China-pub.com
下载

第8章　行 为 建 模

在前几章中，我们已经介绍了使用门和 UDP实例语句的门级建模方式，以及用连续赋值语句的数据流建模方式。本章描述 Verilog HDL中的第三种建模方式，即行为建模方式。为充分使用Verilog HDL，一个模型可以包含所有上述三种建模方式。

8.1　过程结构

下述两种语句是为一个设计的行为建模的主要机制。

1) initial 语句

2) always语句

一个模块中可以包含任意多个 initial或always语句。这些语句相互并行执行，即这些语句的执行顺序与其在模块中的顺序无关。一个 initial语句或always语句的执行产生一个单独的控制流，所有的initial和always语句在0时刻开始并行执行。

8.1.1　initial 语句

initial 语句只执行一次。initial 语句在模拟开始时执行，即在 0时刻开始执行。initial 语句的语法如下：

```
initial
[timing_control] procedural_statement
```

procedural_statement是下列语句之一：

```
procedural_assignment(blocking or non-blocking)//阻塞或非阻塞性过程赋值语句//
procedural_continuous_assignment
conditional_statement
case_statement
loop_statement
wait_statement
disable_statement
event_trigger
sequential_block
parallel_block
task_enable(user or system)
```

顺序过程(begin...end)最常使用在进程语句中。这里的时序控制可以是时延控制，即等待一个确定的时间；或事件控制，即等待确定的事件发生或某一特定的条件为真。 initial语句的各个进程语句仅执行一次。注意 initial语句在模拟的0时刻开始执行。initial语句根据进程语句中出现的时间控制在以后的某个时间完成执行。

下面是initial语句实例。

```
reg Yurt;
. . .
initial
  Yurt = 2;
```

China-pub.com 下载

上述initial语句中包含无时延控制的过程赋值语句。 initial语句在0时刻执行，促使*Yurt*在0时刻被赋值为2。下例是一个带有时延控制的initial语句。

```
reg Curt;
. . .
initial
  #2 Curt = 1;
```

寄存器变量*Curt*在时刻2被赋值为1。initial语句在0时刻开始执行，在时刻2完成执行。

下例为带有顺序过程的initial语句。

```
parameter SIZE = 1024;
reg [7:0] RAM [0:SIZE-1];
reg RibReg;

initial
  begin: SEQ_BLK_A
    integer Index;
  RibReg = 0;
  for (Index = 0; Index < SIZE; Index = Index + 1)
    RAM [Index] = 0;
end
```

顺序过程由关键词begin...end定界，它包含顺序执行的进程语句，与 C语言等高级编程语言相似。*SEQ_BLK_A*是顺序过程的标记；如果过程中没有局部说明部分，不要求这一标记。例如，如果对*Index*的说明部分在initial语句之外，可不需要标记。整数型变量 *Index*已在过程中声明。并且，顺序过程包含 1个带循环语句的过程性赋值。这一 initial语句在执行时将所有的内存初始化为0。

下例是另一个带有顺序过程的 initial语句。在此例中，顺序过程包含时延控制的过程性赋值语句。

```
//波形生成:
parameter APPLY_DELAY = 5;
reg[0:7]port_A;
. . .
initial
  begin
    Port_A ='h20;
  #APPLY_DELAY Port_A= 'hF2;
  #APPLY_DELAY Port_A= 'h41;
  #APPLY_DELAY Port_A= 'h0A;
end
```

执行时，*Port_A*的值如图8-1所示。

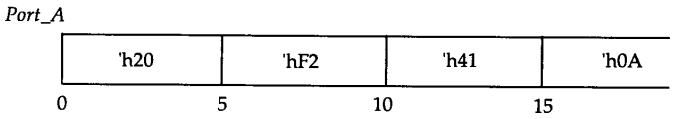

图8-1 使用initial语句产生的波形

如上面举例所示，Initial语句主要用于初始化和波形生成。

China-pub.com
下载

8.1.2 always语句

与initial语句相反，always语句重复执行。与initial语句类似，always语句语法如下：

```
always
  [timing_control] procedural_statement
```

过程语句和时延控制（时序控制）的描述方式与上节相同。

例如：

```
always
  Clk = ~ Clk;
//将无限循环。
```

此always语句有一个过程性赋值。因为always语句重复执行，并且在此例中没有时延控制，过程语句将在0时刻无限循环。因此，always语句的执行必须带有某种时序控制，如下例的always语句，形式上与上面的实例相同，但带有时延控制。

```
always
  #5 Clk = ~ Clk;
//产生时钟周期为10的波形。
```

此always语句执行时产生周期为10个时间单位的波形。

下例是由事件控制的顺序过程的always语句。

```
reg [0:5] InstrReg;
reg [3:0] Accum;
wire ExecuteCycle;

always
  @ (EcecuteCycle)
    begin
     case(InstrReg[0:1])
       2'b00: Store (Accum, InstrReg[2:5]);
       2'b11: Load (Accum, InstrReg[2:5]);
       2'b01: Jump (InstrReg[2:5]);
       2'b10:;
     endcase
  end
//Store、Load和Jump是 在别处定义的用户自定义的任务。
```

顺序过程**(begin...end)**中的语句按顺序执行。这个always语句意味着只要有事件发生，即只要发生变化，*ExecuteCycle*就执行顺序过程中的语句；顺序过程的执行意味着按顺序执行过程中的各个语句。

下例为带异步预置的负边沿触发的D触发器的行为模型。

```
module DFF(Clk, D, Set, Q, Qbar);
  input Clk, D, Set;
  output Q, Qbar;
  reg Q, Qbar;

  always
   wait (Set == 1)
    begin
      #3 Q = 1;
      #2 Qbar = 0;
```

China-pub.com 下载

```
        wait (Set == 0);
      end
  always
    @ (negedge Clk)
      begin
       if (Set != 1)
        begin
          #5 Q = D;
          #1 Qbar = ~ Q;
        end
      end
  endmodule
```

此模型中有2条always语句。第一条always语句中顺序过程的执行由电平敏感事件控制。第二条always语句中顺序过程的执行由边沿触发的事件控制。

8.1.3 两类语句在模块中的使用

一个模块可以包含多条always语句和多条initial语句。每条语句启动一个单独的控制流。各语句在0时刻开始并行执行。

下例中含有1条initial语句和2条always语句。

```
module TestXorBehavior;
  reg Sa, Sb, Zeus;

  initial
  begin
    Sa = 0;
    Sb = 0;
    #5 Sb = 1;
    #5 Sa = 1;
    #5 Sb = 0;
  end

always
  @ (Sa or Sb) Zeus = Sa ^ Sb;

always
  @ (Zeus)
     $display ("At time %t, Sa = %d, Sb = %d, Zeus = %b",
               $time, Sa, Sb, Zeus);
endmodule
```

模块中的3条语句并行执行，其在模块中的书写次序并不重要。 initial语句执行时促使顺序过程中的第一条语句执行，即*Sa*赋值为0；下一条语句在0时延后立即执行。initial语句中的第3行表示“ 等待5个时间单位”。这样*Sb*在5个时间单位后被赋值为 1，*Sa*在另外5个时间单位后被赋值为0。执行顺序过程最后一条语句后， initial语句被永远挂起。

第一条always语句等待*Sa*或*Sb*上的事件发生。只要有事件发生，就执行 always语句内的语句，然后always语句重新等待发生在*Sa*或*Sb*上的事件。注意根据 initial语句对*Sa*和*Sb*的赋值，always语句将在第0、5、10和15个时间单位时执行。

China-pub.com
下载

同样，只要有事件发生在*Zeus*上，就执行第2条always语句。在这种情况下，系统任务 **$display**被执行，然后always语句重新等待发生在*Zeus*上的事件。*Sa*、*Sb*和*Zeus*上产生的波形如图8-2所示。下面是模块模拟运行产生的输出。

```
在时刻          5,  Sa = 0, Sb = 1, Zeus = 1
在时刻          10, Sa = 1, Sb = 1, Zeus = 0
在时刻          15, Sa = 1, Sb = 0, Zeus = 1
```

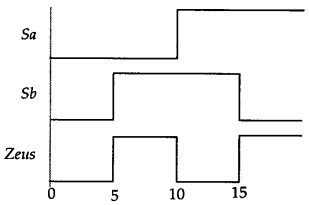

图8-2 *Sa*、*Sb*和*Zeus*上产生的波形

8.2 时序控制

时序控制与过程语句关联。有2种时序控制形式：

1) 时延控制

2) 事件控制

8.2.1 时延控制

时延控制形式如下：

```
#delay procedural_statement
```

实例如下；

```
#2 Tx = Rx-5;
```

时延控制定义为执行过程中首次遇到该语句与该语句的执行的时间间隔。时延控制表示在语句执行前的“ 等待时延”。上面的例子中，过程赋值语句在碰到该语句后的 2个时间单位执行，然后执行赋值。

另一实例如下：

```
initial
  begin
    #3 Wave = 'b0111;
    #6 Wave = 'b1100;
    #7 Wave = 'b0000;
end
```

initial语句在0时刻执行。首先，等待 3个时间单位执行第一个赋值，然后等待 6个时间单位，执行第2个语句；再等待7个时间单位，执行第3个语句；然后永远挂起。

时延控制也可以用另一种形式定义：

```
#delay;
```

这一语句促使在下一条语句执行前等待给定的时延。下面是这种用法的实例。

```
parameter ON_DELAY = 3, OFF_DELAY = 5;
always
  begin
    # ON_DELAY;    //等待ON_DELAY规定的时延。
    RefClk = 0;
    # OFF_DELAY;    //等待OFF_DELAY规定的时延。
    RefClk = 1;
end
```

时延控制中的时延可以是任意表达式，即不必限定为某一常量，见下面的例子。

```
# Strobe
  Compare = TX^ask;
```

China-pub.com 下载

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

如果时延表达式的值为0，则称之为显式零时延。

```
#0;          /显式零时延。
```

显式零时延促发一个等待，等待所有其它在当前模拟时间被执行的事件执行完毕后，才将其唤醒；模拟时间不前进。

如果时延表达式的值为**x**或**z**,其与零时延等效。如果时延表达式计算结果为负值，那么其二进制的补码值被作为时延，这一点在使用时务请注意。

8.2.2 事件控制

在事件控制中，always的过程语句基于事件执行。有两种类型的事件控制方式：

1) 边沿触发事件控制

2) 电平敏感事件控制

1. 边沿触发事件控制

边沿触发事件控制如下：

```
@ event procedural_statement
```

如下例所示：

```
@ (posedge Clock)
Curr_State = Next_State;
```

带有事件控制的进程或过程语句的执行，须等到指定事件发生。上例中，如果*Clock*信号从低电平变为高电平（正沿），就执行赋值语句；否则进程被挂起直到*Clock*信号产生下一个正跳边沿。

下面是进一步的实例。

```
@ (negedge Reset)  Count = 0;
@Cla
Zoo = Foo;
```

在第一条语句中，赋值语句只在 *Reset*上的负沿执行。第二条语句中 ,当*Cla*上有事件发生时，*Foo*的值被赋给*Zoo*,即等待*Cla*上发生事件；当*Cla*的值发生变化时,*Foo*的值被赋给*Zoo*。

也可使用如下形式：

```
@ event ;
```

该语句促发一个等待，直到指定的事件发生。下面是确定时钟在周期的 initial语句中使用的一个例子。

```
time RiseEdge, OnDelay;
initial
  begin
    //等待，直到在时钟上发生正边沿：
    @ (posedge ClockA);
    RiseEdge = $time;
    //等待，直到在时钟上发生负边沿：
    @ (negedge ClockA);
    OnDelay = $time - RiseEdge;
    $display ("The on-period of clock is %t.",Delay);
end
```

事件之间也能够相或以表明“ 如果有任何事件发生”。下例将对此进行说明。

```
@ (posedge Clear or negedge Reset)
```

China-pub.com
下载


```
Q = 0;
@ (Ctrl_A or Ctrl_B)
Dbus = 'bz;
```

注意关键字**or**并不意味着在1个表达式中的逻辑或。

在Verilog HDL中**posedge**和**negedge**是表示正沿和负沿的关键字。信号的负沿是下述转换的一种：

```
1 -> x
1 -> z
1 -> 0
x -> 0
z -> 0
```

正沿是下述转换的一种：

```
0 -> x
0 -> z
0 -> 1
x -> 1
z -> 1
```

2. 电平敏感事件控制

在电平敏感事件控制中，进程语句或进程中的过程语句一直延迟到条件变为真后才执行。电平敏感事件控制以如下形式给出：

```
wait (Condition)
  procedural_statement
```

过程语句只有在条件为真时才执行，否则过程语句一直等待到条件为真。如果执行到该语句时条件已经为真，那么过程语句立即执行。在上面的表示形式中，过程语句是可选的。

例如：

```
wait (Sum > 22)
Sum = 0;

wait (DataReady)
Data = Bus;

wait (Preset);
```

在第一条语句中，只有当 *Sum*的值大于22时，才对Sum清0。在第二条语句中，只有当*DataReady*为真，即*DataReady*值为1时，将*Bus*赋给*Data*。最后一条语句表示延迟至 *Preset*变为真（值为1）时，其后续语句方可继续执行。

8.3 语句块

语句块提供将两条或更多条语句组合成语法结构上相当于一条语句的机制。在 Verilog HDL中有两类语句块，即：

1) 顺序语句块(**begin...end**)：语句块中的语句按给定次序顺序执行。

2) 并行语句块(**fork...join**)：语句块中的语句并行执行。

语句块的标识符是可选的，如果有标识符，寄存器变量可在语句块内部声明。带标识符的语句块可被引用；例如，语句块可使用禁止语句来禁止执行。此外，语句块标识符提供唯一标识寄存器的一种方式。但是，要注意所有的寄存器均是静态的，即它们的值在整个模拟

China-pub.com 下载

运行中不变。

8.3.1 顺序语句块

顺序语句块中的语句按顺序方式执行。每条语句中的时延值与其前面的语句执行的模拟时间相关。一旦顺序语句块执行结束，跟随顺序语句块过程的下一条语句继续执行。顺序语句块的语法如下：

```
begin
  [:block_id{declarations}]
  procedural_statement(s)
end
```

例如：

```
//产生波形:
begin
  #2 Stream = 1;
  #5 Stream = 0;
  #3 Stream = 1;
  #4 Stream = 0;
  #2 Stream = 1;
  #5 Stream = 0;
end
```

假定顺序语句块在第10个时间单位开始执行。两个时间单位后第 1条语句执行，即第12个时间单位。此执行完成后，下 1条语句在第17个时间单位执行(延迟5个时间单位)。然后下1条语句在第20个时间单位执行，以此类推。该顺序语句块执行过程中产生的波形如图 8-3所示。

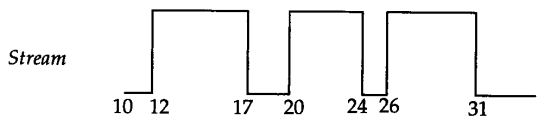

图8-3 顺序语句块中的累积时延

下面是顺序过程的另一实例。

```
begin
  Pat = Mask | Mat;
  @(negedge Clk);
   FF = & Pat
end
```

在该例中，第1条语句首先执行，然后执行第 2条语句。当然，第2条语句中的赋值只有在*Clk*上出现负沿时才执行。下面是顺序过程的另一实例。

```
begin: SEQ_BLK
  reg[0:3] Sat;

  Sat = Mask & Data;
  FF = ^Sat;
end
```

在这一实例中，顺序语句块带有标记 *SEQ_BLK*，并且有一个局部寄存器说明。在执行时，首先执行第1条语句，然后执行第 2条语句。

China-pub.com
下载

8.3.2 并行语句块

并行语句块带有定界符 **fork**和**join**(顺序语句块带有定界符 **begin**和**end**)，并行语句块中的各语句并行执行。并行语句块内的各条语句指定的时延值都与语句块开始执行的时间相关。当并行语句块中最后的动作执行完成时 (最后的动作并不一定是最后的语句)，顺序语句块的语句继续执行。换一种说法就是并行语句块内的所有语句必须在控制转出语句块前完成执行。并行语句块语法如下：

```
fork
  [:block_id{declarations}]
  procedural_statement(s);
join
```

例如：

```
// 生成波形:
fork
  #2 Stream = 1;
  #7 Stream = 0;
  #10 Stream = 1;
  #14 Stream = 0;
  #16 Stream = 1;
  #21 Stream = 0;
join
```

如果并行语句块在第 10个时间单位开始执行，所有的语句并行执行并且所有的时延都是相对于时刻10的。例如，第3个赋值在第20个时间单位执行，并在第 26个时间单位执行第 5个赋值，以此类推。其产生的波形如图 8-4所示。

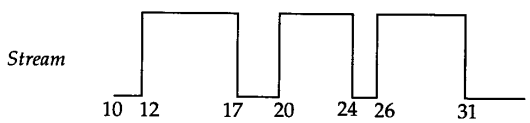

图8-4 并行语句块中的相对时延

下例混合使用了顺序语句块和并行语句块，以强调两者的不同之处。

```
always
  begin:SEQ_A
    #4 Dry = 5;              // S1

    fork: PAR_A               //S2
      #6  Cun = 7;           //P1

      begin: SEQ_B             //P2
       EXE = Box;             //S6
       #5 Jap = Exe;          //S7
      end

      #2 Dop = 3;             //P3
      #4 Gos = 2;             //P4
      #8 Pas = 4;             //P5
    join
```

China-pub.com 下载

```
    #8 Bax = 1;                    //S3
    #2 Zoom = 52;                  //S4
    #6 $stop;                      //S5
end
```

always语句中包含顺序语句块*SEQ_A*，并且顺序语句块内的所有语句(S1、S2、S3、S4和S5)顺序执行。因为always语句在0时刻执行，*Dry*在第4个时间单位被赋值为5，并且并行语句块*PAR_A*在第4个时间单位开始执行。并行语句块中的所有语句(P1、P2、P3、P4和P5)在第4个时间单位并行执行。这样*Cun*在第10个时间单位被赋值，*Dop*在第6个时间单位被赋值，*Gos*在第8个时间单位被赋值，*Pas*在第12个时间单位被赋值。顺序语句块*SEQ_B*在第4个时间单位开始执行，并导致该顺序块中的语句S6、S7依次被执行；*Jap*在时间单位9被赋于新值。因为并行语句块*PAR_A*中的所有语句在第12个时间单位完成执行，语句S3在第12个时间单位被执行，在第20个时间单位*Bax*被赋值，然后语句S4执行，在第22个时间单位*Zoom*被赋值，然后执行下一语句。最终在第28个时间单位执行系统任务**$stop**。always语句执行时发生的事件如图8-5所示。

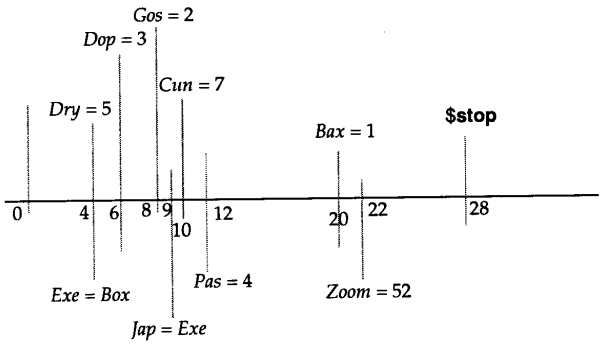

图8-5 顺序语句块和并行语句块混合使用时的时延

8.4 过程性赋值

过程性赋值是在initial语句或always语句内的赋值，它只能对寄存器数据类型的变量赋值。表达式的右端可以是任何表达式。例如：

```
reg[1:4] Enable,A,B;
...
#5 Enable = ~A ^ ~B;
```

*Enable*为寄存器。根据时延控制，赋值语句被延迟5个时间单位执行。右端表达式被计算，并赋值给*Enable*。

过程性赋值与其周围的语句顺序执行。always语句实例如下：

```
always
  @ (A or B or C or D)
    begin:AOI
      reg Temp1,Temp2;

      Temp1 = A & B;
      Temp2 = C & D;
      Temp1 = Temp1 | Temp2;
```

China-pub.com
下载


```
        Z = ~Temp1;
    end
  /*可用一个语句代替上面的4条语句，例如：
      Z = ~((A&B)|(C&D));
    但是，上例的目的主要用于解释说明顺序过程语句的顺序特性 */
```

always语句内的顺序过程在信号 *A*、*B*、*C*或*D*发生变化时开始执行。 *Temp1*的赋值首先执行。然后执行第二个赋值。在以前赋值中计算的 *Temp1*和*Temp2*的值在第三条赋值语句中使用。最后一个赋值使用在第三条语句中计算的 *Temp1*的值。

过程性赋值分两类：

1) 阻塞性过程赋值

2) 非阻塞性过程赋值

在讨论这两类过程性赋值前，先简要地说明语句内部时延的概念。

8.4.1 语句内部时延

在赋值语句中表达式右端出现的时延是语句内部时延。通过语句内部时延表达式，右端的值在赋给左端目标前被延迟。例如：

```
Done = #5 'b1;
```

重要的是右端表达式在语句内部时延之前计算，随后进入时延等待，再对左端目标赋值。下例说明了语句间和语句内部时延的不同。

```
Done = #5 'b1;       /语句内部时延控制
```

与

```
begin
Temp = 'b1;
#5 Done = Temp;          //语句间时延控制
end
```

相同。而语句

```
Q = @(posedge Clk ) D;    //语句内事件控制
```

与

```
begin
Temp = D;
@(posedge Clk)           //语句间事件控制
Q = Temp;
end
```

相同。

除以上两种时序控制(时延控制和事件控制)可用于定义语句内部时延外，还有另一种重复事件控制的语句内部时延表示形式。形式如下：

```
repeat(express) @ (event_expression)
```

这种控制形式用于根据一定数量的1个或多个事件来定义时延。例如，

```
Done = repeat(2) @ (negedge ClkA) A_REG + B_REG
```

这一语句执行时先计算右端的值，即 *A_Reg* + *B_Reg*的值，然后等待时钟 *ClkA*上的两个负沿，再将右端值赋给 *Done*。这一重复事件控制实例的等价形式如下：

China-pub.com 下载

```
begin
  Temp = A_REG + B_REG
  @ (negedge ClkA);
  @ (negedge ClkA);
  Done = Temp;
end
```

这种形式的时延控制方式在给某些边或一定数量的边的同步赋值过程（语句）中非常有用。

8.4.2 阻塞性过程赋值

赋值操作符是“ =”的过程赋值是阻塞性过程赋值。例如，

```
RegA = 52;
```

是阻塞性过程赋值。阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。如下所示：

```
always
  @(A or B or Cin)
   begin: CARRY_OUT
    reg T1,T2,T3;

    T1 = A & B;
    T2 = B & Cin;
    T3 = A & Cin;
    Cout = T1 | T2 | T3;
end
```

*T1*赋值首先发生，计算 *T1*；接着执行第二条语句， *T2*被赋值；然后执行第三条语句， *T3*被赋值；依此类推。

下例是使用语句内部时延控制的阻塞性过程赋值语句。

```
initial
begin
  Clr = #5 0;
  Clr = #4 1;
  Clr = #10 0;
end
```

第一条语句在0时刻执行， *Clr*在5个时间单位后被赋值；接着执行第二条语句，使 *Clr*在4个时间单位后被赋值为 1(从0时刻开始为第9个时间单位)；然后执行第三条语句促使 *Clr*在10个时间单位后被赋值为 0(从0时刻开始为第19个时间单位)。图8-6显示*Clr*上产生的波形。

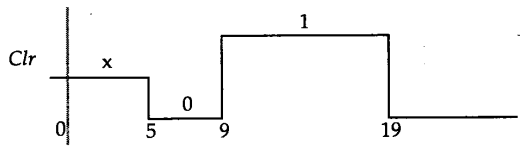

图8-6 带有语句内部时延控制的阻塞性过程赋值

另一实例如下：

```
begin
  Art = 0;
```

China-pub.com
下载


```
    Art = 1;
end
```

在这种情况下，*Art*被赋值为1。这是因为第一个*Art*被赋值为0，然后执行下一条语句促使*Art*在0时延后被赋值为1。因此对*Art*的0赋值被丢弃。

8.4.3 非阻塞性过程赋值

在非阻塞性过程赋值中，使用赋值符号“ <=”。例如：

```
begin
  Load <= 32;
  RegA <= Load;
  RegB <= Store;
end
```

在非阻塞性过程赋值中，对目标的赋值是非阻塞的 (因为时延)，但可预定在将来某个时间步发生(根据时延；如果是 0时延，那么在当前时间步结束)。当非阻塞性过程赋值被执行时，计算右端表达式，右端值被赋于左端目标，并继续执行下一条语句。预定的最早输出将在当前时间步结束时，这种情况发生在赋值语句中没有时延时。在当前时间步结束或任意输出被调度时，即对左端目标赋值。

在上面的例子中，我们假设顺序语句块在时刻 10执行。第一条语句促使*Load*在第10个时间单位结束时被赋值为 32；然后执行第2条语句，*Load*的值不变(注意时间还没有前进，并且第1个赋值还没有被赋新值)，*RegA*的赋值被预定为在第 10个时间步结束时。在所有的事件在第10个时间单位发生后，完成对左端目标的所有预定赋值。

下面的例子更进一步解释这种赋值特征。

```
initial
  begin
    Clr <= #5 1;
    Clr <= #4 0;
    Clr <= #10 0;
end
```

第一条语句的执行使*Clr* 在第5个时间单位被赋于值1；第二条语句的执行使*Clr*在第4个时间单位被赋值为 0(从0时刻开始的第 4个时间单位)；最终，第3条语句的执行使*Clr*在第10个时间单位被赋值为0(从0时刻开始的第10个时间单位)。注意3条语句都是在0时刻执行的。此外，在这种情况下，非阻塞性赋值执行次序变得彼此不相关。*Clr*上产生的波形如图8-7所示。

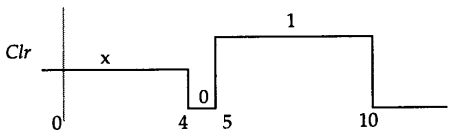

图8-7 带有语句内部时延的非阻塞性过程赋值

下面是带有0时延的例子。

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

在initial语句执行后，因为同时对同一寄存器变量有多个赋值， *Cbn*的值变得不确定，即*Cbn* = **x**。Verilog HDL标准中既没有规定在这种情况下，何种事件被调度，也没有规定事件

China-pub.com 下载

被取消的次序。结果是根据特定的 Verilog模拟器的事件调度算法， *Cbn*将被赋值为0或1[⊖]。

下面是同时使用阻塞性和非阻塞性过程赋值的实例，注意它们的区别。

```
reg [0:2]  Q_State;

initial
  begin
   Q_State = 3'b011;
   Q_State <= 3'b100;
  $display ("Current value of Q_State is %b", Q_State);
  #5;  //等待一定的时延。
  $display ("The delayed value of Q_State is %b", Q_State);
end
```

执行initial语句产生如下结果：

```
Current value of Q_State is 011
The delayed value of Q_State is 100
```

第一个阻塞性赋值使*Q_State*被赋值为3'b011。执行第二条赋值语句(为非阻塞性赋值语句)促使*Q_State*在当前时间步(第0步)结束时被赋值为3'b100。因此当第一个**$display**任务被执行时，*Q_State*还保持来自第一个赋值的值，即 3'b011。当#5时延被执行后，促使被调度的*Q_State*赋值发生，*Q_State*的值被更新。延迟5个时间单位后，执行下一个**$display**任务，此时显示*Q_State*的更新值。

8.4.4 连续赋值与过程赋值的比较

连续赋值与过程赋值有什么不同之处？表8-1列举了它们的差异。

表8-1 过程赋值与连续赋值间的差异

过 程 赋 值	连 续 赋 值
在always语句或initial语句内出现	在一个模块内出现
执行与周围其它语句有关	与其它语句并行执行；在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
使用“=”或“=”赋值符号	使用“=”赋值符号
无assign关键词(在过程性连续赋值中除外，参见第8章第8节)	有assign关键词

下例进一步解释了这些差别。

```
module Procedural;
reg A,B,Z;

always
  @(B) begin
   Z = A;
   A = B;
  end
endmodule
```

⊖ 在上述情况下，使用不同模拟器对其进行模拟所产生的模拟结果可能会有所不同。——译者注

China-pub.com
下载


```
module Continuous
  wire A,B,Z;

  assign Z = A;
  assign A = B;
endmodule
```

假定B在10 ns时有一个事件。在过程性赋值模块中，两条过程语句被依序执行， A在10 ns时得到B的新值。Z没有得到B的值，因为赋值给Z发生在赋值给A之前。在连续性赋值语句模块中，第二个连续赋值被触发，因为这里有一个关于 B的事件。这引起了关于A的事件，A引发第一个连续赋值被执行，这相应引起 Z得到了A的值。Z的新值为A而不是B。然而，如果事件发生在A上，过程性模块中的always语句不执行，因为A不在那个always语句的实时控制事件清单中。然而连续赋值语句中的第一个连续赋值执行，并且 Z得到A的新值。

8.5 if 语句

if语句的语法如下：

```
if(condition_1)
  procedural_statement_1
{else if(condition_2)
  procedural_statement_2}
{else
  procedural_statement_3}
```

如果对*condition_1* 求值的结果为一个非零值，那么*procedural_statement_1*被执行，如果*condition_1* 的值为0、**x**或**z**，那么*procedural_statement_1*不执行。

如果存在一个else分支，那么这个分支被执行。以下是一个例子。

```
if(Sum < 60)
  begin
    Grade = C;
    Total_C = Total _c + 1;
  end
else if(Sum < 75)
  begin
    Grade = B;
    Total_B = Total_B + 1;
  end
else
  begin
    Grade = A;
    Total_A = Total_A + 1;
  end
```

注意条件表达式必须总是被括起来，如果使用 if-if-else格式，那么可能会有二义性，如下例所示：

```
if(Clk)
  if(Reset)
    Q = 0;
  else
    Q = D;
```

China-pub.com 下载

问题是最后一个else属于哪一个if？它是属于第一个if的条件(*Clk*)还是属于第二个if的条件(*Reset*)？这在Verilog HDL中已通过将else与最近的没有else的if相关联来解决。在这个例子中，else与内层if语句相关联。

以下是另一些if语句的例子。

```
if(Sum < 100)
  Sum = Sum + 10;
if(Nickel_In)
  Deposit = 5;
else if(Dime_In)
  Deposit = 10;
else if(Quarter_In)
  Deposit = 25;
else
  Deposit = ERROR;

if(Ctrl)
  begin
    if(~Ctrl2)
      Mux = 4'd2;
    else
      Mux = 4'd1;
  end
else
  begin
    if(~Ctrl2)
      Mux = 4'd8;
    else
      Mux = 4'd4;
  end
```

8.6 case语句

case语句是一个多路条件分支形式，其语法如下：

```
case(case_expr)
  case_item_expr{,case_item_expr}:procedural_statement
. . .
. . .
  [default:procedural_statement]
endcase
```

case语句首先对条件表达式 *case_expr*求值，然后依次对各分支项求值并进行比较，第一个与条件表达式值相匹配的分支中的语句被执行。可以在 1个分支中定义多个分支项；这些值不需要互斥。缺省分支覆盖所有没有被分支表达式覆盖的其他分支。

分支表达式和各分支项表达式不必都是常量表达式。在 case语句中，**x**和**z**值作为文字值进行比较。case语句如下所示：

```
parameter
    MON = 0 , TUE = 1, WED = 2,
    THU = 3, FRI = 4,
    SAT = 5, SUN = 6;
reg [0:2] Day;
```

China-pub.com
下载


```
integer Pocket_Money;

case (Day)
  TUE : Pocket_Money = 6;      /分支1。
  MON ,
  WED : Pocket_Money  = 2;    /分支2。
  FRI,
  SAT,
  SUN : Pocket_Money = 7;     /分支3。
  default : Pocket_Money = 0;        /分支4。
endcase
```

如果*Day*的值为*MON*或*WED*，就选择分支2。分支3覆盖了值*FRI*、*SAT*和*SUN*，而分支4覆盖了余下的所有值，即*THU*和位向量111。case语句的另一实例如下：

```
module ALU (A, B, OpCode, Z);
  input [3:0] A, B;
  input [1:2] OpCode;
  output [7:0] Z;
  reg [7:0] Z;
parameter
    ADD_INSTR = 2'b10,
    SUB_INSTR = 2'b11,
    MULT_INSTR = 2'b01,
    DIV_INSTR = 2'b00;
always
  @ (A or B or OpCode)
    case (OpCode)
      ADD_INSTR:    Z = A + B;
      SUB_INSTR:    Z = A -B;
      MULT_INSTR:   Z = A * B;
      DIV_INSTR:    Z = A / B;
    endcase
endmodule
```

如果case表达式和分支项表达式的长度不同会发生什么呢？在这种情况下，在进行任何比较前所有的case表达式都统一为这些表达式的最长长度。下例说明了这种情况。

```
case (3'b101 << 2)
  3'b100  : $display ( "First branch taken!");
  4'b0100 : $display ( "Second branch taken!");
  5'b10100: $display ( "Third branch taken!");
  default : $display ( "Default branch taken!");
endcase
```

产生：

```
Third branch taken!
```

因为第3个分支项表达式长度为5位，所有的分支项表达式和条件表达式长度统一为5。当计算3'b101<<2时，结果为5'b10100，并选择第3个分支。

case语句中的无关位

上节描述的case语句中，值**x**和**z**只从字面上解释，即作为**x**和**z**值。这里有case语句的其

China-pub.com 下载

它两种形式：casex和casez，这些形式对**x**和**z**值使用不同的解释。除关键字**casex**和**casez**以外，语法与case语句完全一致。

在casez语句中，出现在case表达式和任意分支项表达式中的值**z**被认为是无关值，即那个位被忽略(不比较)。

在casex语句中，值**x**和**z**都被认为是无关位。casez语句实例如下：

```
case(Mask)
  4'b1??? : Dbus[4] = 0;
  4'b01?? : Dbus[3] = 0;
  4'b001? : Dbus[2] = 0;
  4'b0001 : Dbus[1] = 0;
endcase
```

? 字符可用来代替字符**z**,表示无关位。casez语句表示如果Mask的第1位是1(忽略其它位)，那么将*Dbus*[4]赋值为0；如果*Mask*的第1位是0，并且第2位是1(忽略其它位)，那么*Dbus[3]*被赋值为0，并依此类推。

8.7 循环语句

Verilog HDL中有四类循环语句，它们是：

1) forever循环
2) repeat循环
3) while循环
4) for 循环

8.7.1 forever 循环语句

这一形式的循环语句语法如下：

```
forever
  procedural_statement
```

此循环语句连续执行过程语句。因此为跳出这样的循环，中止语句可以与过程语句共同使用。同时，在过程语句中必须使用某种形式的时序控制，否则， forever循环将在0时延后永远循环下去。

这种形式的循环实例如下：

```
initial
  begin
    Clock = 0;
    # 5 forever
      #10 Clock = ~Clock;
  end
```

这一实例产生时钟波形；时钟首先初始化为 0，并一直保持到第 5个时间单位。此后每隔10个时间单位，*Clock*反相一次。

8.7.2 repeat 循环语句

repeat 循环语句形式如下：

```
repeat(loop_count)
  procedural_statement
```

China-pub.com
下载

这种循环语句执行指定循环次数的过程语句。如果循环计数表达式的值不确定，即为 **x**或**z**时，那么循环次数按0处理。下面是一些具体实例。

```
repeat (Count)
  Sum = Sum + 10;

repeat (ShiftBy)
  P_Reg = P_Reg << 1;
```

repeat循环语句与重复事件控制不同。例如，

```
repeat(Count)          //repea循环语句
  @ (posedge Clk) Sum = Sum + 1;
```

上例表示计数的次数，等待*Clk*的正边沿，并在*Clk*正沿发生时，对*Sum*加1。但是，

```
Sum = repeat(Count) @ (posedge Clk) Sum + 1;
  // 重复事件控制
```

该例表示首先计算Sum + 1，随后等待*Clk*上正沿计数，最后为左端赋值。

下面的表达式意味着什么?

```
repeat(NUM_OF_TIMES) @ (negedge ClockZ);
```

它表示在执行跟随在repeat语句之后的语句之前，等待*ClockZ* 的*NUM_OF_TIMES*个负沿。

8.7.3 while 循环语句

while 循环语句语法如下：

```
while(condition)
  procedural_statement
```

此循环语句循环执行过程赋值语句直到指定的条件为假。如果表达式在开始时为假，那么过程语句便永远不会执行。如果条件表达式为 **x**或**z**，它也同样按0（假）处理。例如：

```
while (BY > 0 )
  begin
    Acc = Acc << 1;
    By = By - 1;
  end
```

8.7.4 for 循环语句

for 循环语句的形式如下：

```
for(initial_assignment;condition;step_assignment)
  procedural_statement
```

一个for循环语句按照指定的次数重复执行过程赋值语句若干次。初始赋值*initial_assignment*给出循环变量的初始值。*condition*条件表达式指定循环在什么情况下必须结束。只要条件为真，循环中的语句就执行；而 *step_assignment*给出要修改的赋值，通常为增加或减少循环变量计数。

```
integer K;

for (K=0 ; K < MAX_RANGE ; K = K + 1)
  begin
    if(Abus[K] == 0)
     Abus[K] = 1;
```

China-pub.com 下载

```
    else if(Abus[k] == 1)
     Abus[K] = 0;
    else
     $display("Abus[K] is an x or a z");
  end
```

8.8 过程性连续赋值

过程性连续赋值是过程性赋值的一类，即它不能够在 always语句或initial语句中出现。这种赋值语句能够替换其它所有对线网或寄存器的赋值。它允许赋值中的表达式被连续驱动到寄存器或线网当中。注意，这不是一个连续赋值，连续赋值发生在 initial或always语句之外。

过程性连续赋值语句有两种类型：

1) 赋值和重新赋值过程语句：它们对寄存器进行赋值。

2) 强制和释放过程性赋值语句：虽然它们也可以用于对寄存器赋值，但主要用于对线网赋值。

赋值和强制语句在如下意义上是“连续”的：即当赋值或强制发生效用时，右端表达式中操作数的任何变化都会引起赋值语句重新执行。

过程性连续赋值的目标不能是寄存器部分选择或位选择。

8.8.1 赋值—重新赋值

一个赋值过程语句包含所有对寄存器的过程性赋值，重新赋值过程语句中止对寄存器的连续赋值。寄存器中的值被保留到其被重新赋值为止。

```
module DEF(D,Clr,Clk,Q);
  input D,Clr,Clk;
  output Q;
  reg Q;

  always
    @(Clr) begin
     if(!Clr)
      assign Q = 0;   // D对Q无效。
     else
      deassign Q;
  end

  always
    @(negedge Clk) Q = D;
endmodule
```

如果*Clr*为0，assign赋值语句使*Q*清0，而不管时钟边沿的变化情形，即 *Clk*和*D*对*Q*无效。如果*Clr*变为1，重新赋值语句被执行；这就使得强制赋值方式被取消，以后 *Clk*能够对*Q*产生影响。

如果赋值应用于一个已经被赋值的寄存器， assign赋值在进行新的过程性连续赋值前取消了原来的赋值。实例如下：

```
reg[3:0] Pest;
...
```

China-pub.com
下载


```
Pest = 0;
...
assign Pest = Hty ^ Mtu;
...
assign Pest = 2;    /将对Pest重新赋值，然后赋值。
...
deassign Pest;      //Pes连续地保持值为2。
...
assign Pest[2] = 1;   /错误：对寄存器的位选择不能够作为过程性连续赋值的目标*/
```

第二个赋值语句在进行下一个赋值前促使第一个赋值被重新赋值。在重新分配执行后，*Pest*的值在另一个对寄存器的赋值前保持为 2。

赋值语句在如下意义上是连续的：即在第 1个赋值执行后，第2个赋值开始执行前，*Hty*或*Mtu*上的任何变化将促使第1个赋值语句被重新计算。

8.8.2 force与release

force和release过程语句与assign和deassign非常相似，不同的是 force和release过程语句不仅能够应用于线网，也能够应用于寄存器的赋值。

当force语句应用于寄存器时，寄存器的当前值被 force语句的值覆盖；当 release语句应用于寄存器时，寄存器中的当前值保持不变，除非过程性连续赋值已经生效（在 force语句被执行时），在这种情况下，连续赋值为寄存器建立新值。

当用force过程语句对线网进行赋值时，该赋值方式为线网替换所有驱动源，直到在那个线网上执行release语句为止。

```
wire Prt;
...
or #1 (Prt,Std,Dzx);
initial
  begin
    force Prt = Dzx & Std;
    #5;                        //等待5个时间单位。
    release Prt;
  end
```

执行force语句使*Prt*的值覆盖来自于或门原语的值，直到 release语句被执行，然后或门原语的*Prt*驱动源重新生效。尽管 force赋值有效（在前5个时间单位），*Dzx*和*Std*上的任何变化都促使赋值重新执行。

另一实例如下：

```
reg[2:0] Colt;
...
Colt = 2;
force Colt = 1;
...
release Colt;          // Col保持值为1。
...
assign Colt = 5;
...
force Colt = 3;
...
```

China-pub.com 下载

```
release Colt;          //Colt值变为5。
...
force Colt[1:0] = 3;   /错误：寄存器的部分选择不能设为过程性连续赋值的目标 */
```

*Colt*的第1次释放促使 *Colt*的值被保持为1。这是因为在force语句被应用时没有过程性连续赋值对寄存器赋值。在后面的release语句中，*Colt*因为过程性连续赋值在*Colt*上重新生效而重新获得值5。

8.9 握手协议实例

always语句可用于描述交互进程的行为，如有限状态机的交互。这些模块内的语句用对所有always语句可见的寄存器来相互通信。在always语句间使用在一个always语句内声明的寄存器变量传递信息并不可取（这可以使用层次路径名实现，见第 10章）。

考虑下面两个交互进程的实例：*RX* ,接收器；*MP*,微处理器。*RX*进程读取串行的输入数据。并发送*Ready*信号表明数据可被读入*MP*进程。*MP*进程在将数据分配给输出后，回送一个接收信号*Ack*到*RX*进程以读取新的输入数据。两个进程的语句块流程如图 8-8所示。

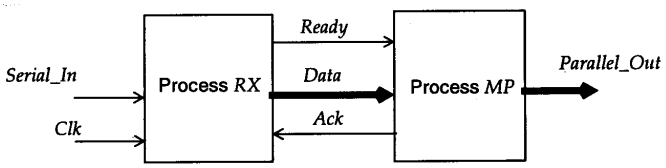

图8-8 两个交互进程

这两个交互进程的行为可用下述行为模型加以描述：

```
'timescale 1ns/100ps
module  Interacting (Serial_In,Clk,Parallel_Out)
  input Serial_In,Clk;
  output [0:7] Parallel_Out;
  reg [0:7] Parallel_Out;

  reg Ready,Ack;
  wire [0:7] data;

  'include "Read_Word.v"   /Read_Word任务在此文件中定义。
always
  begin: RX
    Read_Word(Serial_In,Clk,Data);
    //任务Read_Word在每个时钟周期读取串行数据，将其转换为并行数据并存于 Data中。Read_Word完
    成上述任务需要10ns。
    Ready = 1;
    wait(Ack);
    Ready = 0;
    #40;
  end

always
  begin: MP
    #25;
```

China-pub.com
下载


```
    Parallel_Out = Data;
    Ack = 1;
    #25 Ack = 0;
    wait (ready);
  end
endmodule
```

这两个进程通过寄存器*Ready*和*Ack*的交互握手协议如图8-9中的波形显示。

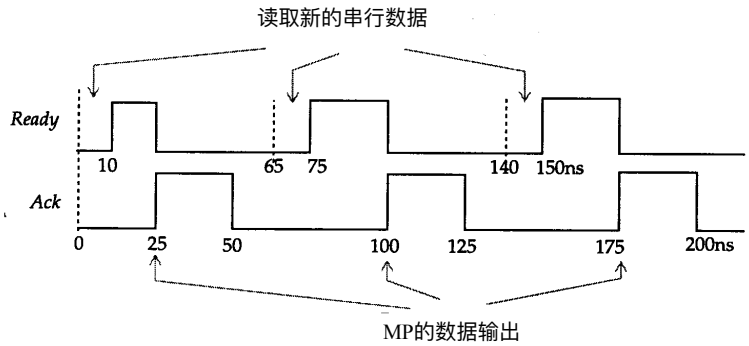

图8-9 两进程间的握手协议

习题

1. initial语句和always语句，哪一种可重复执行?
2. 顺序语句块和并行语句块的区别是什么？举例说明。顺序语句块能否出现在并行语句块中?
3. 语句块在什么时候需要标识符?
4. 在always语句中是否有必要指定时延?
5. 语句内部时延和语句间时延的区别是什么？举例说明。
6. 阻塞性赋值和非阻塞性赋值有何区别?
7. casex语句与case语句有何区别?
8. 能否在always语句中为线网类型（例如wire型线网）赋值?
9. 产生一个在5 ns时刻开始、周期为10 ns的时钟波形。
10. 用一个initial语句和1个forever循环语句替代下述always语句。

```
always
  @(Expected or Observed)
   if (Expected !== Observed) begin
    $display ("MISMATCH: Expected = %b,Observed = %b"
              Expected, Observed);
    $stop;
   end
```

11. 按如下条件，两个always语句中*NextStateA*和*NextStateB*上的值是多少：*ClockP*在5 ns时有1个正沿；*CurrentState*在时钟边沿前值为5，并且在时钟沿3 ns后改变为7?

```
always
  @ (posedge ClockP)
    #7 NextStateA = CurrentState;
```

China-pub.com 下载

```
always
  @(posedge ClockP)
    NextState = #7 CurrentState;
```

12. 使用行为建模方式写出如下有限状态机模型的行为描述。

Inp(Gak)	*PresentState*	*NextState*	*Output(Zuk)*
0	NO_ONE	NO_ONE	0
1	NO_ONE	ONE_ONE	0
0	ONE_ONE	ONE_ONE	0
1	ONE_ONE	TWO_ONE	0
0	TWO_ONE	NO_ONE	0
1	TWO_ONE	THREE_ONE	1
0	THREE_ONE	NO_ONE	0
1	THREE_ONE	THREE_ONE1	

13. 使用always语句描述JK触发器的行为功能。
14. 描述如下电路行为：该电路在每一个时钟下跳沿（负沿）检查输入数据，当输入数据 *Usg* 为1011时，输出*Asm*被置为1。
15. 描述多数逻辑电路行为。输入为 12位的向量。如果其中1的数量超过0的数量，输出设置为1。当*Data_Ready*为1时，才对输入数据进行检查。