目录

Verilog 硬件描述语言参考手册（按英文字母顺序查找部分）

Procedural Assignment 过程赋值语句
Procedural Continuous Assignment 过程连续赋值语句
Programming Language Interface 编程语言接口
Register 寄存器
Repeat 重复执行语句
Reserved Words 关键词
Specify 指定的块延时
Specparam 延时参数
Statement 声明语句
Strength 强度
String 字符串
Task 任务
Task Enable 任务的启动
Timing control 定时控制
User Defined Primitive 用户自定义原语
While 条件循环语句
Compiler Directives 编译器指示
Standard Compiler Directives 标准的编译器指示
Non-Standard Compiler Directives 非标准编译器指示

系统任务和函数 System task and function

标准的系统任务和函数

$display, $monitor, $strobe, $write 等
$fopen 和$fclose
$readmemb 和 $readmemh
$timeformat[(Units,Precision,Suffix,MinFieldWidth)];
$printtimescale
$stop
$finish
$time,$stime,和 $realtime
$realtobits 和 $bitstoreal
$rtoi 和 $itor

随机数产生函数

1) $random
2) $dist_chi_square
3) $dist_erlang
4) $dist_exponential
5) $dist_normal
6) $dist_poisson
7) $dist_t
8) $dist_uniform

指定块内的定时检查系统任务 Specify Block Timing Checks

1) $hold
2) $nochange
3) $period
4) $recovery
5) $setup
6) $setuphold
7) $skew
8) $width

记录数值变化的系统任务 Value Change Dump Tasks

1) $dumpfile
2) $dumpvars
3) $dumpoff;
4) $dumpon;
5) $dumpall;
6) $dumplimit(FileSize);
7) $dumpflush;

非标准的系统任务和函数

$countdrivers
$list
$input
$scope and $showscopes
$key, $nokey, $log and $nolog
$reset[
$reset_count
$reset_value
$save(“FileName”);
$incsave(“FileName”);
$restart(“FileName”);
$showvars[(NetOrRegister,...)];
$getpattern(MemoryElement);
$sreadmemb (Memory, StartAddr, FinishAddr, String, ...);
$sreadmemh (Memory, StartAddr, FinishAddr, String, ...);
$scale(DelayName); {Returns realtime}

常用系统任务和函数的详细使用说明

$display 和 $write
$fopen and $fclose
$monitor 等
$readmemb 和 $readmemh
$strobe
$timeformat
随机模型 Stochastic Modelling

$q_initialize
$q_add
$q_remove
$q_full
$q_exam

定时检查　　Timing Checks

$hold
$nochange
$period
$recovery
$setup
$setuphold
$skew
$width

记录数值变化的系统任务　Value Change Dump Tasks

$dumpfile
$dumpvars
$dumpoff;
$dumpon;
$dumpall;
$dumplimit
$dumpflush;

Command Line Options　命令行的可选项

Verilog 硬件描述语言
参考手册

一. 关于 IEEE 1364 标准

本Verilog 硬件描述语言参考手册是根据IEEE 的标准“*Verilog 硬件描述语言参考手册 1364-1995*”编写的。OVI (Open Verilog International) 根据Cadence 公司推出的Verilog LRM（1.6 版）编写了Verilog 参考手册 1.0 和 2.0 版。OVI又根据以上这两个版本制定了IEEE1364-1995 Verilog标准。在推出Verilog标准前，由于Cadence公司的 Verilog-XL 仿真器广泛使用，它所提供的Verilog LRM成了事实上的语言标准。许多第三方厂商的仿真器都努力向这一已成事实的标准靠拢。

Verilog 语言标准化的目的是将现存的通过 Verilog-XL 仿真器体现的 Verilog 语言标准化。IEEE 的 Verilog 标准与事实上的标准有一些区别。因此，仿真器有可能不完全支持以下的一些功能：

- 在UDP（用户自定义原语）和模块实例中使用数组（见Instantiation说明）。
- 含参数的宏定义（见‘define）。
- ‘undef.
- IEEE标准不支持用数字表示的强度值（见编译预处理命令）。
- 有许多Verilog-XL支持的系统任务、系统函数和编译处理命令在IEEE标准中不支持。
- 若在模块中其Net或寄存类型变量只有一个驱动，IEEE标准允许在一个指定块中， 延迟路径的最终接点可以是一个寄存器或Net类型的变量。而在此标准推出之前，对最终接点的类型有着严格得多的要求（见Specify说明）。
- 指定路径的延迟表达式最多可以达到12个延迟表达式，表达式之间需用逗号隔开。而在此标准推出之前，最多只允许六个表达式（见Specify说明）。
- 在Net类型变量的定义中，标量保留字**scalared**与矢量保留字 **vectored**的位置也做了改动。原先，保留字位于矢量范围的前面。在IEEE标准中，它应位于Net类型的后面（见Net说明）。
- 在最小-典型-最大常量表达式中，对于最小、典型与最大值的相对大小并无限制。而原先最小值必须小于或等于典型值，典型值必须小于或等于最大值。
- 在IEEE标准中，表示延迟的最小-典型-最大表达式不必括在括号里。而原先，它必需括在括号里。

二. Verilog 简介

在Verilog HDL 中，我们可通过高层模块调用低层和基本元件模块，再通过线路连接（即下文中的NET）把这些具体的模块连接在一起，来描述一个极其复杂的数字逻辑电路的结构。所谓基本元件模块就是各种逻辑门和用户定义的原语模块（即下文中的UDPs）。而所谓NET实质上就是表示电路连线或总线的网络。端口连接列表用来把外部NET连接到模块的端口(即引脚）上。寄存器可以作为输入信号连接到某个具体模块的输入口。NET和寄存器的值可取逻辑值 0 ，1 ，x（不确定）和 z（高阻）。除了逻辑值外，NET还需要有一个强度（Strength）值。在开关级模型中，当NET的驱动器不止一个时，还需要使用强度值来表示。逻辑电路的行为可以用Initial和Always 的结构和连续赋值语句，并结合设计层次树上各种层次的模块直到最底层的模块(即UDP及门)来描述。

模块中每个Initial块、Always块、连续赋值、UDP 和各逻辑门结构块都是并行执行的。而Initial及Always块内的语句与软件编程语言中的语句在许多方面非常类似，这些语句根据安排好的定时控制（如时延控制）和事件控制执行。在Begin-End块内的语句按顺序执行，而在Fork-Join块中的语句则并行执行。 连续赋值语句只可用于改变NET的值。寄存器类型变量的值只能在Initial及Always块中修改。Initial及Always块可以被分解为一些特定的任务和函数。PLI （即可编程语言接口的英语缩写）是完整的Verilog语言体系的一个的组成部分，利用PLI便可如同调用系统任务和函数一样来调用C语言编写的各种函数。

编译

Verilog的原代码通常键入到计算机的一个或多个文本文件上。然后把这些文本文件交给Verilog编译器或解释器处理，编译器或解释器就会创建用于仿真和综合必需的数据文件。有时候，编译完了马上就能进行仿真，没有必要创建中间数据文件。

三. 语法总结

典型的 Verilog 模块的结构：

```
module M (P1, P2, P3, P4);
 input P1, P2;
 output [7:0] P3;
 inout P4;
 reg [7:0] R1, M1[1:1024];
 wire W1, W2, W3, W4;
 parameter C1 = “This is a string”;
   initial
        begin : 块名
           // 声明语句

        end

    always @ (触发事件)
         begin
          // 声明语句
```

```
        end
// 连续赋值语句..
assign W1 = Expression;
wire  (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;
// 模块实例引用
   COMP U1 (W3, W4);
   COMP U2 (.P1(W3), .P2(W4));

 task T1;   //任务定义
    input A1;
    inout A2;
    output A3;
      begin
        // 声明语句

      end
  endtask

  function [7:0] F1;     //函数定义
    input A1;
      begin
        // 声明语句
         F1 = 表达式;
      end
  endfunction

endmodule       //模块结束
```

声明语句：

```
  #delay
   wait (Expression)
  @(A or B or C)
  @(posedge Clk)

     Reg= Expression;
     Reg <= Expression;

   VectorReg[Bit] = Expression;
   VectorReg[MSB:LSB] = Expression;
   Memory[Address] = Expression;
   assign Reg = Expression
   deassign Reg;

   TaskEnable(...);
   disable TaskOrBlock;
  EventName;

    if (Condition)
```

```
  ...
else if (Condition)
    ...
else
   ...

case (Selection)
  Choice1 :
    ...
  Choice2, Choice3 :
    ...
  default :
    ...
 endcase

for (I=0; I<MAX; I=I+1)
     ...
    repeat (8)
     …

  while (Condition)
      ...

  forever
      …
```

上面简要语法总结可供读者快速参照，请注意其语法表示方法与本指南中其他地方不同。

四. 编写 Verilog HDL 源代码的标准

编写 Verilog HDL 源代码应按标准进行，其标准可分成两种类别。第一种是语汇代码的编写标准，标准规定了文本布局，命名和注释的约定，其目的是为了提高源代码的可读性和可维护性。第二种是综合代码的编写标准，标准规定了 Verilog 风格，其目的是为了避免常常碰到的不能综合和综合结果存在缺陷的问题，也为了在设计流程中及时发现综合中会发生的错误。

下面列出的代码编写标准可根据所选择的工具和个人的爱好自行作一些必要的改动。

语汇代码的编写标准：

- 每一个Verilog源文件中只准编写一个模块，也不要把一个模块分成几部分写在几个源文件中。
- 源文件的名字应与文件内容有关，最好一致（例 ModuleName.v）。
- 每行只写一个声明语句或说明。
- 如上面的许多例子所示，用一层层缩进的格式来写。
- 用户定义变量名的大小写应自始至终一致（例如，变量名第一个字母大写）。
- 用户定义变量名应该是有意义的，而且含有一定的有关信息。而局部名（例如循环变量）可以是简单扼要。

- 通过注释对Verilog源代码作必要的说明，当然没有必要把Verilog源代码已能说明的再注释一遍，对接口（例如模块参数、端口、任务、函数变量）作必要的注释很重要。。
- 尽可能多地使用参数和宏定义，而不要在源代码的语句中直接使用字母、数字和字符串。

可综合代码的编写标准：

- 把设计分割成较小的功能块，每一块用行为风格去设计这些块。除了设计中对速度响应要求比较临界部分外，都应避免使用门级描述。
- 应建立一个定义得很好的时钟策略，并在Verilog源代码中清晰地体现该策略（例如采用单时钟、多相位时钟、经过门产生的时钟、多时钟域等）。保证在Verilog源代码中时钟和复位信号是干净的（即不是由组合逻辑或没有考虑到的门产生的）
- 要建立一个定义得很好的测试（制造）策略，并认真编写其Verilog代码，使所有的触发器都是可复位的，使测试能通过外部管脚进行，又没有冗余的功能等。
- 每个Verilog源代码都必须遵守并符合在Always声明语句中介绍过的某一种可综合标准模板。
- 描述组合和锁存逻辑的always块，必须在always块开头的控制事件列表中列出所有的输入信号。
- 描述组合逻辑的always块一定不能有不完全赋值，也就是说所有的输出变量必须被各输入值的组合值赋值，不能有例外。
- 描述组合和锁存逻辑的always块一定不能包含反馈，也就是说在always块中已被定义为输出的寄存器变量绝对不能再在该always块中读进来作为输入信号。
- 时钟沿触发的always块必须是单时钟的，并且任何异步控制输入（通常是复位或置位信号）必须在控制事件列表中列出。
- 避免生成不想要的锁存器。在无时钟的always块中，由于有的输出变量被赋了某个信号变量值,而该信号变量没在该always块的电平敏感控制事件中列出，这会在综合中生成不想要的锁存器。
- 避免不想要的触发器。在时钟沿触发的always 块中，用非阻塞的赋值语句对寄存器类型的变量赋值，综合后就会生成触发器；或者当寄存器类型的变量在时钟沿触发的always 块中经过多次循环它的值仍保持不变，综合后也会生成触发器。
- 所有内部状态寄存器必须是可复位的，这是为了使 RTL 级和门级描述能够被复位成同一个已知的状态以便进行门级逻辑验证。（这并不适用于流水线或同步寄存器）
- 对存在无效状态的有限状态机和其他时序电路（例如，四位十进制计数器有六个无效状态），如果想要在这些无效状态下，硬件的行为也能够完全被控制，那么必须用Verilog明确地描述所有的二的N次幂种状态下的行为，当然也包括无效状态。只有这样才能综合出安全可靠的状态机。
- 一般情况下，在赋值语句中不能使用延迟，使用延迟的赋值语句是不可综合的，除了在Verilog的RTL级描述中需要解决零延迟时钟的倾斜问题是个例外。
- 不要使用整型和time型寄存器，否则将分别综合成32位和64位的总线。
- 仔细检查Verilog代码中使用动态指针（例如用指针或地址变量检索的位选择或存储单元）、循环声明或算术运算部分，因为这类代码在综合后会生成大量的门，而且很难进行优化。

五. 设计流程

用 Verilog 和综合工具设计 ASIC 或复杂 FPGA 的基本流程如下：
围绕着设计流程作多次反复是必要的，但下面的流程没有对此加以说明。而且设计流程必须根据所设计的器件的和特定的应用作必要的改动。

1 系统分析和指标的确定
2 系统划分
    2.1 顶级模块
    2.2 模块大小估计
    2.3 预布局

3 模块级设计，对每一模块：
    3.1 写 RTL 级 Verilog
    3.2 综合代码检查
    3.3 写 Verilog 测试文件
    3.4 Verilog 仿真
    3.5 写综合约束、边界条件、层次
    3.6 预综合以分析门的数量和延时

4 芯片综合
    4.1 写 Verilog 测试文件
    4.2 Verilog 仿真
    4.3 综合
    4.4 门级仿真

5 测试
    5.1 修改门级网表以便进行测试
    5.2 产生测试向量
    5.3 对可测试网表进行仿真

6 布局布线以使设计的逻辑电路能放入芯片
7 布局布线后仿真、故障覆盖仿真、定时分析

Verilog 硬件描述语言参考手册（按字母顺序查找部分）

Always 声明语句

包含一个或一个以上的声明语句(如：进程赋值语句、任务启动、条件语句、case语句和循环)，在仿真运行的全过程中，在定时控制下被反复执行。

语法

```
always
    声明语句
```

在程序中位于何处：

module- <here> -endmodule

规则：

在always块中被赋值的只能是寄存器类型的变量，如 reg, integer, real, time, realtime。每个always在仿真一开始时便开始执行，在仿真的过程中不断地执行，当执行完always块中最后一个语句后，继续从always的开头执行。

注意！

如果always块中包含有一个以上的语句，则这些语句必须放在 begin_end或 fork_join块中。如果always中没有时间控制，将会无限循环。

可综合性问题：

always声明语句是用于综合过程的最有用的Verilog声明语句之一，然而always语句经常是不可综合的。为了得到最好的综合结果，always块的Verilog 程序应严格按以下的模板来编写。

always @ (Inputs) // 所有的输入信号都必须列出，在它们之间插入逻辑关系词 or
begin
... ….. // 组合逻辑关系
end

always @(Inputs) // 所有的输入信号都必须列出，在它们之间插入逻辑关系词 or

if (Enable)
begin
……... //锁存动作
end

always @(posedge Clock) // Clock only
begin
…… // 同步动作
end

always @(posedge Clock or negedge Reset)
// Clock and Reset only
begin
if (!Reset) //测试异步复位电平是否有效
…….. // 异步动作
else
……. // 同步动作
end // 可产生触发器和组合逻辑

举例说明：

下面是一个寄存器级 always 的例子

```
always @(posedge Clock or negedge Reset)
    begin
       if (!Reset) // Asynchronous reset
              Count <= 0;
       else
       if (!Load) // Synchronous load
              Count <= Data;
```

```
      else
            Count <= Count + 1;
   End
```

下面是一个描述组合逻辑电路的always块的例子：

```
always @(A or B or C or D)
  begin
         R = {A, B, C, D}
         F = 0;
      begin : Loop
           integer I;
             for (I = 0; I < 4; I = I + 1)
                if (R[I])
                begin
                   F = I;
                   disable Loop;
                end
      end // Loop
 end
```

还请参阅：
Begin, Fork, Initial, Statement, Timing Control

Assign 连续赋值声明语句

每当表达式中 NET（即连线）或寄存器类型变量的值发生变化时，使用连续赋值声明语句就可在一个或更多的电路连接中创建事件。

语法：{either}
assign [Strength] [Delay] NetLValue = Expression,
NetLValue = Expression,
...;
NetType [Expansion] [Strength] [Range] [Delay]
NetName = Expression,
NetName = Expression,
...; {See Net}

NetLValue = {either}
NetName
NetName[ConstantExpression]
NetName[ConstantExpression: ConstantExpression]
{ NetLValue,...}

在程序中位于何处：
module-<HERE>-endmodule

规则：
两种形式的连续赋值语句效果相同。
在连续赋值声明语句之前，赋值语句左边的 NET（即连线类型的变量）必须明确声明。

注意！
连续赋值并不等同于进程连续赋值语句，虽然它们相似。确保把 assign 放在正确的地方。连续赋值语句必须放在任何initial和always块外。进程连续赋值语句可放在该语句被允许放的地方执行（在initial 、always、task、 function等块内部）。

可综合性问题：
- 综合工具不能处理连续赋值语句中的延迟和强度，在综合中被忽略。请用综合工具指定的定时约束来代替。
- 连续赋值语句将被综合成为组合逻辑电路。

提示：
用连续赋值语句去描述那些用简洁的表达式就能够很容易表达的组合逻辑电路。函数能够用来构建表示式。在描述较复杂的组合逻辑电路方面，用always块比用许多句分开的连续赋值语句更好，而且在仿真的速度更快一些。当Verilog需要电路连线时，可用连续赋值语句把寄存器的值传送到电路连线上（即NET上）。例如，把一个initial块中产生的测试激励信号加到一个实例模块的输入输出端口。

举例说明：
```
wire cout, cin;
wire [31:0] sum, a, b;
  assign {cout, sum} = a + b + cin;

wire enable;
reg [7:0] data;
wire [7:0]  #(3,4)  f = enable ? data : 8'bz;
```

还请参照：
Net、Force、进程连续赋值语句

Begin 声明语句

用于把多个声明语句组合起来成为一个语句,而其中每个声明语句的执行是按顺序。Verilog语法经常严格要求只有一个声明语句，例如always就是这样。如果always需要有多个声明语句，那么这些声明语句必须被包含在一个begin-end块中。

语法
```
  begin [: Label
       [ Declarations...]]
      Statements...
  End

Declaration = {either} Register Parameter Event
```

在程序中位于何处：
请参照 Statement 的说明

规则：
begin-end块必须包含至少一个声明语句。声明语句在begin-end块中被顺序执行。定时控制

是相对于前一声明语句的。当最后的声明语句执行完毕后，begin-end块便结束。Begin-end和fork-join块可以自我嵌套或互相嵌套。如果begin-end块包含局部声明，则它必须被命名（即必须有一个标识）。如果要禁止（disable）某个begin-end块，那么被禁止的begin-end块必须有名字。

注意！
Verilog LRM 允许begin-end块在仿真时被交替执行。这就是说如果begin-end块包含两个相邻且其间没有时间控制的声明语句时，仿真器仍有可能在同一时刻在这两个语句之间执行另一个进程的部分语句（例如另一个always块中的语句）。这就是Verilog 语言如果不加约束的话，便不能与硬件有确定对应关系的原因。

提示:
甚至在并不需要局部声明，也不想禁止Begin-end块时，也可以对该Begin-end块加标识命名，以提高其可读性。给不用在别处的寄存器作局部声明，能使声明的意图变得清楚。

举例说明：

```
initial
   begin : GenerateInputs
           integer I;
           for (I = 0; I < 8; I = I + 1)
           #Period {A, B, C} = I;
   end

initial
      begin
          Load = 0; // Time 0
          Enable = 0;
          Reset = 0;
          #10 Reset = 1; // Time 10
          #25 Enable = 1; // Time 35
          #100 Load = 1; // Time 135
      end
```

还请参照：

Fork, Disable, Statement.

Case 声明语句:

如果 case 控制表达式与标号分支表达式相等，则执行该分支的声明语句。

语法：

```
CaseKeyword ( Expression)
Expression,... : Statement {Expression may be variable}
Expression,... : Statement
                ... {Any number of cases}
      [default [:] Statement] {Need not be at the end}
endcase

CaseKeyword = {either} case casex casez
```

在程序中位于何处：

请参照 Statement 的说明

规则：

- 不确定值（Xs）和高阻值（Zs）在casex声明语句中，以及（Zs ）在casez声明语句的表达式匹配中都意味着“不必考虑”。
- 在case语句中最多只允许有一个default项。当没有一个分支标号表达式能与case表达式的值相等时，便执行default项。（标号是位于冒号左边的一个表达式或用逗号隔开的几个表达式，标号也可以是保留字default,在其后面可以跟冒号也可以不跟冒号。）
- 如果某标号是用逗号隔开的两个或两个以上表达式，只要其中任何一个表达式与case表达式的值相等时，就可执行该标号的操作。
- 如果没有一个标号表达式与case表达式的值相等，又没有default声明语句，该case声明语句没有任何作用。

注意：

- 如果在标号分支中有一个以上的声明语句，这些声明语句必须放在一个begin-end或fork-join块中。
- 只有第一个与case表达式的值相等的标号分支才被执行。Case语句的标号并不一定是互斥的，所以当错误地重复使用相同的标号时，Verilog编译器不会提示出错。
- Casex 或 casez 声明语句的语法是用保留字 endcase 作为结束，而不是用 endcasex 或 endcasez 来结束。
- 在 casex 声明语句的表达式中的 X（不定值）或Z（高阻值）可以和任何值相等，casez 中的 Z 也是如此。这有可能会给仿真结果带来混乱。

可综合性问题：

Case 声明语句中的赋值语句通常被综合成多路器。如果变量（如寄存器或 Net 类型）被用作 Case 语句的标号，它就会被综合成优先编码器（priority encoders）。

在一个无时钟触发的 always 块中，如有不完整的赋值（即对某些输入信号的变化其输出仍保持不变，未能及时赋值），它将被综合成透明锁存器。

在一个有时钟触发的 always 块中，如有不完整的赋值，它将被综合成循环移位寄存器。

提示：

- 为了使仿真能顺利进行，常常用 default 作为 case 声明的最后一个分支，以控制无法与标号匹配的 case 变量。
- 通常情况下用casez比用casex更好一些，因为X的存在可能会导致仿真出现令人误解和混乱的结果。

- 在casex 和 casez声明的标号中用“？”来代替“Z”比较好，因为这样做比较清楚，是一个无关项，而不是一个高阻项。

举例说明：

```
case (Address)
     0 : A <= 1;       // Select a single Address value
     1 : begin         // Execute more than one statement
          A <= 1;
          B <= 1;
         end
 2, 3, 4 :   C <= 1;   // Pick out several Address values
default :              // Mop up the rest
        $display(“Illegal Address value %h in %m at %t”, Address, $realtime);
endcase

casex (Instruction)
  8’b000xxxxx : Valid <= 1;
  8’b1xxxxxxx : Neg <= 1;
  default
      begin
         Valid <= 0;
          Neg <= 0;
      end
endcase

casez ({A, B, C, D, E[3:0]})
    8’b1??????? : Op <= 2’b00;
    8’b010????? : Op <= 2’b01;
    8’b001???00 : Op <= 2’b10;
          default : Op <= 2’bxx;
endcase
```

还请参照：
If

Comment　注释语句

注释应该位于 Verillog 源代码文件中。

语法

单行注释
//
多行注释
/* … */

在程序中位于何处：
可以放在源代码的几乎任何地方，但是注意不能把运算符、数字、字符串、变量名和关键字分开。

规则：
单行注释以两个斜杠符开始，结束于该行的末尾 。
多行注释以“/*” 符开始，中间可能有多行，结束于“*/” 符。
多行注释不能嵌套，但是，在多行注释中可以有单行注释，但在这儿它没有别的特殊含义。

注意：
/* ... /* ... */ ... */- 这样的注释会出现语法错误，要注意注释符的匹配。

提示：
建议在源代码文件中自始至终用单行注释。只有在必需注释一大段的地方才用多行注释，例如在代码的开发和调试阶段，常需要详细地注释。

举例说明：
```
// This is a comment
/*
    So is this - across three lines
*/
module ALU /* 8-bit ALU */ (A, B, Opcode, F);
```

请参照：
Coding Standard 编码标准

Defparam 定义参数声明语句

编译时可重新定义参数值。如果是分层次命名的参数，可以在该设计层次内或外的任何地方重新定义参数。

语法：
```
Defparam ParameterName = Constant Expression
ParameterName = ConstantExpression,

...;
```

在程序中位于何处：
module-<HERE>-endmodule

可综合性问题：
一般情况下是不可综合的。

提示：
不要使用 defparam 声明语句！该声明语句过去常用于布线后的时延参数反标中，但现在时延参数反标一般用指定模块和编程语言接口（PLI）来做。在模块的实例引用时可用“#”号后跟参数的语法来重新定义参数。

举例说明：
```
‘timescale 1ns / 1ps
module LayoutDelays;
   defparam Design.U1.T_f = 2.7;
   defparam Design.U2.T_f = 3.1;
    ...
```

```
endmodule
module Design (...);
   ...
   and_gate U1 (f, a, b);
   and_gate U2 (f, a, b);
   ...
endmodule

module and_gate (f, a, b);
  output f;
  nput a, b;
  parameter T_f = 2;
    and #(T_f) (f,a,b);
endmodule
```

还请参照：
Name, Instantiation, Parameter

Delay　时延

可以为 UDP 和门的实例指定时延，也可以为连续赋值语句和线路连接指定时延。时延是在网表中线路连接和元件传输时延的模型。

语法：
{either}
DelayValue
#(DelayValue[, DelayValue[, DelayValue]]) {Rise,Fall,Turn-Off}
DelayValue = {either}
UnsignedNumber
ParameterName
ConstantMinTypMaxExpression

在程序中位于何处：
请参照：连续赋值语句、实例引用、线路连接。

规则：
- 如果只给出一个延迟，则它既表示上升延迟也表示下降延迟 （即从0转变到1或从1转变到0的时延），并且还表示关闭延迟（如果电路中有这样开关）。
- 如果给出两个延迟值，则第一个表示上升延迟，第二个表示下降延迟，除了tranif0, tranif1, rtranif0和rtranif1外，第一个值也可表示接通延迟，第二个表示关闭延迟。
- 如果给出三个延迟，第三个延迟表示关闭延迟（转变到高阻），除了三态电路外，第三个延迟表示电荷衰减时间。
- 延迟到X表示最小的指定延迟。
- 对于向量，从非零到零的转变被看作下降，转变到高阻被看作关闭，其余的变化被看作是上升。

注意！
许多工具要求MinTypMax延迟表达式必须用括号括起来。例如 #（1：2：3）是合法的，而 #1：2：3是非法的。

可综合性问题：
一般综合工具不考虑延迟。综合后网表中的延迟是由综合工具的命令项强制生成的，如在综合工具中可设置本次设计综合生成的门级电路所允许的最高时钟频率。

提示：
指定块的延迟（即线路路径延迟）通常是一种更加精确的延迟建模方法，可提供延迟计算机制和布线后反标信息。

还请参照：
线路连接, 实例引用, 连续赋值, Specify, 定时控制 等声明语句

Disable　禁止

在运行激活的任务或命名的块时 Disable 能使在所在块执行完毕以前，终止该块的执行。

语法：
disable　BlockOrTaskName;

在程序中位于何处：
请参照 Statement.

规则：
- 禁止（disable）命名块（即定义了名称的 begin_end 或 fork_joink 块）或任务便禁止了所有由该块或该任务激活的任务，直达该块或该任务层次树的底层。继续执行禁止（块或任务）语句后的声明语句。
- 命名块或任务可以通过其内部的禁止声明语句实现自我禁止
- 当一个任务被禁止时，以下内容未被指定：任何一个输出值或输入输出值；尚未起作用的非阻塞赋值语句、赋值和强制声明语句所预定的事件。
- 函数不能被禁止。

注意！
如果一个任务被自我禁止，这跟任务返回不一样，因为输出未定义。

可综合性问题：
只有当命名块或任务自我禁止时，禁止才是可综合的，一般情况下是不可综合的。

提示：
用禁止作为一种及早跳出任务的方法，用来跳出循环或继续下一步循环。

举例说明：

```
begin :  Break      //命名 Break 块
     forever
      begin :  Continue     //命名 Continue 块

           ...
         disable Continue;     // Continue with next iteration
            ...
```

```
         disable Break;      // Exit the forever loop
             ...
      end    // Continue
end          // Break
```

Errors　错误

下面列出的是编写 Verilog 源代码时最常犯的错误。前面的五个错误大约占所有错误的 50%。

最容易犯的五大错误：

- 进程赋值语句的左侧变量没有声明为寄存器类型。
- Begin – end 声明语句忘了配套。
- 写二进制数时忘了标明数基（即‘b）。这样，在编译时会把它们当作十进制数来处理。
- 编译引导语句用了错误的撇号，应该用向后的撇号也就是用表示重音的撇号；而表示数基的撇号，应该是普通的撇号，也就是反向的逗号。
- 在声明语句的末尾忘了写上分号。

其他常犯的错误：

- 在定义任务或函数时，试图在任务或函数名后用括号来定义变量。
- 在调试时，忘了在测试文件中引用实例模块。
- 使用进程连续赋值语句而没有使用连续赋值语句（即赋值语句用错了地方）
- 把保留字作为标识符（例如用 xor 做标识符）。
- Always 块忘了声明定时控制（导致无休止的循环）。
- 在事件控制列表中错误地使用了逻辑或操作符（即||）而没有使用或保留字or，例如把 always @(a or b),写成了always @(a||b)。
- 用缺省定义的 wire 类型变量来做矢量端口的连线。
- 模块实例引用时端口的连接次序搞错。
- 在嵌套的 if-else 语句中 begin-end 配套错误。
- 错误地使用等号。“ = ”用于赋值，“ = = ”用于作数值比较，“ = = = ”用于需要对 0、1、X、Z 这四种逻辑状态作准确比较的场合 。

Event　事件

在行为模型中 Events 可以用来描述通信和同步。

语法：

```
event Name ,...; {Declare the event}
-> EventName; {Trigger the event}
```

语法

事件名，...;　（事件声明）

->事件名　　（触发事件）

在程序中位于何处：

请参照为 -> 所作的声明语句。

事件声明语句可以放在下面这些地方：

```
module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction
```

规则：
事件没有值，也没有延迟，它们仅被事件触发声明所触发，由沿敏感定时控制启动检测。

可综合性问题：
通常是不可综合的。

提示：
在测试文件和系统级模块中，命名事件可用于同一个模块的不同 always 块间或不同模块（用层次名）的 always 块间传递信息。

举例说明：

```
event StartClock, StopClock;
always
   fork
      begin : ClockGenerator
              Clock = 0;
              @StartClock
                forever
                    #HalfPeriod Clock = !Clock;
      end
      @StopClock disable ClockGenerator;
   join


initial
    begin : stimulus
              ...
       -> StartClock;
              ...
       -> StopClock;
              ...
       -> StartClock;
              ...
       4-> StopClock;
   end
```

还请参阅：
定时控制

Expression 表达式

表达式可以通过一系列的操作符、变量名、数字以及次级表达式来算出一个值。其中常量表达式是一种其值可在编译过程中计算出来的表达式。标量表达式的值是一比特二进制数。时间延迟可以用最小-典型-最大（即 MinTypMax）表达式来表示。

语法：

Expression = {either}	表达式 = {以下任取其一}
Primary	基本表达式
Operator Primary {unary operator}	运算符 基本表达式 {单目运算符}
Expression Operator Expression {binary operator}	表达式 运算符 表达式 {双目运算符}
Expression ? Expression : Expression	表达式 ？ 表达式 ： 表达式
String	字符串

Primary = {either}	基本表达式 = {以下任取其一}
Number	数字
Name {of parameter, net, or register}	变量名 {参数，网络，或者寄存器的}
Name[Expression] {bit select}	变量名[表达式] {位选择}
Name[Expression: Expression] {part select}	变量名[表达式：表达式] {部分位选择}
MemoryName[Expression]	存储器名[表达式]
{ Expression,...} {concatenation}	{表达式，……} {位拼接}
{ Expression{ Expression,...}} {replication}	{表达式{表达式,……}} {复制}
FunctionCall	函数调用
(MinTypMaxExpression)	（MinTypMax 表达式）
{MinTypMax expressions are used for delays}	{ MinTypMax 表达式用于延迟}
MinTypMaxExpression = {either}	MinTypMax 表达式 = {任取其一}
Expression	表达式
Expression: Expression: Expression	表达式：表达式：表达式

规则：

- 只有矢量类型的 NET 和寄存器、整数及时间类型变量才允许选取某位及某些位。
- 某些位的选取必须将高位列在冒号的左侧，低位列在右侧。（最高位是在 NET 或寄存器类型声明中位于冒号左边的数值。）
- 某位或某些位的选取若其中包含 X 或 Z 的位，或超出位的定义范围，在编译时可能会也可能不会被认定为是错误的。如果不被认定是错的，编译器会给出一个值为 X 的表达式。
- 没有为存储器建立某位或某些位选取的机制。
- 当整型常量在表达式中作为操作数时，未标明进制的有符号数(例如-5)与标明进制的有符号数(例如-‘d5)是有所区别的。前者被视作一个有符号数，而后者被视作一个无符号数。

注意！

许多工具要求在常量 MinTypMax 表达式中必须指定最小、典型和最大延迟值 (例如：min<=typ<=max)。

举例说明：

```
A + B
!A
(A && B) || C
A[7:0]
B[1]
-4'd12/3                          // 是一个很大的正数
"Hello" != "Goodbye"          // 此表达式为真（1）
$realtobits®;                     //  系统函数调用
{A, B, C[1:6]}                    // 位拼接（8 位）
1:2:3                             //最小-典型-最大表达式
```

还请参阅：

延迟，函数调用，变量名，数字，操作数。

For 循环声明语句

一般用途的循环语句。允许一条或更多的语句能被重复地执行。

语法：

```
for ( RegAssignment;  {initial assignment}
      Expression;     {loop condition}
      RegAssignment)  {iteration assignment}
    Statement
```

```
RegAssignment = RegisterLValue = Expression  寄存器赋值 = 寄存器值 = 表达式
RegisterLValue = {either}                    寄存器值 = {任取其一}
RegisterName                                 寄存器名
RegisterName[ Expression]                    寄存器名[表达式]
RegisterName[ ConstantExpression: ConstantExpression]
                                             寄存器名[常量表达式：常量表达式]
Memory[ Expression]                          存储器[表达式]
{ RegisterLValue,...}                        {寄存器值，.....}
```

在程序中位于何处：

请参照 Statement 说明。

规则：

当 for 循环开始执行时，循环计数变量已赋于初始值。在每一次循环执行之前（包括第一次），都必须首先检查表达式的值；如果它为假（即为 0、X、或 Z），则立刻退出循环。而在每一次循环重复执行之后，都要对迭代次数寄存器重新赋值。

注意！
不要使用位宽小的 reg 类型变量作为循环变量。在测试存有负数值的寄存器变量时要格外注意。由于加减操作是可替换的，并且 reg 类型变量被看作是无符号数，所以循环表达式可能永远不会为假，从而导致循环无限止地进行。

```
reg [2:0] i;              //i 始终界于 0 至 7 之间
...
for ( i= 0; i<8; i=i+1 )     //循环永远不会停止
...
for ( i=-4; i<0; i=i+1 )    //  循环不可能执行
...;
```
在以上这些情况中，应将循环变量 i 定义为整型。

可综合性问题：
如果循环的边界是固定的，那么在综合时该循环语句被认为是重复的硬件结构。

举例说明：
```
V = 0;
for ( I = 0; I < 4; I = I + 1 )
begin
F[I] = A[I] & B[3-I];    //   四个独立的与门
V = V ^ A[I];            //  四个级连的异或门
end
```

还请参阅：
Forever, Repeat, While 语句的说明。

Force 强迫赋值

类似于进程连续赋值语句，可对 Net 和寄存器类型变量实行强制赋值。常用于调试。

语法：

{either}	{任取其一}
force NetLValue = Expression ;	force 网络参数值=表达式；
force RegisterLValue = Expression ;	force 寄存器值=表达式；
{either}	{任取其一}
release NetLValue;	release 网络参数值；
release RegisterLValue;	release 寄存器值
NetLValue = {either}	网络参数值={任取其一}
NetName	网络变量名

{NetName,...}	{网络变量名}
RegisterLValue = {either}	寄存器值={任取其一}
RegisterName	寄存器变量名
{RegisterName,...}	{寄存器变量名}

在程序中位于何处：
请参照声明语句

规则：

- 不能对网络变量或寄存器变量的某位或某些位实行强制赋值或释放。 **force**具有比进程连续赋值声明语句更高的优先级。force 将会一直发挥作用直到另一个force对同一Net变量或寄存器变量执行强迫赋值，或者直到这个Net变量或寄存器变量被释放。
- 当作用在某一寄存器上的 force 被释放，寄存器并无必要立刻改变其值。如果此时没有进程连续赋值对这个寄存器赋值，则强制赋入的值会一直保留到下一个进程赋值语句的执行。
- 当作用在某个 Net 变量上的 force 被释放，该 Net 变量的值将由它的驱动决定，其值有可能会立刻更新。

可综合性问题：
强迫赋值语句是不可综合的。

提示：
强迫赋值常用于测试文件的编写，调试时常需要强制对某些变量赋值。不能用于模块的行为建模（此时应使用连续赋值语句）。

举例说明：

```
force   f = a && b;
. . .
release f;
```

还请参照：
进程连续赋值语句。

Forever 声明语句

使一个或一个以上语句无限循环地执行。

语法：

```
forever     Statement
```

在程序中位于何处：

请参阅 Statement 说明。

注意！
forever 循环应包括定时控制或能够使其自身停止循环，否则循环将无限进行下去。

可综合性问题：
一般情况下是不可综合的。如果 forever 循环被@(posedge Clock)形式的时间控制打断，则是可综合的。

提示：
forever 在测试模块中描述时钟时很有用。常用 disable 来跳出循环。

举例说明：

```
initial
    begin : Clocking
        Clock = 0;
        forever
        #10 Clock = !Clock;
    end

initial
    begin : Stimulus
        . . .
        disable Clocking;   // 停止时钟
    end
```

还请参阅：
For, Repeat, While, Disable 说明。

Fork 声明语句

可将多个语句集合在一个块中，以使它们能被并发地执行。

语法：

```
fork [ : Label                          fork   [:块名
[ Declarations...]]                             [块内声明语句……]]
Statements...                            语句……
join                                    join
Declaration = {either}                  块内声明语句={任选其一}
Register                                寄存器变量
Parameter                               参数
```

Event 事件

在程序中位于何处：
请参照 Statement 说明。

规则：
fork-join 块必须至少包括一条语句。Fork-join 块里的语句是并发执行的，因此 Fork-join 块内语句的顺序是无所谓的。时间控制是相对于块的开始时刻的。 当 fork-join 块里所有的语句执行完毕后，块也就执行完毕了。Begin-end 和 fork-join 块可以自身嵌套或互相嵌套。
如果想在某 fork-join 块内包含块内局部声明语句，那么必须对该块命名（即该块必须有一个标识符号）。如果想要禁止某 fork-join 块的运行，则该块必须已被命名。

可综合性问题：
fork 语句不可综合。

注意！
Fork-join 语句在描述并发形式的行为时很有用。

举例说明：
```
initial
    fork : stimulus
        #20 Data = 8'hae;
        #40 Data = 8'hxx; // 本句最后执行
        Reset = 0;        // 本句最先执行
        #10 Reset = 1;
    join                  // 在第 40 个时间单位时结束
```

还请参照：
Begin, Disable, Statement

Function 函数

用于把多个语句组合在一起，来定义新的数学或逻辑函数。函数是在模块内部定义的，并且通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。

语法：

function [RangeOrType] FunctionName;	function [返回值的类型或范围] 函数名；
Declarations...	端口声明...
Statement	语句
endfunction	endfunction

RangeOrType = {either}　　返回值的类型或范围={任取其一}
integer　　整数、
real　　实数
time　　时间
realtime
Range
Range = [ConstantExpression: ConstantExpression]　范围=[常量表达式：常量表达式]
Declaration = {either}　　端口声明={任取其一}
input [Range] Name,...;　　input [范围] 变量名，……;
Register
Parameter
Event

在程序中位于何处：
module　-<HERE>- endmodule

规则：
- 函数必须至少含有一个输入变量。它不能有任何输出或输入/输出双向变量。
- 函数不能包含时间控制语句（如延迟#，事件控制@ 或等待 wait）。
- 函数是通过对函数名赋值的途径返回其值的，（就好比它是一个寄存器）。
- 函数不能启动任务。
- 函数不能被禁用。

注意！
- 函数的输入变量不能象模块的端口那样列在函数名后的括弧里；在声明输入时把这些输入端口列出即可。
- 如果函数包含一条以上的语句，这些语句必须包含在 begin-end 或 fork-join 块中。

可综合性问题：
函数的每一次调用都被综合为一个独立的组合逻辑电路块。

举例说明：
```
function [7:0] ReverseBits;
    input [7:0] Byte;
    integer i;
    begin
        for (i = 0; i < 8; i = i + 1)
        ReverseBits[7-i] = Byte[i];
    end
endfunction
```

还请参阅：
Function Call, Task 语句的说明。

Function Call　函数调用

函数的调用可返回一个供表达式使用的值。

语法：
FunctionName (Expression，...);　　函数名（表达式，……）；

在程序中位于何处：
请参照 Expression 说明。

规则：
函数必须至少含有一个输入变量，所以函数调用时总是至少含有一个表达式。

可综合性问题：
函数的每一次调用在综合后都会生成一个独立的组合逻辑电路块。

举例说明：
Byte = ReverseBits (Byte);

还请参阅：
Function, Expression, Task Enable 说明。

Gate　门

Verilog 已有一些建立好的逻辑门和开关的模型。在所设计的模块中能通过实例引用这些门与开关模型，从而对模块进行结构化的描述。

逻辑门：
and (Output, Input,...)
nand (Output, Input,...)
or (Output, Input,...)
nor (Output, Input,...)
xor (Output, Input,...)
xnor (Output, Input,...)

缓冲器与非门：
buf (Output,..., Input)
not (Output,..., Input)

三态门：

bufif0 (Output, Input, Enable)
bufif1 (Output, Input, Enable)
notif0 (Output, Input, Enable)
notif1 (Output, Input, Enable)

MOS 开关：
nmos (Output, Input, Enable)
pmos (Output, Input, Enable)
rnmos (Output, Input, Enable)
rpmos (Output, Input, Enable)

CMOS 开关：
cmos (Output, Input, NEnable, PEnable)
rcmos (Output, Input, NEnable, PEnable)

双向开关：
tran (Inout1, Inout2)
rtran (Inout1, Inout2)

双向可控开关：
tranif0 (Inout1, Inout2, Control)
tranif1 (Inout1, Inout2, Control)
rtarnif0 (Inout1, Inout2, Control)
rtranif1 (Inout1, Inout2, Control)

上拉源与下拉源：
pullup (Output)
pulldown (Output)

真值表

在这些表中，逻辑值 L 与 H 代表部分未知值。L 表示 0 或 Z，H 表示 1 或 Z。

and	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

nand	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

or	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

nor	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

xor	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

xnor	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

buf	
Input	Output
0	0
1	1
X	X
Z	X

Not	
Input	Output
0	1
1	0
X	X
Z	X

缓冲门、非门都可以有多个输出，但这些输出值都是相同的。

Bufif0		Enable			
		0	1	X	Z
DATA	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

Bufif1		Enable			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

notif0		Enable			
		0	1	X	Z
DATA	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

notif1		Enable			
		0	1	X	Z
DATA	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

PMOS RPMOS		Control			
		0	1	X	Z
DATA	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	Z	Z	Z	Z

NMOS RNMOS		Control			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	Z	Z	Z

```
cmos (W, Datain, NControl, PControl);
```
等价于:
```
nmos (W, Datain, NControl);
pmos (W, Datain, PControl);
```

规则：

- 当 nmos, pmos, cmos, tran, tranif0 和 tranif1 类型的开关开启时，信号从输入传到输出并不改变其强度。

- 当有电阻的开关，如 rnmos, rpmos, rcmos, rtran, rtranif0 和 rtranif1 类型的开关，开启时，信号从输入传到输出会按下表减小其强度：

Strength	减至
supply	pull
strong	pull
pull	weak
large	medium
weak	medium
medium	small
small	small
highz	highz

还请参阅：
UDP（用户自定义原语），Instantiation(实例引用)语句的说明。

IF 条件声明语句

根据条件表达式的逻辑值（真/假），执行两条/块语句中的一条/块。

语法：
if (Expression) if(表达式)
Statement 语句
[else [else
Statement] 语句]

在程序中放在何处：
请参阅 Statement 说明.

规则：
当表达式的值为非零时被认为是真，当值为零、X 或 Z 时被认为是假。

注意：
- 如果在 **if** 或 **else** 分支中有超过一条的语句需要执行，则必须用 begin-end 或 fork-join 将其包括。
- 在使用嵌套的 if-else 语句时，当 else 分支省略时，要特别注意。Else 只与离它最近的前面的那个 if 相关联。 Verilog 编译器不能判别源代码中省略的 else 分支。

可综合性问题：
- If 声明语句中的赋值语句通常被综合为多路器。在无时钟的 **always** 块中，当输入变化

时而输出仍能保持不变的那些赋值语句，将被综合为透明锁存器，而在有时钟的 **always** 块中，它们则被综合为循环锁存器。

- 在某些情况下，嵌套的 if 语句会被综合为多层的逻辑。用 case 语句可以避免出现这种情况。

提示：

如果对某些条件需要先进行测试，在这种情况下应选用嵌套的 if-else 语句。如果所有的条件优先权一致，则应选用 **case** 语句。

举例说明：

```
if (C1 && C2)
    begin
        V = !V;
        W = 0;
            if (!C3)
                X = A;
                else    if (!C4)
                            X = B;
            else
                X = C;
    end
```

还请参阅：

Case, Operators 说明

Initial 声明语句

在仿真一开始就执行并只执行一次的声明语句，可执行只包含一条语句或多条语句组成的块。

语法：

```
initial
    Statement
```

在程序中放在何处：

module-<HERE>-endmodule

可综合性问题：

initial 语句是不可综合的。

注意！
包含多个语句的 **initial** 块需要用 begin-end 或 fork-join 块将这些语句合成一块。

提示：
在仿真测试文件中可使用 **initial** 语句来描述激励。

举例说明：
下面的例子给出如何使用 **initial** 在测试文件中产生矢量：

```
reg Clock, Enable, Load, Reset;
reg [7:0] Data;
parameter HalfPeriod = 5;
   initial
      begin : ClockGenerator
         Clock = 0;
         forever
            #(HalfPeriod) Clock = !Clock;
      end
   initial
      begin
         Load = 0;
         Enable = 0;
         Reset = 0;
         #20     Reset = 1;
         #100    Enable = 1;
         #100    Data = 8'haa;
                 Load = 1;
         #10 Load = 0;
         #500    disable ClockGenerator;     //停止时钟的产生。
      end
```

还请参阅：
Always 语句说明

Instantiation 实例引用

实例（instance）是模块、UDP 或门的唯一拷贝。 通过实例的引用可以生成设计的各个层次。设计的行为也能通过引用 UDP、门和其他的模块的实例，并用电路连线（Net）将它们连接起来，从结构上加以描述。
述。

语法：

{either}	{任取其一}
ModuleName	
[#(Expression,...)] ModuleInstance,...;	模块名 [#（表达式，…）] 实例模块，……;
UDPOrGateName [Strength]	
[Delay] PrimitiveInstance,...;	UDP 或门名 [强度] [延迟] 原始实例，…;
ModuleInstance =	实例模块=
InstanceName [Range] ([PortConnections])	实例名[范围]（[端口连线]）
PrimitiveInstance =	原始实例=
[InstanceName [Range]] (Expression,...)	[实例名[范围]] （表达式，……）
Range =	
[ConstantExpression: ConstantExpression]	范围 = [常量表达式：常量表达式]
PortConnections = {either}	端口连线={任取其一}
[Expression] ,... {ordered connection}	［表达式］，……{有顺序的连线}
・PortName([Expression]) ,... {named connection}	・端口名([表达式]),……{指定连线}

在程序中位于何处：

module-<HERE>-endmodule

规则：

- 命名的端口连线只能用于模块实例。
- 如果给出端口的连线次序列表，则在引用实例时其端口必须按次序与模块或门的端口一一对应。
- 如果给出命名的端口连线列表时，则在引用实例时其端口顺序是无关紧要的，但其端口的名字必须与模块的端口名字一致。
- 如果给出端口的连线次序列表，在引用实例时，其端口列表中若有两个邻近的逗号，则会因为没有表达式而导致相应端口未连线。如果给出命名的端口连线列表时，在引用实例时，其端口列表中若没有某端口的名字或虽有端口的名字但在括号内没有表达式，也会在导致该端口未连线。
- 任何表达式都可用来与输入端口相连，但输出端口只能与 Net（线路）、一位或多位的连线或这些位的拼接线相连。输入表达式生成隐含的连续赋值。
- 如果在模块实例定义时给出了范围，其含义是定义了一个含有同种的多个子实例的实例模块。如果端口表达式的位长与定义的实例模块相应端口位长（即多个同种 UDP 或门端口位数的总和）一致时，整个表达式都将与每个子实例的端口相连。如果位长不一致，太多或太少，都会出错误。
- “#”符号有两种不同的用途。它既可用来强制修正实例模块中的一个或多个参量，也可用于为 UDP 或门实例指定延迟。对于实例模块，“#”符号后的第一个表达式替代模块中声明的第一个参量，第二个表达式替代模块中声明的第二个参量，依次类推。
- 实例引用 pullup、pulldown、tran 和 rtran 这些类型的门时不允许有延迟。
- 对于 nmos、pmos、cmos、rnmos、rpmos、rcmos、tran,、rtran,、tranif0、tranif1,、rtranif0 和 rtranif1 这些类型的开关不能定义强度。

注意！

- 在按顺序的端口的列表中很容易不小心将两个端口的次序弄混。若这些端口的位宽和方

向相同，不会报告出错，只有在仿真出现错误结果后，才能找到。这类错误往往很难发现。使用命名的端口连线能避免实例模块引用中出现这类问题。

- 多个模块、UDP 或门的成组实例引用的语法是最近才加入 Verilog 语言的标准中，目前还没有工具支持这语法。

可综合性问题：

UDP 和开关的实例引用一般是不能综合的。

提示：

使用命名的端口连接方式以提高程序的可读性并减少发生错误的可能性（见前文）。

端口表达式只使用位、部分位和位拼接的变量名。如果需要，则应尽量使用独立的连续赋值语句，对实例模块引入信号。

举例说明：

UDP 实例引用：

```
Nand2 (weak1,pull0) #(3,4) (F, A, B);
```

模块实例引用：

```
Counter  U123 (.Clock(Clk), .Reset(Rst), .Count(Q));
```

在下面的两个例子中 QB 端口没有连接：

```
DFF Ff1 (.Clk(Clk), .D(D), .Q(Q), .QB());
DFF  Ff2  (Q,, Clk, D);
```

下面是在端口连线表中使用门表达式的例子：

```
nor (F, A&&B, C) // 不要这样使用
```

下面是一个多实例模块引用的例子。

```
module Tristate8 (out, in, ena);
output [7:0] out;
input [7:0] in;
input ena;

bufif1 U1[7:0] (out, in, ena);
/* 上面的一条语句等同下面 8 条语句
bufif1 U1_7 (out[7], in[7], ena);
bufif1 U1_6 (out[6], in[6], ena);
bufif1 U1_5 (out[5], in[5], ena);
bufif1 U1_4 (out[4], in[4], ena);
bufif1 U1_3 (out[3], in[3], ena);
bufif1 U1_2 (out[2], in[2], ena);
bufif1 U1_1 (out[1], in[1], ena);
bufif1 U1_0 (out[0], in[0], ena);
*/
```

endmodule

还请参阅：
Module, User Defined Primitive, Gate, Port 语句的说明。

Module 模块定义

在 Verilog 语言中，模块是层次的基本单元。模块中包括声明语句、功能描述和引用一些现存的硬件部件。有些模块只用来声明可被别的模块调用的参量，任务和函数。在这类模块中没有任何 **initial** 块、**always** 块、连续赋值语句和实例引用，因而实际上不存在相应的硬件元件与之对应 。

语法：

{either}	{任取其一}
module ModuleName [(Port,...)];	module 模块名[（端口，……）];
ModuleItems...	模块条款
endmodule	endmodule
macromodule ModuleName [(Port,...)];	macromodule 模块名[（端口,…）];
ModuleItems...	模块条款
endmodule	endmodule
ModuleItem = {either}	模块条款={任取其一}
Declaration	声明
Defparam	参数定义
ContinuousAssignment	连续任务
Instance	实例引用
Specify	详细说明块
Initial	初始化块
Always	总是执行块
Declaration = {either}	声明={任取其一}
Port	端口
Net	网络
Register	寄存器
Parameter	参量
Event	事件
Task	任务
Function	函数

在程序中位于何处：

在其它模块或 UDP 外。

规则：

- 几个模块或几个 UDP（或它们的混合）可以在一个文件中进行描述。（事实上，一个模块也可以分开在两个或更多的文件中描述，但不推荐这种做法）
- 模块也可使用关键字 **macromodule** 来定义。其语法与用关键字 **module** 来定义模块是完全一样的。
- Verilog 编译器在编译宏模块时与编译一般模块时有所不同，比如不必为宏模块实例创建层次。这样，从仿真速度或存储介质的开销两方面来说，宏模块的编译更有效率。为了达到这个目的，宏模块的编译可能受制于实现时的某些特殊条件的限制。如果遇到这种情况，宏模块将被按一般模块编译。

注意！

模块与宏摸块都以关键字 endmodule 作为结束标志。

可综合性问题：

- 每一个模块都被综合为一个独立的分层块，虽然有些工具的缺省配置规定把层次展平（为单层），但仍允许用户对综合后生成的网表层次进行控制，。
- 不是所有的工具都支持宏模块的综合。

提示：

尽量使每一个文件只包含一个模块。在大型设计中，这样做易于源代码的维护。

举例说明：

```
macromodule nand2 (f, a, b);
    output f;
    input a, b;
        nand (f, a, b);
endmodule

module PYTHAGORAS (X, Y, Z);
    input [63:0] X, Y;
    output [63:0] Z;
    parameter Epsilon = 1.0E-6;
    real RX, RY, X2Y2, A, B;
        always @(X or Y)
            begin
                RX = $bitstoreal(X);
            RY = $bitstoreal(Y);
            X2Y2 = (RX * RX) + (RY * RY);
                B = X2Y2;
                A = 0.0;
                 while ((A - B) > Epsilon || (A - B) < -Epsilon)
                    begin
```

```
                A = B;
                B = (A + X2Y2 / A) / 2.0;
            end
        end
    assign Z = $realtobits(A);
endmodule
```

还请参阅：

User Defined Primitive, Instantiation, Name 语句的说明。

Name 名字

任何用 Verilog 语言描述的“东西”都通过它的名字来识别。

语法：

Identifier	标识符
\EscapedIdentifier {terminates with white space}	\ 扩展标识符{空格表示结束}

规则：

- 标识符可由字母，数字，下划线和美元符号构成。第一个字符必须是字母或下划线，而不能是数字或美圆符号。
- 一个扩展标识符用反斜杠引出，用空格结束（空格符、制表符、回车键或换行键），并且可包含除空格外的任何可印刷的字符。反斜杠和空格并不算作标识符的部分，例如，标识符 Fred 与扩展标识符 \ Fred 是相同的。
- 在 Verilog 中变量名是大小写敏感的。
- 在 Verilog 文件中，一个名字不能有多于一个以上的含义。名字的内部声明（例如在 begin-end 块中的名字）能屏蔽外部声明 (例如包含有该命名 begin-end 块的上层模块的变量声明语句)。

Hierarchical Names 分级名字

- Verilog HDL 中的每个标识符都有唯一的分级名字。这意味着任何 Net、寄存器、事件、参量、任务和函数都能通过使用它的分级名，在标识符的声明块外对它进行访问。
- 在名字层次的最上层是不需要实例引用的模块名。顶层测试模块就是一个最上层模块例子（尽管可能会有不止一个顶层模块在同一仿真中运行）。
- 在每个实例模块、命名块、任务和函数的定义时，便定义了名字层次树上的新层。

- Verilog 变量的分级名字是从顶层模块名字开始直到包含该变量的模块实例名、命名块、

任务和函数名构成，其间用小圆点隔开。

Upwards Name Referencing　向上索引名

包含两个标识符中间用点号隔开的分级名可能是下列情况中的一种：

- 当前模块所引用的实例模块中的一项。（这是向下引用）
- 顶层模块中的一项。（这是一个分级名字）
- 当前模块的父模块中的引用的实例模块中的一项。（这是向上引用。）
- 向上引用名字的第一个标识符既可能是一个模块名也可能是一个模块实例的名字。

可综合性问题：

分级名字和向上索引名在一般底档综合工具上是不可综合的。

提示：

- 通常，应选择对读者来说有含义的名字。相对本地名而言，这一点对于全局名显得更为重要。例如，给全局复位信号起名为 G0123，这名字不好，因没有含义，而 I 作为循环变量却是易于接受的。
- 名字中不要使用扩展字符。如网表生成或综合这一类 EDA 工具，它们具有与 Verilog 不同的命名规则，常留给这些扩展字符某些特殊含义。
- 分级名字仅用于测试模块或那些无法改用别的合适的名字的高层系统模型中。
- 避免使用向上索引名，因为它们会导致代码非常难理解，从而给调试和维护带来麻烦。

举例说明：

以下是合法名的例子：

```
A_99_Z
Reset
_54MHz_Clock$
Module       //与“module”是不一样的
\$%^&*()     //扩展标识符
```

以下是因上述原因而不合法的名字：

```
123a                 //名字不能用数字开始
$data                // 名字不能用美圆符号
module               // 名字不能用保留字
```

下面的例子说明了分级名字和向上引用名字：

```
module Separate;
    parameter P = 5;        // Separate.P
endmodule

module Top;
```

```
    reg R;            // Top.R
        Bottom    U1 ();
Endmodule

Module Bottom;
   reg R;         //Top.U1.R

       task T;      // Top.U1.T
          reg R;         //Top.U1.T.R;
           ...
       endtask

initial
     begin : InitialBlock
       reg R; // Top.U1.InitialBlock.R;
          $display(Bottom.R); // 向上索引名指向 Top.U1.R
          $display(U1.R); //  向上索引名指向 Top.U1.R
          ...
     end
endmodule       //end of Bottom module
```

Net 线路连接

Net 是结构描述中为线路连接（连线和总线）建立的模型。net的值是由net的驱动器所决定的。驱动器可以是门、UDP、实例模块或者连续赋值语句的输出。

语法：

{either}

```
NetType [ Expansion] [ Range] [ Delay] NetName,...;
trireg [ Expansion] [ Strength] [ Range] [ Delay]
NetName,...;
{Net declaration with continuous assignment}    用连续声明语句对 net 进行声明
NetType [ Expansion] [ Strength] [ Range] [ Delay]
NetAssign,...;
NetAssign = NetName = Expression
NetType = {either}
wire  tri {equivalent}
wor  trior {equivalent}
wand  triand {equivalent}
tri0
tri1
supply0
supply1
Expansion = {either}
vectored scalared
Range = [ ConstantExpression: ConstantExpression]
```

在程序中位于何处：

module-<HERE>-endmodule

规则：

- **supply0 和 supply1** 类型的 net 分别具有逻辑值0和1，并可以为它定义驱动能力(Supply strength)。
- **tri0** 和 **tri1** 类型的nets，当没有驱动时，分别具有逻辑值0和1，并可以为它定义驱动能力(Pull strength)。
- 如果net的扩展（Expansion）选项选用了关键词 **vectored**，则不允许对它进行某位和某些位的选择，也不允许对它定义强度，PLI会认为该 net 是不可扩展的；如果扩展（Expansion）选项选用了关键词**scalared**，则允许对它进行某位和某些位的选择，也允许对它定义强度，PLI将会认为该 net 是可扩展的，这些关键词是有参考价值的。
- 除了结构描述中的端口和标量连线不用声明其 net 类型外，其他类型的net变量在应用之前必须声明。

Truth Table 真值表

当Net具有两个或两个以上驱动时，同时假定其驱动器强度值均相等，这些真值表则告诉我们输出的结果。如果不相等，则驱动强度大者，驱动该 Net。

wire tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

tri0	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	0

tri1	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	1

注意!

- 当net未被驱动时，对**tri0** 或 **tri1** 类型的net 的连续赋值不影响其值和强度，经常为强度（strength）保持为Pull，和逻辑值保持为 0（对tri0）或 1（对tri1）。
- 在IEEE 标准和已成事实的Cadence公司标准中，扩展可选项的保留字**scalared** 或 **vectored**的位置有所不同，在Cadence标准中，保留字位于范围（range）选项的跟前。

可综合性问题：

- Net类型的变量被综合成线路连接，但是某些线路连接经优化后有可能被删去。
- 综合工具只支持 Net类型中wire型的综合，其它的 Net类型均不支持。

提示:

- 在每个模块的块首明确地声明所有的 nets，即使是缺省的类型也应该明确地加以说明。通过清楚地说明设计意图，可以提高 Verilog 程序的可读性和可维护性。
- 只能用supply0 和 supply1来声明地和电源。
-

举例说明：

```
wire Clock;
wire [7:0] Address;
tri1 [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
```

```
g = a || b; // 连续赋值
```

还请参照：
连续赋值，寄存器类型说明

Number 数

整数或者实数。在Verilog中整数是通过若干位来表示的，其中某些位可以是不定值（X）或高阻态（Z）。

语法：

```
{either}
BinaryNumber                （二进制数）
OctalNumber                 （八进制数）
DecimalNumber               （十进制数）
HexNumber                   （十六进制数）
RealNumber                  （实数）
BinaryNumber = [ Size] BinaryBase BinaryDigit...
OctalNumber  = [ Size] OctalBase OctalDigit...
DecimalNumber = {either}
[ Sign] Digit... {signed number}
[ Size] DecimalBase Digit...
HexNumber = [ Size] HexBase HexDigit...
RealNumber = {either}
[ Sign] Digit... .Digit...
[ Sign] Digit...[. Digit...]e[ Sign] Digit...
[ Sign] Digit...[. Digit...]E[ Sign] Digit...
BinaryBase = {either} 'b 'B
OctalBase = {either} 'o 'O
DecimalBase = {either} 'd 'D
HexBase = {either} 'h 'H
Size = Digit...
Sign = {either} +
-Digit = {either} _ 0 1 2 3 4 5 6 7 8 9
BinaryDigit = {either} _ x X z Z ? 0 1
OctalDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7
HexDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7 8 9 a A
b B c C d D e E f F
UnsignedNumber = Digit...
```

在程序中位于何处：
请参阅表达式。

规则：
- 表示进制的字母、十六进制数、X和Z在数的表示中是不区分大小写的，字符Z和？在数的表示中是等价的。
- 数字中不能有空格，但是在表示进制的字母两侧可以出现空格。
- 负数表示为其二进制的补数。
- 数字的第一个字符不允许出现下划线'_'，但标识符可以。为了提高数字的可读性可用下划线把长的数字分段，在处理数字时下划线将被忽略。
- 位宽指明了数字的准确位数。
- 不指明位宽的数字，它的位宽应为32位或32位以上，取决于主机字长。
- 如果位宽大于实际的二进制位数时，高位部分补0，但除非左边最高位是X或Z，在这种情况下，则补X或Z。
- 如果位宽小于实际的二进制数位时，超过位宽的高位（左边）被舍去。

注意！
- 定义了位宽的负数被赋值到寄存器后，它将被认为是无符号的数。

```
reg [7:0] byte;
reg [3:0] nibble;
initial
begin
nibble = -1;    //例如 4'b1111
byte = nibble;  // 变为 8'b0000_1111
end
```

当寄存器类型的数或者定义了位宽的数被用在表达式中时，其值通常被当作一个无符号数。
```
integer i;
initial
i = -8'd12 / 3;   // i变成 81 (即 8'b11110100 / 3)
```

可综合性问题：
- 0和1分别被综合成接地和接电源的连线，赋值为X的则被认为是无关项。 除了使用 **casex** 语句，如用其它的条件语句，与X的比较都认为是假的。 （case 等式运算符 ＝＝＝ 和！＝＝ 一般情况下都是不可综合的）
- 除了在casex 和 casez语句中Z被认为是无关项，在其它情况下 Z 则被用来表示三态驱动器。

提示：
- 在 **case** 语句的标号中，通常用 ？要比用Z好。在程序的其它地方不要使用 ？号，否则会产生混淆。
- 用下划线来分隔较长的数字，从而提高可读性。

举例说明：

```
-253          // 有符号的十进制数
'Haf          // 未定义位宽的十六进制数
6'o67         // 位宽为六的八进制数
8'bx          // 位宽为8的二进制数，其值为不定值
4'bz1         // 位宽为四的二进制数，最低位为1其余高三位均为高阻值(4'bzzz1)。
```

下面所列的数为不合法的数并解释其原因：

```
_23           //  以_开头
8' HF F       //  包含两个非法空格
0ae           //  十进制数中出现十六进制数字
x             //  是名字，不是数字（应用1'bx）
.17           //  应该是0.17
```

还请参照：
表达式，字符串说明

Operators　运算符

在表达式中，使用运算符便可根据操作数（诸如数字、参量以及其它子表示式）计算出表达式的值。Verilog 语言中的运算符和 C 语言中的很相似。

单目运算符：

+ -	正负号
!	逻辑非
~	按位取反
& ~& | ~| ^ ~^ ^~	缩位运算符（~^ 和 ^~ 等价）

二目运算符：

+ - * /	算术运算符
%	取模运算符
> >= < <=	关系运算符
&& ||	逻辑运算符
== !=	逻辑等式运算符
=== !==	case等式运算符
& | ^ ^~ ~^	逐位运算符（^~ 和 ~^ 等价）
<< >>	移位运算符

其它运算符：

A ? B : C	条件运算符
{ A, B, C }	位拼接运算符
{ N{A} }	重复运算符

在程序中位于何处：
参阅表达式说明。

规则：

- 逻辑运算符把它的操作数当作布尔变量。例如，非零的操作数被认为是真（1'b1)；零被认为是假(1'b0)；不确定的值，例如4'bXX00，因不能判断其值为真还是假，就被认为是不确定的（1'bX)。
- 位运算符(~ & | ^ ^~ ~^)和全等运算符(= = =　　!= =)把它们操作数的逐位分别进行处理。
- 在包含 = = 或　!= 的逻辑比较式中，如果有任何一个操作数为X或Z，其结果便是不确定的(1'bX)。(请仔细看注意事项）
- 在包含(< > <= >=)的比较式中，如果操作数不确定，其结果为不定值。(1'bX)例如：

  2'b10 > 1'b0X　　//　结果为真
  2'b11 > 1'b1X　　//　结果不定(1'bX)
  （请看注意事项）
- 缩位运算符（ & ~& | ~| ^ ~^ ^~)将一个矢量缩减为一个标量。
- 位宽确定的表达式的运算采用溢出的位不计的办法，例如：4'b1111 + 4'b0001 = 4'b0000.
- 整数作除法运算时，小数部分被截掉。
- 取模运算(%)的结果是第一个操作数被第二个操作数除的余数，符号与第一个操作数一致。
- 只有某些特定的运算符允许出现在实数表达式中,例如单目运算符＋和－、算术运算符、关系运算符、逻辑运算符以及条件运算符。实数逻辑或关系运算符的结果是一个只有一位的值。

运算符的优先级：

```
+ - ! ~          单目(unary) -        最高优先级
* / %
+ -              双目(binary)
<< >>
< <= > >=
== != === !==
& ~&
^ ^~
| ~|
&&
||
?:                                    最低优先级
```

注意：

- 应用 ==、 !=、 <、　> 、 <= 和 >= 对某些位不确定的值进行比较的规则并不适用于

所有的仿真器，这点请特别注意!

- 注意单目缩位运算符与逐位逻辑运算符之间的区别，运算符本身是相同的，可根据上下文的关系来判断是哪一种，有时必须要用括号才能表达清楚。

可综合性问题：

- 逻辑运算符、逐位运算符、移位运算符是可综合的，都被综合成逻辑运算。
- 条件运算符是可综合的，被综合成多路器或带使能端的三态门。
- 运算符+、 -、 * 、< 、<= 、>、 >=、 == 和 != 都是可综合的，被分别综合成加法器、减法器、乘法器和比较器。
- 运算符 / 和 % 一般是不可综合的,只有当能用移位寄存器来表示运算时才是可综合的。而常量的 / 和 %运算是可综合，但结果只能用二进制数表示。
- 其它运算符均不能被任何工具所综合。

提示：

在写表达式的时候，运用括号要比依靠运算符的优先级要好，这样可预防错误产生，并且使那些不太了解 Verilog 语言的人更容易理解你的意思。

举例说明：

```
-16'd10              // 这是表达式，是负运算，不是有符号数!
a + b
x % y
Reset && !Enable     // 与 Reset && (!Enable)相同
a && b || c && d     // 与 (a && b) || (c && d)相同
~4'b1101             // 结果为 4'b0010
&8'hff               // 结果为 1'b1,即一位的逻辑值1
```

还请参照：

Expression 的说明

Parameter 参数

参数是为常数命名的一种手段。在 Verilog 代码模块编译时（而不是在仿真期间），可以改写参数的值。使用参数就有可能重新定义 Verilog 代码中的常数，如数组的宽度等。

语法：

```
parameter Name = ConstantExpression,
Name = ConstantExpression,
... ;
```

有些工具支持下列非标准的语法：

```
parameter [ Range] Name = ConstantExpression,
Name = ConstantExpression,
```

... ;
Range = [ConstantExpression: ConstantExpression]

在程序中位于何处：

module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction

规则：

- 参数是常量，在仿真期间更改参数的值是非法的。
- 在编译期间用 defparam 或者当包含参数的模块被引用时， 可以改写其参数的值。

可综合性问题：

有些综合工具能把含有参数的模块当作模板，一旦读入模板，便能够用不同的参数值多次对该模板进行综合。所有的综合工具都支持不带改动参数的模块实例的综合。

提示：

尽可能用参数给常数起一个有含义的名字。

举例说明：

下面的例子是一个 N 位宽的（可通过参数改变位宽的）移位寄存器。实例引用该参数化移位寄存器时可重新定义不同的位宽。

```
module Shifter (Clock, In, Out, Load, Data);
parameter NBits = 8;
input Clock, In, Load;
input [NBits-1:0] Data;
output Out;
always @(posedge Clock)
if (Load)
ShiftReg <= Data;
else
ShiftReg <= {ShiftReg[NBits-2:0], In}
assign Out = ShiftReg[NBits-1];
endmodule

module TestShifter;
...
defparam U2.NBits = 10;
Shifter #(16) U1 (...);      //  6位移位寄存器
Shifter U2 (...)             // 10位移位寄存器
Endmodule
```

还请参阅：
`define, Defparam, Instantiation, Specparam语句的说明。

PATHPULSE$　路径脉冲参数

- 在指定块中用指定参数 (即用specparam)对PATHPULSE$参数赋值可控制脉冲的传输。这里所谓的脉冲是指在模块输出端出现的两个跳变沿和它们之间的一段持续时间，其持续时间必须小于信号从模块的输入端直到输出端的延时。
- 如果使用缺省的PATHPULSE$参数值，仿真器将不考虑脉冲，这就是指因为路径脉冲的持续时间比模块传输延时短，故脉冲不能传过该模块，这种效应被称为“时延惯性”。用指定参数 (即用specparam)可给PATHPULSE$参数赋新的值。

语法：
```
{either}
PATHPULSE$ = ( Limit[, Limit]); {(Reject, Error)}
PATHPULSE$Input$Output = ( Limit[, Limit]);
Limit = ConstantMinTypMaxExpression
```

在程序中位于何处：
```
specify-<HERE>-endspecify
```

规则：
- 如果PATHPULSE$的第二个极限参数（即 Error）没有给定，它就应该与第一个极限参数（即 reject）相同。
- 维持时间比第一个极限参数（即 reject）短的脉冲不会输出。
- 维持时间比第一个极限参数（即 reject）长而比第二个极限参数（即 Error）短的脉冲将输出一位的不确定值（即 1'bX)。
- 维持时间比第二个极限参数长的脉冲将正常地输送出去。
- 用specparam对PATHPULSE$input$output参数重新赋值将改写常规值。
- 在同一个模块中可通过使用specparam对PATHPULSE$赋值来描述从输入到输出的延时。

可综合性问题：
综合工具不考虑延时结构，包括指定块的定义。

举例说明：
```
specify
        (clk => q) = 1.2;
        (rst => q) = 0.8;
        specparam PATHPULSE$clk$q = (0.5,1),
        PATHPULSE = (0.5);
Endspecify
```

还请参阅：
Specify, Specparam 说明

Port 端口

模块的端口是硬件器件的引脚或接口的模型。

语法：

{definition}
{either}
PortExpression {ordered list}
.PortName([PortExpression]) {named list}
PortExpression = {either}
PortReference
{ PortReference,...}
PortReference = {either}
Name
Name[ConstantExpression]
Name[ConstantExpression: ConstantExpression]
{declaration}
{either}
input [Range] Name,...; {of port reference}
output [Range] Name,...; {of port reference}
inout [Range] Name,...; {of port reference}
Range = [ConstantExpression: ConstantExpression]
{在上述部分位选择（即 Range）选项内，冒号左侧常量表达式表示最高位（即 MSB），冒号右侧常量表达式表示最底位（即LSB）}

在程序中位于何处：

module (<HERE>); {definition}
<HERE> {declaration}
...
endmodule

规则：

- 在端口列表中列出的所有端口必须按次序排列或按端口名称排列,这两种排列方式是不同的，不能混合使用。
- 有端口的名称但没有端口表达式，如 .A()，则表示在本模块中定义了不与任何东西相连的端口。
- 每个端口除了必须在端口列表中列出外， 还必须声明该端口是输出(output)、输入

(input)、还是双向端口(inout)。

- 每个端口不但要声明是输出、输入、还是双向端口，而且还要声明是连线（wire）还是寄存器(reg)类型，如果没声明，则会隐含地认为该端口是连线（wire）类型，且其位宽与相应的端口一致。如果某端口已被声明为一矢量，则其端口的方向和类型两个声明中的位宽必须一致。
- 输入和双向端口不能声明为寄存器类型。
- 输出端口的类型不能声明为实型（Real）或实时型（realtime)。

提示：

- 在测试模块中不要定义端口。
- 在模块定义时不建议使用命名的端口的列表，因为很少有人这样来定义模块端口，大家都不了解这种端口的定义形式。

举例说明：

```
module (A, B[1], C[1:2]);
input A;
input [1:1] B;
output [1:2] C;

module (.A(X), .B(Y[1]), .C(Z[1:2]);
input X;
input [1:1] Y;
output [1:2] Z;
```

还请参照：

Module, User Defined Primitive, Instantiation 的说明。

Procedural Assignment　过程赋值语句

改变寄存器的值，或者安排以后的变化。

语法：

```
{Blocking assignment}                              阻塞赋值
RegisterLValue = [ TimingControl] Expression ;
{Non-blocking assignment}                          非阻塞赋值
RegisterLValue <= [ TimingControl] Expression ;
RegisterLValue = {either}
RegisterName
RegisterName[ Expression]
RegisterName[ ConstantExpression: ConstantExpression]
Memory[ Expression]
```

{ RegisterLValue,...}

在程序中位于何处：

请参阅statement语句的说明。

规则：

- 对寄存器的赋值（不包括正负号）
- 对于实型和实时数据类型的寄存器不允许选择某位和某几位的。
- 当赋值语句执行时，右侧的表达式被计算出值，但是直到定时控制事件或延时（也被称为‘内部指定的延时’）发生后，左侧的表达式才更新。
- 直到左侧的表达式更新后（例如内部定义的延时过后）阻塞赋值语句才算完成。在begin-end模块中，只有当前一条语句执行完后，才能执行其后面的一条语句。在fork-join模块中，只有当块中所有的阻塞赋值语句结束后，整个块才算结束。
- 如果仿真时刻相同，要待所有的阻塞赋值语句执行后，非阻塞赋值语句才执行。

```
A <= #5 0;
A = #5 1;      //5个时间单位后，A将变为0，而不是变为1
```

注意！

寄存器变量可以在一个或几个**initial** 或 **always** 语句中赋值。无论何时，寄存器变量的值都是由最近的赋值所决定，与事件的来源无关。这一点与 net 类型的变量不同。net可以由两个或更多的源驱动，其结果值则取决于net变量的类型（wire型,wand型等）。

可综合性问题：

- 综合工具不考虑延时。
- 定时控制或延时是不可综合的。
- 同一个寄存器类型变量虽然可以在几个**always**语句中赋值，但只有在一个 **always**语句中赋值的才有可能被综合。
- 同一个寄存器类型变量不能既用阻塞赋值和非阻塞赋值。
- 在描述组合逻辑的**always**块中，右侧表达式被综合成组合逻辑，左侧的表达式被综合成连线，如有不完整的赋值则综合成锁存器。在描述时序逻辑的用时钟沿触发的**always**块中，非阻塞赋值符的左侧被综合成触发器，阻塞赋值符的左侧则被综合成一个连接，除非它被用在该**always**块之外，或者在赋值之前它的值已被读取。

提示：

- 通常采用非阻塞赋值语句来生成触发器组成的时序逻辑，而阻塞赋值常用于其它方面，这样做可以防止时钟沿触发的 **always** 块中发生竞争冒险。这样做也可使设计意图更加清晰，又能避免生成不需要的触发器。
- 在时钟树的模型已确定的情况下，可用一个简单的内部指定的延时来避免RTL时钟沿对不齐的问题。

举例说明：

```
always @(Inputs)
    begin : CountOnes
        integer I;
```

```
        f = 0;
        for (I=0; I<8; I=I+1)
            if (Inputs[I])
                    f = f + 1;
    end
always @Swap
    fork // 交换a和b的值
        a = #5 b;
        b = #5 a;
    join // 延时5秒后完成

always @(posedge Clock)
    begin
        c <= b;  // 用旧的b值
        b <= a;  // b被a值替换
    end
```

用非阻塞赋值语句时加一个延时来做输出与时钟沿有些偏移的仿真：

```
  always @(posedge Clock)
        Count <= #1 Count + 1;
```

在时钟周期的第五个下降沿插入复位信号：

```
  initial
    begin
       Reset = repeat(5) @(negedge Clock) 1;
       Reset = @(negedge Clock) 0;
    End
```

还请参照：

Timing Control, Continuous Assignment 的说明。

Procedural Continuous Assignment 过程连续赋值语句

启动过程连续赋值语句，将给一个或多个寄存器赋值，并同时防止一般的过程赋值语句影响已赋值的寄存器。

语法：

```
assign RegisterLValue = Expression ;
deassign RegisterLValue ;
RegisterLValue = {either}
RegisterName
```

RegisterName[Expression]
RegisterName[ConstantExpression: ConstantExpression]
MemoryName[Expression]
{ RegisterLValue,...}

在程序中位于何处：
请参阅 Statement的说明。
规则：

- 过程连续赋值语句执行后，它会对指定的寄存器（组）强制地维持过程连续赋值直到解除赋值（deassign）语句的执行，或直到另一个过程连续赋值语句又对该寄存器（组）赋值。
- 用force（强制）语句可以改写已由过程连续赋值语句赋值的寄存器类型变量，直到 release 语句的执行，此时强制赋值被解除而原过程连续赋值对该寄存器类型变量的作用又重新恢复。

注意！
连续赋值语句与过程连续赋值语句尽管很相似，但并不是完全一致。在编写程序时，确认将 assign 写在正确的位置。 过程连续赋值语句可以写在声明语句允许出现的位置（在 initial, always, task, function 等内部），而连续赋值语句则必须写在任何 initial 或 always 块之外。

可综合性问题：
无论用什么综合工具，过程连续赋值语句是都不能综合的。

提示：
过程连续赋值语句可以用来为异步复位和中断建立仿真模型。

举例说明：

```
always @(posedge Clock)
    Count = Count + 1;        //受下面always块控制的计时钟个数的计数器

always @(Reset) // 异步复位
    if (Reset)
            assign Count = 0; // 当 Reset为高时，使Count为0，不计数
        else
            deassign Count; // 当 Reset为低时，解除Count为0，
                            //于是下一个时钟的上升沿又重新开始计数。
```

还请参阅：
Continuous Assignment, Force 的说明

Programming Language Interface 编程语言接口

Verilog编程语言接口 (PLI)为用户提供了在Verilog模块中调用用C语言编写的函数的手段。这些函数可以动态地访问和修改被引用的Verilog数据结构中的数据，用PLI编写的系统任务使上述功能变得容易使用。通过调用用户定义的系统任务和函数的可以来启动PLI，用户编写自己的PLI 模块的目的是扩大系统任务和函数的内容。用户自定义的系统任务和函数在调用时都用以$字符开头的任务和函数名。这与Verilog语言提供的系统任务和函数库名一致。如用户自定义的系统任务和函数名与原系统任务或函数名相同时，则执行用户自定义的系统任务和函数。

下面列举的是PLI在某些方面的应用：
- 延迟计数
- 测试矢量读入
- 波形演示
- 源代码调试

接口模型可用C语言或其他语言（例如VHDL或硬件建模工具）编写或生成。
对于PLI的全面讨论超出了本参考指南的范围。

Register　寄存器

寄存器可存储在initial、 always、task 和 function 块中所赋的值，广泛地应用在行为建模中。

语法：

```
{either}
reg [ Range] RegisterOrMemory,...;
integer RegisterOrMemory,...;
time RegisterOrMemory,...;
real RegisterName,...;
realtime RegisterName,...;
RegisterOrMemory = {either}
RegisterName
MemoryName Range
Range = [ ConstantExpression: ConstantExpression]
```

在程序中位于何处：

```
module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction
```

规则：

- 寄存器类型变量只能用过程赋值语句赋值。
- 在具体实现时，整数（integer）类型的变量至小用32位，时间（time）类型的变量至小用64位寄存器。
- integer 或 time类型的寄存器变量与位数相同的reg类型的寄存器变量行为是相同的。Integer 和 time 型的寄存器变量也可像reg类型的寄存器变量一样对某位或某些位操作。而在表达式中，整数类型的值被当作有符号值，而reg、time类型的值被当作无符号值。
- 存储器类型数组中的每个元素作为整体可以进行读或写操作，如果要单独访问数组中某个元素的个别位，则必须先把这个元素的内容复制到某个位数相同的寄存器变量中才能进行。

注意！

- 虽然 register 这个词指的是硬件寄存器（例如触发器），而寄存器（register）这个名字，在这里是指软件寄存器（即变量）。Verilog寄存器常用于组合逻辑电路、锁存器、触发器和接口电路的描述和综合。
- realtime 类型寄存器变量是Verilog语言新增加的变量类型，目前还没有任何工具支持这种类型的变量。
- 有符号和无符号值的概念，不同版本的Verilog和用不同厂家的仿真器时，并不是完全一致的。因此，当使用位宽大于32位的有符号数或矢量时要特别注意。

可综合性问题： .

- Real, time 和 realtime类型的寄存器变量是不可综合的。
- 在描述组合逻辑的always块中，寄存器被综合成 wire型；如果存在不完整赋值的情况，则被综合成锁存器。在描述时序逻辑的always块中，寄存器根据块内语句的内容被综合成连线（wire）或者触发器。
- 运用目前的综合工具，整数被综合成32位，其值用二进制数表示，负数则用其二进制补码表示。
- 根据所用语句，存储器数组会被综合成触发器或连线，而不会被综合成RAM 或 ROM的器件。

提示：

运用reg类型变量来描述寄存器逻辑，integer类型变量用于循环变量和计数，real类型变量用于系统模块，time 和 realtime类型变量用于测试模块中记录仿真时刻。

举例说明：

```
reg a, b, c;
reg [7:0] mem[1:1024], byte; // byte 不是数组只是一个8位的reg类型矢量
integer i, j, k;
time now;
real r;
realtime t;
```

下面的部分显示了reg 类型 和integer 类型变量的一般用法

```
integer i;
reg [15:0] V;
reg Parity;
always @(V)
    for ( i = 0; i <= 15; i = i + 1 )
        Parity = Parity ^ V[i];
```

还请参阅：
Net的说明。

Repeat　　重复执行语句

把一个或多个声明语句重复地执行指定的次数。

语法 ：

```
repeat ( Expression)
    Statement
```

在程序中位于何处：
参见 Statement 的说明

规则：
重复执行的次数是由表达式的数值所决定的，如果该值为0, X 或 Z，则不会有重复。

可综合性问题：
只有部分综合工具可以综合repeat语句,而且只有当该循环中的每个循环的分支都被时钟事件，如被 @(posedge Clock)，所中断时才有可能被综合成电路。

举例说明：

```
initial
    begin
        Clock = 0;
            repeat (MaxClockCycles)
                begin
                    #10 Clock = 1;
                    #10 Clock = 0;
                end
    end
```

还请参阅：
For, Forever, While, Timing Control的说明。

Reserved Words　　关键词

下列词汇是Verilog语言规定的所有的关键词，请注意，千万不要把这些标识符用作自定义的标识符，除非把他们改写为大写的字符或扩展字符。

And	for	output	strong1
Always	force	parameter	supply0
Assign	forever	pmos	supply1
Begin	fork	posedge	table
Buf	function	primitive	task
bufif0	highz0	pulldown	tran
bufif1	highz1	pullup	tranif0
case	if	pull0	tranif1
casex	ifnone	pull1	time
casez	initial	rcmos	tri
cmos	inout	real	triand
deassign	input	realtime	trior
default	integer	reg	trireg
defparam	join	release	tri0
disable	large	repeat	tri1
edge	macromokule	mmos	vectored
else	medium	rpmos	wait
end	module	rtran	wand
endcase	nand	rtranif0	weak0
endfunction	negedge	rtranif1	weak1
endprimitive	nor	scalared	while
endmodule	not	small	wire
endspecify	notif0	specify	wor
endtable	notif1	specparam	xnor
endtask	nmos	strength	xor
event	or	strong0	

Specify　　指定的块延时

Specify块（指定延时块）用于描述从模块的输入到输出的路径延时以及定时约束，例如信号的建立和保持时间。用指定延时块可以在设计时把模块的信号传输延时与行为或结构分开来进行描述。

语法：

```
specify
SpecifyItems...
endspecify
SpecifyItem = {either}
Specparam
PathDeclaration
TaskEnable {Timing checks only}
PathDeclaration = {either}
SimplePath = PathDelay;
EdgeSensitivePath = PathDelay;
StateDependentPath = PathDelay;
SimplePath = {either}
( Input,... [ Polarity] *> Output,...)  {full}
( Input [ Polarity] => Output)          {parallel}

EdgeSensitivePath = {either}
([ Edge] Input,... *> Output,... [ Polarity]: Expression)
([ Edge] Input => Output [ Polarity]: Expression)
StateDependentPath = {either}
if ( Expression) SimplePath = PathDelay;
if ( Expression) EdgeSensitivePath = PathDelay;
ifnone SimplePath = PathDelay;

Input = {either}
InputName
InputName[ ConstantExpression]
InputName[ ConstantExpression: ConstantExpression]
Output = {either}
OutputName
OutputName[ ConstantExpression]
OutputName[ ConstantExpression: ConstantExpression]
Edge = {either} posedge negedge
Polarity = {either}
 +
 -
PathDelay = {either}
ListOfPathDelays
( ListOfPathDelays)
ListOfPathDelays = {either}
t
t,t                          {Rise,Fall}
t,t,t                        {Rise,Fall,Turn-Off}
```

t,t,t,t,t,t　　{01,10,0Z,Z1,1Z,Z0}
t,t,t,t,t,t,t,t,t,t,t,t　　{01,10,0Z,Z1,1Z,Z0,0X,X1,1X,X0,XZ,ZX}
t = MinTypMaxExpression

在程序中位于何处：
module-<HERE>-endmodule

规则：
- 路径必须从模块的输入端开始，在该模块的输出端结束，而且在模块内部只能有一个驱动器。
- 在路径声明中可使用全连接符(*>)或者并行连接符 (=>)来描述。全连接符指所有从输入端到输出端可能的路径，并行连接符指命名的输入端的某些位到命名的输出端的某些位的路径。
- 模块路径的极性可选是指着路径可以选择正极性或者负极性，分别指路径是同相的或是反相的（即路径的输入端若是正跳沿，输出端也是正跳沿，则路径是同相的。反之是反相的）。无论选哪一个都不影响仿真，但路径的相位可改变的选项便于时序分析等工具使用。
- 跳变沿敏感的路径数据表达式同样也不影响仿真。
- 与状态有关的路径延时(SDPD)表达式只跟端口、常量、局部定义的寄存器或者net类型变量有关。只有部分运算符在SDPD表达式中是有效的：如逐位计算符(~ & | ^ ^~ ~^)、逻辑运算和逻辑等式运算符(== != && || !)、 缩位运算符(& | ^ ~& ~| ^~ ~^)、位拼接运算符、重复拼接运算符和条件运算符({} {{}} ?:)。如果条件表达式的值为真（在SDPD 表达式中 1，X，Z 均被认为是真），路径延时只影响路径。
- 如果没有一个if 条件为真，则用Ifnone来定义缺省的SDPD。如果同一条路径既定义为ifnone 与状态有关的路径延时（SDPD）又定义为一般简单的路径延时，则是非法的。
- 无条件的路径优先于SDPD路径。
- 对于同一条路径，跳变沿敏感的SDPD路径声明必须是唯一的，必须用不同的电平或不同的沿（或者两者）。在每条语句中，必须以同样的方式（整个端口、某一位、某些位）来引用输出端的信号。
- 如果模块的延时既包括指定的块延时（specify delays）又包括分布的延时（即由门、UDP、Net引起的延时），应选用最长的延时作为每条路径的延时。

可综合性问题：
综合工具不能综合指定延时块。指定延时块只用于模块的时延仿真建模。

注意！
- 目前还没有一个仿真工具支持上面语法中列出的那种可用12种不同跳变参数表示某条路径延时的方法。
- 有关路径目的地的规则比当前许多仿真工具所能支持的灵活。

提示：
- 运用指定延时块来描述库中的单元（cell）延时。请注意建立库模型时延时是怎样计算的。以PLI（编程语言接口）为基础的延时计算，需要依据并访问设计中所有单元的指定延时块中的信息。

- 可运用指定延时块来描述“黑匣子”元件的定时特性， 但这时还需要借助于支持指定延时块特性的时序验证工具或综合工具。

举例说明：

```
module M (F, G, Q, Qb, W, A, B, D, V, Clk, Rst, X, Z);
    input A, B, D, Clk, Rst, X;
    input [7:0] V;
    output F, G, Q, Qb, Z;
    output [7:0] W;
    reg C;
        // Functional Description ...功能描述
        specify
            specparam TLH$Clk$Q = 3,
            THL$Clk$Q = 4,
            TLH$Clk$Qb = 4,
            THL$Clk$Qb = 5,
            Tsetup$Clk$D = 2.0,
            Thold$Clk$D = 1.0;
    // 单一路径，全连接
            (A, B *> F) = (1.2:2.3:3.1, 1.4:2.0:3.2);
    // 单一路径，并行连接，正极性
            (V + => W) = 3,4,5;
    // 沿敏感路径，带极性
            (posedge Clk *> Q +: D)  = (TLH$Clk$Q,THL$Clk$Q);
            (posedge Clk *> Qb -: D) = (TLH$Clk$Qb,THL$Clk$Qb);
    // 电平敏感路径
            if (C) (X *> Z) = 5;
            if (!C && V == 8'hff) (X *> Z) = 4;
            ifnone (X *> Z) = 6; // 缺省为SDPD,从 X（不定值）到 Z（高阻值）
    // 时序检测
        $setuphold(posedge Clk, D,Tsetup$Clk$D, Thold$Clk$D, Err);
    endspecify
endmodule
```

还请参阅：

Specparam, PATHPULSE$, $setup 的说明。

Specparam　　延时参数

类似于parameter(参数)，但只能用在指定延时块中。

语法：

```
specparam Name = ConstantExpression,
Name = ConstantExpression,
... ;
```

在程序中位于何处：

```
specify -<HERE>- endspecify
```

规则:

- Specify块中的常量表达式可以用数字和 specparam 来定义，但不能用参数(parameter)来定义，specparam 不能用在 Specify块（即指定延时模块）外。
- 利用 **defparam** 或在模块的实例引用时使用＃，可以改写用 specparam 定义的延时参数值，用编程语言接口（PLI）也可以修改其值。

提示

- 在Specify块中，用specparam来定义命名的延时参数比直接用数字要好。
- 这些延时参数应有一个命名的规则，这样便于对它们进行修改，如果有必要的话，也可以采用PLI的延时计数来进行修改。

举例说明：

```
specify
    specparam   tRise$a$f = 1.0,
                tFall$a$f = 1.0,
                tRise$b$f = 1.0,
                tFall$b$f = 1.0;
                (a *> f) = (tRise$a$f, tFall$a$f);
                (b *> f) = (tRise$b$f, tFall$b$f);
endspecify
```

还请参阅：

PATHPULSE$, Specify的说明。

Statement　　声明语句

运用声明语句可以描述硬件模块的行为。声明语句在定时控制的（延时、控制程序、等待）时刻执行。若两个或两个以上的语句是一起的，必须把它们写在 begin-end 或 fork-join 块中。在 begin-end 块中每条语句是顺序执行的，在fork-join块中，它们是并行执行的。initial 或 always块中的语句是同其它initial 或 always块中的语句是同时执行的。

语法：

```
{either}
```

; {Null statement}
TimingControl Statement {Statement may be Null}
Begin
Fork
ProceduralAssignment
ProceduralContinuousAssignment
Force
If
Case
For
Forever
Repeat
While
Disable
-> EventName; {Event trigger}
TaskEnable

在程序中位于何处：
initial-<HERE>
always-<HERE>
begin-<HERE>-end
fork-<HERE>-join
task-<HERE>-endtask {Null allowed}
function-<HERE>-endfunction
if()-<HERE>-else-<HERE> {Null allowed}
case- label:-<HERE>-endcase {Null allowed}
for(<HERE>)-<HERE>
forever-<HERE>
repeat()-<HERE>
while()-<HERE>

还请参阅：
Timing Control的说明。

Strength　　强度

除了逻辑值外，Net类型的变量还可以定义强度，因而可以更精确地建模。Net的强度来自于动态 Net 驱动器的强度。在开关级仿真时，当 Net由多个驱动器驱动且其值互相矛盾时，常用强度（Strength）的概念来描述这种逻辑行为。

语法：

```
{either}
( Strength0, Strength1)
( Strength1, Strength0)
( Strength0)            {pulldown primitives only}
( Strength1)            {pullup primitives only}
( ChargeStrength)       {trireg nets only}
Strength0 = {either}
supply0
strong0
pull0
weak0
highz0
Strength1 = {either}
supply1
strong1
pull1
weak1
highz1
ChargeStrength = {either}
large
medium
small
```

在程序中位于何处：

请参照： Net、Instantiation、Continuous Assignment 的说明。

规则：

- 关键词Strength0 和 Strength1用于定义Net的驱动器强度。其中Strength表示强度，与紧跟着的0和1连起来分别表示输出逻辑值为0和1时的强度。
- 在强度声明中可选择不同的强度关键字来代替strength，但(highz0,highz1) 和(highz1,highz0)这两种强度定义是不允许的，在pullup （上拉）和 pulldown（下拉）门的强度声明中 highz0 和 highz1是不允许的。
- 默认的强度定义为 (strong0,strong1)，但下述情况除外：

1) 对于pullup and pulldown门，默认强度分别为(pull1) 和 (pull0)。
2) 对于trireg 的 Net，默认强度为 medium

3) 强度定义为supply0 和 supply1的Net，总是能提供强度。

- 在仿真期间，Net 的强度来自于 Net 上的主驱动强度（即具有最大强度值的实例或连续赋值语句）。如果 Net 未被驱动，它会呈现高阻值，但以下情况除外：

1) tri0 和 tri1 类型的 net 分别具有逻辑值0和1，并为pull强度。
2) trireg 类型的 net 保持它们最后的驱动值。
3) 强度为supply0 和 supply1 的 nets分别具有逻辑值0和1，并能提供驱动能力。

- 强度值有强弱顺序，可从 supply（最强的）依次减弱排列到 highz（最弱的），当需要确定Net的确实逻辑值和强度时，或者当 Net由多个驱动器驱动而且驱动相互间出现冲突时，出现冲突的两个强度值在强弱顺序表中的相对位置就会对该Net的真实逻辑值起作用。

Supply
Strong
Pull
Large
Weak
Medium
Small
Highz

可综合性问题：

不可综合

提示：

可以在$display和$monitor等中用特定的格式控制符 %V 显示其强度值。

举例说明：

```
assign (weak1,weak0) f= a + b;
trireg  (large ) c1,c2;
and (strong1,weak0)  u1(x,y,z);
```

请参阅：

Continous Assignment、Instantiation、Net、$display 的说明

String 字符串

字符串能够用在系统任务（诸如$display和$monitor等）中作为变量，字符串的值可以像数字一样储存在寄存器中，也可以像对数字一样对字符串进行赋值、比较和拼接。

语法

见“string”说明。

在程序中位于何处：

请参见 Expression 说明。

规则

- 一条字符串不能占原代码的多行。
- 字符串可以包含下列扩展字符。

• \n	• 换行
• \t	• Tab符
• \\	• 反斜杠字符\
• \”	• 双引号字符”
• \nnn	• 八进制的ASCII字符
• %%	• 百分号%

- 诸如$display和$monitor等的系统任务中的打印字符串可以包含特殊的格式控制符(如%b)(参见$display的说明)。
- 当字符串存储于寄存器中，每个字符要占8位，字符以ASCII代码形式存储。VerilogHDL语言的字符串的定义和 C 语言的不一样。在C 语言中需要用，而在VerilogHDL语言中不需要用ASCII代码的 0字符来表示字符串的结束。

注意！

在表达式中使用字符串时，请注意填加物。对字符串的处理跟对数字的处理方式一样，当字符所占的位数少于寄存器的数目时，则在字符串的左边的寄存器中填加0。

举例说明：

```
reg [23:0] MonthName[1:12];
initial
begin
MonthName[1] = "Jan";
MonthName[2] = "Feb";
MonthName[3] = "Mar";
MonthName[4] = "Apr";
MonthName[5] = "May";
MonthName[6] = "Jun";
MonthName[7] = "Jul";
MonthName[8] = "Aug";
MonthName[9] = "Sep";
MonthName[10] = "Oct";
MonthName[11] = "Nov";
MonthName[12] = "Dec";
end
```

请参阅：

NUMBER, $display 的说明。

Task　　任务

任务常用于把模块代码分割成由若干声明语句构成的较大的块，便于模块代码的理解和维护，也可以从模块代码的不同位置执行这样一个常见的顺序声明语句块。

语法：

```
task TaskName;
[ Declarations...]
Statement
endtask
Declaration = {either}
input [ Range] Name,...;
output [ Range] Name,...;
inout [ Range] Name,...;
Register
Parameter
Event
Range = [ ConstantExpression: ConstantExpression]
```

规则：

- 若用于任务中的命名变量或参数没有在任务块中声明,则指的是在模块中声明的命名变量或参数。
- 任务中的 input、output 和 inout 的个数不受限制（也可以为零个）。
- 任务中的变量（包括输入和双向端口（inout））可以声明为寄存器型。如果没有明确地声明，则默认为寄存器型，且其位宽与相应的变量匹配。
- 当启动任务时，相应于任务的输入和双向端口（inout）的变量表达式的值被存入相应的变量寄存器中。当任务结束时，输入和双向端口（inout）的变量寄存器中的值又被代入启动任务的语句中相应的表达式。

注意！

- 和模块的端口定义不一样，任务的变量不能在任务名后的括号中定义。
- 任务中若包括一句以上的语句，必须要用begin –end 或fork-join 将其包含成块。
- 任务的输入、双向端口 (inout) 、输出和局部寄存器的值都是静态储存的，也就是说即使多次启动任务，也只有一份这些寄存器的拷贝。若第一次启动的任务还未完成，便第二次启动该任务，其输入、双向端口 (inout)、输出和局部寄存器的值便会被覆盖。
- 当被启动的任务运行结束时,输出和双向端口 (inout)的值被代入任务中相应的寄存器表达式。如果任务中的输出和双向端口 (inout)在赋值后有时间的控制，则相应的寄存器只能在定时控制延迟后才被更新。
- 同样，对输出和双向端口 (inout)寄存器变量的非阻塞赋值语句也不会起作用，因为当任务返回时，赋值语句可能还未生效。

可综合性问题：

包含定时控制语句的任务是不可综合的。启动的任务往往被综合成组合逻辑。

提示：

- 复杂 RTL 模块通常需要用多个 always 块来构造。建议最好不要采用一个always块运行多个任务的方案。
- 在测试块中可用任务来产生重复的激励序列。例如，对存储器的数据读写（见例）序列。
- 某任务如果被多个模块引用，可以把它定义为一个独立的模块（只包括该任务），并可用层次命名来引用它。

举例说明：

这个例子表示一个简单的可以综合的RTL任务

```
task Counter;
    inout [3:0] Count;
    input Reset;
        if (Reset)      // 同步复位
            Count = 0;  // 对 RTL 必须用非阻塞方式赋值
        else
            Count = Count + 1;
Endtask
```

下面这个例子说明如何在测试模块中运用任务。

```
module TestRAM;

  parameter AddrWidth = 5;
  parameter DataWidth = 8;
  parameter MaxAddr = 1 << AddrBits;

  reg [DataWidth-1:0] Addr;
  reg [AddrWidth-1:0] Data;
  wire [DataWidth-1:0] DataBus = Data;
  reg Ce, Read, Write;

  Ram32x8 Uut (.Ce(Ce), .Rd(Read), .Wr(Write),
                .Data(DataBus), .Addr(Addr));
  initial
  begin : stimulus
     integer NErrors;
     integer i;
// 错误开始记数
     NErrors = 0;
// 为每个地址写上地址值
```

```
     for ( i=0; i<=MaxAddr; i=i+1 )
        WriteRam(i, i);
// 读且比较
     for ( i=0; i<=MaxAddr; i=i+1 )
     begin
        ReadRam(i, Data);
        if ( Data !== i )
            RamError(i,i,Data);
     end
//小结错误个数
      $display(Completed with %0d errors, NErrors);
end

task WriteRam;
   input [AddrWidth-1:0] Address;
   input [DataWidth-1:0] RamData;
begin
   Ce = 0;
   Addr = Address;
   Data = RamData;
   #10 Write = 1;
   #10 Write = 0;
   Ce = 1;
end
endtask

task ReadRam;
   input [AddrWidth-1:0] Address;
   output [DataWidth-1:0] RamData;
begin
   Ce = 0;
   Addr = Address;
   Data = RamData;
   Read = 1;
   #10 RamData = DataBus;
   Read = 0;
   Ce = 1;
end
endtask

task RamError;
   input [AddrWidth-1:0] Address;
   input [DataWidth-1:0] Expected;
   input [DataWidth-1:0] Actual;
```

```
        if ( Expected !== Actual )
        begin
           $display("Error reading address %h", Address);
           $display(" Actual %b, Expected %b", Actual,
                         Expected);
           NErrors = NErrors + 1;
        end
        endtask

endmodule
```

请参阅：

Task Enable，Function 的说明。

Task Enable　任务的启动

在模块代码中只需用任务名便可启动任务。当任务启动时，输入值通过任务的端口变量（输入和 inout变量）传递到任务中。当任务结束时，返回值通过任务的端口寄存器变量（输出和 inout变量）传出。

语法

```
TaskName[( Expression,...)];
```

规则

- 任务可以从 **initial** 或 **always** 块或其它任务中启动。任务可以多次调用。但任务不能被函数调用。
- 调用任务的语句中，端口表达式的顺序和任务端口变量声明的顺序必须一致。端口的个数必须与任务声明的端口变量的个数一致。
- 若任务的端口变量是输入时，则对应的端口变量可以是任何一种表达式；若端口变量为输出和 **inout** 时，对应的端口变量必须位于进程赋值语句的左边而且必须是有效的。
- 当任务启动时，输入和 **inout** 表达式复制到相应的变量寄存器中。当任务结束时，输出和 inout 寄存器的值会复制到启动任务相应的端口寄存器中。
- 可以在任务内部或任务外部把任务禁止（**disable**）。

注意！

任务中变量寄存器默认为静态的，所以当一个任务正在执行时又启动该任务时，输入和 **inout** 寄存器的值会被覆盖。

可综合性问题：

若任务不包含定时控制，是有可能被综合的。调用的任务往往被综合成组合逻辑。

举例说明：

```
    task Counter;
        inout [3:0] Count;
        input Reset;
        ...
    endtask

 always @(posedge Clock)
      Counter(Count, Reset);
```

请参阅：

Disable、Task、Function Call 的说明。

Timing control　　定时控制

用于延迟语句的执行或按排语句的执行顺序。定时控制可以放在语句的前面，或者在程序的进程赋值语句表达式中的赋值操作符（即 = 或 <= ）之间。前一种延迟语句的执行，后一种延迟声明的语句生效。

语法：

```
    {Timing controls before statements}
     {either}
     DelayControl
     EventControl
     WaitControl
    {Intra-assignment Timing controls}
    {either}
     DelayControl
     EventControl
     repeat ( Expression) EventControl
     DelayControl = {either}
     # UnsignedNumber
     # ParameterName
     # ConstantMinTypMaxExpression
     # ( MinTypMaxExpression)
     EventControl = {either}
     @Name {of Register, Net or Event}
     @( EventExpression)
     EventExpression = {either}
     Expression
     Name {of Register, Net or Event}
     posedge Expression {01, 0X, 0Z, X1 or Z1}
```

```
negedge Expression {10, 1X, 1Z, Z0 or X0}
EventExpression or EventExpression
WaitControl = wait ( Expression)
```

在程序中位于何处：

请参阅：statement、procedural assignment (for intra-assignment timing control) 的说明。

规则：

- 在某声明语句前面插入的事件或延迟控制使原本立刻要执行的该条语句延迟执行。
- 当执行到wait时，如果其表达式为假（0或X），wait 控制只延迟 wait 语句后的下一条语句；当表达式为真（非0）时，下一条语句才执行。当执行到 wait 时，如果表达式为真，下一句不延迟马上执行。
- 执行进程赋值语句时，要检查赋值语句右边的表达式，如果没有内部赋值延迟，若用的是阻塞赋值，则左边的寄存器类型变量立即更新，若用的是非阻塞赋值，则在下一个仿真周期更新。如果有内部赋值延迟，左边的寄存器类型变量只有在发生内部赋值延迟后才更新。
- 内部赋值延迟必须是常数的，但语句前的延迟可以是常数或变量（即Net或reg 型变量）。
- or 列表中的任何一个信号（事件）变化（发生）时，即触发事件控制。
- 对于 posedge（上升沿）和 negedge（下降沿）事件触发控制，只测试表达式的最低位。要不然的话，表达式的任何变化都会触发事件。

注意！

对于阻塞赋值语句而言，指定内部赋值延迟为零（# 0）与不指定是不一样的，也与没有赋值延迟的非阻塞赋值语句不同。 对于阻塞赋值语句而言，指定#0意味着该语句在所有待定事件完成以后，而在非阻塞赋值完成以前进行。（不指定内部赋值延迟和指定内部赋值延迟为零（#0）的赋值语句是一样的。）

可综合性问题：

- 综合时延迟被忽略。
- 综合工具不支持 wait语句和内部赋值延迟以及repeat(重复)语句。
- 事件控制用于控制always块的执行，从而能确定综合出的逻辑是组合的还是时序的。一般情况下，always后紧跟着的就是事件控制，这有时也称为敏感列表。

提示：

在用RTL（寄存器传输级HDL语言）描述电路时，可用内部赋值延迟可来描述在给表示触发器的寄存器变量赋值时的时钟偏移现象。

举例说明：

```
#10
#(Period/2)
#(1.2:3.5:7.1)
@Trigger
@(a or b or c)
```

```
@(posedge clock or negedge reset)
wait (!Reset)
```

非阻塞赋值时使用延迟来克服时钟的偏移：

```
always @(posedge Clock)
        Count <= #1 Count + 1;
```

在周期时钟的第五个下降沿复位：

```
initial
    begin
        Reset = repeat(5)  @(negedge Clock)  1;
        Reset = @(negedge clock) 0;
End
```

请参阅：

Procedural Assignment、Always、Repeat语句的说明

User Defined Primitive　　用户自定义原语

用户自定义原语（UDPS）可以为小型元件建立模型，这也是模块的另一种表示方法。可以用引用由门构建的实例同样的方式来实例引用用户自定义原语（UDPS）。

语法：

```
primitive UDPName (OutputName, InputName,...);
   UDPPortDeclarations ...
   UDPBody
endprimitive
UDPPortDeclaration = {either}
  output OutputName;
  input InputName,...;
  reg OutputName; {Sequential UDP}
UDPBody = {either} CombinationalBody SequentialBody
CombinationalBody =
  table
    CombinationalEntry...
  endtable
SequentialBody =
  [initial OutputName = InitialValue;]
  table
     SequentialEntry...
  endtable
InitialValue =
```

```
  {either} 0 1 1'b0 1'b1 1'bx {not case sensitive}
CombinationalEntry = LevelInputList : OutputSymbol ;
SequentialEntry =
  SequentialInputList : CurrentOutput : NextOutput;
SequentialInputList = {either}
   LevelInputList
   EdgeInputList
LevelInputList = LevelSymbol...
EdgeInputList =
   [ LevelSymbol...] EdgeIndicator [ LevelSymbol...]
CurrentOutput = LevelSymbol
NextOutput = {either} OutputSymbol
EdgeIndicator = {either}
    ( LevelSymbol LevelSymbol)
    EdgeSymbol
OutputSymbol = {either} 0 1 x {not case sensitive}
LevelSymbol = {either} 0 1 x ? b {not case sensitive}
EdgeSymbol = {either} r f p n * {not case sensitive}
```

规则：

- UDP只允许有一个输出端，至少允许有一个输入端。具体实施时，对输入端的个数是有限制的，但必须至少允许10个输入端口。
- 如果某UDP的输出端定义为reg 型（寄存器类型）变量，则该UDP是时序逻辑的UDP,否则为组合逻辑的UDP。
- 如果已对时序逻辑的UDP的输出进行了初始化，则只有待到在仿真开始时，初始值才开始从引用的原语实例的输出传出。
- 描述时序逻辑的UDPS可以是电平敏感的或边沿敏感的。若在真值表中有边沿敏感的指示（至少一个），则该描述时序逻辑的UDP为边沿敏感的。
- UDP的行为在表中定义。表的行定义为不同输入条件下的输出。对于描述组合逻辑的UDP，每一行定义为一个或多个输入的组合逻辑的输出。对于描述时序逻辑的UDP，每一行都要考虑reg类型变量的当前输出值。一行最多只能有一个边沿变化入口。行定义了在指定的边沿发生变化时，由输入值和当前输出值所产生的输出值。
- UDP表中所用的特殊的电平和边沿符号含义如下：

?	0 1 或 x

b or B	0 或 1
-	输出不变
(vw)	由 v 变为 w
r or R	(01)
f of F	(10)
p or P	(01)(0x)或(x1)
n or N	(10)(1x)或(x0)
*	(??)

- 若组合逻辑的输入值和触发边沿没有明确指定将会导致输出的不确定。
- 不支持Z值。输入时Z看成是X；输出值不允许设为X。注意 ？符号的特殊含义，它和在数字中的 ？符号意思不一样，在数字的表示中符号？和 z 含义相同。

注意！

在描述时序逻辑的UDP中，若在表中任何地方出现边沿触发的条件，则输入信号所有可能的边沿都要认真考虑并列出，因为默认的只是一种边沿的触发的条件，这将导致输出的不确定性。

可综合性问题：

任何一种工具都不能综合UDP，它只被用来建立基本的门级逻辑器件的逻辑仿真模型。

提示：

- 和行为模块相比较，用UDP来做仿真非常有效。为ASIC单元库的元件建立模型时应该使用UDP。
- 输出端口在一个以上的元件，应该对每个输出建立独立的UDP。
- 在表的第一行加上注释，指明每一列的含义。

举例说明：

```
primitive Mux2to1 (f, a, b, sel); // 组合UDP
  output f;
  input a, b, sel;
  table
// a b sel : f
    0 ? 0   : 0;
    1 ? 0   : 1;
    ? 0 1   : 0;
    ? 1 1   : 1;
    0 0 ?   : 0;
    1 1 ?   : 1;
  endtable
endprimitive

primitive Latch (Q, D, Ena);
  output Q;
  input D, Ena;
```

```
    reg Q; // Level sensitive UDP
    table
  // D Ena : old Q : Q
      0 0   :  ?      : 0;
      1 0   :  ?      : 1;
      ? 1   :  ?      : -; // 保持原先值
      0 ?   :  0      : 0;
      1 ?   :  1      : 1;
   endtable
  endprimitive

  primitive DFF (Q, Clk, D);
     output Q;
     input Clk, D;
     reg Q; // Edge sensitive UDP
     initial
       Q = 1;
      table
           / / Clk  D : old Q : Q
               r    0 :  ?    : 0; // Clock '0'
               r    1 :  ?    : 1; // Clock '1'
               (0?) 0 :  0    : -; // Possible Clock
               (0?) 1 :  1    : -; // " "
               (?1) 0 :  0    : -; // " "
               (?1) 1 :  1    : -; // " "
               (?0) ? :  ?    : -; // Ignore falling clock
               (1?) ? :  ?    : -; // " " "
                ?   * :  -    : -; // Ignore changes on D
      endtable
endprimitive
```

请参阅：

Module、Gate、Instantiation 的语法说明。

While　条件循环语句

只要控制表达式为真（即不为零），循环语句就重复进行。

语法

```
  wile {Expression}
     Statement
```

可综合性问题：
只有当循环块有事件控制（即@(posedge Clock)）才可综合。

举例说明：

```
reg [15:0]  Word
while (Word)
begin
  if  (Word[0])
     CountOnes = CountOnes + 1;
  Word = Word >>1;
End
```

请参阅：
For、Forever、Repeat 语句的说明。

Compiler Directives　　编译器指示

编译器指示是在源代码中对Verilog 编译器所发出的指令。在编译指示需要用反引号（`）做前导。编译器指示从它在源代码出现的地方开始生效，并一直继续生效到随后运行的所有的文件，直到编译器指示结束的地方或一直运行的最后的文件。

下面有Verilog 编译指示的摘要。摘要后面详细介绍了一些比较重要的编译指示。

注意!
编译器指示的生效依赖于编译时源代码中所包含文件的执行顺序。

Standard Compiler Directives　标准的编译器指示

在Verilog LRM中定义了以下编译器指示：

1) `celldefine 和 `endcelldefine
可用来作为分别加在模块的前面和后面的标记，以表示该模块是一个库单元（cell）。单元可被 PLI 子程序调用来做某种应用，比如延迟的计算。
例子:

```
`celldefine
module  Nand2 {...};
 . . .
endmodule
```

`endcelldefine

2) `default_nettype

改变Net类型的默认类型。如果没有该声明，默认的Net类型是wire 型。

例子: `default_nettype tril

3) `define 和 `undef

`define 定义一个文本宏，`undef 取消已定义的文本宏定义。

在编译的第一阶段期间, 宏（macro）被它所定义的文本字符串取代。宏也可以用来控制条件编译（请参阅 `ifdef)。想要知道关于 `define 应用的更多细节见下面说明。

4) `ifdef, `else 和 `endif

根据是否定义了特殊的宏，来指示编译器是否要编译这一段 Verilog 源代码。详细细节见下面。

5) `include

指示编译器读入包含文件的内容，并在`include所在的地方编译该文件。

例子: `include “definitions.v”

6) `resetall

把现行的已启动的所有编译器指示复位到原默认值。该编译指示可以写在每个Verilog源文件的第一行，以防止前面别的源文件的编译指示在该源文件编译时产生不需要的结果。

例子: `resetall

7) `timescale

定义仿真的时间单位和精度。细节请见下面说明。

8) `unconnected_drive 和 `nounconnected_drive

`unconnected_drive 编译指示把模块没连接的输入端口设置为上拉 pull up（pull1，即逻辑1）或为下拉 pull down（pull0, 即逻辑0)。`nounconnected_drive 编译指示把模块没连接输入端口的设置恢复到默认值，即把没连接的输入端口值设置为高阻浮动（Z)。

例子: `unconnected_drive pull0 //或 pull1 (即逻辑值为1)

Non-Standard Compiler Directives 非标准编译器指示

下面的编译指示并不属于Verilog HDL 语言的IEEE标准。但在CADENCE公司的Verilog LRM中提及。并不是所有的Verilog工具都支持以下这些编译指示。

1) `default_decay_time

若未明确给定衰减时间，则由该编译指示将其设置为默认的三态寄存器(trireg) 类型的线路连接（Net）的衰减时间。

例子:

```
`default_decay_time  50
`default_decay_time  infinite   //表示无衰减时间
```

2) `default_trireg_strength

把三态寄存器(trireg) 类型的线路连接（Net）的默认强度设置为整数。用整数来表示强度并不符合IEEE规定的Verilog语言标准，但仍属于Verilog语言非标准扩展部分。

例子：

```
`default_trireg_strength  30
```

3) `delay_mode_distribute、`delay_mode_path、`delay_mode_unit 和`delay_mode_zero 这些编译指示都会影响延迟的仿真方式。分布式延迟是在原语实例中的延迟、赋值延迟和线路连接延迟。路径延迟是在Specify(指定)块中定义的延迟。若用单位和零延迟代替分布式延迟和路径延迟将加快仿真的过程，但会丢失真实的延迟信息。在默认情况下，仿真器会自动选择最长的延迟仿真方式，即分布式延迟和路径延迟仿真方式。

4) `define

`define 定义一个文本宏。宏在编译的第一阶段被由它定义的文本所代替。在用参数和函数表达不适合或不允许的情况下，用宏可以提高 Verilog 源代码的可读性和可维护性。

语法：

```
{declaration}
`define Name[(Argument,...)] Text
{usage}
`Name [(Expression,...)]
```

在程序中位于何处：

宏可以在模块内或模块外定义。

规则：

- 像所有的编译指示一样，宏定义在整个文件中生效，除非被后面的`define、`undef 和`resetall编译指示改写或清除。宏定义没有范围的限制。
- 若定义的宏内有参数，即在宏文本中用到参数，则当宏调用时，宏的参数被实际的参数表达式所代替。

```
`define add(a , b)  a + b
  f = `add(1, 2);              //f= 1 + 2;
```

- 宏定义可以用反斜杠（\）跨越几行。新的一行是宏文本中的一部分。
- 宏文本不允许分下列语言记号：注释，数字，字符串，名称，保留名称，操作符。
- 不能把编译器指示名用作宏名。

注意！

- 所有的具体电路实现工具都不支持带参数的宏。
- 若定义了宏，则必须把撇号（`）写在宏名的紧前面才能调用该宏。没有撇号（`）打头的名，即使名称与宏名一致，则为独立的标识符与宏定义无关。
- 要区别撇号（`）和表示数制的前引号（‘）的不同。
- 不要用分号来结束宏定义，除非真要在用宏代替分号。否则会引起语法错误。

提示：

- 通常更喜欢用参数而不是用宏给无含义的字符起一个有含义的名字。
- 仿真时，用带参数的宏要比用同样功能的函数效率高。

举例说明：

本例子说明在分层设计中如何用文本宏来选择不同的模块实现。这在综合时很有用，特别是当必须用RTL源代码模块和已综合成门级电路的模块做混合仿真时。

```
`define  SUBBLOCK1 subblock1_rtl
`define  SUBBLOCK2 subblock2_rtl
`define  SUBBLOCK3 subblock3_gates
module TopLevel ...
  `SUBBLOCK1 sub1_inst (...);
  `SUBBLOCK2 sub2_inst(...);
  `SUBBLOCK3 sub3_inst(...);
 ...
endmodule
```

下面的例子说明带参数的文本宏的定义和调用：

```
`define nand (delay)  nand #(delay)
 nand(3)  (f,a,b);
 nand(4)  (g,f,c);
```

请参阅：

‘ifdef 的说明。

5) `ifdef

根据是否定义了特定的宏，来决定是否编译这部分Verilog源代码。

语法：

```
`ifdef MacroName
   VerilogCode...
[ `else
  VerilogCode...]
`endif
```

规则：

- 如果宏名已经用`define定义，只编译Verilog编码的第一块。
- 如果宏名没有定义和`else 指示出现，只编译第二块。
- 这些编译指示是可以嵌套的。
- 没被编译的代码仍然必须是有效的 Verilog 代码。
-

提示：

这些编译指示可以用来调试模块。例如，可以在同一个模块的两种形式之间切换（如布线前仿真模块和带布线延迟的门级仿真模块之间）或有选择地开启诊断信息的打印输出。

例子

```
`define primitiveModel
```

```
module Test
...
`ifdef primitiveModel
  Mydesign_primitives  UUT (...);
`else
  Mydesign_RTL UUT(...);
`endif
endmodule
```

请参阅：

`define的说明。

6) `timescale

定义时间单位和仿真精度

语法：

```
`timescale TimeUnit / PrecisionUnit
TimeUnit = Time Unit
PrecisionUnit = Time Unit
Time = {either} 1 10 100
Unit = {either} s ms us ns ps fs
```

规则：

- 像所有的编译指示一样，`timescale影响在该指示后的所有模块，无论位于同一个文件的还是位于独立编译的多个文件中的模块，直到碰到下一个`timescale或`resetall指示将其改写或复位到默认为止。
- 精度单位必须小于或等于时间单位。
- 仿真器运行的精度就是在`timescale指示中所定义的最小精度单位。所有的延迟时间都以精度单位为准取整。

提示：

在每个模块文件的第一句应写上`timescale指示，即使在模块中没有延迟，也是如此，因为有的仿真器必需要有`timescale指示才能正常工作。

举例说明：

```
`timescale  10ns / 1ps
```

请参阅：

$timeformat 的说明。

System task and function

系统任务和函数

Verilog语言包含一些很有用的系统命令和函数。用户可以像自己定义的函数和任务一样调用它们。所有符合IEEE标准的Verilog工具中一定都会有这些系统命令和函数。CADENCE公司的Verilog 工具中还有另外一些常用的系统任务和函数，它们虽并不是标准的一部分，但在一些仿真工具中也经常见到。

请注意，各种不同的 Veriog 仿真工具可能还会加入一些厂商自己特色的系统任务和函数。用户也可以通过编程语言接口（PLI）把用户自定义的系统任务和函数加进去，以便于仿真和调试。

所有的系统任务和系统函数的名称（包括用户自定义的系统任务），前面都要加$以区别于普通的任务和函数。下面是Verilog 工具中常用的系统任务和函数的摘要。详细资料在后面介绍。

标准的系统任务和函数

Verilog HDL的IEEE标准中包括下面的系统任务和函数：

- **$display, $monitor, $strobe, $write 等**
  用于把文本送到标准输出和或写入一个或多个文件中的系统任务。详细说明在后面介绍。

- **$fopen 和$fclose**
  $fopen(“FileName”);　　{Return an integer}
  $fclose(Mcd);
  $fopen 是一个系统函数，它可以打开文件为写文件做准备。
  而$fclose也是一个系统函数，它关闭由 $fopen 打开的文件。
  有关的详细说明在后面介绍。

- **$readmemb 和 $readmemh**
  $readmemb(“File”, MemoryName [,StartAddr[,FinishAddr]]);
  $readmemh(“File”, MemoryName [,StartAddr[,FinishAddr]]);
  把文本文件中的数据赋值到存储器中。有关的详细说明在后面介绍。

- **$timeformat[(Units,Precision,Suffix,MinFieldWidth)];**
  定义用$display等显示仿真时间的格式。有关的详细说明在后面介绍 。

- **$printtimescale**
  $printtimescale([ModuleInstanceName]);
  以如下格式显示一个模块的时间单位和精度：

Time scale of (module_name) is unit /precision。
如果没有参数，则显示模块的时间单位和精度。

- **$stop**

  $stop[(N)]; {N is 0,1,2}
  暂停仿真。可选的参数决定诊断输出的类型。 0 输出最少， 1 个多点， 2 输出最多。

- **$finish**

  $finish[(N)]; {N is 0,1,2}
  退出仿真，把控制权返回给操作系统。如果给出参数N，则根据N值打印不同的诊断信息，见下面的解释：
  - 0 ——不打印
  - 1 ——打印仿真时间和地点（默认值）。
  - 2 ——打印仿真时间和地点和仿真所使用的CPU时间和内存的统计数据。

- **$time,$stime,和$realtime**

  $time;
  $stime;
  $realtime;
  系统函数返回仿真的当前时间值。返回时间值的单位由调用该系统函数语句的模块的`timescale定义。
  - $time 返回一个根据时间单位四舍五入取整的64位无符号整数。
  - $stime返回一个截去高位保留低32位的无符号整数。
  - $realtime 返回一个实数。

  请注意，这些系统函数没有输入，与Verilog的其它函数不同。

- **$realtobits 和 $bitstoreal**

  $realtobits(RealExpression) {return a 64 bit value}
  $bitstoreal(BitValueExpression) {return a real value}
  实数和用位（bit）表示的数之间的互相转换。因为模块的端口不允许传输实数，故需要把实数转换为用位表示的数后才能输入/输出模块。请参阅Module的说明。

- **$rtoi 和 $itor**

  $rtoi(RealExpression) {return an integer}
  $itor(IntegerExpression) {rerurn a real number}
  实数和整数之间的互相转换。$rtoi把实数截断后转换为整数。$itor把整数转换为实数。

- **随机数产生函数**

  1) $random[(Seed)];
  2) $dist_chi_square(Seed, DegreeOfFreedom);
  3) $dist_erlang(Seed, K_ stage, Mean);
  4) $dist_exponential(Seed, Mean);
  5) $dist_normal(Seed, Mean, StandardDeviation);
  6) $dist_poisson(Seed, Mean);

7) $dist_t(Seed, DegreeOfFreedom);
8) $dist_uniform(Seed, Start, End);

当重复调用上述函数时，根据不同概率分布的随机数产生函数条件返回其相应的随机数序列。若伪随机序列的源种相同，则伪随机序列也总是一样的。请参考关于概率与统计理论的教科书，详细了解其中分布函数及其应用部分。

- **Specify Block Timing Checks 指定块内的定时检查系统任务**

1) $hold(ReferenceEvent, DataEvent, Limit [, Notifier]);
2) $nochange(ReferenceEvent, DataEvent,
StartEdgeOffset, EndEdgeOffset [, Notifier]);
3) $period(ReferenceEvent, Limit [, Notifier]);
4) $recovery(ReferenceEvent, DataEvent, Limit [, Notifier]);
5) $setup(DataEvent, ReferenceEvent, Limit [, Notifier]);
6) $setuphold(ReferenceEvent, DataEvent,
SetupLimit, HoldLimit [, Notifier]);
7) $skew(ReferenceEvent, DataEvent, Limit [, Notifier]);
8) $width(ReferenceEvent, Limit [, Threshold [, Notifier]]);

以上8个系统任务均为常用的定时检查系统任务。这些专用的系统任务只能在specify block（指定块）里被调用，详细说明请参阅后面的材料。

- **Value Change Dump Tasks 储存数值变化的系统任务**

1) $dumpfile(“FileName”);
2) $dumpvars[(Levels, ModuleOrVariable,...)];
3) $dumpoff;
4) $dumpon;
5) $dumpall;
6) $dumplimit(FileSize);
7) $dumpflush;

以上七个系统任务用于把数值的变化储存到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序（例如一个波形显示程序）的一种手段。详见后面。

非标准的系统任务和函数

以下这些系统任务和函数在 CADENCE 公司的 Verilog 工具中有，但它们并不属于 IEEE 标准必须包括的范围。其中有部分系统任务和函数与 Verilog 仿真工具操作时的交互方式有关。如果仿真工具支持交互方式的操作，则接受这些系统任务和函数作为其指令。

$countdrivers

$countdrivers (Net, [IsForced, NoOfDrivers, NoOfDriversTo0,
NoOfDriversTo1, NoOfDrivrsToX]);

该系统函数能返回某指定的 Net 类型标量或 Net 型矢量的某个选定位上的驱动器个数。驱动器包括原语的输出和连续赋值语句（强迫（force）启动的除外）。若 Net 含有一个以上的驱动器时，该系统函数（$countdrivers）返回 0；其他情况下返回 1。在该系统任务中除第一个变量外，其余的都返回整型数。若 Net 为 force ，则 IsForced 返回 1 ，否则返回 0 。

NoOfDrivers 返回驱动器个数。其他变量返回数的总和等于 NoOfDrivers 。

- **$list**

  $list [(ModuleInstance)];

在交互模式中调用此系统函数可列出在本设计中当前（或指定）范围内的源程序。.

- **$input**

  $input(“FileName”);

从某个文本文件中读出交互命令。

- **$scope and $showscopes**

  $scope(ModuleInstance);

  $showscopes[(N)];

本系统命令用于在交互模式中设置和显示当前范围，若给定 N 并为非零，则还显示下面的范围。

- **$key, $nokey, $log and $nolog**

  $key[(“FileName”)];

  $nokey;

  $log[(“FileName”)];

  $nolog;

“key”文件记录用交互方式输入的命令，“log”文件记录在仿真期间所有写入标准设备的信息，而运行 $nokey 和 $nolog 系统任务可分别禁止这两项功能。用$key 和$log（无参数）可恢复其记录功能。如有参数，则$key 和$log 创建新的记录文件。

- **$reset、$reset_count 和 $reset_value**

  $reset[(StopValue[, ResetValue[, DiagnosticsValue]]);

  $reset_count; {Returns an integer}

  $reset_value; {Returns an integer}

系统任务$reset 使仿真器复位，并使仿真从头重新开始执行。StopValue 为 0 表示仿真器复位到交互模式，允许用户自己来启动和控制仿真。而非 0 值表示仿真将会自动地从头开始仿真。ResetValue 的值可以通过$reset_value 系统函数读出。DiagnosticsValue 是指复位前仿真工具所显示信息的类型。$reset_count 返回已调用$reset 系统任务的次数。$reset_value 返回传给$reset.系统任务的值。

- **$save、$restart 和 $incsave**

  $save(“FileName”);

  $incsave(“FileName”);

  $restart(“FileName”);

$save 将完整的仿真状态保存在文件中，$restart.可以读出保存的文件。$incsave 只保存自上次调用$save 后的变化。$restart 将仿真复位并把完整的或只记录变化的文件读出。若是$restart 只记录变化的文件，原完整的仿真状态记录文件必须存在，记录变化的文件会引用完整的仿真状态文件。

- **$showvars**

  $showvars[(NetOrRegister,...)];

在标准输出设备显示 Net 和寄存器的状态。这个系统任务用于交互模式。所显示的状态信息

在 Verilog LRM 工具中未作定义。状态信息可以包括当前的 Net 和寄存器值、这些 Net 和寄存器上的预定事件、以及 Net 的驱动器。如果未给出变量表，将显示所有当前范围的 Net 和寄存器。

- **$getpattern**

```
$getpattern( MemoryElement);
```

$getpattern 是一个只能用于连续赋值语句的系统函数，连续赋值语句的左边必须为 Net 类型标量的位拼接。$getpattern 常与$readmemb 和 $readmemh 一起使用，可从文本文件中提取测试矢量。当有大量的标量需要输入时，$getpattern 能提供快速的处理。

- **$sreadmemb and $sreadmemh**

```
$sreadmemb (Memory, StartAddr, FinishAddr, String, ...);
$sreadmemh (Memory, StartAddr, FinishAddr, String, ...);
```

这两个任务与$readmemb 和$readmemh 类似，只是存储器中的初始数据不是由文件输入，而是由一个或多个字符串输入。字符串格式与 $readmemb 和$readmemh 系统任务所要求的相应文件格式一致。

- **$scale**

```
$scale(DelayName); {Returns realtime}
```

将一模块的时间值转换为调用$scale 系统任务的模块中所定义的时间单位来表示。$scale 可以引用用模块层次命名的参数（如延迟值），并将它转换为调用$scale 的模块中所定义的时间单位来表示。

常用系统任务和函数的详细使用说明

$display 和 $write

把格式化文本输出到标准输出设备及仿真器日志或其他文件。

- **语法：**

```
$display( Argument,...);
$fdisplay( Mcd, Argument,...);
$write( Argument,...);
$fwrite( Mcd, Argument,...);
Mcd = Expression {Integer value}
```

- **规则：**

$display与 $write的唯一区别为前者在输出结束后会自动换行而后者不会自动换行。Arguments 可以是字符串或表达式或空格（,,）。字符串内可包含以下格式控制符。若包含格式控制符（%m 除外），则每个字符串后必须有足够的表达式来为字符串中的格式控制符提供数值。

字符串中也可包含以下扩展字符：

- \n 换行（Newline）
- \t 制表符（Tab）
- \” 双引号
- \\ 反斜杠
- \nnn 用八进制表示的 ASCII 字符

不定值和高阻值这样表示（注意：八进制数的每一个数字代表 3 位，而十制数和十六进制数的每一个数字代表 4 位）：对十进制数而言，若有某个数字为不定值和高阻值则写作 x，z，X，Z。若用大写的 X，Z 表示，则该数字中并非所有位（bit）为不定值和高阻值，若用小写 x，z 表示，则表示该数字中所有位（bit）为不定值和高阻值时。若参数表含二相邻逗号，则输出显示或打印一空格。

- **格式控制符**

在字符串中允许出现下面这些格式控制符：

1) %b %B 二进制数（Binary）
2) %o %O 八进制数 （Octal）
3) %d %D 十进制数 （Decimal）
4) %h %H 十六进制数（Hexadecimal）
5) %e %E %f %F %g %G 实型数（Real）
6) %c %C 字符（Character）
7) %s %S 字符串（String）
8) %v %V 二进制数和强度（Binary and Strength）
9) %t %T 时间类型数（Time）
10) %m %M 分级实例名 （Hierarchical Instance）

格式控制符 %v 按如下的形式打印出变量的强度值：若强度值为supply， 则打印 Su; 若为strong，则打印St；若为Pull ，则打印Pu；若为Large，则打印La；若为Weak，则打印We；若为Medium，则打印Me；若为Small，则打印Sm；若为Highz，则打印Hi。 %v也能把变量值打印为H和 L（这些值若用 %b 格式控制符，则只能打印为X）。% 号后常跟有一个数用于表示打印变量值区域的宽度（例如%10d，表示至少保留 10位宽度给要打印的十进制数）。对十进制数，高位不足此值者以空格代替，其他进制以0代替，若% 号后的数为0，则表示打印变量值区域的宽度随其值的位数自动调节。

Verilog HDL 实型数的格式符(%e, %f and %g) 其格式控制功能和C语言的格式符完全一样。例如，%10.3g指至少保留 10位宽度给要打印的十进制数，小数点后还保留3个数字位。
若相应的变量未用格式控制符声明，则默认为是十进制数。有些系统打印任务有其自己的缺省值，如$displayb, $fwriteo，$displayh 的缺省值分别是二进制，八进制，十六进制。

- **举例说明：**

```
$display ( “Illegal opcode %h in %m at %t”, Opcode, $realtime);
$writeh  (“Register values (hex .): “ , reg1 , ,  reg2, ,  reg3, ,  reg4, ” \n”);
```

请参阅：

$monitor, $strobe 的说明

$fopen and $fclose

$fopen 是用于打开某个文件并准备写操作的系统任务，而 $fclose 则是关闭文件的系统任务。把文本写入文件还需要用$fdisplay、$fmonitor 等系统任务。

语法：

```
$fopen(“FileName”); {Returns an integer}
$fclose( Mcd);
Mcd = Expression {Integer value}
```

在程序中位于何处：

请参阅 Statement 的说明;

规则：

- 一般情况下一次最多可打开32个文件，但若所用的操作系统不同，一次最多可打开的文件数可能不到32。当调用$fopen时，它返回一个32 位（bit）（与文件有关）的无符号多通道描述符或者返回0值，0值表示文件不能打开。多通道描述符可以被认为是32个标志每个代表32个文件中的一个。多通道描述符的第0位与标准输出设备有关，第1位为第 1 个文件打开的标志位，第 2 位为第 2 个文件的标志位，依次类推。当输出文件的系统任务，如$fdisplay，被调用时，其第一个参数为多通道描述符，它表示向何处写。文本被写入那些多通道描述符内标志位已设的相应文件中。

举例说明：

```
integer MessagesFile, DiagnosticsFile, AllFiles;
    initial
        begin
            MessagesFile = $fopen(“messages.txt”);
            if (!MessagesFile)
                begin
                    $display(“Could not open \”messages.txt\””);
                    $finish;
                end
            DiagnosticsFile = $fopen(“diagnostics.txt”);
            if (!DiagnosticsFile)
                begin
                    $display(“Could not open \”diagnostics.txt\””);
                    $finish;
                end

        AllFiles = MessagesFile | DiagnosticsFile | 1;
        $fdisplay(AllFiles, “Starting simulation ...”);
```

```
        $fdisplay(MessagesFile, “Messages from %m”);
        $fdisplay(DiagnosticsFile, “Diagnostics from %m”);
            ...
        $fclose(MessagesFile);
        $fclose(DiagnosticsFile);
    end
```

请参阅：

$display, $monitor, $strobe 的说明。

$monitor 等

当$monitor 系统任务所指定的参数表中中任何一个或多个 Net 或寄存器类型变量值发生变化时，便立即显示一行文本。此系统任务常用于测试模块中，以监测仿真行为的细节。

语法：

```
$monitor( Argument,...);
$fmonitor( Mcd, Argument,...);
$monitoron;             {turns monitor flag on}
$monitoroff;        {turns monitor flag off}
Mcd = Expression   {Integer value}
```

规则：

上面这些系统任务在变量使用的语法上与$display 系统任务完全相同。有一点与$display 系统任务不同：只能同时运行一个$monitor 系统任务。但$fmonitor 系统任务却能同时运行多个。第二次或下一次调用$monitor 系统任务，就把上一次正在执行的$monitor 系统任务取消了，用新的$monitor 系统任务取而代之。$monitoroff 系统任务关闭监视的功能，而$monitoron 则恢复监视的功能，它能把现存的$monitor 进程所监测到的信号不管其值是否变化立即显示出来。对$fmonitor 而言，没有与之对应的$monitoron 和 $monitoroff 系统任务。系统函数$time、 $stime 和 $realtime 不会从$monitor 或 $fmonitor 等系统任务触发出一行显示。

提示：

在测试模块里使用$monitor 可以从任何一种 Verilog 兼容的仿真器获得仿真结果。用于生成波形图显示的任务往往与仿真器相关。

举例说明：

```
initial
$monitor ( “ %t : a = %b, f = %b ”,$realtime, a, f );
```

请参阅：
$display, $strobe, $fopen 语句的说明。

$readmemb 和 $readmemh

把文本文件中的数据读到存储器阵列中，以对存储器变量进行初始化。此文本文件的内容可以是二进制格式（用 $readmemb）的， 也可以是十六进制格式（用$readmemh）的。

语法：

```
{System task call}
$readmemb (“File”, MemoryName [, StartAddr[, FinishAddr]]);
$readmemh (“File”, MemoryName [, StartAddr[, FinishAddr]]);
{Text file}
{either} WhiteSpace DataValue @ Address
WhiteSpace = {either} Space Tab Newline Formfeed
DataValue = {either}
BinaryDigit... {$readmemb}
HexDigit... {$readmemh}
Address = HexDigit...
```

规则：

- 第一个参数是 ASCII 文件名，文件中可以包含空格、Verilog 注释语句、十六进制地址和二进制或十六进制数据。
- 第二个参数是存储器阵列名。
- 数据的位宽必须与存储器阵列的每个存储单元的位宽相同,而且每个数据之间必须用空格间隔开。数据被一个挨一个地读入连续相邻的存储器阵列中，从存储器阵列的第一个地址（若指定起始地址，则从指定的起始地址）开始，直到数据文件结束或直到存储器阵列的最后一个地址（若指定结束地址，则到指定的结束地址）为止。
- 地址均用十六进制数字表示且以@符号开头（对$readmemb 亦然）。当遇到一个地址后，下一个文本数据将被读入这个地址的存储单元。

可综合性问题：
不可综合。 综合工具忽略这些系统任务的存在。在可综合的设计里，从存储器阵列导出的触发器不能用这种方法初始化。如果需要上电复位对存储器阵列(RAM)初始化，则必须对其明确地编码。

提示：
存储器阵列可以储存从文本文件读出的激励源。这是把数据读进 Verilog 仿真器的唯一方式，而无须另外使用编程语言接口（PLI）或非标准语言扩展来做到这一点。

举例说明：

```
module Test;
```

```
    reg a,b,c,d;
    parameter NumPatterns = 100;
    integer Pattern;
    reg [3:0] Stimulus[1:NumPatterns];
    MyDesign  UUT (a, b, c, d, f);
    initial
        begin
            $readmemb("Stimulus.txt", Stimulus);
            Pattern = 0;
            repeat (NumPatterns)
                begin
                    Pattern = Pattern + 1;
                    {a,b,c,d} = Stimulus[Pattern];
                    #110;
                    end
        end
    initial
        $monitor("%t a= %b b= %b c= %b d= %b : f= %b", $realtime, a, b, c, d, f);

endmodule
```

$strobe

在所有事件都已处理完毕后的时刻打印出一行格式化的文本。

语法：

```
$strobe( Argument,...);
$fstrobe( Mcd, Argument,...);
Mcd = Expression {Integer value}
```

规则：

本系统任务（$strobe）有关参数以及文本打印的语法与系统任务$display 完全一样。 但$strobe 只打印调用此系统任务的时刻且当所有活动事件都已结束后的信息，其中可包括所有阻塞和非阻塞赋值产生的效果。

提示：

在写仿真激励模块时，若想打印出仿真结果，应优先考虑使用$strobe系统任务。因为与使用$display或$write比较，系统任务$strobe 可以保证显示出写入Net和寄存器类型变量的是一个稳定的数值。

举例说明：

```
    initial
        begin
```

```
        a = 0;
        $display(a); // displays 0
        $strobe(a);  // displays 1 ...
        a = 1;       //... because of this statement
    end
```

请参阅：

$display, $monitor, $write 语句的说明。

$timeformat

定义仿真时间的打印格式。系统任务 $timeformat 应配合格式控制符 %t 使用。

语法：

$timeformat[(Units, Precision, Suffix, MinFieldWidth)];

规则：

- Units（单位）是指打印的时间单位，它是一个 0 到 -15 之间整型数，0 表示秒（s），-3 表示毫秒（ms），-6 表示微秒（us），-9 表示纳秒（ns），-12 表示皮秒（ps），-15 表示浮秒（femtosecond），中间的整数也可用，如 -10 表示 100 皮秒（ps），依此类推。
- Precision 是指打印的十进制数小数点后保留的位数。
- Suffix 指打印时间值后跟的字符串。
- MinFieldWidth 指打印出的字符的最少个数，其中包括前面的空格。若需要打印的字符多，则需要取较大的整数。
- 缺省形式，即不指定参数，自动设置为：Units（单位）为仿真的时间精度；Precision（精度）为 0；Suffix 无；MinFieldWidth 为 20。

提示：

在使用$display、$monitor 或其他显示任务时，应使用 ‘timescale, $timeformat 和 $realtime（并配合 %t）来指定和显示仿真时间。

举例说明：

```
$timeformat  (-10, 2, “ x100ps”, 20);   // 20.12 x100ps
```

请参阅

‘timescale, $display 的说明。

Stochastic Modelling 随机模型

Verilog 提供了一整套系统任务和函数，可用来启动随机序列的生成和管理以支持建立随机

模型。

语法：

```
$q_initialize( q_id, q_type, max_length, status);
$q_add( q_id, job_id, inform_id, status);
$q_remove( q_id, job_id, inform_id, status);
$q_full( q_id, status); {Returns an integer}
$q_exam( q_id, q_stat_code, q_stat_value, status);
```

在程序中位于何处：

请参阅 Statement 的说明。

概论：

所有这些系统任务和函数的参数都是整型数。每个系统任务和函数都返回一整数型的状态（status）值，它为下列值之一：

0- OK
1- 队列已满：不能再增加工作($q_add)
2- 未定义的 q_id
3- 队列空：不能再删除工作($q_remove)
4- 不支持的队列形式：不能创建这个队列($q_initialize)
5- 最大长度小于等于 0：不能创建这个队列($q_initialize)
6- 两个相同的 q_id:：不能创建这个队列($q_initialize)
7- 内存不足：不能创建这个队列($q_initialize)

系统任务 $q_initialize

创建一个队列。q_id (输出)是唯一的队列标识符，当程序需要调用多个队列任务和函数时，可用该标识符来区别各个队列。 q_type (输入)可为 1 或 2，1 表示 FIFO（先进先出）队列，2 表示 LIFO（后进先出）队列。max_length (输入)为队列所允许的最多的输入个数（即最大长度）。

系统任务 $q_add

向队列加进一个入口。q_id (输入)表示向哪个队列加输入口。job_id (输入)表示是哪个工作（job），它通常为一整型数，每次向队列加入一个新元素其值加 1，这样当队列的某一元素需要移走，可以用 job_id 来识别。 inform_id (输入)用于定义与队列入口有关的信息，由用户自己来定义。

系统任务 $q_remove

从队列取一个入口。 q_id (输入)表示从哪个队列取走该入口。 job_id (输出)确定是哪个工作(请参阅 $q_add 的说明)。inform_id (输出)是由$q_add 储存的数值。

系统任务 $q_full

检查队列是否满。若返回值为 1 则队列满；为 0 则不满。

系统任务 $q_exam

取得队列不同类型的统计信息。下列描述中提及的时间是基于何时队列元素被加入到队列中（到达时间）以及队列元素从加入队列到被删除出去的时间差（等待时间）。时间单位为仿真的时间精度。其中 q_stat_value (输出)参数返回取得的消息，而其中 q_stat_code (输入) 参数可以取 1—6，分别表示要求取得的信息类型：

1. 当前队列长度。
2. 平均达到时间间隔。
3. 最大队列长度。
4. 最短等待时间
5. 当前队列中队列元素的最长等待时间。
6. 本队列的平均等待时间。

举例说明：

```
module Queues;
    parameter Queue = 1; // Q_id
    parameter Fifo = 1, Lifo = 2;
    parameter QueueMaxLen = 8;
    integer Status, Code, Job, Value, Info;
    reg     IsFull;

   task Error; // Write error message and quit
       ...
   endtask

   initial
    begin
       // 生成后进先出队列，其标号为为 1, 队列长度为 8。
          $q_initialize (Queue, Lifo, QueueMaxLen, Status);
            if ( Status )
              Error(“Couldn’t initialize the queue”);
       // 向1号队列加入从1号到8号共8个工作，每个job 之间间隔10个单位时间
       // 每次从1号队列加入的信息为job号加100
          for (Job = 1; Job <= QueueMaxLen;  Job = Job + 1)
            begin
                  #10  Info = Job + 100;
                $q_add (Queue, Job, Info, Status);
                  if ( Status )
                      Error(“Couldn’t add to the queue”);
                  $display(“Added Job %0d, Info = %0d”, Job, Info);
                  $write(“Statistics: “);
   /**要求取得有关当前队列长度、平均达到时间间隔、最大队列长度、最短等待时间、
   当前队列中队列元素的最长等待时间，和本队列的平均等待时间共 6 种队列信息 **/

                      for ( Code = 1; Code <= 6; Code = Code + 1 )
```

```
                        begin
                           $q_exam(Queue, Code, Value, Status);
                            if ( Status )
                                Error("Couldn't examine the queue");
                           $write("%8d", Value);  //显示 6 种队列信息
                        end
                    $display("");
                end
    //  队列此时应是满的
        IsFull = $q_full(Queue, Status);
            if ( Status )
                Error("Couldn't see if queue is full");
            if ( !IsFull )
            Error("Queue is NOT full");
    // 去除工作
             repeat (10 )
             begin
               #5  $q_remove(Queue, Job, Info, Status);
               if ( Status )
                    Error ("Couldn't remove from the queue");
               $display("Removed Job %0d, Info = %0d", Job,Info);
               $write("Statistics: ");
               for ( Code = 1; Code <= 6; Code = Code + 1 )
                begin
                    $q_exam(Queue, Code, Value, Status);
                   if ( Status )
                        Error("Couldn't examine the queue");
                    $write("%8d", Value);
                end
               $display("");
            end
     end
endmodule
```

请参阅：

$random、$dist_chi_square 等系统任务的说明。

Timing Checks　定时检查

Verilog 提供了一些系统任务，这些系统任务仅能在“specify block”（ 指定块）里调用，以进行常见的定时检查。

语法：

```
$hold( ReferenceEvent, DataEvent, Limit [, Notifier]);
$nochange( ReferenceEvent, DataEvent, StartEdgeOffset, EndEdgeOffset [, Notifier]);
$period( ReferenceEvent, Limit [, Notifier]);
$recovery( ReferenceEvent, DataEvent, Limit [, Notifier]);
$setup( DataEvent, ReferenceEvent, Limit [, Notifier]);
$setuphold( ReferenceEvent, DataEvent, SetupLimit, HoldLimit [, Notifier]);
$skew( ReferenceEvent, DataEvent, Limit [, Notifier]);
$width( ReferenceEvent, Limit [, Threshold [, Notifier]]);

ReferenceEvent = EventControl PortName [&&& Condition]
DataEvent = PortName
Limit = {either} ConstantExpression SpecparamName
Threshold = {either} ConstantExpression SpecparamName
EventControl = {either}
   posedge
   negedge
edge [ TransitionPair,... ]
TransitionPair = {either}  01 0x 10 1x x0 x1
Condition = {either}
ScalarExpression
~ ScalarExpression
  ScalarExpression == ScalarConstant
  ScalarExpression === ScalarConstant
  ScalarExpression != ScalarConstant
  ScalarExpression !== ScalarConstant
```

规则：

- 参考事件（ReferenceEvent）的变化提供了定时检查的时间基准，参考事件（ReferenceEvent）必须通过模块的输入口（input）或输入/输出口（inout）引入。
- 数据事件（DataEvent）的变化会启动定时检查，数据事件也必须通过模块的输入口（input）或输入/输出口（inout）引入。
- 如果参考事件与数据事件同时发生，这时虽不会产生建立违例报告，但会产生保持违例报告。
- 对于系统任务$width, 脉冲如果低于门限（Threshold）参数的设定（若设置了门限），则不会发生违例报告。
- 下列定时检查系统任务中的参考事件（ReferenceEvent）必须是沿触发的：$width, $period, $recovery, $nochange。
- 以上系统任务中 ReferenceEvent 参数都可以用关键字 **edge**，除了$recovery 和 $nochange 这两个系统任务例外，它们的 ReferenceEvent 参数只能用 **posedge** 和 **negedge**。
- 使用&&&做注释的条件定时检查仅在条件为真时才执行。
- 在以上的系统任务里如果设置了 notifier 参数，则其必须为寄存器类型变量，当违例发生时，寄存器变量数值发生变化：若原为不定值则变为 0，若原为 0 值则变为 1，若

原为 1 则变为 0，若原为高阻则不变。

注意！
这些系统任务仅能在 specify （指定）块中调用 ，而不能用作程序声明语句。
参数 ReferenceEvent 和 DataEvent 在系统任务$setup 里是颠倒的。

提示：
若条件比较复杂，应在指定块（specify block）外描述条件，而把驱动条件的信号（wire 或 reg 类型）放在指定块内。

举例说明：

```
reg Err, FastClock;     // Notifier registers
specify
     specparam Tsetup = 3.5, Thold = 1.5,
     Trecover = 2.0, Tskew = 2.0,
     Tpulse = 10.5, Tspike = 0.5;
     $hold(posedge Clk, Data, Thold);
     $nochange(posedge Clock, Data, 0, 0 );
     $period(posedge Clk, 20, FastClock)];
     $recovery(posedge Clk, Rst, Trecover);
     $setup(Data, posedge Clk, Tsetup);
     $setuphold(posedge Clk &&& !Reset, Data, Tsetup, Thold, Err);
     $skew(posedge Clk1, posedge Clk2, Tskew);
     $width(negedge Clk, Tpulse, Tspike);
endspecify
```

请参阅：
Specify, Specparam 的说明。

Value Change Dump

以下七个系统任务用于把数值的变化储存到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序（例如一个波形显示程序）的一种手段。

语法：

```
   $dumpfile(“FileName”);
   $dumpvars[( Levels, ModuleOrVariable,...)];
   $dumpoff;    {suspend dumping}
   $dumpon;         {resume dumping}
   $dumpall;        {dump a checkpoint}
   $dumplimit ( FileSize);
   $dumpflush;  {update the dump file}
```

在程序中位于何处：

请参阅 Statement 说明。

规则：

- 系统任务$dumpvars 中的参数 Levels 表示想要把指定模块的哪些层次的数值变化记录到 VCD 文件中：若设置为 1 表示仅记录指定层次模块中的变化，0 表示不但记录该层次模块还记录所有与该模块有关的下层模块中的变化。
- 如果没有设置任何参数，则设计中所有变量的变化均记录到 VCD 文件。
- FileSize 参数是用于设置 VCD 文件可记录的最多字节数。
- 在测试程序中可写多个系统任务$dumpvars 调用，但是每个调用必须在同一时刻（通常在仿真开始时）。

举例声明：

```
module Test;
    ...
    initial
        begin
            $dumpfile(“results.vcd”);
            $dumpvars(1, Test);
        end
// Perform periodic checkpointing of the design.
    initial
        forever
            #10000 $dumpall;
endmodule
```

Command Line Options 命令行的可选项

虽然怎样选用启动 Verilog 仿真器工作的命令行可选项并不是的 Verilog 语法的一部分，大多数有关 Verilog 语法学习参考材料里也不提供这方面的资料，但是为了更快掌握仿真工具还是有必要介绍一些这方面的资料，因为绝大多数仿真器都支持一些常见共同的 Verilog 编译命令选项（尽管有的 Verilog 仿真工具还有自己的一些命令选项），熟练地掌握这些共同的选项能更有效地进行仿真，提高 Verilog 仿真工具的使用效果。

UNIX Verilog 编译命令选项分为两类：一类是一个字符的，前面带有一个减号 -（如 -s）；另一类是多个字符的，前面带有一个加号（如 +word）。有些 UNIX Verilog 编译命令选项后还可跟一个值，例如跟一个文件名（如 -f file）。

下面介绍一些最有用和最常见的 Verilog编译命令选项（注意：并非所有仿真器都支持这些选项）：

-f CommandFile	除从命令行输读入命令选项外，还从命令文件读入更多的命令选项
-k KeyFile	在 KeyFile 里记录仿真期间所有键入的交互命令
-l LogFile	除了在显示器输出外还把所有仿真信息（包括$display 等的输出）记录

到 LogFile 中。

-r SaveFile　从由非标准的系统任务$save 生成的文件再次开始仿真。

-s　在 0 时刻中断仿真器，以便采用交互方式来控制仿真的进行。

-u　将 Verilog 源代码中的字符都视为大写字符（字符串除外），用此选项时要小心。

-v LibraryFile　在 LibraryFile 中寻找设计文件中缺少的 UDP 或模块。只有那些在设计文件中并未定义但已实例引用的 UDP 或模块编译器才会在 LibraryFile 中寻找。而设计中没有引用的在 LibraryFile 中 UDP 或模块不会进行编译。

-y LibraryDirectory

在LibraryDirectory中的文件中寻找设计文件中缺少的UDP或模块，期望在该库的目录中有一个文件已定义了一个与其同名的模块。如果给出+libext+扩展名 的命令行选项，则是指在搜寻文件时将带扩展名搜寻。例如，-y mylib + libext+.v 是指在mylib目录中，在扩展名为.v的文件范围内搜寻在设计中未定义的同名的模块。

+define+ MacroName

定义一段文字作宏名（无值）。这种零值的宏可以用于‘ifdef 中来控制（条件）编译的范围。

+incdir+ Directory[+ Directory...]

定义搜索路径，搜索用‘include 包含的文件。搜索开始于当前路径，如果没有找到，就去由+incdir+定义的搜索路径依次去找。

+libext+ Extension

定义库文件扩展名。（请参阅 –y 的说明）

+notimingchecks

关闭指定块的定时检查，此举可以加速仿真，或抑制伪定时错误信息，使用此选项时需小心。

+mindelays, +typdelays, +maxdelays

以上分别是指仿真时使用最小，典型和最大延迟，缺省为使用典型延迟。仿真时不要混淆以上三种延迟。

注意！

Verilog 仿真器不能检查出+号后选项参数的拼写错误，这是因为 Verilog 命令行允许用户自己来定义+号后的参数，所以一定小心注意选项的拼写，例如“ +maxdelays”要注意拼写正确。