Verilog®黄金参考指南

V1.0 1996 年 8 月

DOULOS
Church Hatch,
22 Market Place,
Ringwood.
Hampshire.
BH24 1AW
England.

Tel (+44) (0)1425 471223
Fax (+44) (0)1425 471573
Email：info@doulos.co.uk
URL：http://www.doulos.co.uk

序言

Verilog 黄金参考指南是 Verilog 硬件描述语言及其语法、语义、合并以及将它应用到硬件设计的一个简明的快速参考指南。

Verilog 黄金参考指南并不是要代替 IEEE 的标准 Verilog 语言参考手册。它不像 IEEE 的标准手册提供了 Verilog 完整、正式的描述。相反，黄金参考指南以一种方便的参考格式解答了在 Verilog 的实践应用过程中经常遇到的问题。

Verilog 黄金参考指南也不想成为介绍性的指南。这里所提出的信息是一种扼要的参考格式，而不是学习像 Verilog 这种复杂的主题所必要的渐进和共鸣方式。但必须承认的是已经熟悉计算机语言的人希望将这个参考指南作为 Verilog 的课本，因此在开始本指南就会对这个主题进行一个简单的非正式介绍。

Verilog 黄金参考指南的主要特征是它包含了从许多 Verilog 项目中积聚起来的大量实践知识。指南不仅提供方便的语法参考（很多类似的书也有），还对最常见的语言错误提出警告、在代码不能编译的时候给出线索指出要看什么地方、警告你注意合并问题并对改善你的编码形式提出建议。

Verilog 黄金参考指南是为了给 Doulos 的系列 Verilog 训练课程增值，也是 HDL PaceMaker——Doulos Verilog 计算机基础训练包的补充。

指南的使用

本指南的主体被分成 3 个主要部分，每个部分都按字母顺序组织。每一部分都以每页顶部的关键字词条作为索引。通常，你只要快速翻阅指南查找相应的关键字词条的就可以找到想要的信息。如果找不到，也可以用最后的完整索引查找。

本指南的很多信息都围绕 Verilog 的句法组织，但也有另外一些有关编码标准、设计流程、错误、保留字以及在正文按字母顺序参考部分后面的编译器伪指令、系统任务和函数以及命令行选项等特殊的部分。

如果你是 Verilog 的新手，请先阅读次页的 Verilog 简单介绍。

索引

粗体的索引条目在指南的主体有对应的页。剩下的索引条目按重要性的顺序在字符参考部分的页参考后面。

用于定义 Verilog 语法的记号说明

在任何可能的时候，语法定义要写得和例子相似，但有需要引入一些额外的记号。简单地来说，方括号[ ]包含可选的项目，三个点号…表示重复，花括号{ }包含的是注释。斜体表示语法该部分在别处定义。记号的完整描述如下：

包含注释的花括号{ }所不是被定义的 Verilog 语法的一部分，但它向你提供了有关语法定义的补充信息。粗体的花括号**{ }**是 Verilog 句法的一部分（连接运算符）。

包含在方括号[ ]中的语法是可选的。粗体的方括号是**[ ]**Verilog 语法的一部分（矢量范围、位和部分选择、存储器元素）。

…表示前面的项目或行的重复 0 次或者多次，或者表示一个列表，例如：

Item…表示 Item 重复 0 次或多次。

,…表示在用逗号分隔的列表重复（例如：A,B,C）。

列表中至少有一个项目。列表的结尾没有,号。

小写的字是保留字，是 Verilog 语言的组成部分（例如：module）。

以大写字母开头的字（不是斜体）是 Verilog 的标识符，即用户定义的名字，它们都不是保留的标识符（例如：InstanceName）。

斜体的字是句法范畴，即在别处完整给出句法名字的定义。句法范畴可以在相同的页里定义，也可以在独立的页定义，还可以在下面定义的其中一个特殊范畴中定义。

斜体=表示在同一页定义和使用的句法范畴。

特殊的语法范畴：

MinTypMaxExpression 以 *Expression* 定义。

UnsignedNumber 以 *Number* 定义。

SomethingExpression=Expression，其中 *Something* 给出有关表达式的含意信息（例如：*ConstantExpression*，*ConstantMinTypMaxExpression*）。

Verilog 的简单介绍

下面向不懂 Verilog 语言的读者在技术上简要介绍一下 Verilog。

背景

Verilog 硬件描述语言（HDL）是描述电子电路行为和结构的一种语言，是一种 IEEE 标准（IEEE Std.1364-1995）。

Verilog 用于模拟从随机和纯行为到门级和开关级的抽象范围等层次的数字电子电路功能，也用于从许多抽象（寄存器传输级）描述合并（即自动产生）门级描述。Verilog 一般用于支持高层次的设计（或基于语言的设计），其中电子设计在用自动合并工具进行详细设计前要通过高层次的完全抽象仿真来检验。Verilog 也广泛应用于 IC 的门级检验，包括仿真、故障仿真和定时检验。

Verilog 最初是在 1984 年由 Gateway Design Automation 公司开发 Verilog-XL 仿真器的时候一起开发出来。1989 年，Cadence Design Systems 公司并购 Gateway 公司，同时拥有对 Verilog 语言和 Verilog-XL 仿真器的权力。1990 年，Cadence 将 Verilog 语言（不是 Verilog-XL）放到公共领域。为了使 Verilog 语言通过 IEEE 标准化过程，一个非赢利性组织 Open Verilog International（OVI）将它不断推进，结果在 1995 年 Verilog 成为一个 IEEE 标准。此后，OVI 仍继续不断维护和开发这种语言。

语言

在本部分和指南剩下的部分，以大写字母开头的斜体字都是技术术语，都可以在本指南的主体中找到。

硬件设计的层次部分在 Verilog 中用模块（*Module*）描述。模块定义了硬件单元的接口（即输入和输出）及其内部结构或行为。

大量的原语或者门（*Gates*）都内置在 Verilog 语言内。它们表示基本的逻辑门（例如：and、or）。另外，还可以定义用户定义的原语（*User Defined Primitives*，UDPs）。

电子电路的结构通过在高层模块内对模块和原语（UDPs 和门）举实例（*Instances*）来描述，而且实例之间通过线网（*Nets*）连接。线网表示一个电气连接、一条线路或总线。端口（*Port*）连接列表用于将线网连接到模块的端口或者连接到原语的实例，其中一个端口表示一个管脚。寄存器（*Registers*，见下面）也可以连接到实例的输入端口（只能连接到输入端口）。

线网（和寄存器）的值由逻辑值 0、1、X（未知或未初始化的）和 Z（高阻或悬空）组成。除了逻辑值外，线网还有强度（*Strength*）值。强度广泛地用于开关级模型以及解释网络有超过一个驱动器的情况。

电子电路的行为是用 *Initial* 和 *Aways* 结构以及连续赋值（*Continuous Assignments*）描述。这些结构以及 *UDPs* 和门表示设计的层次树的叶子。每个 *Initial*、*Always*、连续赋值、*UDP* 和门实例相对于所有其他结构是同时执行的，*Initial* 或 *Always* 内的语句（*Statements*）在很多方面上都和软件编程语言的语句相似。它们在用定时控制（*Timing Controls*）规定的时间例如延迟以及用（仿真）事件控制触发执行。语句在 *Begin-End* 块顺序地执行，在 *Fork-Join* 块并行地执行。连续赋值语句修改线网的值。*Initial* 和 *Aways* 修改寄存器的值。*Initial* 或 *Always* 可以分解成有给定变量的指定的任务（*Tasks*）和函数（*Functions*）。Verilog 语言还有大量内置的系统任务（*System Tasks*）和函数（*Functions*）。编程语言接口（*Programming Language Interface*，PLI）是 Verilog 语言的一个组成部分，它提供了一种和调用系统任务和函数相同的方法调用以 C 写的函数。

编译

Verilog 源代码通常输入到计算机的一个或多个文本文件中。然后，这些文本文件被提交到 Verilog 编译器或解释器，编译用于仿真或合并的数据文件。有时候仿真在编译后立即进行，不创建中间的数据文件。

模块结构

```
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 : BlockName
    // 语句
  end

  always
  begin
    // 语句
  end

  // 连续赋值...
  assign W1 = Expression;
  wire (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;

  // 模块实例...
  COMP U1 (W3, W4);
  COMP U2 (.P1(W3), .P2(W4));

  task T1;
    input A1;
    inout A2;
    output A3;
    begin
      // 语句
    end
  endtask

  function [7:0] F1;
    input A1;
    begin
```

```
        // 语句
        F1 = Expression;
      end
  endfunction

endmodule
```

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
   ...
```

这个快速参考语法摘要不遵从本指南剩余部分所使用的符号惯例。

Verilog 黄金参考指南

按字母顺序参考的部分

Always

包含一条或多条语句（过程赋值、任务使能、if、case 和循环语句），这些语句在仿真运行中重复执行，由定时控制管理。

语法

```
always
  Statement
```

在何处使用

module-<HERE>-endmodule

规则

- always 只能赋值寄存器（reg、integer、real、time、realtime 类型）
- 启动仿真时，所有 always 都开始执行，而且在仿真过程中持续执行。当到达 always 的最后一条语句时，程序返回到 always 的第一条语句继续执行。

注意

- 如果 Always 包含超过一条语句，语句要包含在 begin-end 或 fork-join 块中。
- 没有定时控制的 always 将永远循环。

合并

always 是其中一条很有用的 Verilog 合并语句，但 always 通常是不合并的。为了使结果最好，代码应受到下面其中一种模板的限制：

```
always @(Inputs)                    // 所有输入
begin
  ...                               // 组合逻辑
end

always @(Inputs)                    // 所有输入
  if (Enable)
  begin
    ...                             // 锁存器的动作
  end

always @(posedge Clock)             // 只是时钟
begin
  ...                               // 同步的动作
end

always @(posedge Clock or negedge Reset)
```

```
                                    // 只是时钟和复位
begin
  if (!Reset)                       // 测试异步复位的有效激活电平
    ...                             // 异步行动
  else
    ...                             // 同步行动
end                                 // 给出触发器+逻辑
```

举例

下面的例子是一个寄存器传送级 always：

```
always @(posedge Clock or negedge Reset)
begin
  if (!Reset)                       // 异步复位
    Count <= 0;
  else
    if (!Load)                      // 同步载入
      Count <= Data;
    else
      Count <= Count + 1;
end
```

接下来的例子是描述组合逻辑的 always：

```
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 // 循环
end
```

Begin

用于组合语句，使它们按顺序执行。Verilog 的语法通常要求例如在 always 中只有一条语句。如果需要超过一条语句，语句就要被包含在一个 begin-end 块内。

语法

```
begin [: Label
  [ Declarations...]]
```

```
  Statements...
end
```

Declaration = {either} *Register Parameter Event*

在何处使用

见语句。

规则

- begin-end 块必须包含至少一条语句。
- Begin-end 块内的语句按顺序执行。定时控制与前一语句有关。执行完 begin-end 块的最后一条语句后，整个 begin-end 块的执行也完成。
- Begin-end 和 fork-join 块可在自己内部或相互之间嵌套。
- 如果 begin-end 块要包含局部声明，必须先命名这个 begin-end 块（即必须有标记）。
- 如果要禁能一个 begin-end 块，必须先命名这个块。

注意

仿真中，Verilog LRM 允许 begin-end 块被交叉执行。这就是说，即使 begin-end 块包含的两个相邻语句之间没有定时控制，仿真器也可以选择执行另一个过程的部分（例如：另一个 always 的语句）。这就是 Verilog 语言有不确定性的原因。

提示

- 在没有局部声明和块不会被禁能的情况下也可以给 begin-end 块加上标记，增加其可读性。
- 不会在别处使用的寄存器可以用局部声明来声明。这使声明更明确。

举例

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

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

Case

一个语句基于 case 表达式的值条件地执行最多一个分支。

语法

CaseKeyword (*Expression*)

Expression,... : Statement	{表达式可以是变量}
Expression,... : Statement	
...	{任意数量的情况}
[default [:] *Statement*]	{不一定在结尾}

endcase

CaseKeyword = {either} case casex casez

在何处使用

见语句。

规则

- casex 语句的 X 和 Z 以及 casez 语句的 Z 表示“无关”。
- 最多可以有一条 default 语句。当没有标记表达式符合 case 表达式时执行 default 语句。(“标记”是在冒号左面的一个表达式或是用逗号分隔的表达式列表，也可以是保留字 default，default 后面可以有或者没有冒号)。
- 如果标记是用逗号分隔的两个或多个表达式列表，只要 case 表达式符合其中一个标记表达式就认为标记就符合 case 表达式。
- 如果没有标记表达式符合 case 表达式而且没有 default 语句，case 语句不起作用。

注意

- 如果特殊的标记要执行超过一条语句，语句必须包含在 begin-end 或 fork-join 块内。
- 如果相应的标记是第一个符合 case 表达式的标记，case 只会执行这个分支。Case 的标记不需要互斥，因此当相同的标记被错误地重复使用时 Verilog 编译器不会报告出错。
- Casex 和 casez 语句的要用保留字 endcase 而不是 endcasex 或 endcasez 结束。
- Casex 表达式中的 X 或 Z 或者 casez 表达式的 Z 符合所有 case 的标记。这会使仿真结果混淆。

合并

- case 语句内的赋值一般合并到多路复用器。如果 case 标记使用变量（例如寄存器或线网），那么可以合成优先权译码器。
- 无时钟的 always 中的不完全赋值（即没有指定某些输入条件的输出）被合并到透明锁存器。有时钟 always 中的不完全赋值合并到循环寄存器。

提示

- 仿真时，通常用 default 作为最后一条 case 语句，以捕捉非法的情况。
- casez 通常比 casex 好，因为在仿真中出现 X 会导致产生误导或混乱的结果。
- casex 和 casez 标记的 Z 可用字符?代替。这可以清楚地表示值“无关”而不是高阻。

举例

case (Address)

```
    0 : A <= 1;                          // 选择一个单独的地址值
    1 : begin                            // 执行超过一条语句
          A <= 1;
          B <= 1;
        end
    2, 3, 4 : C <= 1;                    // 选出几个地址值
    default :                            // 其他剩余的地址
      $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
```

编码标准

编码标准共有两类。词汇编码标准负责控制文本格式、命名惯例和注释，这种标准增加程序的可读性并简化维护。合并编码标准控制 Verilog 的结构样式，避免普通的合并缺陷并在设计流程的早期发现合并错误。

下面列出的编码标准需要根据所选的工具和个人喜好修改。

词汇编码标准

- 将每个 Verilog 源文件的内容限制到一个模块，不从文件分割模块。
- 源文件的名字应与文件内容有关（例如：ModuleName.v）
- 每行只写一条声明或语句。
- 使用和例子一样的缩进。
- 用户定义的名字要注意大小写一致（例如：第一个字符是大写）。
- 尽管局部名字（例如：循环变量）可能很简洁，但用户定义的名字应当是有意义并包含信息的。
- 写注释来解释（不是复制）Verilog 代码。其中，注释接口特别重要（例如：模块参数、端口、任务和函数变量）。

- 在任何可能的时候用参数或`定义宏，避免在声明和语句中直接嵌入文本数字和字符串。

合并编码标准

- 将设计分割成小的功能块，并为每个块使用一个行为样式。除了设计的主要部分外，其他部分要避免门级描述。
- 有定义好的时钟策略，而且在 Verilog 明确执行该策略（例如：单时钟、多相时钟、门时钟、多时钟域）。确保 Verilog 的时钟和复位信号是清楚的（即：不从组合逻辑或无意识的门产生）。
- 有定义好（制造好）的测试策略，并适当地编码 Verilog（例如：所有触发器可复位、从外部管脚进行测试、无功能冗余）。
- 所有 Verilog 的 always 应遵守标准合并处理模板的其中一个（见 Always）。
- 描述组合和锁存逻辑的 always 必须在 always 顶部的事件控制列表列出所有输入。
- 组合的 always 必须不能包含不完全赋值，即所有输入值的组合必须赋值给所有输出。
- 描述组合和锁存逻辑的 always 必须不能包含反馈，即 always 中被赋值为输出的寄存器不能作为 always 的输入。
- 带时钟的 always 在事件控制列表中只能有时钟和异步控制输入（一般是复位或置位）。
- 避免不必要的锁存器。这些多余的锁存器是由于无时钟 always 的不完全赋值产生。
- 避免不必要的触发器。当用非阻塞赋值在带时钟的 always 内给寄存器赋值或者当寄存器在连续重复的带时钟 always 之间和时钟周期之间保持值不变时，触发器被合并。
- 所有内部状态寄存器必须可复位，这样寄存器传输极和门级描述在检验的时候可以复位到相同的已知状态（这不能应用到流水线寄存器或同步寄存器）。
- 对于有不能到达的状态的有限状态机器和其他时序电路（例如：一个 4 位的十进制计数器有 6 个不可到达的状态），如果硬件在这些状态下可以被控制，那么所有 2^N 个可能的状态的行为必须在 Verilog 中明确描述，包括不能到达的状态的行为。这就允许保留状态机器进行合并。
- 避免在赋值中使用延时，除非要求在寄存器传输级解决 0 延时时钟的时滞问题。
- 不要使用 integer 或 time 类型的寄存器，否则它们会分别合并到 32 位总线和 64 位总线。
- 请仔细检查使用动态索引的 Verilog 代码（即用可变的索引或地址作为位选择或存储器元素）、循环语句或算术运算符，因为这些代码要被合并成大量难以优化的门。

注释

可以（应当!）在 Verilog 源代码中包含说明性的注释。

语法

{单行注释}

//

{多行注释}

/* ... */

在何处使用

几乎可以在任何地方使用，但不能分割运算符、数字、字符串、名字和关键字。

规则

- 单行注释用两个斜杠字符开始，在行的末端结束。
- 多行注释用/*开始，注释可能跨过多行，直到*/处结束。
- 多行注释不能嵌套。但多行注释中可以有单行注释，它们没有特别的含意。

注意

/* ... /* ... */ ... */——注释在第一个*/处结束，第二个*/将被忽略。这个例子无疑会出现句法错误。

提示

全部使用单行注释。只在例如在开发和调试代码的过程中需要注释大段代码时使用多行注释，。

举例

```
// 这是一个注释
/*
共有三行的多行注释
*/
module ALU /* 8 位 ALU */ (A, B, Opcode, F);
```

连续赋值

当表达式中的线网或寄存器的值改变时，连续赋值在一个或多个线网创建事件。

语法

{either}

assign [*Strength*] [*Delay*] *NetLValue* = *Expression*,
NetLValue = *Expression,*
...;

NetType [*Expansion*] [*Strength*] [*Range*] [*Delay*]
NetName = *Expression*,
NetName = *Expression*,
…; {见线网}

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

在何处使用

module-<HERE>-endmodule

规则

两种形式的连续赋值效果相同。

Assign 左边的线网必须在连续赋值语句的前面的源代码中明确声明。

注意

尽管连续赋值和过程连续赋值很相似，但它们是不一样的。请确保将 assign 放在正确的地方。连续赋值在 initial 或 always 外。过程连续赋值在允许使用语句的地方都可以使用（initial、always、任务、函数

等的内部）。

合并

- 合并工具忽略延时和强度；使用指定定时限制的工具代替。
- 连续赋值作为组合逻辑合并。

提示

- 用连续赋值描述组合逻辑可以容易地用一个简单表达式描述。函数可以用于组成表达式。always 一般能较好地描述许多复杂的组合逻辑，而且其仿真速度比用大量独立的连续赋值语句快得多。
- 当 Verilog 要求使用线网时，连续赋值对传输寄存器的值到线网非常有用。例如，将在 initial 中描述的测试激励应用到模块实例的输入输出口。

举例

```
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;
```

Defparam

编译时覆盖参数值。通过使用层次名字，参数值可以在设计层次内部或外部的任何地方被覆盖。

语法

```
defparam   ParameterName = ConstantExpression,
           ParameterName = ConstantExpression,
           ... ;
```

在何处使用

module-<HERE>-endmodule

合并

一般不合并

提示

不要使用 defparam！它提供了一种有用的方法逆向注释布局延时，但现在通常通过使用特殊的程序块和编程语言接口实现。对要覆盖的参数，请在模块实例化时使用#语法。

举例

```
`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;
  input a, b;
  parameter T_f = 2;
  and #(T_f) (f,a,b);
endmodule
```

Delay

延时可以在 UDP 和门的实例、连续赋值和线网中使用。这些延时能为线网元件和连接的传播延时建模。

语法

{either}

 # DelayValue

#(DelayValue[, *DelayValue*[, *DelayValue*]]) {上升、下降、关断}

DelayValue = {either}

UnsignedNumber

ParameterName

ConstantMinTypMaxExpression

在何处使用

见连续赋值、实例化、线网

规则

- 如果只给出一个延时的值，这个值既表示上升和下降传播延时（即分别从 0 或 1 跳变）又表示关断延时（如果可用）。
- 如果给出两个延时的值，第一个值是上升延时，第二个值是下降延时；但 tranif0、tranif1、rtranif0 和 rtranif1 除外，它们的第一个值表示开启延时，第二个值是关断延时。
- 如果给出三个延时的值，第三个延时是关断延时（转换到 Z）；但 trireg 线网除外，它的第三个延时是电荷衰减时间。
- X 的延时是最小的指定延时。

- 对于矢量来说，从非 0 到 0 的转换被认为是“下降”，转换到 Z 是“关断”，所有其他转换都被认为是“上升”。

注意

很多工具强调：延时的 MintypMax 表达式必须包含在括号中。例如：#(1:2:3)是允许的，但#1:2:3 是不允许的。

合并

合并工具忽略延时。合并线网列表的延时受到合并工具命令例如设置最大的时钟周期约束。

提示

指定的块延时（路径延时）通常是建模延时的一种很精确的方式，而且它还提供了计算延时的机制以及逆向注释布局信息。

设计流程

使用 Verilog 和合并设计 ASIC 或复杂 FPGA 的基本流程如下。设计流程需要进行迭代，但这里不作介绍。而且，设计流程必须根据设计的器件种类和特殊的应用进行修改。

1 系统分析和说明
2 系统划分
  2.1 顶级块的捕捉
  2.2 块的大小估算
  2.3 最初的平面布置
3 块级设计。对于每个块：
  3.1 写寄存器传输级 Verilog
  3.2 合并编码检查
  3.3 写 Verilog 测试程序
  3.4 Verilog 仿真
  3.5 写合并脚本——约束、边界条件、层次
  3.6 初始的合并——分析门计数和定时
4 芯片集成。对于完整的芯片：
  4.1 写 Verilog 测试程序
  4.2 Verilog 仿真
  4.3 合并
  4.4 门级仿真
5 测试阶段
  5.1 为测试修改门级线网列表
  5.2 产生测试矢量
  5.3 仿真可测试的线网
6 芯片的放置和布线（或装配）
7 接线柱布局仿真、故障仿真和定时（时序）分析

Disable

使激活的任务或有名字的块在执行完所有语句前终止其执行。

语法

disable BlockOrTaskName;

在何处使用

见语句。

规则

- 禁能一个有名字的模块（begin-end 或 fork-join）或者一个任务就禁能从该模块或任务使能的所有任务，而且向下禁能到使能的任务层次。然后继续执行被禁能的任务后面使能的语句或有名字的块。
- 有名字的模块或任务可以在自己内部用 disable 语句自我禁能。
- 当任务被禁能时，下面的事件不能确定：任何输出或输入输出的值、仍未生效的非阻塞赋值预定的事件、assign 和 force 语句。
- 函数不能被禁能。

注意

任务自我禁能与任务返回不一样，因为任务自我禁能的输出没有定义。

合并

disable 只在有名字的模块或任务自我禁能时合并。

提示

用 disable 可以从任务早期退出、退出循环或继续下一个迭代的循环。

举例

```
begin : Break
  forever
  begin : Continue
    ...
    disable Continue;          // 继续下一个迭代
    ...
    disable Break;             // 退出 forever 循环
    ...
  end // 继续
end // 终止
```

错误

下面列出的是最常见的 Verilog 错误。前五位占了所有错误的 50%。

前 5 位 Verilog 错误

- 过程赋值的左边不声明为寄存器类型。
- 缺少 begin-end 语句，或 begin-end 语句不匹配。
- 二进制数缺少基（'b）（也就是说编译器将它们看作是十进制数）。

- 在编译器伪指令中错误使用撇号（应当是后撇号或重音符号`）和数字基（应当是一般的单引号或倒转的逗号'）。
- 语句的结尾缺少分号。

其他常见错误

- 尝试在任务或函数名字后面的方括号中定义任务和函数参数。
- 在测试程序中测试时，忘了要将模块实例化。
- 用过程连续赋值代替连续赋值（即“assign”在错误的地方使用）。
- 尝试用保留字作为标识符（例如：xor）。
- Always 内没有定时控制（导致无限循环）。
- 在事件控制（例如：@(a or b)）中用逻辑或运算符（||）代替保留字 or。
- 用隐式线网连接矢量端口。
- 模块实例中端口连接的顺序错误。
- 嵌套的 if-else 语句中有包含错误（begin-end 的位置不正确）。
- “等号”的形式错误。‘=’用于赋值；‘==’用于比较数字值；‘===’用于匹配 0、1、X、和 Z 的精确序列。

事件

事件用于在行为模型中描述通讯和同步。

语法

```
event Name ,...;                {声明事件}
-> EventName;                   {触发事件}
```

在何处使用

见->的语句。

在以下的地方允许有事件声明：

module-<HERE>-endmodule

begin : Label-<HERE>-end

fork : Label-<HERE>-join

task-<HERE>-endtask

function-<HERE>-endfunction

规则

事件没有值或延时；它们被事件触发语句触发，在跳变沿敏感的定时控制中测试。

合并

一般不合并。

提示

有名字的事件通常用于在测试程序和系统级模型中与相同模块或不同模块（使用层次名字）的 always 之间通讯。

举例

```
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;
  ...
  -> StopClock;
end
```

表达式

表达式从一组运算符、名字、文本值和子表达式计算值。常数表达式是在编译时可以计算值的表达式。标量表达式只计算 1 位的值。延时可以用 MinTypMax 表达式表示。

语法

```
Expression = {either}
  Primary
  Operator Primary                          {一元运算符}
  Expression Operator Expression            {二元运算符}
  Expression ? Expression : Expression
  String

Primary = {either}
  Number
  Name                                      {参数、线网或寄存器的名字}
  Name[ Expression]                         {位选择}
  Name[ Expression: Expression]             {部分选择}
  MemoryName[ Expression]
```

{ *Expression,...***}** {串联}
{ *Expression***{** *Expression,...***}}** {复制}
FunctionCall
(*MinTypMaxExpression*)
{延时使用 MinTypMax 表达式}
MinTypMaxExpression = {either}
Expression
Expression: Expression: Expression

规则

- 只有矢量的线网和 reg 以及 integer 和 time 允许位和部分选择。
- 部分选择必须在分号的左边寻址一个比分号右边更高的位。(最高位是线网或寄存器声明中左边的范围表达式的值)。
- 有 X 或 Z 或超出范围的位和部分选择可能会也可能不会被捕捉为编译器错误。它们给出的表达式结果是 X。
- 存储器没有位或部分选择机制。
- 当表达式用整数常数作为操作数，无基但带符号的整数（例如：-5）的处理和有基且带符号的整数（例如：-'d5）不一样。前者作为带符号数处理，后者作为无符号数处理。

注意

很多工具要求常数 MinTypMax 表达式的最小、典型和最大延时值有序（例如：min<=typ<=max）。

举例

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

For

通用循环语句。允许一条或多条语句重复执行。

语法

```
for ( RegAssignment;        {初始值}
      Expression;           {循环条件}
      RegAssignment)        {循环值}
   Statement
```

RegAssignment = *RegisterLValue* = *Expression*
RegisterLValue = {either}
RegisterName
RegisterName**[** Expression**]**
RegisterName**[** *ConstantExpression: ConstantExpression***]**
Memory**[** *Expression***]**
{ *RegisterLValue,...***}**

在何处使用
见语句。

规则
执行 for 循环的时候，一定要建立初始值。在每次重复前（包括第一次）都要检验表达式：如果是 false（例如：0、X 或 Z）循环终止。每次循环后，都要进行循环赋值。

注意
请小心使用小型宽度的 reg 作为循环变量。测试负值的 reg 也要非常小心。如果重复的加、减运算和 reg 的值都作为无符号数处理，循环表达式就永远不会出错。

```
reg [2:0] i;                                    // i 的值永远在 0 到 7 之间
...
for ( i=0; i<8; i=i+1 )                         // 循环不会停止
...
for ( i=-4; i<0; i=i+1 )                        // 不会执行
...
```

在像上面的情况中，循环变量 i 应该用 integer 类型。

合并
如果循环的边界固定，For 循环被合并到重复的硬件结构。

举例

```
V = 0;
for ( I = 0; I < 4; I = I + 1 )
begin
  F[I] = A[I] & B[3-I];                         // 4 个独立的与门
  V = V ^ A[I];                                 // 4 个级联的异或门
end
```

Force
Force 和过程连续赋值相似，覆盖线网和寄存器的行为。它主要用于帮助调试。

语法
{either}
force *NetLValue* = *Expression* ;

force *RegisterLValue* = *Expression* ;

{either}
release *NetLValue*;
release *RegisterLValue*;

NetLValue = {either}
  NetName
  {NetName,...**}**
RegisterLValue = {either}
  RegisterName
  {RegisterName,...**}**

在何处使用
见语句。

规则
- 线网或寄存器的位或部分选择不能被 force 或 release。
- Force 优于过程连续赋值（assign 是过程语句）。
- Force 保持有效，直到在相同的线网或寄存器执行另一个 force 或者线网或寄存器被释放。
- 当寄存器的 force 释放后，它不立即修改寄存器的值。Force 的值会维持直到发生下一个过程赋值，除非寄存器的过程连续赋值有效。
- 当线网的 force 释放后，线网的值由线网的驱动器决定，而且值可以立即更新。

合并
不合并。

提示
在测试程序使用，为了调试的目的对行为进行覆盖。请不要用于建模行为（用连续赋值代替）。

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

Forever
使一条或多条语句执行无限循环。

语法
forever
  Statement

在何处使用

见语句。

注意

无限循环应包含定时控制或可以自我禁能，否则它将无限循环。

合并

一般不合并。如果连续重复被形式为@(posedge Clock)的定时控制终止，可以合并。

提示

- 用于描述测试程序的时钟。
- 用 disable 跳出循环。

举例

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

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

Fork

将语句组合成一个并行的块，使它们能并发执行。

语法

```
fork [ : Label
   [ Declarations...]]
   Statements...
join
```

Declaration = {either} *Register Parameter Event*

在何处使用

见语句。

规则

- fork-join 块必须包含至少一个语句。
- Fork-join 块的语句是并发执行的。所以 fork-join 块内语句的顺序无关紧要。定时控制与进入模块

的时间有关。当 Fork-join 块的所有语句都执行完后，fork-join 块也执行完。

- Begin-end 和 fork-join 块可以在自己内部或者互相嵌套。
- 如果 fork-join 块要包含局部声明，它必须被命名（即它必须有一个标记）。
- 如果要禁能 fork-join 块，它必须被命名。

合并

不合并。

提示

Fork-join 语句用于描述并发的激励。

举例

```
initial
fork : stimulus
   #20 Data = 8'hae;
   #40 Data = 8'hxx;              // 最后执行的语句
   Reset = 0;                     // 首先执行的语句
   #10 Reset = 1;
join                              // 在时刻 40 完成
```

函数

用于组合语句来定义新的算术或逻辑函数。函数通常在模块内声明而且只从该模块调用，但它也可以用层次名字从别处调用。

语法

```
function [ RangeOrType] FunctionName;
   Declarations...
   Statement
endfunction

RangeOrType = {either} Range integer time real realtime
Range = [ ConstantExpression: ConstantExpression]
Declaration = {either}
   input [ Range] Name,...;
   Register
   Parameter
   Event
```

在何处使用

```
module-<HERE>-endmodule
```

规则

- 函数至少有一个输入变量，可以没有任何输出或输入输出。

- 函数不包含定时控制（延时、事件控制或等待）。
- 通过赋值函数名，函数可以返回一个值，就象是寄存器一样。
- 函数不能使能任务。
- 函数不能被禁能。

注意

- 函数的输入不像模块的端口一样列在函数名后面的方括号中；它们只在输入声明中声明。
- 如果函数包含超过一个语句，语句必须包含在 begin-end 或 fork-join 块内。

合并

每次调用函数都被合并为独立的组合逻辑块。

举例

```
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
```

函数调用

调用一个函数，返回一个可以在表达式中使用的值。

语法

FunctionName (*Expression,...*);

在何处使用

见表达式。

规则

函数必须至少有一个输入变量，因此函数调用通常至少有一个表达式。

合并

每次函数调用都作为组合逻辑一个独立块合并。

举例

```
Byte = ReverseBits(Byte);
```

门

Verilog 有大量的内置逻辑门和开关模型。这些门和开关可以在模块里实例化，创建模块行为的结构化

描述。

逻辑门

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

缓冲门和非门

buf (Output,..., Input)
not (Output,..., Input)

三态逻辑门

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)

带控制的双向开关

tranif0 (Inout1, Inout2, Control)
tranif1 (Inout1, Inout2, Control)
rtarnif0 (Inout1, Inout2, Control)
rtranif1 (Inout1, Inout2, Control)

上拉和下拉源

pullup (Output)
pulldown (Output)

真值表

表中的逻辑值 L 和 H 表示结果的值部分未知。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	
输入	输出
0	0
1	0
X	0
Z	0

not	
输入	输出
0	1
1	1
X	1
Z	1

缓冲门和非门有多个输出，这些输出的值都相等。

bufif0		使能端（EN）			
		0	1	X	Z
数据	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

bufif1		使能端（EN）			
		0	1	X	Z
数据	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

notif0		使能端（EN）			
		0	1	X	Z
数据	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

notfif1		使能端（EN）			
		0	1	X	Z
数据	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

pmos rpmos		控制			
		0	1	X	Z
数据	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	Z	Z	Z	Z

nmos rnmos		控制			
		0	1	X	Z
数据	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);

相当于

nmos (W, Datain, NControl);

pmos (W, Datain, PControl);

规则

当开关 nmos、pmos、coms、tran、tranif0 和 tranif1 开路时，输入到输出所传播的信号强度不会改变。电阻性开关 rnmos、rpmos、rcoms、rtran、rtranif0 和 rtranif1 会减弱传播的信号强度：

强度	减弱为
电源	拉
强	拉
拉	弱
大	中等
弱	中等
中等	小
小	小
高阻	高阻

IEEE1364

Verilog HDL 是由 IEEE 标准 Verilog 硬件描述语言参考手册 1364-1995 定义。这个文档是从基于 Cadence Verilog LRM 版本 1.6 的 OVI Verilog 参考手册 1.0 和 2.0 衍生出来。在标准化进程以前，Cadence Verilog-XL 仿真器已经有一个事实语言标准。很多第三方的仿真器都试图遵从这个事实标准。

尽管标准化进程的目的是标准化 Verilog-XL 使用的已有的 Verilog 语言，但 IEEE 标准语言和事实标准仍有一些差别。结果仿真器可能支持也可能不支持以下的特性：

- 原语和模块实例数组（见实例化）。
- 带变量的宏定义（见`define）。
- `undef。
- IEEE 标准不支持的数字强度值（见编译器伪指令）。
- 许多 Verilog-XL 支持但 IEEE 标准不支持的系统任务、函数以及编译器伪指令。
- 如果线网或寄存器在模块内只有一个驱动器，那么 IEEE 标准允许特殊模块的通路延时目的地是寄存器或者线网。事实标准对目的地有许多限制（见 Specify）。
- 指定的通路延时表达式可以由高达 12 个用逗号分隔的独立延时表达式组成。事实标准允许最多 6 个表达式。（见 Specify）
- 线网声明中保留字 scalared 和 vectored 的位置有变化。在事实标准中，保留字的后面紧跟着矢量范围。IEEE 标准中，保留字紧跟在线网类型后面。（见线网）
- 对常数 MinTypMax 表达式的最小值、典型值和最大值的相对大小没有限制。事实标准规定最小

延时的值要小于或等于典型延时，典型延时的值也要小于或等于最大延时的值。

- IEEE 标准指出当用 MinTypMax 表达式表示延时的时候，该表达式不需要包含在方括号中。事实标准要求有方括号。

If

If 根据条件表达式执行两条语句或两个语句块的其中一条（个）。

语法

```
if ( Expression)
  Statement
[else
  Statement]
```

在何处使用

见语句。

规则

如果表达式的值非 0，表达式为真；如果值是 0、X 或 Z，表达式为假。

注意

- 如果 if 或 else 分支要执行超过一条语句，这些语句必须包含在 begin-end 或 fork-join 块内。
- 注意省略 else 部分的嵌套 if-else 语句。else 与前面最近的 if 相关联，除非已经出现对应的 begin-end。Verilog 编译器会省略源代码中的缩进。

合并

- if 语句中的赋值通常合并到多路复用器。
- 某些输入不改变输出的不完全赋值合并到无时钟 always 的透明锁存器而且在带时钟 always 中再循环。
- 在一些情况下，嵌套的 if 语句合并到多个逻辑级。用 case 语句代替 if 就不会出现这种情况。

提示

一组嵌套的 if-else 语句可以用于给出条件测试的优先级。如果想不给出优先级而译码一个值，可以使用 case 语句。

举例

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

```
    X = C;
end
```

Initial

包含只执行一次的语句或语句块，initial 在仿真启动时开始执行。

语法

```
initial
  Statement
```

在何处使用

module-<HERE>-endmodule

合并

无合并

注意

如果 initial 包含超过一条语句，这些语句要包含在 begin-end 或 fork-join 块内。

提示

在测试程序中用 initials 来描述激励。

举例

下面的例子展示了在测试程序中用 initial 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;        // 停止时钟发生器
end
```

实例化

实例是模块、UDP 或门的唯一副本。设计中的层次是通过实例化模块来创建；设计的行为在结构上可以通过建立连接到线网的 UDP、门和其他模块的实例来描述。

语法

{either}

ModuleName [**#**(*Expression,...*)] *ModuleInstance,...;*

UDPOrGateName [*Strength*] [*Delay*] *PrimitiveInstance*,...;

ModuleInstance =

InstanceName [*Range*] ([*PortConnections*])

PrimitiveInstance =

[InstanceName [*Range*]] (*Expression,...*)

Range = **[** *ConstantExpression: ConstantExpression***]**

PortConnections = {either}

[*Expression*] ,... {有序的连接}

.PortName([*Expression*]) ,... {有名字的连接}

在何处使用

module-<HERE>-endmodule

规则

- 有名字的端口连接只允许用于模块实例。
- 给出的有序端口连接表中，第一个元素连接到模块或门的第一个端口，第二个元素连接到第二个端口，如此类推。
- 如果给出有名字的端口连接表，名字必须和模块的端口对应，它与连接的顺序无关。
- 在有序端口连接表中，用两个相邻的逗号省略表达式表示该模块端口不连接。有名字的端口连接表中，省略所有名字或使方括号内的表达式为空表示端口不连接。
- 输入端口可以连接任意的表达式，但输出端口只能连接到线网、线网的位或部分选择或者是它们的级连。输入表达式创建隐性的连续赋值。
- 如果给出的范围是实例名的部分，它表示一个实例数组。当端口表达式的位长度和模块、被实例化的 UDP 或门相应端口的位长度相等时，整个表达式被连接到每个实例的这个端口。如果位长度不同，每个实例都得到在从指定范围的右边开始的表达式部分选择。连接到所有实例的位太多或太少都会出现错误。
- #符号有两种不同的用法。它可以用于覆盖模块实例中一个或多个参数的值，也可以指定 UDP 或门实例的延时。对于模块实例，第一个表达式替代在模块中声明的第一个参数的值；第二个表达式替代第二个参数的值，如此类推。
- 门 pullup、pulldown、tran 和 rtran 的实例不允许有延时。

- 以下的开关不能指定强度：nmos、pmos、coms、rnmos、rpmos、rcmos、tran、rtran、tranif0、tranif1、rtranif0 以及 rtranif1

注意

- 在有序表中很容易意外地交换两个端口。如果端口的宽度和方向相同，第一次提示出错是在仿真中得到错误结果的时候。这些错误很难调试。在模块实例中用有名字的端口连接表可以避免这个问题。
- 模块、UDP 或门实例的阵列最近才添加到 Verilog 语言中，并不是所有工具都支持。

合并

UDP 和开关的实例一般不合并。

提示

- 模块端口使用有名字的连接增加可读性并减少错误的可能性（见上面）。
- 除了位或部分选择以及级连外，不要使用端口表达式。用独立的连续赋值代替。

举例

UDP 实例：

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

模块实例：

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

在下面的两个例子中，端口 QB 不连接：

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

下面是一个与或非，显示了端口连接表的表达式：

```
nor (F, A&&B, C)                    // 不推荐使用
```

下面的例子是一个实例数组：

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

  bufif1 U1[7:0] (out, in, ena);

/* 相当于(除了实例名字)...
  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

模块

模块是 Verilog 的层次基本单元。模块包含声明和功能描述而且并代表硬件元件。

模块也用于声明可以在各处使用的参数、任务和函数。这样的模块不代表实际的硬件元件，因为它们不需要包含 initial、always、连续赋值或实例。

语法

{either}

module ModuleName [(*Port,...*)];
  ModuleItems...
endmodule

macromodule ModuleName [(*Port,...*)];
  ModuleItems...
endmodule

ModuleItem = {either}
  Declaration
  Defparam
  ContinuousAssignment
  Instance
  Specify
  Initial
  Always
Declaration = {either}
  Port
  Net
  Register
  Parameter
  Event
  Task
  Function

在何处使用

模块要在任何其他模块或 UDP 的外部声明。

规则

- 几个模块或 UDP（或者都有）可以在一个文件内描述。（实际上，一个单独的模块可被分割成两个或多个文件，但不提倡这样做。）
- 模块可以用关键字 macromodule 定义。语法和模块一样。Verilog 对宏模块的编译和模块不一样，例如它不为宏模块实例创建层次结构的层。这可能会使仿真在速度或存储器方面更有效。为此，

宏模块会受到某些执行指定约束的限制。如果没有碰到这些问题，宏模块和普通模块的处理一样。

注意

module 和 macromodule 都用关键字 endmodule 结束。

合并

- 每个模块作为一个独立的层次块合并，允许你控制合并线网的层次结构，但一些工具默认地拉平层次结构。
- 不是所有工具都支持宏模块。

提示

每个文件只有一个模块。这简化了大型设计的源代码维护。

举例

```
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
```

名字

任何 Verilog 的“东西”都用它的名字来识别。

语法

Identifier

\EscapedIdentifier {用空白结束}

规则

- 标识符可以由字母、数字、下划线和美元符号组成。第一个字符必须是字母或下划线，不能是数字或美元符号。
- 用反斜杠引入的转义标识符以空白（空格、制表符、换页符或换行符）结束，它由任意可打印的字符（除了空白字符）组成。反斜杠和空白不是标识符的组成部分；因此，例如标识符 Fred 和转义标识符\Fred 是一样的。
- Verilog 中的名字是大小写敏感的。
- Verilog 正文中的名字在任何特殊地方都不能有超过一种含意。名字的内部声明（例如：在有名字的 begin-end 块中的名字）在声明外是隐藏的（例如：模块中的名字，其中有名字的 begin-end 块是模块的一部分）。

层次名字

- 在 Verilog HDL 描述中的每个标识符有一个唯一的层次名字。这就意味着通过这个层次名字所有线网、寄存器、事件、参数、任务和函数都能从声明它们的模块外访问。
- 在层次名字的顶层是没有实例化的模块名字。尽管在一次独立的仿真运行中可能有超过一个顶级模块，但顶层的测试程序是一个实例。
- 层次名字的新层次由每个模块实例、有名字的块、任务或函数定义来定义。
- Verilog 对象的唯一的层次名字是由层次根部的顶级模块名字和包含用点号分隔的对象的模式实例、有名字的块、任务或函数的名字组成。

向上的名字引用

- 层次名字形式的名字由两个用点号分隔的标识符组成，它涉及到以下的其中一种：
  - 当前模块内模块实例的一项（这是一个向下引用）。
  - 顶级模块内的一项（这是一个层次名字）。
  - 在当前模块的父模块内模块实例的一项（这是一个向上的名字引用）。
- 向上的名字引用中的第一个标识符可以是模块名或模块实例的名字。

合并

合并工具一般不支持层次名字和向上的名字引用。

提示

- 一般选择对读者来说有意义的名字。而且这对全局名比对局部名更重要。例如：G0123 对于全局复位来说是一个坏名字，而对于循环变量来说 I 是一个可接受的名字。
- 不要使用转义名。这些名字是被 EDA 工具（例如线网制表器或合并工具）使用，这些工具的命名规则和 Verilog 不一样。
- 只在测试工具或者在没有合适选择情况的高层系统模型中使用层次名。
- 避免向上的名字引用，因为这会使代码很难懂，也对调试和维护不利。

举例

下面例子是合法的名字：

```
A_99_Z
Reset
_54MHz_Clock$
Module                                  // 与'module'不一样
\$%^&*()                                // 转义标识符
```

下面的名字是非法的，原因如下：

```
123a                                    // 第一个字符是数字
$data                                   // 第一个字符是美元符号
module                                  // 是保留字
```

下面的例子是层次名字和向上的名字引用：

```
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);         // 名字向上引用到 Top.U1.R
    $display(U1.R);             // 名字向上引用到 Top.U1.R
    ...
  end
endmodule
```

线网

线网用于建模结构化描述中的连接（线路和总线）。线网的值由线网的驱动器决定。驱动器可以是门、UDP 或模块的实例或者连续赋值的输出。

语法

{either}

NetType [*Expansion*] [*Range*] [*Delay*] NetName,...;

trireg [*Expansion*] [*Strength*] [*Range*] [*Delay*]

NetName,...;

{用连续赋值进行线网声明}

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*]

在何处使用

module-<HERE>-endmodule

规则

- 线网 supply0 和 supply1 的值分别是 0 和 1，强度都属于电源。
- 当线网 tri0 和 tri1 不被驱动时，它们的值分别是 0 和 1，强度属于上拉。
- 如果使用关键字 vectored，那么位和部分选择以及强度说明都不允许出现，PLI 认为线网是“不可扩展的”。如果使用关键字 scalared，允许出现位和部分选择，PLI 认为线网是“可扩展的”。建议使用这两个关键字。
- 除了结构化声明中的端口和标量导线，线网必须在使用前声明。

真值表

下面的真值表显示了当线网有两个或多个驱动器时（假设每个驱动器的强度值相等）如何解决其冲突。否则，驱动器用最大的驱动强度值驱动线网。

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	X	X	Z

wor trior	0	1	X	Z
0	1	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 tri0 和 tri1 线网不被驱动时，向它们连续赋值强度到不会影响线网的值和强度。此时，强度一般是上拉，逻辑值是 0（tri0）或 1（tri1）。
- IEEE 标准和 Cadence 事实标准的可选扩展保留字 scalared 或 vectored 的位置不一样。Cadence 事实标准的保留字后面紧跟着范围。

合并

- 线网被合并到连接，但这些可以被优化掉。
- 除了 wire 外，合并工具不支持其他线网类型。

提示

- 甚至在建立隐式声明时，也要在每个模块的顶部明确声明所有线网。这样通过明确意图，增加了 Verilog 代码的可读性和可维护性。
- 只用 supply0 和 supply1 声明地和电源线网。

举例

```
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
```

数字

一个整数或实数数字。整数在 Verilog 中用位表示，其中有些位可能是未知的（X）或高阻（Z）。

语法

{either} *BinaryNumber OctalNumber DecimalNumber*
HexNumber RealNumber

BinaryNumber = [*Size*] *BinaryBase BinaryDigit...*
OctalNumber = [*Size*] *OctalBase OctalDigit...*
DecimalNumber = {either}
[*Sign*] *Digit...* {有符号数}
[*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...*

在何处使用

见表达式。

规则

- 基字符、十六进制数字、X 和 Z 在数字中是大小写不敏感的。
- 数字中字符 Z 和?意义相同。
- 数字不包含空格，但基的两边允许有空格。
- 负数用 2 的补码表示。
- 数字的第一个字符不允许是下划线。（这是一个有效的标识符）。下划线可以提高可读性，但在读出的时候被忽略。
- Size 表示实际的位数。
- 没有说明 size 的数字默认是 32 位或更多位，由器件决定。
- 如果 size 比数字的位数大，数字将在左边填补 0，如果最左边的一位是 X 或 Z，就在左边填补 X

或 Z。

- 如果 size 比数字的位数小，数字将被从左边删节。

注意

- 一个有大小的负数在赋值到寄存器时不带符号扩展。

```
reg [7:0] byte;
reg [3:0] nibble;

initial
begin
  nibble = -1;                    // 即 4'b1111
  byte = nibble;                  // 字节变为 8'b0000_1111
end
```

- 在表达式中使用寄存器或有大小的数字，它的值一般被解释为无符号数。

```
integer i;

initial
  i = -8'd12 / 3;                 // i 变为 81 (即：8'b11110100 / 3)
```

合并

- 0 和 1 分别被合并作为到地和电源的连接。
- 赋值 X 被解释为“无关”。与 X 比较的结果是假（FALSE），除了 casex 语句。（全等运算符===和!==一般不合并。）
- 除了在 casex 和 casez 语句中 Z 表示“无关”外，Z 还用于表示三态驱动器。

提示

- 在 case 语句的标记中，优先使用?而不是 Z。请不要在其他地方使用?，因为可能会引起混淆。
- 在长的数字中加入下划线使其更有可读性。

举例

```
-253                              // 带符号的十进制数
'Haf                              // 无大小的十进制数
6'o67                             // 一个 6 位的 8 进值数
8'bx                              // 一个 8 位的未知数字(8'bxxxx_xxxx)
4'bz1                             // 除了最低位外所有位都是 Z (4'bzzz1)
```

下面的数字是非法的，原因如下：

```
_23                               // 第一个字符是_
8' HF F                           // 有两个非法的空格
0ae                               // 带十六进制数字符的十进制数
x                                 // 是一个名字，不是一个数字(应用 1'bx)
.17                               // 应当是 0.17
```

运算符

运算符用于从表达式的数字、参数和其他子表达式等操作数产生值。Verilog 中的运算符和 C 编程语言的相似。

一元运算符

+ -	符号
!	逻辑非
~	位取反
& ~& | ~| ^ ~^ ^~	归约 (~^ 和 ^~ 相等)

二进制运算符

+ - * /	算术
%	取模
> >= < <=	比较
&& ||	逻辑
== !=	逻辑等
=== !==	全等
& | ^ ^~ ~^	按位操作(^~ 和 ~^相等)
<< >>	移位

其他运算符

A ? B : C	条件
{ A, B, C }	级连
{ N{A} }	复制

在何处使用

见表达式

规则

- 逻辑运算符将它们的操作数看作是布尔量。非零的操作数被解释为真（1’b1），零被解释为假（1’b0）。不明确的值（即应是真或者假，例如 4’bXX00）是未知的（1’bx）。
- 按位运算符（~ & | ^ ^~ ~^）和全等运算符（=== !==）将操作数的各个位独立处理。
- 如果操作数的某一位是 X 或 Z，==或!=的逻辑比较值是未知（1’bX）。（见注意）
- 如果用（< > <= >=）运算符的比较不明确，比较的结果是未知的（1’bX）。例如：

```
2'b10 > 1'b0X                    // 真(1'b1)
2'b11 > 1'b1X                    // 未知(1'bX)
```

（见注意）

- 归约运算符（& ~& | ~| ^ ~^ ^~）将矢量简化为标量值。
- 有大小的表达式的算术运算是四舍五入的。因此，例如 4'b1111 + 4'b0001 的结果是 4'b0000。
- 整数除法将删除结果的小数部分。
- 模（%）是第一个操作数被第二个操作数除所得到的余数，结果取第一个操作数的符号。
- 实数表达式只允许出现某些运算符：一元的+和-以及算术、关系、逻辑、等号和条件运算符。在实数中使用逻辑或关系运算符的结果得到一个位值。

运算符的优先级

- + - ! ~ (一元) – 最高优先级

- * / %
- + - (二元)
- << >>
- < <= > >=
- == != === !==
- & ~&
- ^ ^~
- | ~|
- &&
- ||
- ?: - 最低优先级

注意

- 并不是所有仿真器都遵从用== != < > <= >=进行未知和不明确比较的规则。请注意！
- 请注意一元归约运算符和按位逻辑运算符的差别。其意义由上下文和包含特殊解释的方括号给出。

合并

- 逻辑、按位和移位运算符被作为逻辑运算符合并。
- 条件运算符被作为多路复用器或三态使能端来合并。
- 运算符+ - * < <= > >= == !=被分别作为加法器、减法器、乘法器和比较器合并。
- 运算符/和%只作为移位或在常数表达式中合并。（例如：/2 表示右移）。
- 所有工具都不合并其他运算符。

提示

用括号而不是运算符优先级来组成表达式。这样能防止错误而且使不是很懂 Verilog 语言的人可以理解你的表达式。

举例

```
-16'd10                    // 一个表达式，不是一个带符号数!
a + b
x % y
Reset && !Enable           // 和 Reset && (!Enable)一样
a && b || c && d           // 和(a && b) || (c && d)一样
~4'b1101                   // 结果是 4'b0010
&8'hff                     // 结果是 1'b1 (所有位都是 1)
```

参数

参数是向常数值给定名字的一种方法。当设计被编译时（不是仿真时），参数的值可以被覆盖，因此可以将总线宽度等参数化。

语法

parameter Name = *ConstantExpression*,

Name = *ConstantExpression*,
... ;

{一些工具支持下面的非标准语法}
parameter [*Range*] Name = *ConstantExpression,*
Name = *ConstantExpression,*
... ;
Range = **[** *ConstantExpression: ConstantExpression* **]**

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

规则
参数是常量：在仿真时修改它们的值是非法的。但在编译时用 defparam 命令或者当模块包含的参数被实例化时，参数值可以改变。

合并
一些合并工具可以像“模板”一样处理参数化的模块，也就是说模块一旦被读入后可以用不同的参数值合并几次。所有合并工具都能合并包含不被覆盖的参数的模块实例。

提示
请给参数指定在字面上有意义的名字。

举例
这是一个 N 位的参数化移位寄存器例子。不同的移位器实例有不同的宽度。

```
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 (...);          // 16 位的移位寄存器
  Shifter U2 (...)                 // 10 位的移位寄存器
endmodule
```

PATHPULSE$

这是一个在特殊模块中控制脉冲传播的 specparam。脉冲（*pluse*）是模块输出的两个预定的转换，在比从模块到输出的延时还要短的时间内产生。

默认地，仿真器拒绝接受脉冲；这意味着只有比通过模块的延时长的转换才被传播。这个作用被称为“惯性延时”。PATHPULSE$ specparam 允许修改这个默认的特性。

语法

```
{either}
PATHPULSE$ = ( Limit[, Limit]);                    {(拒绝,错误)}
PATHPULSE$Input$Output = ( Limit[, Limit]);

Limit = ConstantMinTypMaxExpression
```

在何处使用

specify-<HERE>-endspecify

规则

- 如果没有给定错误界限，它将被设置为和拒绝界限一样。
- 比拒绝界限短的脉冲不能传播到输出。
- 比拒绝界限长但比错误界限短的脉冲作为 1’bX 传播。
- 比错误界限长的脉冲将被如常传播。
- 对于从“输入”到“输出”的延时，PATHPULSE$input$output specparam 覆盖了相同模块中的普通 PATHPULSE$ specparam。

合并

合并工具忽略延时结构（包括指定的块）。

举例

```
specify
  (clk => q) = 1.2;
  (rst => q) = 0.8;
  specparam PATHPULSE$clk$q = (0.5,1),
            PATHPULSE = (0.5);
endspecify
```

端口

模块的端口建模硬件元件的管脚或印刷板插座。

语法

{definition}

{either}
PortExpression {有序列表}
.PortName([*PortExpression*]) {名字列表}

PortExpression = {either}
PortReference
{ *PortReference,...***}**
PortReference = {either}
Name
Name**[** *ConstantExpression* **]**
Name**[** *ConstantExpression: ConstantExpression* **]**

{declaration}

{either}
input [*Range*] Name,...; {端口引用}
output [*Range*] Name,...; {端口引用}
inout [*Range*] Name,...; {端口引用}

Range = **[** *ConstantExpression: ConstantExpression* **]**
{在部分选择或范围中，左边的表达式是 MSB，右边的表达式是 LSB}

在何处使用

module (<HERE>); {定义}
<HERE> {声明}
...
endmodule

规则

- 端口定义列表中的所有端口必须按顺序或用名字指定。但两种方式不能混合使用。
- 无端口表达式的名字（例如：.A()）定义了一个不连接到模块任意部分的端口。
- 除了在端口列表定义外，每个端口必须声明是作为输入、输出还是输入输出（双向）口。
- 除了声明端口是作为输入、输出还是输入输出（双向）口外，还必须声明端口是线网还是寄存器；如果没有声明，它被隐含地声明为和相应的输入、输出或输入输出有相同范围的线。如果端口被声明为矢量，两个声明的范围必须一样。
- 输入和输入输出不能声明为寄存器类型。
- （输出）端口不能声明为 real 或 realtime 类型。

提示

- 不要将端口放在测试装置模块。
- 不使用名字端口列表会使端口不易被理解。所以我们建议使用名字端口列表。

举例

```
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;
```

过程赋值

改变寄存器的值或为以后排定改变。

语法

{阻塞性赋值}

RegisterLValue = [*TimingControl*] *Expression ;*

{非阻塞性赋值}

RegisterLValue <= [*TimingControl*] *Expression ;*

RegisterLValue = {either}
RegisterName
RegisterName**[** Expression **]**
RegisterName**[** *ConstantExpression: ConstantExpression* **]**
Memory**[** *Expression* **]**
{ *RegisterLValue,...***}**

在何处使用

见语句。

规则

- 赋值到 reg 不带符号扩展。
- 类型是 real 或 realtime 的寄存器不允许位和部分选择。
- 当执行赋值语句时，算出右边的表达式的值。但左边的值不更新，直到产生定时控制事件或延时（被称为“内部赋值延时”）。
- 直到左边被更新后（即经过内部赋值延时后）阻塞性赋值才完成。Begin-end 块中的下一个语句直到此时才开始执行。Fork-join 块要直到所包含的所有阻塞性赋值都完成后才结束。
- 非阻塞性赋值预定的事件要直到在相同的仿真时间内所有模块化赋值事件被处理完后才开始生

效。

```
A <= #5 0;
A = #5 1;                          {A 在时刻 5 的时候值是 0}
```

注意

寄存器可以在超过一个 initial 或 always 内赋值。寄存器的值在任何时候都由最近的事件决定，而和事件的源无关。这和线网不一样。线网可以被两个或多个不同的源驱动，得到的值由线网的类型决定（wire、wand 等等）。

合并

- 合并忽略延时。
- 内部赋值事件不被合并。
- 相同的寄存器不能被超过一个 always 赋值。
- 相同的寄存器不能用非阻塞性和阻塞性赋值。
- 右边的表达式被合并到组合逻辑。在组合的 always 中，左边被合并到非完整赋值的线或锁存器。在带时钟的 always 中，非阻塞性赋值的左边作为触发器合并，阻塞性赋值的右边作为连接赋值，除非表达式在 always 外使用或者值在被赋值前读出。

提示

- 用非阻塞性赋值表示触发器和其他阻塞性赋值。这防止了有时钟的 always 之间的竞争。这样也使目的更清楚，帮助避免多余的触发器。
- 用一个简单的内部赋值延时避免 RTL 时钟时滞的问题，此时时钟树被建模。

举例

```
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                                   // 在 a 延时 5 个时间单位后完成

always @(posedge Clock)
begin
  c <= b;
  b <= a;                              // 用 b 的旧值
end
```

延迟一个非阻塞性赋值来克服时钟的时滞。

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

在时钟的第 5 个负跳变沿进行一个时钟周期的复位

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

过程连续赋值

当激活过程连续赋值时，它可以赋值给一个或多个寄存器，并防止普通的过程赋值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,...***}**

在何处使用

见语句。

规则

- 在执行完后，过程连续赋值对赋值的寄存器仍有效，直到寄存器被解除赋值或直到另一个过程连续赋值对相同的寄存器赋值。
- 当连续赋值再次有效时，寄存器的 force 会覆盖该寄存器的连续赋值直到它被释放。

注意

连续赋值和过程连续赋值尽管很相似，但它们是不一样的。请确保将 assign 放在正确的地方。过程连续赋值在允许使用语句的地方使用（例如 initial、always、任务、函数等内部）。连续赋值是在所有 initial 或 always 外。

合并

所有工具都不合并过程连续赋值。

提示

过程连续赋值用于建模异步复位和中断。

举例

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

always @(Reset)                    // 异步复位
  if (Reset)
    assign Count = 0;              // 禁止计数，直到 Reset 为低
  else
    deassign Count;                // 继续计数下一个 posedge Clock
```

编程语言接口

Verilog 编程语言接口（PLI）为用户从 Verilog 模块调用用 C 编程语言写的函数提供了一种方法。这些函数可以在实例化的 Verilog 数据结构动态地访问和修改数据，而且 PLI 还为此提供了一个 C 语言函数库。

通过用户定义的系统任务和函数（用于扩充内置的系统任务和函数）可以调用 PLI。用户定义的系统任务和函数和内置的系统任务和函数一样都有以$字符开始的名字。如果用户定义了和内置系统任务或函数名字一样的系统任务或函数，这些新建的系统任务或函数将使内置的系统任务或函数隐藏起来。

下面是 PLI 可能的应用：

- 延时计算器
- 测试矢量阅读器
- 图像波形显示
- 源代码调试器
- 接口用 C 或其他语言例如 VHDL 或硬件建模器编写的模块。

完整的编程语言接口讨论在本参考指南的范围外。

寄存器

寄存器保存在 initial、always、task 和 function 内赋的值。它们在行为建模中被广泛使用。

语法

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

RegisterOrMemory = {either}
  RegisterName
  MemoryName Range
Range = [ ConstantExpression: ConstantExpression ]
```

在何处使用

module-<HERE>-endmodule

begin : Label-<HERE>-end

fork : Label-<HERE>-join

task-<HERE>-endtask

function-<HERE>-endfunction

规则

- 寄存器只能用过程赋值来赋值。
- 在给定的设备中，integer 有最大大小，但它至少有 32 位。Time 寄存器的长度也有相似的保证，但它至少有 64 位。
- 类型是 integer 或 time 的寄存器一般像有相同数量位数的 reg 一样运转。Integer 和 time 的个别位和部分选择可以和 reg 一样建立。但在表达式中，integer 的值作为有符号数处理，而 reg 或 time 的值作为无符号数处理。
- 在同一时间只能访问（读或写）存储器数组的一个完整元素。要访问存储器数组一个元素的个别位，该元素的内容首先要被复制到一个适当的寄存器。

注意

- “寄存器”这个术语也表示一个硬件寄存器（即触发器），名字是假设表示一个软件寄存器（即一个变量）。Verilog 寄存器可以并用于描述和合并组合逻辑、锁存器、触发器和连接。
- Realtime 是最近添加到 Verilog 语言的一种类型，并不是所有工具都支持。
- 有符号数和无符号数的概念在 LRM 和不同仿真器的运转方式之间不完全一样。请小心使用宽度超过 32 位的带符号数和矢量。

合并

- Real、time 和 realtime 不合并。
- 在组合的 always 中，寄存器被合并到线，如果是非完全赋值则合并到锁存器。在有时钟的 always 中，寄存器被合并到线或触发器，由上下文决定。
- 整数用当前的工具合并到 32 位。值用二进制数表示。负数用 2 的补码表示。
- 存储器数组将合并到触发器或线，由上下文决定。它们不会合并到 RAM 或 ROM 元件。

提示

用 reg 描述逻辑，integer 描述循环变量和计算，real 在系统模型中使用，time 和 realtime 保存测试装置的仿真时间。

举例

```
reg a, b, c;
reg [7:0] mem[1:1024], byte;                    // ‘byte’不是存储器数组
integer i, j, k;
time now;
real r;
realtime t;
```

下面的片段显示了 reg 和 integer 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];
```

Repeat

按指定的次数重复一条或多条语句。

语法

```
repeat ( Expression)
  Statement
```

在何处使用

见语句。

规则

循环重复的次数由表达式的数字值决定。如果值是 0、X 或 Z，不进行循环。

合并

只和某些工具合并，而且只在循环被每个可执行分支的时钟事件打断（例如：@(posedge Clock)）时合并。

举例

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

保留字

下面是 Verilog 保留标识符的完整列表。这些保留的标识符必须不能作为用户定义的标识符，除非它们被转义或不是小写。

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	macromodule	rnmos	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	nmos	xor
event	or	or	

Specify

Specify 块用于描述从模块的输入到输出的通路延时以及设置和保持时间的定时限制。指定的块允许对设计的定时和设计的行为或结构分别进行描述。

语法

specify
  SpecifyItems...
endspecify

SpecifyItem = {either}
  Specparam
  PathDeclaration
  TaskEnable {只是定时检查}
PathDeclaration = {either}
  SimplePath = PathDelay;
  EdgeSensitivePath = PathDelay;
  StateDependentPath = PathDelay;
SimplePath = {either}
  (*Input*,... [*Polarity*] *> *Output*,...) {满}
  (*Input* [*Polarity*] => *Output*) {并行}

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	{上升、下降}
t,t,t	{上升、下降、关断}
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

在何处使用
module-<HERE>-endmodule

规则

- 路径从模块的输入开始，在模块的输出结束，而且模块内只有一个驱动。
- 满（*>）或并行（=>）连接在路径声明中描述。满连接表示所有从输入到输出的可能路径。并行连接表示一个指定输入的位到对应的指定输出的位。
- 在模块路径中可选的极性表示路径是反向的（正极）或不是反向的（负极）；它不影响仿真，但其他工具例如定时校验器可以使用。
- 跳变沿敏感的路径的数据表达式同样不会影响仿真。
- 基于状态的路径延时（SDPD）表达式可以只引用端口、常数和局部定义的寄存器或线网。一个有限的运算符集合在 SDPD 表达式中有效：按位（~ & | ^ ^~ ~^）、逻辑和相等（== != && || !）、归约（& | ^ ~& ~| ^~ ~^）、并置、复制和条件（{} {{}} ?:）。如果条件表达式为真，路径延时只影响路径。（SDPD 表达式中，1、X 和 Z 都被认为是真）。

- 如果没有 if 的条件是真，ifnone 定义了默认的 SDPD。在相同的路径同时有 ifnone SDPD 和简单的路径延时是非法的。
- 无条件路径的优先权比 SDPD 路径高。
- 在不同的条件或不同的边沿（或两个跳变沿）的情况下相同路径的跳变沿敏感的 SDPD 路径声明必须唯一。输出必须用相同的方法（整个端口、位选择或部分选择）在声明中引用。
- 如果模块包含指定的延时和分布式的延时（在门和 UDP 实例和线网上），每条路径使用它们中的最大值。

合并

合并工具忽略或禁止指定的块。

注意

- 并不是所有设备都支持列出的 12 种路径延时。
- 路径目的地的规则比很多设备当前支持的规则灵活。

提示

- 用指定的块描述库的单元的延时。请注意当建模库时如何计算延时。基于 PLI 的延时计算器需要访问设计中所有单元指定块的信息。
- 用指定的块描述与定时检验或合并工具（支持指定的块）相连接的“黑盒”元件的定时。

举例

```
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;

// 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;
// 简单路径，满连接
  (A, B *> F) = (1.2:2.3:3.1, 1.4:2.0:3.2);
// 简单路径，并行连接，正极
  (V + => W) = 3,4,5;
// 跳变沿敏感的路径，带极性
  (posedge Clk *> Q +: D) = (TLH$Clk$Q,THL$Clk$Q);
  (posedge Clk *> Qb -: D) = (TLH$Clk$Qb,THL$Clk$Qb);
// 基于状态的路径
```

```
  if (C) (X *> Z) = 5;
  if (!C && V == 8'hff) (X *> Z) = 4;
  ifnone (X *> Z) = 6;                                    // 默认 SDPD，X 到 Z
// 定时检查
  $setuphold(posedge Clk, D,
             Tsetup$Clk$D, Thold$Clk$D, Err);
  endspecify
endmodule
```

Specparam

像参数一样，但只在指定的块内使用。

语法

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

在何处使用

specify-<HERE>-endspecify

规则

- 在指定块内的常数表达式可以使用数字和 specparams，但不能用参数。Specparams 不能在指定的块外使用。
- Specparams 不能用 defparam 或在模块实例中用#覆盖。它们可以用编程语言接口（PLI）修改。

提示

- 在指定的模块使用 specparams 而不是文字数字。
- Specparams 应采用命名惯例，这样在必要的时候可以通过使用 PLI 的延时计数器修改。

举例

```
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
```

语句

硬件部件的行为可以用语句来描述。语句在定时控制（延时、事件控制和等待）定义的时间执行。当要求一起执行两条或多条语句时，语句要包含在 begin-end 或 fork-join 块内。Begin-end 块中的语句按顺

序执行；fork-join 块的语句并行执行。一个 initial 或 always 中的语句和其他 initial 或 always 的语句是同时执行的。

语法

{either}

; {空语句}

TimingControl Statement {语句可以为空}

Begin

Fork

ProceduralAssignment

ProceduralContinuousAssignment

Force

If

Case

For

Forever

Repeat

While

Disable

-> EventName; {事件触发器}

TaskEnable

在何处使用

initial-<HERE>

always-<HERE>

begin-<HERE>-end

fork-<HERE>-join

task-<HERE>-endtask {允许为空}

function-<HERE>-endfunction

if()-<HERE>-else-<HERE> {允许为空}

case- label:-<HERE>-endcase {允许为空}

for(<HERE>)-<HERE>

forever-<HERE>

repeat()-<HERE>

while()-<HERE>

强度

线网除了有逻辑值外还有强度值，这个值可以让你更精确地建模。线网的强度动态地从线网驱动器的强度取得。强度广泛地应用于开关级仿真以及线网有多个驱动器且驱动的值不一致的情况。

语法

{either}

(*Strength0, Strength1*)

(*Strength1, Strength0*)
(*Strength0*) {只是下拉原语}
(*Strength1*) {只是上拉原语}
(*ChargeStrength*) {只是 trireg 线网}

Strength0 = {either} supply0 strong0 pull0 weak0 highz0
Strength1 = {either} supply1 strong1 pull1 weak1 highz1
ChargeStrength = {either} large medium small

在何处使用
见线网、实例化和连续赋值。

规则

- 关键字 Strength0 和 Strength1 指定了线网驱动器分别驱动值 0 和 1 时的强度。
- 不允许强度说明（highz0、highz1）和（highz1、highz0）。Pullup 和 pulldown 门的强度说明不允许有 highz0 和 highz1。
- 默认的强度是（strong0、strong1），除了以下的情况：
  - pullup 和 pulldown 门的默认值分别是（pull1）和（pull0）。
  - Trireg 线网的默认值是（medium）。
  - Supply0 和 supply1 线网的强度通常是电源强度。
- 仿真中线网的强度是从线网的主驱动器（即实例或用最强的值连续赋值）取得。如果线网没有被驱动，它的值应该是高阻，除了以下的情况：
  - tri0 和 tri1 线网的值分别是 0 和 1 强度是上拉。
  - Trireg 线网保持最近被驱动的值。
  - Supply0 和 supply1 线网的值分别是 0 和 1，强度是电源。
- 强度值有“阶层”，范围从电源（最强）到 highz（最弱）。当解释有不一致驱动器的线网的逻辑值和强度值时，要用到两个强度值的相对位置。

电源
强
拉
大
弱
中等
小
Highz

合并
合并工具忽略强度。

提示
强度值可以在$display、$monitor 等中用格式说明符%v 显示。

举例

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

字符串

字符串可以作为例如$display 和$monitor 等系统任务的变量。字符串值可以作为数字保存在寄存器，而且可以和数字一样赋值、比较和并置。

语法

"String"

在何处使用

将表达式。

规则

- 源代码中字符串不能跨越多行。
- 字符串可以包含下面的转义字符：

\n	新行
\t	制表符
\\	反斜杠
\”	双引号
%%	百分号字符
\nnn	八进制 ASCII 码

- 系统任务例如$display 和$monitor 打印的字符串可以包含格式说明符（例如：%b），它们用特殊的方法解释（见$display）。
- 当字符串保存在寄存器中时，用 ASCII 码保存的每个字符都要求有 8 位。它不像 C 编程语言的字符串，不是用零字符（ASCII 的 0）终止。

注意

当在表达式中使用字符串时，请注意填补。字符串和数字的处理方法一样，如果字符的数量少于寄存器的位数就要在字符串的左边补 0。

举例

```
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
```

任务

任务是用于分割大块的语句或从几个地方执行一个公共的语句序列。

语法

```
task TaskName;
  [ Declarations...]
  Statement
endtask

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

在何处使用

```
module-<HERE>-endmodule
```

规则

- 不在任务中声明但在任务中使用的名字对应在调用模块中的名字。
- 任务可以有任意数量（包括没有）的输入、输出和输入输出变量。
- 变量（包括输入和输入输出）可以作为寄存器声明。如果它们没有被明确声明，变量就作为与对应的变量有相同范围的 reg 声明。
- 当使能一个任务后，对应于任务输入和输入输出的变量表达式的值被复制到相应的变量寄存器。当任务完成后，输入输出和输出变量寄存器的值被复制到任务使能语句相应的表达式。

注意

- 不像模块端口，任务的变量不在 task 后面的括号中定义。
- 如果任务包含超过一个语句，这些语句要用 begin-end 块或 fork-join 块包含起来。
- 任务的输入、输出和输入输出以及所有局部寄存器都被静态保存。这就是说甚至任务被使能（即调用）超过一次，这些寄存器也只有一个副本。如果任务在第一次使能没有完成前再次使能，输

入和输入输出寄存器以及可能的局部寄存器的值将会被覆盖。

- 当任务完成后，输出和输入输出的值只能被复制到使能任务的对应寄存器表达式。如果在赋值到输出或输入输出后任务有定时控制，使能任务中的相应寄存器将只在定时控制延时后更新。
- 相似地，不能非阻塞性赋值到输出或输入输出，因为任务返回时赋值仍无效。

合并

要合并，任务就不能包含定时控制。任务使能作为组合逻辑合并。

提示

- 程序有太多 always 会使 RTL 代码复杂化。考虑用一个 always 使能几个任务。
- 在测试设备中使用任务可以重复应用的激励序列。例如：从存储器读或写数据（见举例）。
- 被超过一个模块使用的任务可以在一个独立的模块中定义，它只包含任务而且用层次名字引用。

举例

下面是一个可被合并的简单 RTL 任务。

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

  if (Reset)                                    // 同步复位
    Count = 0;                                  // RTL 必须使用非阻塞性赋值
  else
    Count = Count + 1;
Endtask
```

下面的例子显示了在测试设备中使用任务。

```
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;
```

```
// 初始化错误计数器
   NErrors = 0;

// 向每个地址写地址值
   for ( i=0; i<=MaxAddr; i=i+1 )
      WriteRam(i, i);

// 读然后比较
   for ( i=0; i<=MaxAddr; i=i+1 )
   begin
      ReadRam(i, Data);
      if ( Data !== i )
         RamError(i,i,Data);
   end

// 错误数量的和
   $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
```

任务使能

一个使能（或“调用”）任务的语句。当任务被使能后，值被输入和输入输出变量传递到任务。当任务完成后，值通过任务的输出和输入输出变量从任务输出。

语法

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

在何处使用

见语句。

规则

- 任务可以从 initial 或 always 或者从其他任务使能。它们可以被递归调用。任务不能从函数调用。
- 任务使能语句的表达式顺序对应于变量在任务中声明的顺序。表达式的数量必须和变量的数量一样。
- 当任务变量是输入时，相应的表达式可以是任意的表达式。当任务变量是输入输出或输出时，表达式必须是过程赋值的左边有效的表达式。
- 当任务被使能后，输入和输入输出表达式被复制到相应的变量寄存器。当任务结束后，输出和输入输出寄存器的值被复制到列在对应的任务使能表达式中的寄存器。
- 任务可以被内部禁能或外部禁能。

注意

任务变量定义的隐式寄存器是静态的。因此如果任务被使能的同时相同的任务正在执行，那么输入和输入输出寄存器将被覆盖。

合并

要合并，任务就不能包含定时控制。任务使能作为组合逻辑合并。

举例

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

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

定时控制

用于延时或确定语句执行的时间。定时控制可以放在语句前或者放在过程赋值的=或<=与表达式之间。前者延迟执行它后面的语句，后者延迟赋值结果。

语法

{定时控制前的语句}

{either}
DelayControl
EventControl
WaitControl

{内部赋值定时控制}

{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 {寄存器、线网或事件的名字}

posedge Expression {01, 0X, 0Z, X1 or Z1}

negedge Expression {10, 1X, 1Z, Z0 or X0}

EventExpression or EventExpression

WaitControl = wait (*Expression*)

在何处使用

见语句和过程赋值（内部赋值定时控制）。

规则

- 语句前的事件或延时控制一般使紧接着的语句延时执行。
- 当到达 wait 时表达式是假（0 或 X），wait 控制只延时跟在它后面的语句；语句在表达式变真的时候执行。如果到达 wait 时表达式为真（非 0），跟在它后面的语句立即执行并不延时。
- 过程赋值右边的表达式在赋值执行的时候算出。如果没有内部赋值延时，左边的寄存器由于阻塞性赋值将立即更新，而非阻塞性赋值则下一个仿真周期才更新左边的寄存器。如果有内部赋值延时，左边的寄存器只在发生内部赋值延时后更新。
- 内部赋值延时必须是常数，但语句前的延时可以是常数或变量（即可能包括线网或寄存器）。
- 当“or”列表的其中一个事件发生，事件控制都会被触发。
- 对于 posedge 和 negedge，它们只测试表达式的最低位。另外，表达式的任何改变都触发事件。

注意

内部赋值延时是 0（#0）和无内部赋值延时并不一样。它和没有延时的非阻塞性赋值也不一样。#0 表示在当前所有挂起的事件赋值完成后但在非阻塞性赋值进行前执行的事件。(无内部赋值延时的非阻塞性赋值和内部赋值延时是#0 的延时一样）

合并

- 合并忽略延时。
- 合并工具不支持等待控制和内部赋值事件以及重复控制。
- 事件控制用于控制 always 的执行并决定合并什么逻辑。通常这些都放在 always 的顶部。这有时会被称为制约列表（*sensitivity list*）。

提示

内部赋值延时可以在 RTL 描述中使用，克服用赋值寄存器表示触发时的时钟时滞问题。

举例

```
#10
#(Period/2)
#(1.2:3.5:7.1)
@Trigger
@(a or b or c)
@(posedge clock or negedge reset)
wait (!Reset)
```

延迟一个非阻塞性赋值来克服时钟时滞。

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

在时钟的第 5 个负跳变沿产生长度是一个时钟周期的复位信号。

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

用户定义的原语

模块可以选择用用户定义的原语（UDP）建模小型的部件。它们用与内置的门一样的方式实例化。

语法

```
primitive UDPName (OutputName, InputName,...);
   UDPPortDeclarations ...
   UDPBody
endprimitive

UDPPortDeclaration = {either}
   output OutputName;
   input InputName,...;
   reg OutputName;                                    {时序 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                        {大小写不敏感}
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 {大小写不敏感}
LevelSymbol = {either} 0 1 x ? b {大小写不敏感}
EdgeSymbol = {either} r f p n * {大小写不敏感}

在何处使用

UDP 和模块一样在其他模块或 UDP 外声明。

规则

- UDP 有一个输出和至少一个输入。设备会限制输入的数量，但允许至少有 10 个输入。
- 如果 UDP 的输出被定义为 reg，UDP 是一个时序 UDP。否则是一个组合 UDP。
- 如果时序 UDP 的输出被初始化，初始值只在仿真开始的时候从任意原语实例的输出启动传播。
- 时序 UDP 可以是电平敏感或是边沿敏感的。如果表中至少有一个边沿指示器，时序 UDP 是边沿敏感的。
- UDP 的行为在一个表中定义。表中的行定义了不同输入条件的输出值。对于组合 UDP 来说，每行定义了一个或多个输入值组合后的输出。对于时序 UDP 来说，每行也定义了输出 reg 的当前值。一行可以有最多一个边沿变化的表项。当产生指定的边沿时，行定义了对应输入的输出值和当前输出 reg 的值。
- 在表中使用的特殊电平和边沿符号有下面的含意：

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

- 未指定的输入值和边沿组合的输出是未知（1'bX）。
- 不支持 Z 这个值。输入的 Z 作为 X 处理；输出值不允许有 Z。注意?的特殊含意，它在数字中的含意（和 Z 的含意相同）和现在的含意不一样。

注意

对于时序 UDP，如果在表的任何地方出现边沿，必须考虑输入所有可能的边沿，因为默认是边沿使输出的值为未知。

合并

所有工具都不合并 UDP。

提示

- 有时候，UDP 的仿真比行为模型更有效。当建模库部件例如 ASIC 单元时，请使用 UDP。

- 对于有多于一个输出的部件，请对每个输出使用一个独立的 UDP。
- 将注释放在 table 的顶部，指出该列是什么内容。

举例

```
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;                                             // 电平敏感的 UDP

  table
  // D Ena : old Q :   Q
     0   0 :    ?    :  0;
     1   0 :    ?    :  1;
     ?   1 :    ?    :   -;                         // 维持前面的值
     0   ? :    0    :  0;
     1   ? :    1    :  1;
  endtable
endprimitive

primitive DFF (Q, Clk, D);
  output Q;
  input Clk, D;

  reg Q;                                             // 边沿敏感的 UDP

  initial
    Q = 1;
```

```
  table
  // Clk     D    :  old Q  :    Q
     r       0    :  ?      :    0;                    //  时钟'0'
     r       1    :  ?      :    1;                    //  时钟  '1'
     (0?)    0    :  0      :    -;                    //  可能的时钟
     (0?)    1    :  1      :    -;                    //  " "
     (?1)    0    :  0      :    -;                    //  " "
     (?1)    1    :  1      :    -;                    //  " "
     (?0)    ?    :  ?      :    -;                    //  忽略下降时钟
     (1?)    ?    :  ?      :    -;                    //  " " "
     ?       *    :   -     :    -;                    //  忽略 D 的变化
  endtable
endprimitive
```

While

一个重复语句或语句块直到控制表达式为假的循环语句。

语法

```
while ( Expression)
  Statement
```

在何处使用

见语句。

合并

只有循环被时钟事件控制“中断”（例如：@(posedge Clock)）时可合并。

举例

```
reg [15:0] Word;

while (Word)
begin
  if (Word[0])
    CountOnes = CountOnes + 1;
  Word = Word >> 1;
end
```

Verilog 黄金参考指南

编译器伪指令

编译器伪指令

编译器伪指令是 Verilog 编译器的指令。编译器伪指令用后引号（`）有时也叫重音符号作为前缀。

编译器伪指令从它在源代码中出现的地方开始起作用，而且对随后处理的所有文件都有效，直到伪指令被取代或运行到最后一个文件的结束。

下面是 Verilog 编译器伪指令的摘要。一些重要伪指令的详细信息请看摘要后面的详细描述。

注意

编译器伪指令的作用由包含设计被编译的源代码的文件顺序决定。

标准编译器伪指令

下面的编译器伪指令在 Verilog LRM 中定义。

`celldefine 和`endcelldefine

分别在模块的前面和后面使用，将模块标记为库单元。这些单元被应用程序的某些 PLI 子程序例如延时计算器使用。

举例：

```
`celldefine
module Nand2 (...);                                    // Nand2 是一个“单元”
  ...
endmodule
`endcelldefine
```

`default_nettype

为隐式声明改变默认的线网类型。如果不出现这个伪指令，默认的线网类型是 wire。

举例：

```
`default_nettype tri1                                  // 允许所有 Verilog 的线网类型
```

`define 和`undef

`define 定义了一个文本宏。`undef 取消宏定义。

宏在编译的第一阶段就被取代。它们也用于控制条件编译（见`ifdef）。详细的信息请参考后面的`define。

`ifdef、`else 和`endif

由是否定义了指定的宏决定条件编译 Verilog 代码。详细信息请参看后面。

`include

使编译器读文件的内容，并在`include 伪指令的地方编译文件。

举例：

```
`include "definitions.v"
```

`resetall

将所有激活的编译器伪指令复位到它们的默认值。它们在每个 Verilog 源文件的顶部，防止编译上一个文件的编译器伪指令影响现在的文件，产生不期望的结果。

举例

`resetall

`timescale

定义仿真时间单位和精度。详细信息请参考后面。

`unconnected_drive 和`nounconnected_drive

`unconnected_drive 使不连接的模块输入上拉或下拉。`nounconnected_drive 恢复默认值，使不连接的输入悬空而且值是 Z。

举例：

```
`unconnected_drive pull0                    // 或‘pull1’
```

非标准的编译器伪指令

下面的伪指令不是 IEEE 标准的一部分，它们在 LRM 中起提供信息的作用。并不是所有 Verilog 工具都支持这些伪指令：

`default_decay_time

当没有明确给出衰减时间时，这个指令指定了 trireg 线网的默认衰减时间。

举例：

```
`default_decay_time 50
`default_decay_time infinite                // 意味着不会衰减
```

`default_trireg_strength

用整数指定 trireg 线网的默认强度。用整数建模强度是 Verilog 语言的非标准扩展。

举例：

```
`default_trireg_strength 30
```

`delay_mode_distributed、`delay_mode_path、`delay_mode_unit 和`delay_mode_zero

这些伪指令影响仿真延时的方法。

分布延时（*Distributed delays*）是原语实例延时、连续赋值延时和线网延时。路径延时（*Path delays*）定义在指定的块中。单元（*Unit*）和零（*zero*）延时取代所有分布延时和路径延时，而且会导致仿真加快，但损失了实际的延时信息。

默认地，仿真器会选择最长的分布延时和路径延时。

`define

`define 定义了一个文本宏。宏在编译的第一阶段被替代。宏可以增加 Verilog 代码的可读性和可维护性，找出参数或函数不正确或不允许的地方。

语法

{声明}

`define Name[(Argument,...)] *Text*

{用法}

`Name [(*Expression,...*)]

在何处使用

宏可以在模块的内部或外部定义。

规则

- 和所有编译器伪指令一样，宏定义在超过文件边界的时候仍有效，除非被后面的`define、`undef 或 `resetall 伪指令覆盖。这个指令不受范围限制。
- 当用变量定义宏时，变量可以在宏正文使用，而且在使用宏的时候可以用实际的变量表达式代替。

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

- 通过用反斜杠（\）转义中间换行符，宏定义可以跨越几行。新的行是宏正文的一部分。
- 宏正文不能分离以下的语言记号：注释、数字、字符串、名字、保留的名字、运算符。
- 编译器伪指令不允许作为宏的名字。

注意

- 不是所有设备都支持带变量的宏。
- 定义了宏后，在使用的时候要在宏的名字前加前缀后引号（`）。没有后引号的宏名是一个独立的标识符。
- 请注意区别后引号（`）和在数字中使用的单引号（'）。
- 不要用分号结束宏定义，除非你想使用宏的时候有分号替代。否则使用宏的时候会出现语法错误。

提示

最好使宏的参数有在字面上有意义的名字。

仿真带变量的宏比仿真功能相同的函数更高效。

举例

这个例子展示了如何在层次设计中用文本宏选择模块的不同设备。这在合并时很有用，特别是在仿真 RTL 和合并门电路的混合设计时。

```
`define SUBBLOCK1 subblock1_rtl
`define SUBBLOCK2 subblock2_rtl
`define SUBBLOCK3 subblock3_gates

module TopLevel ...
  `SUBBLOCK1 sub1_inst (...);
  `SUBBLOCK2 sub2_inst (...);
  `SUBBLOCK3 sub3_inst (...);
  ...
endmodule
```

下面的例子是一个带变量的文本宏。

```
`define nand(delay) nand #(delay)

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

`ifdef
由是否定义了指定的宏决定条件编译 Verilog 代码。

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

在何处使用
任何地方。

规则
- 如果宏的名字已经用了`define 定义，那么只编译 Verilog 代码的第一个块。
- 如果没有定义宏的名字，而且出现`else 伪指令，那么只编译第二个块。
- 这些伪指令可以嵌套。
- 不被编译的代码都应是有效的 Verilog 代码。

提示
可以用于切换模块的其他设备或者选择启动写诊断信息。

举例
```
`define primitiveModel

module Test;
...
`ifdef primitiveModel
  MyDesign_primitives UUT (...);
`else
  MyDesign_RTL UUT (...);
`endif

endmodule
```

`timescale
定义了时间单位和仿真精度（最小的增量）。

语法
`timescale *TimeUnit* / *PrecisionUnit*

TimeUnit = *Time Unit*

PrecisionUnit = *Time Unit*

Time = {either} 1 10 100

Unit = {either} s ms us ns ps fs

在何处使用

<HERE>-module

规则

- `timescale 伪指令像其他所有编译器伪指令一样，影响该指令后面的所有模块，不管是相同的文件还是独立编译的文件，直到出现下一个`timescale 或`resetall 伪指令。
- 精度单位必须小于或等于时间单位。
- 仿真运行的精度是`timescale 伪指令中所有精度单位的最小值。所有延时都按最近的精度单元四舍五入。

提示

甚至在模块没有延时的情况下也在每个模块的顶部加入`timescale 伪指令，因为有些仿真器对此有要求。

举例

`timescale 10ns / 1ps

Verilog 黄金参考指南

系统任务和函数

系统任务和函数

Verilog 语言包含大量有用的系统任务和函数。它们和用户定义的任务和函数一样使能和调用。它们保证在任何遵从 IEEE Verilog 标准的工具中都能使用。Verilog LRM 也提出了其他一些经常使用的系统任务和函数，但这些不是标准的一部分，有部分工具支持。

注意：不同厂商的 Verilog 仿真器可以包含额外的、专有的系统任务和函数，用户可以用编程语言接口（PLI）添加用户定义的系统任务和函数。

所有系统任务和系统函数名（包括用户定义的）都要以美元字符$开始，使它们和普通的任务和函数区分开来。

下面是 LRM 提出的所有系统任务和函数的摘要。一些重要的系统任务和函数的详细信息请参考摘要后面的部分。

标准的系统任务和函数

下面的系统任务和函数是 IEEE 标准的一部分。

$display, $monitor, $strobe, $write 等

这是写文本到标准输出或者到一个或多个文件的整个系统任务系列。完整的信息请参考后面。

$fopen 和$fclose

```
$fopen("FileName");                                    {返回一个整数}
$fclose( Mcd);
```

$fopen 是打开一个文本文件进行写操作的一个系统函数。$fclose 是关闭用$fopen 打开的文件。

完整的信息请参考后面。

$readmemb 和$readmemh

```
$readmemb("File",MemoryName[, StartAddr[, FinishAddr]]);
$readmemh("File",MemoryName[, StartAddr[, FinishAddr]]);
```

从文本文件初始化存储器数组的任务。完整的信息请参考后面。

$timeformat

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

定义了用$display 等写仿真时间的格式。完整的信息请参考后面。

$printtimescale

```
$printtimescale[(ModuleInstanceName)];
```

用下面的格式显示模块的时间单位和精度：

Time scale of (module_name) is unit / precision.

如果没有给出变量，显示调用$printtimescale 得到的模块时间单位和精度。

$stop

```
$stop[( N)];                                           {N 是 0、1 或 2}
```

使仿真停止。可选的变量决定了诊断输出产生的类型。0 给出最少数量的输出，1 给出多一点，2 给出最大数量的输出。

$finish

$finish[(N)]; {N 是 0、1 或 2}

使退出仿真，将控制传输回到操作系统。如果提供变量，诊断信息的打印如下：

- 0——不打印。
- 1——打印仿真时间和位置（如果没有提供变量，这是默认的）
- 2——打印仿真时间和位置，以及存储器的统计信息和在仿真中 CPU 所用的时间。

$time、$stime 和$realtime

$time;

$stime;

$realtime;

返回当前仿真时间的系统函数。返回时间的单位和系统函数被调用的模块中用`timescale 定义的单位一样。

- $time 返回一个 64 位的无符号数，四舍五入到最接近的单位。
- $stime 返回一个 32 位的无符号数，删节大的时间值。
- $realtime 返回一个实数。
- 注意这些函数和 Verilog 的其他函数不一样，它们没有输入。

$realtobits 和$bitstoreal

$realtobits(RealExpression) {返回一个 64 位的值}

$bitstoreal(BitValueExpression) {返回一个实数值}

在实数和位级表达之间转换，因此实数可以通过模块的端口传输。（端口不允许被声明为 real）。例子请见模块。

$rtoi 和$itor

$rtoi(*RealExpression*) {返回一个整数}

$itor(*IntegerExpression*) {返回一个实数}

在实数和整数之间转换。$rtoi 删节实数以形成整数。

PLA 建模任务

下面的系统任务提供用于建模 PLA。

$async$and$array(MemoryName,{Inputs,...},{Outputs,...})

$async$nand$array(MemoryName,{Inputs,...},{Outputs,...})

$async$or$array(MemoryName,{Inputs,...},{Outputs,...})

$async$nor$array(MemoryName,{Inputs,...},{Outputs,...})

$async$and$plane(MemoryName,{Inputs,...},{Outputs,...})

$async$nand$plane(MemoryName,{Inputs,...},{Outputs,...})

$async$or$plane(MemoryName,{Inputs,...},{Outputs,...})

$async$nor$plane(MemoryName,{Inputs,...},{Outputs,...})

$sync$and$array(MemoryName,{Inputs,...},{Outputs,...})

$sync$nand$array(MemoryName,{Inputs,...},{Outputs,...})

$sync$or$array(MemoryName,{Inputs,...},{Outputs,...})

$sync$nor$array(MemoryName,{Inputs,...},{Outputs,...})

$sync$and$plane(MemoryName,{Inputs,...},{Outputs,...})

$sync$nand$plane(MemoryName,{Inputs,...},{Outputs,...})

$sync$or$plane(MemoryName,{Inputs,...},{Outputs,...})

$sync$nor$plane(MemoryName,{Inputs,...},{Outputs,...})

这些任务的第一个变量是存储器数组的名字，数组保存了被建模的 PLA 特性。数组可以用升序声明（例如：reg [1:NInputs] Mem[1:NOutputs]）。特性可以动态改变。

异步的任务被模拟为过程连续赋值。当其中一个输入改变或者特性改变后，输出立即更新。同步任务只在任务被调用的时候更新输出。

随机建模的任务

$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*); {返回一个整数}

$q_exam(*q_id, q_stat_code, q_stat_value, status*);

通过使能队列的创建和管理，4 个系统任务和一个系统函数支持随机建模。完整的详细信息请看后面。

随机数发生函数

$random[(*Seed*)];

$dist_chi_square(*Seed, DegreeOfFreedom*);

$dist_erlang(*Seed, K_ stage, Mean*);

$dist_exponential(*Seed, Mean*);

$dist_normal(*Seed, Mean, StandardDeviation*);

$dist_poisson(*Seed, Mean*);

$dist_t(*Seed, DegreeOfFreedom*);

$dist_uniform(*Seed, Start, End*);

当重复调用这些系统函数时，返回的是根据不同的概率分布的一系列伪的随机数。如果开始的种子值一样，序列将永远是相同的。

详细信息请查阅分布函数及其应用的统计或可能性理论的文章。

指定的块定时检查

$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*]]);

指定的系统任务执行公共定时检查。这些系统任务只能从指定的块调用。完整的详细信息请看后面。

值改变转储任务

$dumpfile("FileName");

$dumpvars[(Levels, ModuleOrVariable,...)];

$dumpoff;

$dumpon;

$dumpall;

$dumplimit(FileSize);

$dumpflush;

这系列系统任务保存在值改变转储（VCD）文件内值的改变。VCD 文件是将仿真激励或结果传递到其他程序的一种方法，例如图像波形浏览器。完整的信息请参考后面。

非标准系统任务和函数

下面的系统任务和函数在 LRM 中提出，但不是 IEEE 标准的一部分。

某些这类系统任务和函数被认为是 Verilog 仿真器的“交互模式”。如果仿真器支持交互模式的操作，这些任务和函数可以作为命令接收。

$countdrivers

$countdrivers(*Net,* [*IsForced, NoOfDrivers,*

NoOfDriversTo0, NoOfDriversTo1, NoOfDriversToX]);

这个系统函数是显示在指定的标量线网或矢量线网的位选择上驱动器的数量。驱动包括原语的输出和连续赋值，不包括激活的 force。如果线网有超过一个驱动器$countdrivers 返回值 0，否则返回值 1。所有变量（除了第一个）都返回整数值。

- 如果线网是 forced 的，isforce 返回 1，否则返回 0。
- *NoOfDrivers* 返回驱动器的数量。
- 剩余的变量返回值可以添加到 *NoOfDrivers*。

$list

$list[(*ModuleInstance*)];

交互调用以列出当前范围或设计的指定范围的源代码。

$input

$input("FileName");

从文本文件读出交互命令。

$scope 和$showscopes

$scope(*ModuleInstance*);

$showscopes[(*N*)];

用交互命令设置当前和范围并显示当前范围以及后面（如果 N 存在而且非 0）的范围。

$key、$nokey、$log 和$nolog

$key[("FileName")];

$nokey;

$log[("FileName")];

$nolog;

“key”文件记录了交互地输入的命令。“log”文件记录了在仿真运行中写到标准输出的所有信息。

$nokey 和$nolog 禁能记录。没有变量的$key 和$log 重新使能记录。如果有文件名变量，它们将创建

新的文件。

$reset、$reset_count 和$reset_value

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

$reset_count; {返回一个整数}

$reset_value; {返回一个整数}

$reset 复位仿真器，使仿真器重新启动。

- *StopValue* 的值是 0 表示仿真器在交互模式复位，允许用户启动和控制仿真。非 0 的值表示仿真将自动重新启动。
- *ResetValue* 的值可以由$teset_value 函数读出。
- *DiagnosticsValue* 指定了复位前工具显示的信息种类。

$reset_count 返回$reset 被调用的次数。

$reset_value 返回传递到$reset 的值。

$save、$restart 和$incsave

$save("FileName");

$incave("FileName");

$restart("FileName");

$save 保存了仿真文件的完整状态，因此可以用$restart 读出。

$incsave 只保存最近一次调用$save 后的改变。

$restart 从完整或增量 save 文件复位仿真。对于增量保存，前面必须有完整的 save 文件，因为它要在增量 save 文件中引用。

$showvars

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

显示线网和寄存器在标准输出的状态。这个任务可以交互使用。显示的状态信息不在 LRM 中定义。它可能包含线网和寄存器的当前值、线网和寄存器的任意预设事件以及线网的驱动。

如果没有给出变量列表，显示的是当前范围的所有线网和寄存器的信息。

$getpattern

$getpattern(*MemoryElement*);

$getpattern 是系统函数，它只在连续赋值中使用。连续赋值的左边必须是标量线网的并置。$getpattern 还和$readmemb 以及$readmemh 一起从文本文件应用测试矢量。当包含大量的标量输入时，用$getpattern 可以快速处理。

$sreadmemb 和$sreadmemh

$sreadmemb(*Memory, StartAddr, FinishAddr, String, ...*);

$sreadmemh(*Memory, StartAddr, FinishAddr, String, ...*);

这些任务相似于$sreadmemb 和$sreadmemh，除了存储器是由一个或多个字符串提供的数据初始化而不是从文件初始化。字符串格式和$sreadmemb 和$sreadmemh 对应的文本文件一样。

$scale

$scale(DelayName); {返回 realtime}

转换一个模块的时间值到被调用的模块的$scale 时间单位。$scale 对延时值采取层次引用，例如参数

在另一个模块，将参数的时间单位转换成参数被调用的模块的时间单位。

$display 和$write

写格式化的文本到标准输出或仿真器的 log 或者一个文件。

语法

$display(*Argument,...*);

$fdisplay(*Mcd, Argument,...*);

$write(*Argument,...*);

$fwrite(*Mcd, Argument,...*);

Mcd = Expression {整数值}

规则

- $display 和$write 的唯一区别是$display 在文本的结束写一个换行字符而$write 不写。
- 变量可以是字符串或表达式或者空白（两个相邻的逗号）。
- 写出的字符串可能包含格式说明符（见下面）。如果是，每个字符串的后面必须紧跟足够的表达式为字符串的所有格式表达式提供值（%m 除外）。
- 字符串可以包含以下的转义字符：

\n	换行
\t	制表符
\”	双引号
\\	反斜杠
\nnn	（八进制）字符的 ASCII 值

- 未知和高阻值被如下写出。注意八进制和十六进制数一个数字分别表示 3 位或 4 位。对于十进制数未知和高阻值用一个数字表示。
  - 小写的 x 或 z 表示这个数字代表的所有位都未知。
  - 大写的 X 或 Z 表示这个数字代表的某些位，不是所有位，未知。
- 如果一个变量列表包含两个相邻的逗号，这里将写入一个空格。

格式说明符

- 字符串允许有下面的格式说明符：

%b %B	二进制
%o %O	八进制
$d $D	十进制
%h %H	十六进制
%e %E %f %F %g %G	实数
%c %C	字符
%s %S	字符串
%v %V	二进制和强度
%t %T	时间
%m %M	层次实例

- %v 打印的强度如下：电源- Su、强- St、拉- Pu、大- La、弱- We、中等- Me、小- Sm、高- Hi。%v 也打印值 H 和 L（这些用%b 打印时是 X）。
- 最小的区域宽度值在%字符后面（例如%10d）。对于十进制数，前面的零用空格代替，但对于其他基数前面的零也要打印出来。最小的区域宽度是 0 表示区域已经足够大可以显示值。
- 实数的说明符格式(%e、%f 和%g)和 C 编程语言的格式完全兼容。例如%10.3g 指出最小的区域宽度是 10，小数点后面有 3 位。
- 不用格式说明符书写的表达式用十进制格式书写。但一些额外的任务有不同的默认格式，例如：$displayb、$fwriteo 和$displayh 在不用格式说明符时分别将数字表达式写成二进制、八进制和十六进制值。

举例

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

$fopen 和$fclose

$fopen 是打开一个文件进行写操作的系统函数，$fclose 是关闭一个文件的系统任务。用$fdisplay、$fmonitor 等系统任务将文本写入到文件。

语法

```
$fopen("FileName");                          {返回一个整数}
$fclose( Mcd);

Mcd = Expression                             {整数值}
```

在何处使用

见语句。

规则

- 一次可以打开高达 32 个文件，但最大值可能低一些，由操作系统决定。
- 当调用$fopen 函数时，它返回一个与文件关联的 32 位的无符号多信道描述符或者是 0（如果文件不能打开进行写操作）。
- 多信道描述符可以认为是 32 个标志，每个表示一个独立的文件（信道）。位 0 和标准输出关联，位 1 和第一个打开的文件关联，位 2 和第二个打开的文件关联，如此类推。当一个文件的输出系统任务例如$fdisplay 被调用时，第一个变量是一个多信道描述符，它表示在哪里写文本。文本写在多信道描述符的标志被置位的文件。

举例

```
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
```

$monitor 等

当一个或多个指定的线网或寄存器列表改变值的时候，写出一行文本。这个命令用于在测试设备中监控仿真行为。

语法

```
$monitor( Argument,...);
$fmonitor( Mcd, Argument,...);
$monitoron;                                  {打开监控标志}
$monitoroff;                                 {关闭监控标志}

Mcd = Expression                             {整数值}
```

规则

- 这些系统任务的变量的语法和它们所写的文本和几乎和$display 任务一样。
- 只有一个$monitor 进程，但同时可以运行任意数量的$fmonitor 进程。
- 第二次或者再次调用$monitor 会取消任何现有的$monitor 进程，而且用新的$monitor 进程代替。
- $monitoroff 禁能监控，$monitoron 重新使能监控。基于当前的$monitor 进程而不管值是否发生改变，$monitoron 立即产生监控显示。
- 没有$fmonitor 相当于$monitoron 和$monitoroff。
- 系统函数$time、$stime 和$realtime 不从$monitor 等或$fmonitor 等触发显示。

提示

在测试设备中用$monitor 从任何符合 Verilog 的仿真器获得仿真结果。用于创建波形的图像显示的任务通常由仿真器决定。

举例

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

$readmemb 和$readmemh

用来自文本文件的值初始化存储器数组。文本文件的内容应该是二进制格式（$readmemb）或十六进制格式（$readmemh）。

语法

{系统任务调用}

```
$readmemb("File",MemoryName[, StartAddr[, FinishAddr]]);
$readmemh("File",MemoryName[, StartAddr[, FinishAddr]]);
```

{文本文件}

{either} *WhiteSpace DataValue* @ *Address*

WhiteSpace = {either} Space Tab Newline Formfeed

DataValue = {either}

BinaryDigit...	{$readmemb}
HexDigit...	{$readmemh}

Address = *HexDigit...*

规则

- 第一个变量是一个 ASCII 文件的名字。这个文件可以只包含空白、Verilog 注释、（hex）地址值以及二进制或十六进制数据。
- 第二个变量是存储器数组的名字。
- 数据值必须和存储器数组的宽度相同，而且用空白分隔。它们可以被读入从数组起始处开始或者从起始地址开始（如果有指定）的连续的存储器单元。数组值被连续读出，直到文件结束或到达指定的结束地址。
- 地址值是带前缀@的十六进制数（对于$readmemb 也是）。当遇到地址值时，下一个数据字被读入该地址。

合并

不合并。合并工具忽略它们的作用。从存储器数组得到的触发器将不在合并的设计中初始化；如果要求上电复位，就要明确地编码。

提示

存储器数组可以用于保存从文本文件读出的激励。这是不用编程语言接口（PLI）扩展语言或非标准语言扩展将数据读入 Verilog 仿真的唯一一种方法。

举例

```
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 =%b : f=%b",
             $realtime, a, b, c, d, f);
endmodule
```

$strobe

当该时刻的所有事件处理完后，在这个时间步的结尾打印一行格式化的文本。

语法

$strobe(*Argument,...*);

$fstrobe(*Mcd, Argument,...*);

Mcd = Expression {整数值}

规则

- 这些系统任务的变量的语法和它们所写的文本和几乎和$display 任务一样。
- 当$strobe 被调用的时刻所有活动都完成了，$strobe 才打印文本。这包括所有阻塞性和非阻塞性赋值的作用。

提示

在写仿真结果时请尽量使用$strobe 少用$display 或$write。这保证了选通的线网和寄存器被写入稳定的值。

举例

```
initial
begin
  a = 0;
  $display(a);                    // displays 0
  $strobe(a);                     // displays 1 ...
  a = 1;                          // ... 因为这条语句
end
```

$timeformat

定义了打印仿真时间的格式。$timeformat 用于连接格式说明符%t。

语法

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

规则

- *Unit* 是 0 到-15 之间的整数值，表示打印的时间的单位：0 表示秒，-3 表示毫秒，-6 表示微秒，-9 表示毫微秒，-12 表示微微秒，-15 表示毫微微秒。中间值也可以使用：例如-10 表示 100ps 单位。
- *Precision* 是在小数点后面要打印的小数位数。
- *Suffix* 是在时间值后面打印的一个字符串。
- *MinFieldWidth* 是打印的最小数量字符，包括前面的空格。如果要求更多字符，那么打印的字符更多。
- 如果没有指定变量，默认地使用下面的值：Units：仿真精度；Precision：0；Suffix：空字符串；MinFieldWidth：20 个字符。

提示

用`timescale、$timeformat 和$realtime（带%t）指定和显示仿真时间，用$display、$monitor 或其他显示任务。

举例

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

随机建模

Verilog 提供了一组通过使能创建和管理队列来支持随机建模的系统任务和函数。

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*); {返回一个整数}
$q_exam(*q_id, q_stat_code, q_stat_value, status*);

在何处使用
见语句。

总体注释

- 这些系统任务和函数的所有变量都是整数。
- 这些系统任务和函数都返回一个状态整数，值以及说明如下：
  - 0——Okay
  - 1——队列满：不能增加工作（$q_add）
  - 2——未定义的 q_id
  - 3——队列空：不能删除工作（$q_remove）
  - 4——不支持的队列类型：不能创建队列（$q_initialze）
  - 5——最大长度是 0 或小于 0：不能创建队列（$q_initialze）
  - 6——重复的 q_id：不能创建队列（$q_initialze）
  - 7——存储器不足够：不能创建队列（$q_initialze）

$q_initialze
创建一个队列

- q_id（输出）是唯一的队列标识符，用于在调用其他队列任务和函数时指出队列。
- q_type（输入）是 1 表示先进先出（FIFO）队列，或者是 2 表示后进先出（LIFO）队列。
- max_lengtb（输入）是队列允许的最大数量条目。

$q_add
向队列添加一个条目。

- q_id（输入）指出在哪个队列添加条目。
- job_id（输入）识别工作。这通常是一个整数，在队列每增加一个元素时由用户加 1，可以用于识别被删除的元素。
- inform_id（输入）用于联系信息和队列条目。它的含意是用户定义的。

$q_remove
从队列取出一个条目。

- q_id（输入）指出在哪个队列删除条目。
- job_id（输出）识别工作（见#q_add）。
- inform_id（输出）是$q_add 保存的值。

$q_full
查看是否队列已满。如果返回值是 1，表示队列满；如果返回值是 0，队列未满。

$q_exam
要求对队列进行统计。下面描述中所提出的时间是基于元素添加到队列的时间（到达时间）和元素被

添加和删除（等待时间）的时间差。时间的单位是仿真精度。

- q_stat_code（输入）指出要求的信息：
  - 1——当前的队列长度。
  - 2——平均交互到达时间
  - 3——最大队列长度
  - 4——通常的最短等待时间
  - 5——仍在队列中的工作的最长等待时间
  - 6——队列中的平均等待时间
- q_stat_value（输出）返回要求的信息。

举例

```
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;                                             // 写错误信息然后退出
    ...
  endtask

  initial
  begin
  // 创建队列
    $q_initialize(Queue, Lifo, QueueMaxLen, Status);
    if ( Status )
      Error("Couldn't initialize the queue");
// 添加工作
    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: ");
      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
// 队列现在应当是满的
      IsFull = $q_full(Queue, Status);
      if ( Status )
        Error("Couldn't see if queue is full");
      if ( !IsFull )
        Error("Queue is NOT full");
// 删除工作
      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
```

定时检查

Verilog 提供了大量只能从指定的块调用的系统任务，它们用于执行公共的定时检查。

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

- 参考事件的转换为定时检查建立了参考时间。参考事件必须被引用到模块的输入或输入输出。
- 数据事件的转换初始化定时检查。数据时间必须被引用到模块的输入或输入输出。
- 当引用事件和数据时间同时发生时不会报告设置违犯。
- 对于$width，脉冲比阀值（如果给出）还短不会产生违犯。
- 后面的时间检查的引用事件必须是边沿触发的语句：$width、$period、$recovery、$nochange。
- 引用事件要用关键字 edge，$recovery 和$nochange 除外，它们只允许用 posedge 和 negedge。
- 只有条件为真时才进行条件定时检查（使用&&&记号）。
- 如果存在通告变量，它必须是一个寄存器。出现违规的时候，寄存器的值改变。如果原来是 X 现在变为 0，如果原来是 0 现在就是 1，如果原来是 1 现在就为 0。如果值是 Z，则不改变。

注意

- 注意：这些系统任务只能在指定的块中调用。它们不能作为过程语句调用。
- 对于$setup，ReferenceEvent 和 DataEvent 变量的顺序是颠倒的！

提示

对于复杂的条件，请在指定的块的外部描述条件，并且驱动一个条件信号（wire 或 reg）在指定的块内使用。

举例

```
reg Err, FastClock;                                    // 通告寄存器

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
```

值改变转储

一系列系统任务可以将值的改变保存到值改变转储（VCD）文件中。VCD 文件是将仿真激励或结果传递到另一个程序（例如一个图像波形浏览器）的一种方法。

语法

```
$dumpfile("FileName");
$dumpvars[( Levels, ModuleOrVariable,...)];
$dumpoff;                                   {挂起转储}
$dumpon;                                    {继续转储}
$dumpall;                                   {转储一个检查点}
$dumplimit( FileSize);
$dumpflush;                                 {更新转储文件}
```

在何处使用

见语句。

规则

- Levels 是任何指定文件要转储的层次级别数量，1 表示只是指定的层次级别，0 表示指定的级别以及下面的所有实例。
- 如果没有给出变量，将转储设计中的所有变量。
- FileSize 是按字节计算的最大转储文件大小。
- $dumpvars 可以调用超过一次，但每次调用必须在相同的时间（通常是在仿真的开始）。

举例

```
module Test;

  ...

  initial
  begin
    $dumpfile("results.vcd");
    $dumpvars(1, Test);
```

```
    end

// 执行设计的周期性检查点
    initial
        forever
            #10000 $dumpall;
endmodule
```

Verilog 黄金参考指南

命令行选项

命令行选项

命令行选项在调用不是 Verilog 语言一部分的 Verilog 仿真器时使用，而且没有在 LRM 中提出，但很多 Verilog 仿真器支持命令行选项的公共组合以及它们自己的所有的一些选项。

共有两类命令行选项：UNIX 命令选项——一个减号加上一个字符（例如：-s）以及“加变量”，形式是+word。一些 UNIX 命令行选项后面跟的是一个值，譬如文件名（例如：-f 文件）。

下面是最常用的命令行选项。注意：不是所有仿真器都支持这些选项。

- **-f** *CommandFile*——从 *CommandFile* 以及从命令行进一步读命令行选项。
- **-k** *KeyFile*——将在仿真期间输入的所有交互命令记录在文件 *KeyFile*。
- **-l** *LogFile*——将仿真器信息（包括$display 等的输出）以及标准输出记录在 *LogFile*。
- **-r** *SaveFile*——从（非标准）系统任务$save 创建的文件重启仿真。
- **-s**——在时刻 0 中断仿真器。这允许交互地控制仿真。
- **-u**——将 Verilog 源代码（除了字符串）看作完全由大写字符组成。请小心地使用这个选项。
- **-v** *LibraryFile*——在 *LibraryFile* 中查找缺少的模块或 UDP。只有被实例化但不在设计的剩余部分定义的模块或 UDP 可以从 *LibraryFile* 编译。不在设计中使用的模块或 UDP 不被编译。
- **-y** *LibraryDirectory*——在 *LibraryDirectory* 的文件中查找缺少的模块或 UDP。模块期望被定义在库目录中与模块名字相同的文件中。如果给出命令行选项+libext+ *extension*，它指定了模块名后面的文件扩展名，然后就能得到文件名。例如：-y mylib +libext+.v 是指在文件 mylib/mycell.v 中查找缺少的模块‘mycell’。
- **+define+** *MacroName*——定义了名叫 *MacroName* 的文本宏，值是零。这种宏可以在`ifdef 语句中使用。
- **+incdir+** *Directory*[+ *Directory*...]——定义了一个查找包含在`include 里文件的目录列表。查找从当前的目录开始，如果在当前的目录没有找到包含文件，查找会按+incdir 目录的顺序继续进行。
- **+libext+** *Extension*——定义了库文件的扩展名。见上面的-y。
- **+notimingchecks**——在指定的块关闭定时检查。这可以加速仿真或者抑制假的定时错误信息。请小心使用这个选项。
- **+mindelays**、**+typdelays**、**+maxdelays**——在整个设计中分别使用最小延时、典型延时或最大延时。默认是使用典型延时。不能在同一仿真运行中混合使用最小延时、典型延时或最大延时。

注意

Verilog 仿真器不能查出拼写错误的“加”变量。这是因为用户可以自定义“加”变量。因此在拼写譬如“+maxdelays”这样的选项时要格外小心。