第10章 其他论题

本章讲述诸如函数、任务、层次结构、值变转储文件和编译程序指令等多种论题。

10.1 任务

一个任务就像一个过程，它可以从描述的不同位置执行共同的代码段。共同的代码段用任务定义编写成任务，这样它就能够从设计描述的不同位置通过任务调用被调用。任务可以包含时序控制，即时延控制，并且任务也能调用其它任务和函数。

10.1.1 任务定义

任务定义的形式如下：

```
task task_id;
  [declarations]
  procedural_statement
endtask
```

任务可以没有或有一个或多个参数。值通过参数传入和传出任务。除输入参数外（参数从任务中接收值），任务还能带有输出参数（从任务中返回值）和输入输出参数。任务的定义在模块说明部分中编写。例如：

```
module Has_Task;
  parameter MAXBITS = 8;

  task Reverse_Bits;
    input [MAXBITS -1:0] Din;
    output [MAXBITS -1:0] Dout;
    integer K;

    begin
      for (K = 0; K < MAXBITS; K = K+1)
        Dout [MAXBITS-K] = Din[K];
    end
  endtask
  ...
endmodule
```

任务的输入和输出在任务开始处声明。这些输入和输出的顺序决定了它们在任务调用中的顺序。下面是另一个例子：

```
task Rotate_Left;
  inout [1:16] In_Arr;
  input [0:3] Start_Bit,Stop_Bit,Rotate_By;
  reg Fill_Value;
  integer Mac1,Mac3;

  begin
```

```
    for (Mac3 = 1; Mac3 <= Rotate_By; Mac3 = Mac3 + 1)
    begin
      Fill_Value = In_Arr[Stop_Bit];
      for (Mac1 = Stop_Bit; Mac1 >= Start_Bit + 1;
           Mac1 = Mac1 - 1 )
        In_Arr[Mac1] = In_Arr [Mac1 - 1];

      In_Arr[Start_Bit] = Fill_Value;
    end
  end
endtask
```

*Fill_Value*是任务的局部寄存器，只能在任务中直接可见。任务的第 1个参数是输入输出数组*In_Arr*，随后是3个输入，*Start_Bit*、*Stop_Bit*和*Rotate_By*。

除任务参数外，任务还能够引用说明任务的模块中定义的任何变量。在下一节将举例说明这种情况。

10.1.2 任务调用

一个任务由任务调用语句调用。任务调用语句给出传入任务的参数值和接收结果的变量值。任务调用语句是过程性语句，可以在 always 语句或initial 语句中使用。形式如下：

```
task_id [(expr1,expr2,...,exprN)];
```

任务调用语句中参数列表必须与任务定义中的输入、输出和输入输出参数说明的顺序匹配。此外，参数要按值传递，不能按地址传递。在其它高级编程语言中，例如 Pascal，任务与过程的一个重要区别是任务能够被并发地调用多次，并且每次调用能带有自己的控制，最重要的一点是在任务中声明的变量是静态的，即它决不会消失或重新被初始化。因此一个任务调用能够修改被其他任务调用读取的局部变量的值。

下面是调用任务*Reverse_Bits*的实例，该任务定义已在前面章节中给出，

```
//寄存器说明部分：
reg [MAXBITS-1:0] Reg_X,New_Reg;

Reverse_Bits(Reg_X,New_Reg);      //任务调用。
```

*Reg_X*的值作为输入值传递，即传递给 *Din*。任务的输出*Dout*返回到*New_Reg*。注意因为任务能够包含定时控制，任务可在被调用后再经过一定时延才返回值。

因为任务调用语句是过程性语句，所以任务调用中的输出和输入输出参数必须是寄存器类型的。在上面的例子中，*New_Reg*必须被声明为寄存器类型。

下面的例子不通过参数表向任务调用传入变量。尽管引用全局变量被认为是不良的编程风格，它有时却非常有用。

```
module Global_Var;
  reg [0:7] RamQ [0:63];
  integer Index;
  reg CheckBit;

  task GetParity;
    input  Address;
    output ParityBit;
```

```
    ParityBit = ^RamQ[Address];
  endtask

  initial
    for (Index = 0; Index <= 63; Index = Index+1)begin
      GetParity(Index,CheckBit);
      $display("Parity bit of memory word %d is %b.",
               Index, CheckBit);
    end
endmodule
```

存储器*RamQ*的地址被作为参数传递，而存储器本身在任务内直接引用。

任务可以带有时序控制，或等待特定事件的发生。但是，输出参数的值直到任务退出时才传递给调用参数。例如：

```
module TaskWait;
  reg NoClock;

  task GenerateWaveform;
    output ClockQ;
    begin
      ClockQ = 1;
      #2 ClockQ = 0;
      #2 ClockQ = 1;
      #2 ClockQ = 0;
   end
  endtask

  initial
    GenarateWaveform (NoClock);
endmodule
```

任务*GenerateWaveform*对*ClockQ*的赋值不出现在*NoClock*上，即没有波形出现在*NoClock*上；只有对*ClockQ*的最终赋值0在任务返回后出现在*NoClock*上。为避免这一情形出现，最好将*ClockQ*声明为全局寄存器类型，即在任务之外声明它。

10.2 函数

函数，如同任务一样，也可以在模块不同位置执行共同代码。函数与任务的不同之处是函数只能返回一个值，它不能包含任何时延或时序控制（必须立即执行），并且它不能调用其它的任务。此外，函数必须带有至少一个输入，在函数中允许没有输出或输入输出说明。函数可以调用其它的函数。

10.2.1 函数说明部分

函数说明部分可以在模块说明中的任何位置出现，函数的输入是由输入说明指定，形式如下：

```
function [range] function_id;
  input_declaration
  other_declarations
  procedural_statement
endfunction
```

如果函数说明部分中没有指定函数取值范围，则其缺省的函数值为 1位二进制数。函数实例如下：

```
module Function_Example
  parameter  MAXBITS = 8;

  function [MAXBITS-1:0] Reverse_Bits;
    input [MAXBITS-1:0] Din;
    integer  K;
    begin
      for (K=0; K < MAXBITS; K = K + 1)
        Reverse_Bits [MAXBITS-K] = Din [K];
    end
    endfunction
    ...
endmodule
```

函数名为*Reverse_Bits*。函数返回一个长度为 *MAXBITS*的向量。函数有一个输入 *Din*.*K*，是局部整型变量。

函数定义在函数内部隐式地声明一个寄存器变量，该寄存器变量与函数同名并且取值范围相同。函数通过在函数定义中显式地对该寄存器赋值来返回函数值。对这一寄存器的赋值必须出现在函数定义中。下面是另一个函数的实例。

```
function Parity;
  input [0:31] Set;
  reg [0:3] Ret;
  integer J;
  begin
    Ret = 0;

    for (J = 0;J <= 31; J = J + 1)
     if (Set[J]==1)
       Ret = Ret + 1;

    Parity = Ret % 2;
  end
endfunction
```

在该函数中，*Parity*是函数的名称。因为没有指定长度，函数返回 1位二进制数。*Ret*和*J*是局部寄存器变量。注意最后一个过程性赋值语句赋值给寄存器，该寄存器从函数返回值(与函数同名的寄存器在函数中被隐式地声明)。

10.2.2 函数调用

函数调用是表达式的一部分。形式如下：

```
func_id(expr1,expr2,...,exprN)
```

以下是函数调用的例子：

```
reg [MAXBITS-1:0] New_Reg,Reg_X;  //寄存器说明。

New_Reg = Reverse_Bits(Reg_X); //函数调用在右侧表达式内。
```

与任务相似，函数定义中声明的所有局部寄存器都是静态的，即函数中的局部寄存器在

函数的多个调用之间保持它们的值。

10.3 系统任务和系统函数

Verilog HDL提供了内置的系统任务和系统函数，即在语言中预定义的任务和函数。它们分为以下几类：

1) 显示任务（display task）
2) 文件输入/输出任务 （File I/O task）
3) 时间标度任务 (timescale task)
4) 模拟控制任务 (simulation control task)
5) 时序验证任务 (timing check task)
6) PLA建模任务 （PLA modeling task）
7) 随机建模任务 （stochastic modeling task）
8) 实数变换函数 (conversion functions for real)
9) 概率分布函数 (probabilistic distribution function)

PLA建模任务和随机建模任务不在本书的讨论范围内。

10.3.1 显示任务

显示系统任务用于信息显示和输出。这些系统任务进一步分为：

- 显示和写入任务
- 探测监控任务
- 连续监控任务

1. 显示和写入任务

语法如下：

```
task_name (format_specification1,argument_list1,
           format_specification2,argument_list2,
           ...,
           format_specificationN,argument_listN);
```

*task_name*是如下编译指令的一种：

```
$display  $displayb  $displayh  $displayo
$write    $writeb    $writeh    $writeo
```

显示任务将特定信息输出到标准输出设备，并且带有行结束字符；而写入任务输出特定信息时不带有行结束符。下列代码序列能够用于格式定义：

```
%h  或  %H  :  十六进制
%d  或  %D  :  十进制
%o  或  %O  :  八进制
%b  或  %B  :  二进制
%c  或  %C  :  ASCI字符
%v  或  %V  :  线网信号长度
%m  或  %M  :  层次名
%s  或  %S  :  字符串
%t  或  %T  :  当前时间格式
```

如果没有特定的参数格式说明，缺省值如下：

$display与$write ：十进制数
$displayb与$writeb ：二进制数
$displayo与$writeo ：八进制数
$displayh与$writeh ：十六进制数

可以用如下代码序列输出特殊字符：

```
\n    换行
\t    制表符
\\    字符\
\" 字符”
\OOO  值为八进制值OOO的字符
%%    字符%
```

例如：

```
$display("Simulation time is %t",$time);
$display($time,":R=%b,Q=%b,QB=%b", R,S,Q,QB);
   //因为没有指定格式，时间按十进制显示。
$write("Simulation time is:);
$write("%t\n",$time);
```

上述语句输出$time、R、S、Q和QB**等值的执行结果如下：**

```
Simulation time is                  10
                          10:R=1, S=0, Q=0, QB=1
Simulation time is                  10
```

2. 探测任务

探测任务有：

```
$strobe  $strobeb  $strobeh  $strobeo
```

这些系统任务在指定时间显示模拟数据，但这种任务的执行是在该特定时间步结束时才显示模拟数据。“ 时间步结束 ”意味着对于指定时间步内的所有事件都已经处理了。

```
always
  @(posedge Rst)
    $strobe("the flip-flop value is %b at time %t",Q,$time);
```

当*Rst***有一个上升沿时，**$strobe**任务输出** *Q* **的值和当前模拟时间。下面是** *Q* **和**$time**的一些值的输出。这些值在每次***Rst***的上升沿时被输出。**

```
The flip-flop value is 1 at time         17
The flip-flop value is 0 at time         24
The flip-flop value is 1 at time         26
```

其格式定义与显示和写入任务相同。

探测任务与显示任务的不同之处在于：显示任务在遇到语句时执行，而探测任务的执行要推迟到时间步结束时进行。下面的例子有助于进一步区分这两种不同任务。

```
integer  Cool;

initial
  begin
    Cool = 1;
    $display("After first assignment,Cool has value %b",Cool);
    $strobe("When strobe is executed,Cool has value %b",Cool);
    Cool = 2;
    $display("After second assignment,Cool has value %b",Cool);
end
```

产生的输出为：

```
After first assignment,Cool has value          1
When strobe is executed,Cool has value         2
After second assignment,Cool has value         2
```

第一个$display任务输出*Cool*的值1（*Cool*的第一个赋值）。第二个$display任务输出*Cool*的值2（*Cool*的第二个赋值）。$strobe任务输出*Cool*的值2，这个值保持到时间步结束。

3. 监控任务

监控任务有：

```
$monitor  $monitorb  $monitorh  $monitoro
```

这些任务连续监控指定的参数。只要参数表中的参数值发生变化，整个参数表就在时间步结束时显示。例如：

```
initial
  $monitor ("At %t,D = %d, Clk = %d"),
    $time,D,Clk,"and Q is %b",Q);
```

当监控任务被执行时，对信号 *D*、*Clk*和*Q*的值进行监控。若这些值发生任何变化，则显示整个参数表的值。下面是*D*、*Clk*和*Q*发生某些变化时的输出样本。

```
At          24, D = x, Clk = x and Q is 0
At          25, D = x, Clk = x and Q is 1
At          30, D = 0, Clk = x and Q is 1
At          35, D = 0, Clk = 1 and Q is 1
At          37, D = 0, Clk = 0 and Q is 1
At          43, D = 1, Clk = 0 and Q is 1
```

监控任务的格式定义与显示任务相同。在任意时刻对于特定的变量只有一个监控任务可以被激活。

可以用如下两个系统任务打开和关闭监控。

```
$monitoroff;  //禁止所有监控任务。

$monitoron;  //使能所有监控任务。
```

这些提供了控制输出值变化的机制。 $monitoroff任务关闭了所有的监控任务，因此不再显示监控更多的信息。$monitoron任务用于使能所有的监控任务。

10.3.2 文件输入/输出任务

1. 文件的打开和关闭

系统函数$fopen用于打开一个文件。

```
integer file_pointer = $fopen(file_name);
  //系统函数$fopen返回一个关于文件的整数（指针）。
```

而下面的系统任务可用于关闭一个文件：

```
$fclose(file_pointer);
```

这是一个关于它的用法的例子。

```
integer  Tq_File;

initial
```

```
begin
  Tq_File = $fopen("~/jb/div.tq");
  ...
  $fclose(Tq_File);
end
```

2. 输出到文件

显示、写入、探测和监控系统任务都有一个用于向文件输出的相应副本，该副本可用于将信息写入文件。这些系统任务如下：

```
$fdisplay  $fdisplayb  $fdisplayh  $fdisplayo
$fwrite    $fwriteb    $fwriteh    $fwriteo
$fstrobe   $fstrobeb   $fstrobeh   $fstrobeo
$fmonitor  $fmonitorb  $fmonitorh  $fmonitoro
```

所有这些任务的第一个参数是文件指针，其余的所有参数是带有参数表的格式定义序列。下面的实例将作进一步解释说明。

```
integer  Vec_File;

initial
  begin
    Vec_File = $fopen("div.vec");
    ...
    $fdisplay(Vec_File,"The simulation time %t",$time);
    //第一个参数Vec_File是文件指针。
    $fclose(Vec_file);
  end
```

等到$fdisplay任务执行时，文件“div.vec”中出现下列语句：

```
The simulation time is          0
```

3. 从文件中读取数据

有两个系统任务能够用于从文件中读取数据，这些任务从文本文件中读取数据并将数据加载到存储器。它们是：

```
$readmemb      $readmemh
```

文本文件包含空白空间、注释和二进制（对于 $readmemb）或十六进制（对于$readmemh）数字。每个数字由空白空间隔离。当执行系统任务时，每个读取的数字被指派给存储器内的一个地址。开始地址对应于存储器最左边的索引。

```
reg [0:3] Mem_A [0:63];

initial
  $readmemb("ones_and_zero.vec",Mem_A);
  //读入的每个数字都被指派给从0开始到63的存储器单元。
```

显式的地址可以在系统任务调用中可选地指定，例如：

```
$readmemb("rx.vex",Mem_A,15,30);
  //从文件“rx.vec”中读取的第一个数字被存储在地址15中，下一个存储在地址
  //16，并以此类推直到地址30。
```

也可以在文本文件中显式地给出地址。形式如下：

```
@address_in_hexadecimal
```

在这种情况下，系统任务将数据读入指定地址。后续的数字从指定地址开始向后加载。

10.3.3 时间标度任务

系统任务

```
$printtimescale
```

给出指定模块的时间单位和时间精度。若$printtimescale任务没有指定参数，则用于输出包含该任务调用的模块的时间单位和精度。如果指定到模块的层次路径名为参数，则系统任务输出指定模块的时间单位和精度。

```
$printtimescale;

$printtimescale(hier_path_to_module);
```

下面是这些系统被调用时输出的样本。

```
Time scale of (C10) is 100ps/100ps
Time scale of (C10.INST) is lus/100ps
```

系统任务

```
$timeformat
```

指定%t格式定义如何报告时间信息，该任务形式如下：

```
$timeformat(units_number,precision,
            suffix,numeric_field_width);
```

其中*units_number*为：

```
0  : 1 s
-1 : 100 ms
-2 : 10 ms
-3 : 1 ms
-4 : 100 us
-5 : 10 us
-6 : 1 us
-7 : 100 ns
-8 : 10 ns
-9 : 1 ns
-10 : 100 ps
-11 : 10 ps
-12 : 1 ps
-13 : 100 fs
-14 : 10 fs
-15 : 1 fs
```

系统任务调用

```
$timeformat(-4, 3, "ps", 5);
$display("Current simulation time is %t",$time);
```

将显示$display任务中%t说明符的值，如下：

```
Current simulation time is 0.051 ps
```

如果没有指定$timeformat，%t按照源代码中所有时间标度的最小精度输出。

10.3.4 模拟控制任务

系统任务

```
$finish;
```

使模拟器退出，并将控制返回到操作系统。

系统任务

```
$stop
```

使模拟被挂起。在这一阶段，交互命令可能被发送到模拟器。下面是该命令使用方法的例子。

```
initial  #500 $stop;
```

500个时间单位后，模拟停止。

10.3.5 定时校验任务

系统任务：

```
$setup(data_event,reference_event,limit);
```

如果

```
(time_of_reference_event - time_of_data_event) < limit
```

则报告时序冲突（timing violation）；

系统调用实例如下：

```
$setup(D, posedge Ck, 1, 0);
```

系统任务：

```
$hold(reference_event,data_event,limit);
```

如果

```
(time_of_data_event - time_of_reference_event) < limit,
```

则报数据保持时间时序冲突。

例如：

```
$hold(posedge Ck,D,0.1);
```

系统任务$setuphold是$setup和$hold任务的结合：

```
$setuphold(reference_event,data_event,setup_limit,hold_limit);
```

而系统任务：

```
$width(reference_event,limit,threshold);
```

则检查信号的脉冲宽度限制，如果

```
threshold <(time_of_data_event-time_of_reference_event) < limit
```

则报告信号上出现脉冲宽度不够宽的时序错误。

数据事件来源于基准事件：它是带有相反边沿的基准事件，例如：

```
$width(negedge Ck,0.0,0);
```

系统任务：

```
$period(reference_event,limit)
```

检查信号的周期，若

```
( time_of_data_event - time_of_reference_event) < limit
```

则报告时序错误。

基准事件必须是边沿触发事件。数据事件来源于基准事件：它是带有相同边沿的基准事件。

系统任务：

```
$skew(reference_event,data_event,limit)
```

检查信号之间（尤其是成组的时钟控制信号之间）的偏斜（skew）是否满足要求，若

```
time_of_data_event - time_of_reference_event > limit
```

则报告信号之间出现时序偏斜太大的错误。如果 data_event的时间等于reference_event的时间，则不报出错。

系统任务:

```
$recovery(reference_event,data_event,limit);
```

主要检查时序状态元件（触发器、锁存器、 RAM和ROM等）的时钟信号与相应的置/复位信号之间的时序约束关系，若

```
( time_of_data_event - time_of_reference_event) < limit
```

则报告时序冲突。该系统任务的基准事件必须是边沿触发事件。该系统任务在执行定时校检前记录新基准事件时间；因此，如果数据事件和基准事件在相同的模拟时间同时发生，就报告时序冲突错误。

系统任务：

```
$nochange(reference_event,data_event,start_edge_offset,
          end_edge_offset);
```

如果在指定的基准事件区间发生数据变化，就报告时序冲突错误。基准事件必须是边沿触发事件。例如：

```
$nochange(negedge Clear,Perset,0,0);
```

如果在*Clear*为低时*Preset*发生变化，将报告时序冲突错误。

上述每个系统任务均可以带有一个可选参数 *notifier*。当发生时序冲突时，系统任务根据下列case语句改变自身的值来更新参数 *notifier*。

```
case  (notifier)
  'bx : notifier= 'b0;
  'b0 : notifier= 'b1;
  'b1 : notifier= 'b0;
  'bz : notifier= 'bz;
end
```

*notifier*参数可提供关于时序冲突或传播 x到输出的时序冲突。使用参数 *notifier*的例子如下：

```
reg NotifyDin;
...
$setuphold (negedge Clock,Din,tSETUP,tHOLD,NotifyDin);
```

在这一实例中，*NotifyDin*是参数notifier。如果时序冲突发生，寄存参数 *NotifyDin*根据前面对参数notifier描述的case语句改变其值。

10.3.6 模拟时间函数

下列系统函数返回模拟时间。

- $time：返回64位的整型模拟时间给调用它的模块。
- $stime：返回32位的时间。
- $realtime：向调用它的模块返回实型模拟时间。

例如

```
`timescale 10ns/1ns
module TB;
  ...
  initial
```

```
    $monitor ("Put_A=%d Put_B=%d",Put_A,Put_B,
              "Get_O=%d",Get_O,"at time %t",$time);
endmodule
```

该例产生的输出如下：

```
Put_A=0 Put_B=0 Get_O=0 at time 0
Put_A=0 Put_B=1 Get_O=0 at time 5
Put_A=0 Put_B=0 Get_O=0 at time 16
```

$time按模块*TB*的时间单位比例返回值，并且被四舍五入。注意 $timeformat描述了时间值如何被输出。下面是另一例子及其输出。

```
initial
  $monitor ("Put_A=%d Put_B=%d",Put_A,Put_B,
            "Get_O=%d",Get_O,"at time %t",$realtime);
    Put_A=0 Put_B=1 Get_O=0 at time 5.2
    Put_A=0 Put_B=0 Get_O=0 at time 15.6
```

10.3.7 变换函数

下列系统函数是数字类型变换的功能函数：

- $**rtoi**(*real_value*)：通过截断小数值将实数变换为整数。
- $**itor**(*integer_value*)：将整数变换为实数。
- $**realtobits**(*real_value*)：将实数变换为64位的实数向量表示法（实数的IEEE 745表示法）
- $**bitstoreal**(*bit_value*)：将位模式变换为实数（与 $realtobits相反）

10.3.8 概率分布函数

函数：

```
$random [(seed)]
```

根据种子变量（seed）的取值按32位的有符号整数形式返回一个随机数。种子变量（必须是寄存器、整数或时间寄存器类型）控制函数的返回值，即不同的种子将产生不同的随机数。如果没有指定种子，每次 $random函数被调用时根据缺省种子产生随机数。

例如，

```
integer Seed, Rnum;
wire Clk;

initial Seed = 12;

always
  @ (Clk) Rnum= $random (Seed);
```

在*Clk*的每个边沿，$random被调用并返回一个32位有符号整型随机数。

如果数字在取值范围内，下述模运算符可产生－ 10～+10之间的数字。

```
Rnum = $random(Seed) % 11;
```

下面是没有显式指定种子的例子。

```
Rnum = $random / 2;  //种子变量是可选的。
```

注意数字产生的顺序是伪随机排序的，即对于一个初始种子值产生相同的数字序列。

表达式：

```
{random} % 11
```

产生0～10之间的一个随机数。并置操作符（{}）将$random函数返回的有符号整数变换为无符号数。

下列函数根据在函数名中指定的概率函数产生伪随机数。

```
$dist_uniform ( seed,start,end)

$dist_normal ( seed,mean,standard_deviation,upper)

$dist_exponential ( seed,mean)

$dust_poisson ( seed,mean)

$dist_chi_square ( seed,degree_of_freedom)

$dist_t ( seed,degree_of_freedom)

$dist_erland ( seed,k_stage,mean)
```

这些函数的所有参数都必须是整数。

10.4 禁止语句

禁止语句是过程性语句（因此它只能出现在always或initial语句块内）。禁止语句能够在任务或程序块没有执行完它的所有语句前终止其执行。它能够用于对硬件中断和全局复位的建模。其形式如下：

```
disable task_id;
disable block_id;
```

在禁止语句执行后，继续执行任务调用或被禁止的程序块的下一条语句。

```
begin: BLK_A
  //语句1。
  //语句2。
  disable BLK_A;
  //语句3。
  //语句4。
  end
  //语句5。
```

语句3和语句4从未被执行。在禁止语句被执行后，执行语句5。又如：

```
task Bit_Task;
  begin
    //语句6。
    disable Bit_Task;
    //语句7。
  end
endtask

    //语句8。
    Bit_Task;  /任务调用
    //语句9。
```

当禁止语句被执行时，任务被放弃，即语句 7永远不会被执行。紧跟在任务调用后面继续执行的语句，在此例中是语句 9。

建议最好在任务定义中不要使用 disable禁止语句，尤其是当任务具有一定的返回值时更是如此。这是因为当任务被禁止时， Verilog语言给出的输出和输入参数值是不确定的。如果必须在任务中这样做，一种比较稳妥的方法是：如果有，就在任务中禁止顺序程序块，例如，

```
task  Example;
  output [0:3] Count;
  begin：LOCAL_BLK
    //语句10。
    Count = 10;
    disable LOCAL_BLK;
    //语句11。
  end
endtask
```

当禁止语句开始执行时，禁止语句促使顺序程序块 *LOCAL_BLK*退出。由于这是任务中的唯一语句，因此任务退出，并且 *Count*的值为10。如果禁止语句被替换为：

```
disable Example；
```

那么在禁止语句开始执行后， *Count*的值不确定。

10.5 命名事件

考虑下述两个always语句。

```
reg Ready ,Done;

//获取always语句的交互：
initial
  begin
    Done = 0;
    #0 Done = 1;
  end

always
  @(Done) begin
    ...
    //完成处理这个always语句。
    //触发下一个always语句。
    //在Ready信号上创建一个事件。
    Ready = 0;
    #0 Ready = 1;
  end

always
  @ (Ready) begin
    ...
    //完成处理这个always语句。
    //创建事件触发前面的always语句。
    Done = 0;
    #0 Done = 1;
end
```

每个always语句中的两个赋值必须要确认在 *Ready*和*Done*上创建一个事件。这表明 *Ready*和*Done*的目的是作为两个always语句间的握手信号。

Verilog HDL提供一种替代机制实现这一功能——使用命名事件。命名事件是Verilog HDL语言的另外一种数据类型（Verilog语言中的其它两类数据类型是寄存器类型和线网数据类型）。命名事件必须在使用前声明。声明形式如下：

```
event Ready, Done;
```

事件声明说明*Ready*和*Done*为两个命名事件。声明命名事件后，可以使用事件触发语句创建事件。形式如下：

```
->Ready;
->Done;
```

命名事件上的事件能够同变量上的事件一样被监控，即使用 @机制，例如：

```
@ (Done) <动作语句>
```

所以只要*Done* 上的事件触发语句被执行，一个事件就在 *Done*上发生，这使<动作语句>执行。

可以用命名事件重写两类always 语句的简例：

```
event  Ready, Done

initial
  ->Done;

always
  @(Done) begin
    ...
    //触发下一个always语句。
    //在Ready上创建一个事件。
    ->Ready;
  end

always
  @(Ready)begin
    ...
    //创建事件来触发前面的always语句。
    ->Done;
  end
```

也可以使用事件描述状态机。下面是一个异步状态机实例：

```
event State1, State2, State3;

//状态复位
initial
  begin
    //复位状态逻辑。
    ->State1;
  end

always
  @(State1) begin
    // State1 逻辑。
    ->State2;   //在State2 上创建事件。
```

```
  end

always
  @(State2) begin
    //State2 逻辑。
    ->State3;    /在State3 上创建事件。
  end

always
  @(State3) begin
    //State3 逻辑。它可以有如下语句：
    if (InputA)
      ->State2;       /在State2 上创建事件。
    else
      ->State1;       /在State1 上创建事件。
    end
```

initial语句描述复位逻辑。在initial 语句执行结束时，触发第2条always 语句，该语句中最后一条语句的执行促使在 *State2*上发生一个事件；这促使第 3条always 语句执行，然后第 4条always 语句执行。在最后一条 always 语句中，根据 *Input A*的值决定事件发生在 *State2*还是 *State1*上。

10.6 结构描述方式和行为描述方式的混合使用

在前面的章节中，我们描述了硬件建模的几种不同方式。 Verilog HDL允许所有这些不同风格的建模在单一模块中结合使用。模块语法如下：

```
module  module_name(port_list);
  Declarations:
    Input ,ouput and inout declarations.
    Net declarations.
    Reg declarations.
    Parameter declarations.

    Initial statement.
    Gate instantiation statement.
    Module instantiation statement.
    UDP instantiation statement.
    Always statement.
    Continuous assignment.
endmodule
```

下面的实例采用不同的风格进行硬件建模：

```
module  MUX2x1 (Ctrl,A,B,Ena,Z)
  //输入说明：
  inout Ctrl,A,B,Ena;
  //输出说明：
  output Z;
  //线网说明：
  wire Mot,Not_Ctrl;
  //带赋值的线网说明：
  wire Z = Ena == 1 ? Mot : 'bz;
```

```
    //门实例语句：
    not (Not_Ctrl,Ctrl);
    or (Mot,Ta,Tb);

    //连续赋值
    assign Ta = A & Ctrl;
    assign Tb = B & Not_Ctrl;
  endmodule
```

模块包含内置逻辑门（结构化组件）和连续赋值（数据流方式）的混合描述形式。

10.7 层次路径名

Verilog HDl中的标识符具有一个唯一的层次路径名。层次路径名通过由句点（.）隔开的名字组成。新层次由以下定义：

1) 模块实例化
2) 任务定义
3) 函数定义
4) 命名程序块

任何标识符的全称路径名由顶层模块（不被任何其它模块实例化的模块）开始。这一路径名可在描述的任何层次使用。实例如下。图 10-1显示了层次。

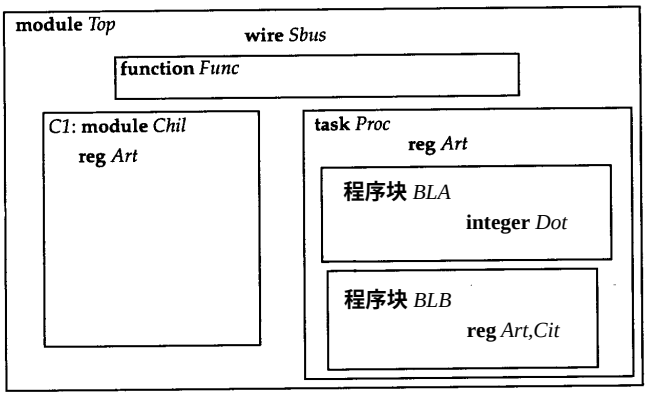

图10-1 模块层次

```
module Top;
  wire  Sbus;

  function Func...
    ...
  endfunction

  task Proc
   ...
   reg Art;

   begin : BLA
     integer  Dot;
```

```
      ...
    end

    begin : BLB
      reg Art, Cit;
      ...
  end
endtask

 Chil C1(...) ;   //一个模块实例。
endmodule         //模块Top。

module Chil;
  reg Art;
  ...
endmodule
```

本例中的层次名为：

```
Top.C1.Art
Top.Proc.Art
Top.Proc.BLB.Art
Top.Proc.BLA.Dot
Top.Proc.BLB.Cit
Top.Sbus
```

这些层次名允许自由访问层次结构中任一层次的任一数据项。数据不仅可读，而且可以通过路径名更新任何层次中的数据项的值。

较低层模块能够通过使用模块实例名限定变量引用高层（称为向上引用）或低层(称为向下引用)模块。形式如下：

```
module_instance_name.variable_name
```

对于向下路径引用，模块实例必须与较低层模块在同一层。例如：

```
module Top;
  wire Sbus;

  Chil Cl (. . .);        //一个模块实例。

  $display (Cl.Art);     //向下引用。
endmodule

module Chil;
  reg Art;
  . . .
endmodule
```

10.8 共享任务和函数

一种在不同模块间共享任务和函数的方法是在文本文件中编写共享任务和函数的定义，然后使用`include编译指令在需要的模块中包含这些定义。假设我们在文件“ share.h ”中有如下函数和任务定义：

```
function SignedPlus;
```

```
  . . .
endfunction

function SignedMinus;
 . . .
endfunction

task PresetClear;
  . . .
endtask
```

下面是在模块中使用文件的方式：

```
module SignedAlu(A, B, Operation, Z);
  input [0:3] A, B;
  input Operation;
  output [0:3] Z;
  reg [0:3] Z;

  //包含共享函数的定义。
  `include "share.h"

  always
    @ (A or B or Operation)
      if (Operation)
        Z = SignedPlus (A, B);
      else
        Z = SignedMinus (A, B);
endmodule
```

注意因为文件“Share.h”中任务和函数定义没有被模块说明限定，`inclade编译指令必须在模块说明内出现。

有一种可选的方法是在模块内定义共享任务和函数，然后用层次名在不同的模块中引用需要的任务或函数。下面的实例与前面相同，但是这一次任务和函数定义在模块说明内出现。

```
module Share;
  function SignedPlus;
    . . .
  endfunction

  function SignedMinus;
    . . .
  endfunction

  task PresetClear;
    . . .
  endtask
endmodule
```

下面是在其它模块中引用共享函数的方法。

```
module SignedAlu2 (A, B, Operation, Z);
  input [0:3] A, B;
  input Operation;
  output [0:3] Z;
```

```
  reg [0:3] Z;

  always
    @ (A or B or Operation)
      if (Operation)
        Z = Share.SignedPlus (A, B);
      else
        Z = Share.SignedMinus (A, B);
endmodule
```

10.9 值变转储文件

值变转储（VCD）文件包含设计中指定变量取值变化的信息。它的主要目的是为其它后处理工具提供信息。

下面的系统任务用于创建和将信息导入 VCD文件。

1) $dumpfile：本系统任务指定转储文件名。

例如：

```
$dumpfile ("uart.dump");
```

2) $dumpvars：本系统任务指定哪些变量值变化时转储进转储文件。

```
$dumpvars;
  //无参数，它指定在设计中转储所有变量。

$dumpvars (level, module_name);
  //在指定模块和所有指定层次下面的模块中的转储变量。

$dumpvars (1, UART);
  //只转储模块UART中的变量。

$dumpvars (2, UART);
  //转储UART及其下一层模块中的所有变量。

$dumpvars (0, UART);
  //第0层导致UART模块及其下面层次中所有模块中的各个变量被转储。

$dumpvars (0, P_State, N_State);
  //转储关于P_State和N_State的变量信息。层次数与此例无关，但是必须给出//层次数。

$dumpvars (3, Div.Clk, UART);
  //层次数只作用于模块本身及其下面两个层次中的所有变量。在此例中，只作用于模块 UART，即UART中
    所有变量及UART下面两个层次中各模块的所有变量，同时转储变量Div.Clk上值的变化。
```

3) $dumpoff：本系统任务促使转储任务被挂起。

```
$dumpoff;
```

4) $dumpon：本系统任务促使所有转储任务被挂起。语法如下：

```
$dumpon;
```

5) $dumpall：本系统任务执行时转储所有当前指定的变量值。语法如下：

```
$dumpall
```

6) $dumplimit：本系统任务为VCD文件指定最大长度(字节)。转储在到达此界限时停止。

例如，

```
$dumplimit (1024);    //VC文件的最大值为1024字节。
```

7) $ dumpflush：本系统任务刷新操作系统 VCD文件缓冲区中的数据，将数据存到 VCD 文件中。执行此系统任务后，转储任务处于唤醒状态。

```
$dumpflush;
```

10.9.1 举例

下面是在5～12之间计数的可逆计数器的例子：

```
module CountUpDown (Clk, Count, Up_Down);
  input Clk, Up_Down;
  output [0:3] Count;
  reg [0:3] Count;

  initial Count = 'd5;

  always
    @ (posedge Clk) begin
      if (Up_Down)
        begin
          Count = Count + 1;

          if (Count > 12)
            Count = 12;
        end
      else
        begin
          Count = Count -1;

          if (Count < 5)
            Count = 5;
        end
    end
  endmodule

  module Test;
    reg Clock, UpDn;
    wire [0:3] Cnt_Out;
    parameter ON_DELAY = 1, OFF_DELAY = 2;

    CountUpDown C1(Clock, Cnt_Out, UpDn);

    always
      begin
        Clock = 1;
        #ON_DELAY;
        Clock = 0;
        #OFF_DELAY;
      end
    initial
```

```
    begin
      UpDn = 0;
      #50 UpDn = 1;
      #100 $dumpflush;
      $stop;        /停止模拟。
    end

  initial
    begin
      $dumpfile ("count.dump");
      $dumplimit (4096);
      $dumpvars (0, Test;)
      $dumpvars (0, Cl.Count, Cl.Clk, C1.Up_Down);
    end
  endmodule
```

10.9.2 VCD文件格式

VCD文件是ASCII文件。VCD文件包含如下信息：

- 文件头信息：提供日期、模拟器版本和时间标度单位。
- 节点信息：定义转储作用域和变量类型。
- 取值变化：实际取值随时间变化。记录绝对模拟时间。

VCD文件实例如图10-2所示。

```
$date                                   $dumpvars
Fri Sep 27 16:23:58 1996                1#
$end                                    0$
$version                                b1 !
Verilog HDL Simulator 1.0               b10 "
$end                                    b101 +
$timescale                              1(
100ps                                   0'
$end                                    1&
$scope module Test $end                 1)
$var parameter 32 ! ON_DELAY            0*
$end                                    $end
$var parameter 32 " OFF_DELAY           #10
$end                                    0#
$var reg 1 # Clock $end                 0)
$var reg 1 $ UpDn $end                  #30
$var wire 1 % Cnt_Out (0) $end          1#
$var wire 1 & Cnt_Out (1) $end          1)
$var wire 1 ' Cnt_Out (2) $end          b100 +
$var wire 1 ( Cnt_Out (3) $end          b101 +
$scope module C1 $end                   #40
$var wire 1 ) Clk $end                  0#
$var wire 1 * Up_Down $end              0)
$var reg 4 + Count (0:3) $end           #60
$var wire 1 ) Clk $end                  1#
$var wire 1 * Up_Down $end              1)
$upscope $end                           b100 +
$upscope $end                           b101 +
$enddefinitions $end                    #70
#0                                      0#
接右栏                                  ...
```

图10-2 VCD文件

10.10 指定程序块

迄今为止，我们讨论的时延，如门时延和线网时延，都是分布式时延。模块中关于路径的时延，称为模块路径时延，可以使用指定程序块来指定。通常，指定程序块有如下用途：

1) 指定源和目的之间的通路。

2) 为这些通路分配时延。

3) 对模块进行时序校验。

指定程序块出现在模块说明内。形式如下：

```
specify
  spec_param_declarations                  //参数说明
  path_declarations                        //通路说明
  system_timing_checks                     //系统时序校验说明
endspecify
```

specparam(或指定参数)说明指定程序块内使用的参数。实例如下：

```
specparam tSETUP = 20, tHOLD = 25;
```

在指定程序块内能够描述下述三类模块通路：

- 简单通路
- 边沿敏感通路
- 与状态有关的通路

可以使用如下两种形式说明简单通路。

```
source *> destination
  //指定一个完全连接：源参数上的每一位都与目的参数的所有位相连接。

source => destination
  //指定一个并行连接：源参数上的每一位分别与目的参数的位一一连接。
```

以下是一些实例。

```
input Clock;
input [7:4] D;
output [4:1] Q;

(Clock => Q) = 5;
  //从输入Clock到Q的每位延迟为5。

(D *> Q) =  (tRISE, tFALL);
  /* 包括下述通路：
    D[7] 到 Q[4]
    D[7] 到 Q[3]
    D[7] 到 Q[2]
    D[7] 到 Q[1]
    D[6] 到 Q[4]
    . . .
    D[4] 到 Q[1]
  */
```

在边沿敏感通路中，通路描述与源的边沿相关。例如，

```
(posedge Clock => (Qb +: Da)) = (2:3:2);
  /*通路时延从Clock的正沿到Qb。数据通路从Qb到Da, 当Da传播到Qb时，Da不翻转。*/
```

与状态有关的通路在某些条件为真时指定通路时延。例如，

```
if (Clear)
  (D => Q) =  (2.1, 4.2);
    //只在Clear为高电平时，为指定通路时延。
```

下面是可在指定程序块内使用的时序验证系统任务。

```
$setup            $hold
$sethold          $period
$skew             $recovery
$width            $nochange
```

下面是指定程序块的实例。

```
specify
  //指定参数：
  specparam tCLK_Q = (5:4:6);
  specparam tSETUP = 2.8, tHOLD = 4.4;

 //指定通路的路径时延：
  (Clock *> Q) = tCLK_Q;
  (Data *> Q) = 12;
  (Clear, Preset *> Q) = (4,5);

  //时序验证：
  $setuphold (negedge Clock, Data, tSETUP, tHOLD);
endspecify
```

沿着模块通路，只有长度大于通路时延的脉冲才能传播到输出。但是，还可以通过用称为PATHPULSE$的专门指定程序块参数进行控制。除用于指定被舍弃的脉冲宽度范围外，还可用于指定促使通路结束处出现x的脉冲宽度范围。参数说明的简单形式如下：

```
PATHPULSE$= (reject_limit, [, error_limit]);
```

如果脉冲宽度小于 *reject_limit*，那么脉冲就不会传播到输出。如果脉冲宽度小于 *error_limit*(如果未指定就与*reject_limit*相同)。但是大于*reject_limit*，在通路的目标处产生x。

也可以按如下形式使用：

```
PATHPULSE$input_terminal$output_terminal
```

PATHPULSE $参数为特殊通路指定脉冲界限。

下面是指定程序块的实例。

```
specify
specparam PATHPULSE$ = (1, 2);
  // Reject limit = 1, Error limit = 2.
specparam PATHPULSE$Data$Q = 6;
  // Reject limit = Error limit = 6在从 Data 到 Q 的路径上。
endspecify
```

10.11 强度

在Verilog HDL中除了指定四个基本值0、1、 x和z外，还可以对这些值指定如驱动强度和电荷强度等属性。

10.11.1 驱动强度

驱动强度可以在如下情况中指定：

1) 在线网说明中带赋值的线网。

2) 原语门实例中的输出端口。

3) 在连续赋值语句中。

驱动强度定义有两个值：一个是线网被赋值为 1时的强度值；另一个是线网被赋值为 0时的强度值。形式如下：

```
(strength_for_1, strength_for_0)
```

值的顺序并不重要。对于值1的赋值，只允许如下的信号驱动强度：

- supply1
- strong1
- pull1
- weak1
- highz1(禁止对门级原语使用)

对于值0的赋值，允许如下的信号驱动强度：

- supply0
- strong0
- pull0
- weak0
- highz0 (禁止对门级原语使用)

缺省的信号驱动强度定义为(strong0, strong1)。例如：

```
//线网的强度：
wire (pull1, weak0) # (2,4) Lrk = Pol && Ord;
//信号的驱动强度定义仅适用于标量类型的信号，如： wire、wand、wor、tri、triand、trior、
trireg、//tri0和tri1。


//门级原语输出端口的驱动强度定义：
nand (pull1, strong0) # (3:4:4) Al (Mout, MinA, MinB, MinC);
 //信号驱动强度仅适用于定义下列门级原语的外部端口： and、or、xor、nand、nor、xnor、//buf
 bufif0、bufif1、not、notif0、notif1、pulldown和pullup。


   //连续赋值语句中的强度定义：
   assign (weak1, pull0) #2.56 Wrt = Ctr;
```

线网的驱动强度可以在显示任务中用 %v格式定义输出。例如：

```
$display (" Prq is %v", Prq);
```

结果为：

```
Prq is Wel
```

10.11.2 电荷强度

三态寄存器线网也能有选择地规定其存储的电荷强度。三态寄存器线网存储的电荷强度与三态线网相关的电容大小有关。电荷强度分为三类：

- 小型
- 中型（如果没有特别强调，则为缺省值）
- 大型

此外，三态寄存器线网存储的电荷衰退时间也可被指定。实例如下：

```
trireg (small ) # (5,4,20)Tro;
```

三态寄存器线网*Tro*有一个小型电容。上升时延是5个时间单位，下降时延是4个时间单位，并且放电时间(当线网处于高阻状态时的电容器放电)是20个时间单位。

10.12 竞争状态

如果在连续赋值或always 语句中未使用时延，即是零时延时，会产生竞争状态。这是因为Verilog HDL没有定义同时发生的事件的模拟顺序。

下面用一个简例解释说明使用非阻塞性赋值的零时延情况。

```
begin
  Start <= 0;
  Start <= 1;
end
```

在时间步结束时，值0和值1都被调度为*Start*赋值。根据事件的排序 (由模拟器内部决定)，*Start*上的结果可以是0，也可以是1。

下面的另一例子显示由于事件排序产生的竞争状态。

```
initial
  begin
    Pal = 0;
    Ctrl = 1;
    #5 Pal = 1;
    Ctrl = 0;
  end

always
  @ (Cot or Ctrl) begin
    $display ("The value of Cot at time",$time, "is", Cot);

  end

assign Cot = Pal;
```

在时刻0，当*Pal*和*Ctrl*在initial语句内被赋值时，连续赋值语句和 always语句都已为执行做好准备。那么哪一个先执行呢？Verilog HDL语言没有定义这种顺序。如果连续赋值语句首先执行，*Cot*赋值为0，反过来触发always语句。但是因为它已为执行做好准备，所以没有发生任何改变。always语句开始执行，*Cot*的值显示为0。

如果我们假设首先执行 always语句，*Cot*的当前值被输出 (连续赋值语句还没有执行)，然后连续赋值语句开始执行，更新*Cot*的值。

因此，在处理零时延赋值时要格外注意。下面是竞争状态的另一实例。

```
always @ (posedge GlobalClk)
  RegB = RegA;

always @ (posedge GlobalClk)
  RegC = RegB;
```

语言没有指出在*GlobalClk*上有正沿时，哪一条 always 语句首先执行。如果执行第1条always 语句，*RegB*将立即获得*RegA*的值；随后第2条always语句执行，*RegC*将获得*RegB*最新

的取值(第1条always语句中的赋值)。

如果首先执行第2条always语句，*RegC*将获得*RegB*的旧值(*RegB*还没有被赋值)，随后*RegB*将被赋于*RegA*的值。所以根据首先执行哪一条always语句，*RegC*取不同的值。因为过程性赋值立即发生，即没有任何时延，所以会产生一些问题。避免这种问题的一种方法是插入语句内时延。但最好是使用非阻塞性赋值语句。如：

```
always @ (posedge GlobalClk)
  RegB < = RegA;

always @ (posedge GlobalClk)
  RegC <= RegB;
```

当always语句通过变量通信时，对变量赋值时使用非阻塞性赋值可以避免产生竞争状态。

习题

1. 函数可以调用任务吗?
2. 任务能够带有时延吗?
3. 函数能够带有零个输入参数吗?
4. 系统任务$display和$write有何区别?
5. 系统任务$strobe和$monitor有何区别?
6. 编写一个函数，执行BCD(二进制码的十进制数)到7段显示码的转换。
7. 编写一个函数，将只包含十进制数字的四字符字符串转换为整数值。例如，如果 *MyBuffer* 包含字串符“4298”，将其转换为值为4298的整型数*My Int*。
8. 在Verilog HDL中除使用$readmemb和$readmemh系统任务外，还有其它读文件的方法吗?
9. 系统任务$stop和$finish的区别是什么?
10. 编写一个从存储器指定的开始和结束位置转储内容的任务。
11. 什么是标识符参数notifier?举例说明它的用法。
12. 如何向存储器的位置0到位置15装载数据？从文本文件“ram.txt”中读取十六进制值。
13. 编写一个任务，模拟上跳沿触发计数器异步清零的行为。
14. 什么语句可用于从任务返回?
15. 什么系统任务会对$time值如何输出产生影响?
16. 什么机制可用于规定被忽略的脉冲的界限?
17. 编写一个执行10位二进制向量做算术移位的函数。
18. 说明禁止语句如何用于模仿C编程语言中“continue”和“break”语句的行为。
19. 给定一个绝对的UNIX文件路径名，假定形式为*/D1/D2/D3/fileA*，编写如下函数：

    -GetDirectoryName：返回文件路径(即*/D1/D2/D3)*

    -GetBaseName：返回文件名(即*fileA*)

    假定路径名中的最大字符数为512个字符。