Z = 0;
              if (~A)
                N_State = ST2;
              else
                N_State = ST1;
            end
        endcase
    end      //顺序块COMB_PART结束。
endmodule
```

在这种类型的有限状态机中，因为状态机的输出可以直接依赖独立于时钟的输入，将输入信号放入组合的部分时序程序块的事件列表中是非常重要的。因为 Moore有限状态机的输出只依赖于状态，并且状态转换在时钟上同步发生，在 Moore有限状态机中不会发生这种情况。

12.11 简化的21点程序

本节介绍简化的21点程序的状态机描述。玩21点程序需要一幅扑克牌。从2到10的牌取值与面值相同，牌A的值可以为1或者为11。游戏的目标是接收一定数量随机产生的牌，总分（所有牌值的总和）尽可能接近21而又不超过21。

当插入新牌时，*Card_Rdy*为真，并且*Card_Value*为牌的值。*Request_Card*表明程序何时就绪准备接收新牌。如果接收牌的序列总和超过 21，*Lost*置为真，以表明牌序列已经输了；否则*Won*置为真以表明游戏已获胜。状态排序由时钟控制。图 12-17显示了21点程序模块的输入和输出。

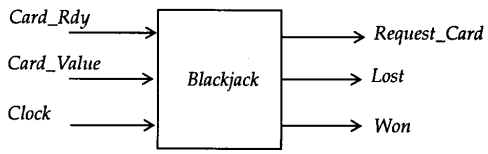

图12-17 21点程序模块的外部视图

程序的行为在如下的模块说明中描述。程序在总得分不大于 17前一直接收新牌。得分不超过21时，第一个A按11取值；得分超过21时，牌A按减10处理，即取值为1。三个寄存器用于存储程序的值：*Total*保存总和，*Current_Card_Value*保存读取的牌值（取值从 1到10），*Aec_As_11*用于记忆牌A是否取值为11而不是1。21点程序的状态存储在寄存器*BJ_State*中。

```
module Blackjack(Card_Rdy, Card_Value,
                 Request_Card, Won, Lost, Clock);
  input Card_Rdy, Clock;
  input [0:3] Card_Value;
  output Request_Card, Lost, Won;
  reg Request_Card, Lost, Won;

  Parameter INITIAL_ST = 0,GETCARD_ST= 1,
            REMCARD_ST= 2, ADD_ST = 3, CHECK_ST = 4,
```

```
          WIN_ST = 5 , BACKUP_ST = 6, LOSE_ST = 7;
reg [0:2] BJ_State;
reg [0:3] Current_Card_Value;
reg [0:4] Total;
reg Ace_As_11;

always
  @ (negedge Clock)
    case (BJ_State)
      INITIAL_ST:
       begin
         Total = 0;
         Ace_As_11 = 0;
         Won = 0;
         Lost = 0;
         BJ_State = GETCARD_ST
       end

      GETCATD_ST:
       begin
         Request_Card= 1;

         if (Card_Rdy)
           begin
             Current_Card_Value = Card_Value;
             BJ_State = REMCATD_ST;
           end          //否则停留在状态GETCARD_ST上不变。
        end

      REMCARD_ST :   //等待牌被移走。
       if (Card_Rdy)
         Request_Card=0;
       else
         BJ_State = ADD_ST;

      ADD_ST:
        begin
          if (~Ace_As_11 && Current_Card_Value)
            begin
              Current_Card_Value= 11;
              Ace_As_11 = 1;
            end

          Total = Total + Current_Card_Value;
          BJ_State = CHECK_ST;
       end

      CHECK_ST:
        if (Total < 17)
          BJ_State = GETCARD_ST;
        else
```

```
            begin
              if (Total <22)
                BJ_State = WIN_ST;
              else
                BJ_State = BACKUP_ST;
            end

          BACKUP_ST:
            if (Ace_As_11)
              begin
                Total = Total -10;
                Ace_As_11 = 0;
                BJ_State = CHECK_ST;
               end
             else
             BJ_State = LOSE_ST;

          LOSE_ST:
            begin
              Lost = 1;
              Request_Card= 1;
              if (Card_Rdy)
                BJ_State = INITIAL_ST;
                //否则停留在这个状态上不变。
             end

          WIN_ST:
            begin
              Won = 1;
              Request_Card= 1;

              if (Card_Rdy)
                BJ_State = INITIAL_ST;
                //否则停留在这个状态上不变。
              end
          endcase
    endmodule       //21点程序结束。
```

习题

1. 为芒果汁饮料机编写一个 Verilog HDL模型。机器分发价值 15美分一听的芒果汁。只接收 5分镍币和一角硬币。任何变化必须被返回。使用测试验证程序测试该模型。
2. 编写一个模型，描述带有同步预置和清空的触发器模型行为。
3. 编写带有串行数据输入、并行数据输入、时钟和并行数据输出的 4位移位寄存器模型。使用测试验证程序测试该模型。
4. 使用行为构造方式描述D型触发器。然后使用这一模块，编写一个 8位寄存器模型。
5. 编写测试12.11节描述的21点游戏模型的测试验证程序。
6. 使用移位操作符，描述解码器模块，然后使用测试验证程序测试该模块。解码器的输入数量指定为：

```
`define NUM_INPUTS 4
```

[提示：使用移动操作符计算解码器输出的数量。]

7. 编写并行到串行的8位转换器模型。输入为8位向量，在时钟上升沿从左位开始一次发送出一位。只有在前面的输入向量的所有位都发送出去以后，才读入下一个输入。
8. 编写与练习7行为相反的串行到并行的8位转换器。为体现传输时延，在时钟上升沿经过一小段时延后对输入流采样。使用高层模块将本练中编写的模型与练习7编写的模型连接，并使用测试验证程序测试其输出。
9. 编写具有保持控制的*N*位计数器模型。如果保持为1，计数器保持它的值，如果保持变为0，计数器被重置为0，并再次启动计数器。编写测试验证程序测试该模型。
10. 编写通用队列模型，队列中的字长为 *N*，字数为*M*。*InputData* 是被写入队列的字；当*Addword*为1时，向队列中添加字。从队列中读出的字存储在 *Output Data*中；当*Read Word*为1时，从队列中读取字。设置相应的标志 *Empty* 和*Full*。所有的事务发生在时钟*ClockA*的下降沿。编写测试验证程序测试该模型。
11. 编写可参数化的时钟分频器模型。输出时钟的周期是输入时钟周期的 2**N*倍，输出时钟与输入时钟的上升沿同步。编写测试验证程序测试该模型的正确性。