

度,通常强度(strength)保持为 Pull,逻辑值保持为 0(对 tri0)或 1(对 tri1)。

(2) 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:

(3) 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。

(4) 综合工具只支持 Net 类型中 wire 型的综合,其他的 Net 类型均不支持。

提 示:

(1) 在每个模块的块明确地声明所有的 nets,即使是默认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。

(2) 只能用 supply0 和 supply1 来声明地和电源。

附表 12 各种真值表

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

举例说明:

```
wire Clock;
wire [7:0] Address;
tril [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
      g = a || b; // 连续赋值
```

参阅连续赋值和寄存器类型语句的说明。

## 27. Number 数

数指整数或者实数。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...

```

在程序中所处位置:

参阅 Number 表达式的说明。

规 则:

- (1) 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。
- (2) 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。
- (3) 负数表示为其二进制的补数。
- (4) 数字的第一个字符不允许出现下画线“\_”, 但标识符可以。为了提高数字的可读性,

可用下画线把长的数字分段,但在处理数字时下画线将被忽略。

(5) 位宽指明了数字的准确位数。

(6) 不指明位宽的数字,它的位宽应为 32 位或 32 位以上,这取决于主机字长。

(7) 如果位宽大于实际的二进制位数时,高位部分补 0,但若左边最高位是 X 或 Z,在这种情况下,则补 X 或 Z。

(8) 如果位宽小于实际的二进制数位时,超过位宽的高位(左边)将被舍去。

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:

(1) 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 语句外,如用其他的条件语句,与 X 的比较都认为是假的。case 等式运算符 == 和 != 在一般情况下都是不可综合的。

(2) 除了在 caseX 和 caseZ 语句中的 Z 被认为是无关项外,在其他情况下 Z 则被用来表示三态驱动器。

提示:

(1) 在 case 语句的标号中,通常用? 要比用 Z 好。在程序的其他地方不要使用? 号,否则会产生混淆。

(2) 用下画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 6 的八进制数
8'bx          // 位宽为 8 的二进制数,其值为不定值
4'b1          // 位宽为 4 的二进制数,最低位为 1 其余高三位均为高阻值(4'bzzz)
```

下面所列的数为不合法的数并解释其原因:

\_23 // 以 \_ 开头

```

8' HFF          // 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

```

参阅表达式和字符串语句的说明。

## 28. 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、参数以及其他子表达式)计算出表达式的值。Verilog 语言中的运算符和 C 语言很相似。

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

其他运算符:

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

在程序中所处位置:

参阅运算符表达式的说明。

规则:

(1) 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)。

(2) 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。

(3) 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)。

(4) 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( $1'bX$ )
```

- (5) 缩位运算符( $\&$ 、 $\sim\&$ 、 $|$ 、 $\sim|$ 、 $\wedge$ 、 $\sim\wedge$ )将一个矢量缩减为一个标量。
- (6) 位宽确定的表达式的运算采用溢出的位不计的办法,如: $4'b1111 + 4'b0001 = 4'b0000$ 。
- (7) 整数作除法运算时,小数部分被截掉。
- (8) 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。
- (9) 只有某些特定的运算符允许出现在实数表达式中,例如单目运算符十和一、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:

$+$ $-$ $!$ $\sim$	单目(unary)	最高优先级
$*$ $/$ $\%$		
$+$ $-$	双目(binary)	
$<<$ $>>$		
$<<=$ $>>=$		
$==$ $!=$ $=====$ $! ==$		
$\&$ $\sim\&$		
$\wedge$ $\sim\wedge$		
$ $ $\sim $		
$\&\&$		
$  $		
$:$		最低优先级

注意:

- (1) 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。
- (2) 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:

- (1) 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。
- (2) 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。
- (3) 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。
- (4) 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 $\%$ 运算是可综合的,但结果只能用二进制数表示。其他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代码的含义。

**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
$\sim 4'b1101$	// 结果为 4'b0010
$\& 8'hff$	// 结果为 1'b1,即一位的逻辑值 1

参阅 Expression 语句的说明。

## 29. 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 ]
```

**在程序中所处位置:**

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

**规 则 :**

- (1) 参数是常量,在仿真期间更改参数的值是非法的。
- (2) 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：

下面的例子是一个以 N 位宽度的(可通过参数改变位宽度)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
```

参阅`define, Defparam, Instantiation 和 Specparam 语句的说明。

### 30. PATHPULSE \$ 路径脉冲参数

(1) 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。

(2) 如果使用默认的 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
```

在程序中所处位置：

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

规 则：

- (1) 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。
- (2) 维持时间比第一个极限参数(即 reject)短的脉冲不会输出。
- (3) 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)。
- (4) 维持时间比第二个极限参数长的脉冲将正常地输送出去。
- (5) 用 specparam 对 PATHPULSE \$ input \$ output 参数重新赋值将改写常规值。
- (6) 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
```

参阅 Specify 和 Specparam 语句的说明。

## 31. 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)}

在程序中所处位置:

```

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

```

规 则:

(1) 在端口列表中列出的所有端口必须按顺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。

(2) 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。

(3) 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)。

(4) 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这两个声明中的位宽必须一致。

(5) 输入和双向端口不能声明为寄存器类型。

(6) 输出端口的类型不能声明为实型(real)或实时型(realtime)。

提 示:

(1) 在测试模块中不要定义端口。

(2) 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;

```

参阅 Module, User Defined Primitive 和 Instantiation 语句的说明。

## 32. 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, ...}

```

在程序中所处位置:

参阅 statement 语句的说明。

规 则:

- (1) 对寄存器的赋值(不包括正负号)。
- (2) 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。
- (3) 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。
- (4) 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。
- (5) 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:

- (1) 综合工具不考虑延时。
- (2) 定时控制或延时是不可综合的。
- (3) 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。
- (4) 同一个寄存器类型变量不能既用阻塞赋值又用非阻塞赋值。
- (5) 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。在描述时序逻辑时用时钟沿触发的 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:

(1) 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他方面,这样做可以防止时钟沿触发的 always 块中发生竞争冒险,也可使设计意图更加清晰,又能避免生成不需要的触发器。

(2) 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 s 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 5 个下降沿插入复位信号:

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

参阅 Timing Control 和 Continuous Assignment 语句的说明。

### 33. 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, ... }
```

在程序中所处位置：

参阅 Statement 语句的说明。

## 规 则：

(1) 过程连续赋值语句执行后,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。

(2) 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,应确认将 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：

参阅 Continuous Assignment 和 Force 语句的说明。

### 34. 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 C 语言编写的函数的方法。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可以启动 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 \$ 符号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:

- (1) 延迟计数;
- (2) 测试矢量读入;
- (3) 波形演示;
- (4) 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手册的范围,可参阅其他参考资料。

### 35. 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]
```

在程序中所处位置:

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

### 规 则：

- (1) 寄存器类型变量只能用过程赋值语句赋值。
- (2) 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。
- (3) 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 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 型的值被当作无符号值。

(4) 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：

- (1) 虽然 register 这个词指的是硬件寄存器(例如触发器),而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。
- (2) 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。

(3) 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：

- (1) Real, time 和 realtime 类型的寄存器变量是不可综合的。
- (2) 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。
- (3) 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。
- (4) 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,而 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];

```

参阅 Net 语句的说明。

### 36. 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

```

在程序中所处位置：

参阅 Statement 语句的说明。

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中断,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

```

参阅 For, Forever, While 和 Timing Control 语句的说明。

### 37. Reserved Words 关键词

下列词汇是 Verilog 语言规定的所有的关键词,在这里,千万不要把这些标识符用作自定义的标识符,除非把它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	pullo	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	tril
edge	macromodule	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	

### 38. 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}
```

在程序中所处位置：

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

## 规 则：

- (1) 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。
  - (2) 在路径声明中可使用全连接符(  $\ast >$  )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。
  - (3) 模块路径的极性可选是指路径可以选择正极或者负极,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。
  - (4) 跳变沿敏感的路径数据表达式同样也不影响仿真。
  - (5) 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( $\sim$  & |  $\sim\sim$   $\sim\sim$ )、逻辑运算和逻辑等式运算符(== ! = && || !)、缩位运算符(& |  $\sim$  &  $\sim$  |  $\sim\sim$   $\sim\sim$ )、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 均被认为是真的),路径延时只影响路径。

(6) 如果没有一个 if 条件为真,则用 Ifnone 来定义默认的 SDPD。如果同一条路径既定义为 ifnone 与状态有关的路径延时(SDPD),又定义为简单的路径延时,则是非法的。

(7) 无条件的路径优先于 SDPD 路径。

(8) 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。

(9) 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:

(1) 目前还没有一个仿真工具支持上面语法中列出的 12 种不同跳变参数表示某条路径延时的方法。

(2) 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:

(1) 运用指定延时块来描述库中的单元(cell)延时。应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。

(2) 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; // 默认为 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

```

参阅 Specparam, PATHPULSE \$ 和 \$ setup 语句的说明。

### 39. 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,
⋮ ;

```

在程序中所处位置:

```

specify -<HERE>- endspecify

```

规 则:

(1) 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。

(2) 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

- (1) 在 Specify 块中,用 specparam 来定义命名的延时参数比直接用数字要好。
- (2) 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

```

参阅 PATHPULSE\$ 和 Specify 语句的说明。

## 40. 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 块中的语句是同其他 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

```

在程序中所处位置:

```

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>

参阅 Timing Control 语句的说明。

## 41. 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,可常用强度(Strength)的概念来描述这种逻辑行为。

语 法:

```

Strength = {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

```

在程序中所处位置:

参照 Net, Instantiation 和 Continuous Assignment 语句的说明。

规 则:

(1) 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。

(2) 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。

(3) 默认的强度定义为 strong0, strong1, 但下述情况除外:

对于 pullup and pulldown 门,默认强度分别为 pull1 和 pull0。

对于 trireg 的 Net, 默认强度为 medium。

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

(4) 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:

tri0 和 tri1 类型的 net 分别具有逻辑值 0 和 1, 并为 pull 强度。

trireg 类型的 net 保持它们最后的驱动值。

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。

(5) 强度值有强弱顺序, 可从 supply(最强的)依次减弱并排列到 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。强度值强弱顺序排列如右列表所示。

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);
```

Supply
Strong
Pull
Large
Weak
Medium
Small
Highz

参阅 Continous Assignment, Instantiation, Net 和 \$display 语句的说明。

## 42. 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”语句的说明。

在程序中所处位置:

参见 Expression 语句的说明。

规则:

(1) 一条字符串不能占源代码的多行。

(2) 字符串可以包含右列表中的扩展字符。

(3) 诸如 \$display 和 \$monitor 等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)。

(4) 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。Verilog HDL 语言的字符串的定义和 C 语言不一样。在 C 语言中需要用而在 Verilog HDL 语言中不需要用 ASCII 代码的 0 字符来表示字符串的结束。

\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 %

注 意：

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

参阅 NUMBER 和 \$display 语句的说明。

### 43. 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：

- (1) 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。
- (2) 任务中的 input, output 和 inout 的个数不受限制(也可以为零个)。
- (3) 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。
- (4) 当启动任务时，相当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：

- (1) 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。
- (2) 任务中若包括一句以上的语句，必须用 begin-end 或 fork-join 将其包含成块。
- (3) 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，也只有一份寄存器的复制。若第一次启动的任务还未完成，则第二次启动该任务，其输入、双向端口 (inout)、输出和局部寄存器的值便会被覆盖。
- (4) 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时序控制延迟后才被更新。
- (5) 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：

包含时序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：

- (1) 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。
- (2) 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(见以下例说明)序列。
- (3) 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;
end

```

```

#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

```

参阅 Task Enable 和 Function 语句的说明。

#### 44. 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：

(1) 任务可以从 initial 或 always 块或其他任务中启动。任务可以多次调用,但任务不能被函数调用。

(2) 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。

(3) 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。

(4) 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。

(5) 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);
```

参阅 Disable, Task 和 Function Call 语句的说明。

## 45. Timing control 时序控制

用于延迟语句的执行或安排语句的执行顺序。时序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)

```

在程序中所处位置：

参阅 statement 和 procedural assignment (for intra—assignment timing control) 的说明。

规 则：

(1) 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。

(2) 当执行到 wait 语句时,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 语句时,如果表达式为真,下一句不延迟马上执行。

(3) 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。

(4) 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)。

(5) 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。

(6) 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,否则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：

(1) 综合时延迟被忽略。

(2) 综合工具不支持 wait 语句和内部赋值延迟以及 repeat(重复)语句。

(3) 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,可用内部赋值延迟来描述当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 5 个下降沿复位:

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

参阅 Procedural Assignment, Always 和 Repeat 语句的说明。

## 46. 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,这是模块的另一种表示方法。可以引用由门构建的实例,并以同样的方式认实例引用用户自定义原语(UDP)。

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：

- (1) 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。
- (2) 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。
- (3) 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。
- (4) 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。
- (5) 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。

(6) UDP 表中所用的特殊的电平和边沿符号含义如附表 13 所列。

(7) 若组合逻辑的输入值和触发边沿没有明确规定将会导致输出的不确定。

(8) 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:

(1) 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。

(2) 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。

(3) 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

```

附表 13 电平与边沿符号之含义

?	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)
*	(??)

```

// DEna :old Q:Q
0 0:?:0;
1 0:?:1;
? 1: ?: -;      // 保持原值
0 ?: 0:0;
1 ?:1:1;
endtable
endprimitive

```

综合器忽略对端口的赋值语句

```

primitive DFF (Q, Clk, D);
    output Q;
    input Clk, D;
    reg Q; // Edge sensitive UDP
    initial
        Q = 1;
    table
        / * Clock transitions * /
        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

```

参阅 Module, Gate 和 Instantiation 的语法说明。

## 47. 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。

```

while {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。下面举一个例子来举例说明:

```

reg [15:0] Word
while (Word)

```

```

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

```

参阅 For, Forever, Repeat 语句的说明。

## 48. Compiler Directives 编译器指示

编译器指示是在源代码中对 Verilog 编译器发出的指令。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。

## 49. 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。例如：

```

`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。

例如：`default\_nettype tri。

(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。

例如：`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。

例如：`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)。

例如: `unconnected\_drive pull0 //或 pull1 (即逻辑值为 1)。

## 50. 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。

例如:

```
'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。

例如:

```
'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,...)]
```

在程序中所处位置:

宏可以在模块内或模块外定义。

规则:

① 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。

② 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;
```

③ 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。

④ 宏文本不允许分下列语言记号:注释、数字、字符串、名称、保留名称和操作符。

⑤ 不能把编译器指示名用作宏名。

注意:

① 所有的具体电路实现工具都不支持带参数的宏。

② 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。

③ 要区别撇号(`)和表示数制的前引号(')的不同。

④ 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:

① 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。

② 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_rt1
#define SUBBLOCK2 subblock2_rt1
#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);
```

参阅“ifdef 语句的说明”。

(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：

- ① 如果宏名已经用'define 定义, 只编译 Verilog 编码的第一块。
- ② 如果宏名没有定义和'else 指示出现, 只编译第二块。
- ③ 这些编译指示是可以嵌套的。
- ④ 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。

例如：

```
'define primitiveModel
module Test
...
`ifndef primitiveModel
    Mydesign_primitives UUT (...);
`else
    Mydesign_RTL UUT(...);
`endif
endmodule
```

参阅'define 语句的说明。

#### (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：

- ① 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。

- ② 精度单位必须小于或等于时间单位。

- ③ 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
```

参阅 \$ timeformat 语句的说明。

## 二、系统任务和函数 (System task and function)

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, 它们虽不是标准的一部分, 但在一些仿真工具中也经常见到。

注意: 各种不同的 Verilog 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。

### 1. 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。

(1) \$ display, \$ monitor, \$ strobe, \$ write 等

用于把文本送到标准输出和或写入一个或写入多个文件中的系统任务。有关的详细说明在后面介绍。

(2) \$ 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。

(3) \$ 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。

(4) \$ 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。

(5) \$ 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。

(6) \$ 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, 即 0 输出最少, 1 输出多点, 2 输出最多。

## (7) \$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:

① 0——不打印;

② 1——打印仿真时间和地点(默认值);

③ 2——打印仿真时间、地点和仿真所使用的 CPU 时间及内存的统计数据。

## (8) \$time, \$stime 和 \$realtime:

```
$time;
```

```
$stime;
```

```
$realtime;
```

系统函数返回仿真的当前时间值,而返回时间值的单位由调用该系统函数语句模块的'timescale 定义。

① \$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;

② \$stime 返回一个截去高位保留低 32 位的无符号整数;

③ \$realtime 返回一个实数。

注意:这些系统函数没有输入,与 Verilog 的其他函数不同。

## (9) \$realtobits 和 \$bitstoreal:

```
$realtobits(RealExpression) {return a 64 bit value};
```

```
$bitstoreal(BitValueExpression) {return a real value};
```

完成实数和用位(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。参阅 Module 语句的说明。

## (10) \$rtoi 和 \$itor:

```
$rtoi(RealExpression) {return an integer};
```

```
$itor(IntegerExpression) {return a real number};
```

完成实数和整数之间的互相转换,即 \$rtoi 把实数截断后转换为整数,而 \$itor 把整数转换为实数。

## 2. 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。可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。

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

- (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(1)~(8)的 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,详细说明参阅后面的材料。

### 4. 存储数值变化的系统任务 Value Change Dump Tasks

- (1) \$ dumpfile("FileName");
- (2) \$ dumpvars[( Levels, ModuleOrVariable, … )];
- (3) \$ dumpoff;
- (4) \$ dumpon;
- (5) \$ dumpall;
- (6) \$ dumplimit( FileSize);
- (7) \$ dumpflush;

以上(1)~(7)的 7 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,详见后面资料。

### 5. 非标准的系统任务和函数(Nonstandard System Task and Function)

以下的系统任务和函数在 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。

- (1) \$ 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 类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。

- (2) \$ 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。

(3) **\$ input:**

**\$ input("FileName");**

从某个文本文件中读出交互命令。

(4) **\$ 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。

(5) **\$ 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。

(6) **\$ 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,并从头重新开始执行仿真。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。

(7) **\$ 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,原先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。

(8) **\$ 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,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。

(9) \$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。

(10) \$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。

(11) \$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。

### 三、常用系统任务和 函数的详细使用说明

#### 1. 标准的系统任务和函数

(1) \$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、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(在这里八进制数的每一个数字代表 3 位,而十进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。若参数表含两相邻逗号,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:

- %b %B 二进制数(Binary)
- %o %O 八进制数(Octal)
- %d %D 十进制数(Decimal)
- %h %H 十六进制数(Hexadecimal)
- %e %E %f %F %g %G 实型数(Real)
- %c %C 字符(Character)
- %s %S 字符串(String)
- %v %V 二进制数和强度(Binary and Strength)
- %t %T 时间类型数(Time)
- %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 X)。% 号后常跟有一个数,该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 个数位。若相应的变量未用格式控制符声明,则默认为十进制数。有些系统打印任务有其自己的默认值,如 \$displayb, \$fwriteo, \$displayh 的默认值分别是二进制、八进制和十六进制。

**举例说明:**

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

参阅 \$monitor 和 \$strobe 语句的说明。

(2) \$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}
```

在程序中所处位置：

参阅 Statement 语句的说明。

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
```

参阅 \$display, \$monitor 和 \$strobe 语句的说明。

(3) \$ monitor 等：

当 \$monitor 系统任务所指定的参数表中的任何一个或多个 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,即只能同时运行一个 \$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 系统任务时,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,而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 );
```

参阅 \$display, \$strobe 和 \$fopen 语句的说明。

#### (4) \$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:

- ① 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。

② 第二个参数是存储器阵列名。

③ 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。

④ 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
```

(5) \$ 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
```

参阅 \$ display, \$ monitor 和 \$ write 语句的说明。

**(6) \$ 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：**

① Units(单位)是指打印的时间单位,是一个 0~ -15 之间的整型数,0 表示秒(s),-3 表示毫秒(ms),-6 表示微秒(μs),-9 表示纳秒(ns),-12 表示皮秒(ps),-15 表示飞秒(femtosecond),中间的整数也可用,如 -10 表示 100 皮秒(ps),依次类推。

② Precision 是指打印的十进制数小数点后保留的位数。

③ Suffix 指打印时间值后跟的字符串。

④ 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。

⑤ 默认形式,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 为无; 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
```

参阅 timescale 和 \$display 语句的说明。

## 2. 随机模型 Stochastic Modelling

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);
```

在程序中所处位置:

参阅 Statement 语句的说明。

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)。

(1) 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)。

(2) 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。

(3) 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(参阅 \$ q\_add 的说明)。inform\_id (输出)是由 \$ q\_add 储存的数值。

(4) 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。

## (5) 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队列元素被何时加入到队列中(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。它分别表示要求取得的信息类型:

- ① 1——当前队列长度;
- ② 2——平均达到时间间隔;
- ③ 3——最大队列长度;
- ④ 4——最短等待时间;
- ⑤ 5——当前队列中队列元素的最长等待时间;
- ⑥ 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 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 1~8 号共 8 个工作,每个 job 之间间隔为 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 */
    end
endmodule
```

```

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

```

参阅 \$random 和 \$dist\_chi\_square 等系统任务的说明。

### 3. 时序检查(Timing Checks)

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：

- (1) 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。
- (2) 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。
- (3) 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。
- (4) 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。
- (5) 时序检查系统任务中的参考事件(ReferenceEvent)必须是沿触发的,如 \$ width, \$ period, \$ recovery 和 \$ nochange。
- (6) 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。
- (7) 使用 &.&.& 做注释的条件,其时序检查仅在条件为真时才执行。
- (8) 在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 值则变为 0,若原为高阻值则不变。

**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
```

参阅 Specify 和 Specparam 语句的说明。

#### 4. 记录数值变化的系统任务(Value Change Dump Tasks)

以下 7 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：**

- (1) \$ dumpfile("FileName");
- (2) \$ dumpvars[( Levels, ModuleOrVariable, ... )];
- (3) \$ dumpoff; {suspend dumping}
- (4) \$ dumpon; {resume dumping}
- (5) \$ dumpall; {dump a checkpoint}
- (6) \$ dumplimit ( FileSize );
- (7) \$ dumpflush; {update the dump file}

在程序中所处位置：

参阅 Statement 语句的说明。

**规则：**

(1) 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。

(2) 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。

(3) FileSize 参数是用于设置 VCD 文件可记录的最多字节数。

(4) 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 $dumppall;
    endmodule
```

## 四、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的可选项并不是 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,例如可跟一个文件名,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。而设计中没有引用在 LibraryFile 中的 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,参阅-y 的说明。

**+notimingchecks**

关闭指定块的定时检查,这样可以加速仿真,或抑制伪定时错误信息,使用此选项时须小心。

**+mindelays, +typdelays, +maxdelays**

这三项分别指仿真时使用最小、典型和最大的延迟,默认为使用典型延迟。仿真时不要混淆以上三种延迟。

**注意:**

Verilog 仿真器不能检查出十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十号后的参数,所以一定注意选项的拼写,例如“+maxdelays”要注意拼写正确。

## 五、IEEE Verilog 1364 – 2001 标准简介

**摘要** 2001 年 3 月 IEEE 正式批准了 Verilog – 2001 标准(即 IEEE 1364 – 2001)。Verilog – 2001 标准在 Verilog – 1995 的基础上有几个重要的改进。新标准有力地支持可配置的 IP 建模,大大提高了深亚微米(DSM)设计的精确性,并对设计管理作了重大改进。这些改进使其更加容易使用;这些改进将会影响每一个 Verilog 用户和 EDA 工具的设计人员。阅读了最近出版的几篇有关英文文献后,作者对 Verilog – 2001 新标准中的若干个改进作了简要的总结和介绍。

**关键词** Verilog, HDL, 硬件描述语言, EDA。

## 1. Verilog 语言发展历史回顾

Verilog 硬件描述语言诞生于 1984 年。最初它只用在 Gateway 公司的一个名为 Verilog - XL 的数字逻辑仿真器上。1989 年 Cadence 公司收购了 Gateway 公司。1990 年 Cadence 公布了 Verilog 硬件描述语言和它的编程语言接口(PLI)。不久,Verilog 国际组织 Open Verilog International (简称 OVI)正式成立,其宗旨是规范 Verilog 的公用部分和推广它的应用。OVI 有关 Verilog 1.0 版本的资料最初来自 Cadence 公司的 Verilog - XL 手册。1993 年 OVI 公布了 Verilog 2.0 版本,新版本对 Verilog 作了一些改进。随后,OVI 正式向 IEEE 提出申请要求批准它为标准。不久,IEEEVerilog 标准化工作组成立,并在 1995 年正式批准其为 Verilog 语言的标准,即 IEEE 1364 - 1995。这是第一个得到 IEEE 官方批准的 Verilog 标准。我们应该注意到 IEEE Verilog 标准化工作组当时并没有对 Verilog 语言做任何本质上的提高。工作组的目标只是把当时应用比较普及的 Verilog 仿真工具所使用的 Verilog 语言标准化,而并没有决心彻底重新编写新标准的文档。因为最初的标准来自于手册,所以 IEEE 1364 - 1995 和最新推出的 IEEE 1364 - 2001 标准也与用户使用手册很类似。

## 2. IEEE 1364 - 2001 Verilog 标准想要达到的目标

编写 IEEE 1364 - 2001 Verilog 标准的工作开始于 1997 年 1 月,当时确立了 3 个工作目标:

- (1) 改进 Verilog,使其有助于深亚微米设计和 IP(知识产权模块)建模问题。
- (2) 确保上述所有改进既有用,又切实可行,并使得仿真器和综合器厂商能在其产品中支持 Verilog - 2001。
- (3) 纠正 IEEE1364 - 1995 Verilog 参考手册中的所有错误,明确那些原来模棱两可的概念。

Verilog - 2001 标准化工作组由 20 个成员组成,他们代表了各种不同的 Verilog 用户、仿真器厂商以及综合器厂商等多方的利益。

整个工作组分为三个执行小分队:ASIC 任务小分队,提高了语言的性能以满足超深亚微米设计的时间精确性;行为级任务小分队,提高了行为级和 RTL 级语言建模的性能;PLI 任务小分队,增强了 Verilog 编程语言接口的功能,使其能支持前面两个小分队所做的改进,同时使 PLI 又新增加了一些功能。

## 3. 新标准使建模性能得到很大提高

本节列出了 21 个改进之处,可以为 Verilog 设计人员提供更强的 Verilog 建模能力。许多改进使得编写可综合 RTL 模型变得更容易也更准确。别的一些改进使得模型的维数可扩展性和可重复利用性更强。本节只是列出了新增加的功能和语法,没有涉及 Verilog - 1995 的相关说明。

(1) 设计管理——Verilog 配置 Verilog - 1995 标准把设计管理工作交给独立的软件工具来承担,设计管理不是语言的一部分。各个仿真器厂商的设计管理工具在处理不同版本的 Verilog 模型时有各自的方法,但是这些和工具有关的方法不能适用不同版本的 Verilog 编写的模型。

而 Verilog - 2001 中增加了配置块(configuration block),它属于新版本 Verilog 语言的一部分,可以用它对每一个 Verilog 模型指定其具体版本和其源代码的位置。出于可移植性的考虑,在配置块中,可以使用多个虚拟模型库,还需要配合独立的库映像文件(library map

files)指出其具体物理位置。配置块位于模块定义外。配置块的名字和模块名、原语块(primitive)名存在于同一个命名空间中。在 Verilog-2001 中新增加了关键字:config 和 endconfig; 其他新增加的关键字:design、instance、cell、use 和 liblist 留给配置块使用。

本文不想介绍 Verilog 配置块的完整语法和使用规则等方面的内容。下面的例子是一个简单的设计配置。其 Verilog 源代码是典型的,其中测试(test bench)模块包含了设计层次树顶层的实例调用,设计顶层还包括其他模块中的实例。

```

module test;
...
myChip dut (...); /* instance of design */
...
endmodule
module myChip(...);
...
adder a1 (...);
adder a2 (...);
...
endmodule

```

配置模块可以指定全部或个别模块实例的源代码的位置。因为配置模块位于 Verilog 模块之外,所以需要重新配置时,Verilog 模型的源代码不需要做任何修改。在下面这个配置例子中,加法器实例 a1 由 RTL 库编译生成,实例 a2 则来自一个门级库。

```

config cfg4
design rtlLib.top /* 给配置块命名 */
default liblist rtlLib gateLib; /* 指定从哪里找到顶层模块 */
instance test.dut.a2 liblist gateLib; /* 设置查找实例模块的默认顺序 */
endconfig

```

配置模块使用虚拟库来指定 Verilog 模型源代码的位置。使用库映射文件将虚拟库的名字和实际的文件位置联系起来。例如:

```

library rtlLib ./*.v; /* RTL 库模型的位置(当前目录) */
library gateLib ./synth_out/*.v; /* 门级库模型的位置 */

```

(2) 用 Verilog 生成维数可扩展的模块 Verilog-1995 标准在设计维数可扩展和可重复使用的模块时功能有限。它虽然具有强大的实例阵列结构,但是并不具备真正维数可扩展性和设计复杂结构所需要的灵活性。

而 Verilog-2001 增加了生成循环(generate loop),允许生成多个模块和原型的实例,同时生成多个变量、网络、任务、函数、连续赋值、初始化过程块以及 always 过程块。可以使用 if - else 或 case 语句有条件地生成声明语句和实例引用语句。

Verilog-2001 中定义了四个与此相关的新关键字:generate、endgenerate、genvar 以及 localparam。其中,关键字 genvar 是一种新的数据类型,这个数据类型用于存储正的整型变量;与其他 Verilog 变量不同,它的值可以在编译和详细描述(elaboration)时改变。生成循环

中的索引变量必须定义成 genvar 类型。

关键字 Localparam 表示一个常数, 和 parameter 类似, 但是 Localparam 与 parameter 的不同点在于它不能够使用参数重定义(parameter redefinition)来改变赋值。生成模块(generate block)同样可以通过专门的 Verilog 语句, 来控制生成的将是何种对象。这些语句包括: for 循环; if - else 语句以及 case 语句等。

下面的例子说明了使用生成(generate)语句创建维数可扩展的乘法器模块实例。当乘法器的 a 和 b 的位宽参数小于 8, 可生成 CLA 乘法器实例; 如果 a 和 b 的位宽大于或等于 8, 则生成 Wallace 数乘法器。

```
module multiplier (a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width;
input [a_width-1:0] a;
input [b_width-1:0] b;
output [product_width-1:0] product;

generate
if((a_width < 8) || (b_width < 8))
  CLA_multiplier #(a_width, b_width)
    ul (a, b, product);
else
  WALLACE_multiplier #(a_width, b_width)
    ul (a, b, product);
endgenerate

endmodule
```

下面的例子说明了如何使用 generatefor 循环语句引用原语(primitive)实例和原语实例的内部互联来创建多位宽加法器, 其位宽和引用实例数目由可重定义的参数常数指定。

```
module Nbit_adder (co, sum, a, b, ci);
parameter SIZE = 4;
output [SIZE-1:0] sum;
output co;
input [SIZE-1:0] a, b;
input ci;
wire [SIZE:0] c;
genvar i;
assign c[0] = ci;
assign co = c[SIZE];
for(i=1; i<SIZE; i=i+1)
  assign c[i] = a[i] + b[i] + c[i-1];
```

```

begin: addbit
  wire n1, n2, n3; //internal nets
  xor g1 ( n1, a[i], b[i] );
  xor g2 ( sum[i], n1, c[i] );
  and g3 ( n2, a[i], b[i] );
  and g4 ( n3, n1, c[i] );
  or g5 ( c[i+1], n2, n3 );
end
endgenerate

endmodule

```

前面的例子中,每个生成的网络都有一个独立的名字,而且每个生成的原语(primitive)实例也具有独立的名字。该名字由 For 循环中程序块的名字和作为循环变量的 genvar 类型变量值组成。例如 n1 网络中的名字如下:

```

addbit[0].n1
addbit[1].n1
addbit[2].n1
addbit[3].n1

```

为第一个 xor 原语(primitive)生成的实例名字如下:

```

addbit[0].g1
addbit[1].g1
addbit[2].g1
addbit[3].g1

```

需要注意的是这些生成的名字中包含了方括号,这在用户定义的标识符中是非法的,但是在生成的名字中是合法的。

(3) 常数函数 Verilog 语法规定必须使用数值或常数表达式来定义向量的宽度和阵列的规模。例如:

```

parameter WIDTH = 8;
wire [WIDTH-1:0] data;

```

Verilog-1995 标准的局限之一是上述表达式必须基于算术操作。使用编程语句定义上述表达式的值是不允许的。

Verilog-2001 给 Verilog 函数定义了一种新用法,称为常数函数。常数函数的定义和任何 Verilog 函数的定义相同。但是常数函数只能使用这样一种结构,它的具体数值是在编译或详细描述(elaboration)过程中被确定的。

常数函数有助于创建可改变维数和规模的可重用模型。下面的例子定义了一个称为 clogb2 的函数,该函数返回一个整数(即以 2 为底的对数的极限值)。这一常数函数可用于根据 RAM 中的单元数目,以确定 RAM 地址总线必需的宽度。

```

module ram (address_bus, write, select, data);

```

```

parameter SIZE = 1024;
input [clogb2(SIZE)-1:0] address_bus;
...
function integer clogb2 (input integer depth);
begin
for(clogb2=0; depth>0; clogb2=clogb2+1)
depth = depth >> 1;
end
endfunction
...
endmodule

```

(4) 选择索引向量的一部分 Verilog-1995 标准中对向量某些位的选择是允许的,但被选择的部分必须是固定的。因此使用变量来选择长字中某个字节是不符合语法的。而 Verilog-2001 中增加了一个新的语法,称为索引的部分选择(indexed part selects)。索引的部分选择由基表达式、宽度表达式和偏移方向三部分组成,其形式如下:

[base_expr +: width_expr]	//正偏移
[base_expr -: width_expr]	//负偏移

其中,基表达式可以随着仿真过程的运行而变化,但是宽度表达式必须是常数。偏移方向表示选择区间究竟是基表达式加上宽度表达式,还是减去宽度表达式。例如:

```

reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num * 8 +: 8];

```

上例中,如果 byte\_num 的值是 4,则把 word[39:32]赋给 byteN。其中,起始位 32 由基表达式确定,终止位 39 则由基正偏移和宽度确定。

(5) 多维矩阵 Verilog-1995 标准只允许一维矩阵变量。Verilog-2001 打破了这一限制,允许使用:

- 多维矩阵;
- 含有变量和网络(net)数据类型的矩阵。

这一改进需要改变矩阵声明和矩阵索引的语法。下例说明了一维矩阵和三维矩阵的声明和索引语法。

Verilog-1995 和 Verilog-2001 标准都支持一维矩阵的描述,下面两条语句表示的是变量为 8 位寄存器组的一维矩阵:

```

reg [7:0] array1 [0:255]; //变量为 8 位寄存器组的一维矩阵
wire [7:0] out1 = array1[address]; //变量为 8 位的一维矩阵

```

只有 Verilog-2001 标准才支持三维矩阵,用语句表示如下:

```

wire [7:0] array3 [0:255][0:255][0:15]; //变量为 8 位寄存器组的三维矩阵
wire [7:0] out3 = array3[addr1][addr2][addr3]; //变量为 8 位的三维矩阵

```

(6) 选择矩阵内的某位和某几位 Verilog-1995 不允许直接访问矩阵字的某一位或某几位。必须将整个矩阵字复制到一个暂存变量中,从暂存变量中访问。Verilog-2001 去除了这一限制,允许直接访问矩阵字的某一位或某几位。举例说明如下:

```
//选择变量为 32 位的 2 维矩阵中某一单元[100][7]的最高字节[31:24]
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```

(7) 带符号的算术运算的扩展 对于整型算术操作,Verilog 根据操作数的数据类型来决定作无符号运算还是带符号运算。通常(也有例外),只要表达式中有无符号操作数,就执行无符号操作;只有当表达式所有的操作数都是带符号数时,才执行带符号操作。

Verilog-1995 中,整型数据是有符号数,寄存器和网络型数据是无符号数。Verilog-1995 的局限之一是整型数据必须具有固定的矢量位宽,大多数的 Verilog 仿真器将其定义为 32 位。因此,根据 Verilog-1995 标准,有符号的整数算术运算只限于 32 位矢量。对此,Verilog-2001 标准作了五点改进,大大增强了有符号数算术运算的能力:

- 寄存器和网络型数据可以定义为有符号;
- 函数返回值可以定义为有符号;
- 任何进制的整数都可以定义为有符号;
- 操作数可以由无符号变为有符号;
- 可以进行算术移位操作。

Verilog-1995 标准中保留了一个关键字 signed,但是没有用到。Verilog-2001 标准使用了这个关键字,使得寄存器数据类型、网络数据类型、端口以及函数都可以定义成带符号的类型。下面举几个例子说明:

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

Verilog-1995 标准中,没有指定基数(进制)的整型数被认为是有符号数,相反,指定了基数(进制)的整型数被认为是无符号数。Verilog-2001 标准增加了一个额外的标识符,字母 's' 和基数标识符一起指定该数是带符号数。举例说明如下:

16'hC501	//16 位十六进制无符号数
16'shC501	//16 位十六进制有符号数

除了可以定义有符号数据类型和数值外,Verilog-2001 还增加了两个新的系统函数,即 \$signed 和 \$unsigned。这两个系统函数可以将无符号数变换为带符号数,或相反。举例说明如下:

```
reg [63:0] a; //无符号数据类型
always @(a) begin
    result1 = a / 2; //无符号运算
    result2 = $signed(a) / 2; //有符号运算
```

```
end
```

Verilog - 2001 标准中,另一个关于带符号数运算的改进是算术移位操作符,右移和左移分别用符号 $>>>$ 和 $<<<$ 表示。算术右移操作不改变数值的符号,移位时,用符号位填充空缺位。

例如,如果 8 位带符号变量 D 的值为 8'b10100011,3 位的逻辑右移和算术右移的结果如下:

D >> 3	//逻辑右移的结果是 8'b00010100
D >>> 3	//算术右移的结果是 8'b11110100

(8) 幂运算符 \* \* Verilog - 2001 标准中增加了幂运算,用符号 $* *$ 表示,实现与 C 语言中 pow() 函数相同的功能。两个操作数中只要有一个实型数,函数就返回实型数;如果两个操作数都是整型数,函数才返回整型数。幂运算常常用来计算  $2^n$  的值。

举例说明:

```
always @(posedge clock)
    result = base * * exponent;
```

(9) 可重入任务和递归函数 Verilog - 2001 标准中增加了一个新的关键字—automatic。这个关键字可以用于定义可重入(Re - entry)的自动的任务(task)。自动任务中的每一条声明语句,在每次当前任务的调用中,都会进行动态的分配定位。函数(function)也可以定义为自动的,使得函数可以递归调用(函数中的每条声明语句在每次递归调用中都将动态分配定位)。自动任务或函数中的声明语句不能通过层次名调用。

Verilog - 2001 标准中不用关键字 automatic 声明的任务或函数是静态的,与 Verilog - 1995 中的任务和函数的表现完全相同。静态任务或函数中的每条声明语句是静态分配定位的,即由该任务或函数的每次调用所共享。下面的例子说明了一个递归调用自己的函数,实现 32 位无符号整型操作数的 n! (阶乘)的功能。

```
function automatic [63:0] factorial;
    input [31:0] n;
    if (n == 1)
        factorial = 1;
    else
        factorial = n * factorial(n-1);
endfunction
```

(10) 组合逻辑的电平敏感符号@ \* 为了使用 Verilog always 过程块正确地为组合逻辑建立模型,敏感列表必须包含逻辑块中用到的所有输入信号。编写大型复杂的组合逻辑模块时很容易在敏感列表中遗漏某一个输入信号,从而导致仿真和综合的结果不一致。

Verilog - 2001 中增加了一个新的符号@ \*,用于表示组合逻辑的敏感列表。@ \* 表示仿真器和综合工具应该能够自动地对该符号下逻辑块中每条语句中被赋于的值敏感。举例说明:下例中@ \* 符号使得逻辑在 sel、a 或 b 变化时,y 值能自动地跟着变化。

```
always @ * //组合逻辑的敏感性问题
```

```
if (sel)
y = a;
else
y = b;
```

(11) 用逗号分隔的敏感列表 Verilog - 2001 增加了表示敏感列表的另一种写法, 即用逗号而不是 or 关键字来分隔敏感列表中的各个信号。下例表示了功能完全一致的敏感列表的两种不同的写法:

```
always @ (a or b or c or d or sel)
always @ (a, b, c, d, sel)
```

用逗号分隔的敏感列表并没有增加新的功能, 但是更加直观, 与 Verilog 中的其他信号列表更加一致。

(12) 增强的文件输入输出操作 Verilog - 1995 标准中, Verilog 语言在文件的输入/输出操作方面功能非常有限。因而, 文件操作经常是借助于 Verilog PLI(编程语言接口), 通过与 C 语言中文件输入/输出库的访问来处理的。Verilog - 1995 标准规定, 可以同时打开的 I/O 文件数目不能超过 31 个。

Verilog - 2001 中增加了新的系统任务和函数, 为 Verilog 语言提供了强大的文件输入/输出操作, 而不需要使用 PLI。同时, 扩展了可以同时打开的文件数目至 230。这些新的文件操作任务和函数包括: \$ferror、\$fgetc、\$fgets、\$fflush、\$fread、\$fscanf、\$fseek、\$fscanf、\$ftel、\$rewind 和 \$ungetc。Verilog - 2001 还给出了读写字符串的系统任务, 包括 \$sformat、\$swrite, \$swriteb, \$swriteh, \$swriteo 和 \$sscanf。它们可以用来生成格式化的字符串或从字符串中读取信息。

(13) 超过 32 位宽的自动宽度扩展 Verilog - 1995 中对于不指定位数的位宽超过 32 的总线赋高阻时, 只会将低 32 位赋成高阻, 高位将赋 0。为了将整个总线的所有位都置为高阻态, 必须明确指定总线的位数。

举例说明(按照 Verilog - 1995 规定):

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;                                // data ='h00000000zzzzzzzz
data = 64'bz;                             // data ='hzzzzzzzzzzzzzzzz
```

Verilog - 1995 中的填充法则使得难于编写便捷的矢量位宽可变的模型。参数重定义是一种方法, 但是必须修改 Verilog 源代码, 改变其赋值语句中的位宽值。

Verilog - 2001 改变了赋值扩展法则, 将高阻或不定态赋给未指定位宽的信号时, 可以自动地扩展到信号的整个位宽范围。

举例说明(按照 Verilog - 2001 规定):

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;                                // data ='hzzzzzzzzzzzzzzzz
```

(14) 使用参数名在线传递参数 Verilog - 1995 有两种方法实现在模块中重新定义参

数:一是使用 defparam 语句显式地重新定义;二是在模块实体调用时使用 # 符号隐式地重新定义参数。第二种方法更为简洁,但是由于参数的重新定义与声明的位置有关,因而容易出错而且代码的含义不易于理解。

```
module ram (...);
parameter WIDTH = 8;
parameter SIZE = 256;
...
endmodule
module my_chip (...);
...
//用参数名显式地重新定义参数
RAM raml (...);
defparam raml.SIZE = 1023;
//根据位置隐式地重新定义参数
RAM #(8,1023) ram2 (...);
endmodule
```

Verilog - 2001 提出了第三种方法:在线显式重新定义。这种新方法允许在线参数值按照任意的顺序排列,并且可以对重新定义的参数加注释。

```
//在线显式参数的重新定义
RAM #( .SIZE(1023) ) ram2 (...);
```

(15) 端口声明和数据类型声明结合起来的声明语句 Verilog 语法要求每一个连接到模块输入或输出端口的信号必须声明两点:输入、输出方向和数据类型。Verilog - 1995 中这两个声明语句是分开写的。Verilog - 2001 提出了更简单的写法,即将两个声明结合在一条语句中。见下例:

```
module mux8 (y, a, b, en);
output reg [7:0] y;
input wire [7:0] a, b;
input wire en;
```

(16) ANSI 风格的输入和输出的声明语句 Verilog - 1995 使用早期的 Kernighan 和 Ritchie C 语言的语法来定义模块端口,端口的次序在小圆括号内定义,而端口的声明在小圆括号之后。

Verilog - 1995 标准中,任务和函数定义时不用写出圆括号和其内部的端口列表,只需要用输入、输出声明语句的排列次序来定义输入、输出端口的次序。Verilog - 2001 改进了模块、任务和函数的输入、输出序列定义,将端口声明语句写在包含输入、输出端口序列的圆括号内,使其更接近于 ANSI C 语言。

```
module mux8 (output reg [7:0] y,
             input wire [7:0] a,
             input wire [7:0] b,
             input wire en);
```

```
function [63:0] alu ( input [63:0] a, b, input [7:0] opcode );
```

(17) 寄存器声明时进行初始化赋值 Verilog-2001 允许在变量声明时对其进行初始化,可以不需要用专门的变量初始化进程块来对变量进行初始化。采用这种方式进行的变量初始化与在初始化进程块中进行初始化一样,在仿真的 0 时刻执行。举例说明:

在 Verilog-1995 标准中:

```
reg clock;
initial
  clk = 0;
```

在 Verilog-2001 标准中可以写为:

```
reg clock = 0;
```

这两种表达方式的实际含义是一样的。

(18) 将寄存器改变为变量 自 1984 年 Verilog 语言诞生以来, register 一词就一直用来描述 Verilog 语言中变量的一种类型。register 并不是一个关键字,只是一种数据类型 (reg、integer、time、real 和 realtime) 的名称。register 一词的使用通常会给 Verilog 初学者带来困扰,他们通常认为 register 和硬件中的寄存器 (flip-flops) 概念是一致的。因此,IEEE 1364-2001 Verilog 参考手册中将 register 一词改为 variable。

(19) 对条件编辑的改进 Verilog-1995 支持 `ifdef、`else 和 `endif 等条件编译语句。Verilog-2001 中增加了一些条件编译控制语句,包括 `ifndef 和 `elsif。

(20) 文件和行编译指示 Verilog 工具需要不断地跟踪 Verilog 源代码的行号和文件名。Verilog 的可编程语言接口 (PLI) 可以取得并利用行号和源文件名信息,显示程序运行的错误信息时,也需要知道这些信息。但是,如果 Verilog 源码经过其他工具的处理,源码的行号和文件名信息可能会丢失。Verilog-2001 中增加了一个 `line 编译指示,用来指定源码的行号和文件名。这样可以使源码文件中的指定位置在经过其他工具的修改(例如增加或删除一行)后,也能够保持不变。

(21) 属性 创建 Verilog 语言的最初目的是建立一种用于数字仿真的硬件描述语言。随着仿真器以外的其他工具采用 Verilog 作为设计输入,这些工具需要 Verilog 语言能够加入跟指定工具有关的信息和命令。在 Verilog-1995 标准中没有这一机制,只能通过一些非标准的方式来实现,例如通过 Verilog 注释语句加入可控制综合器功能的命令。

Verilog-2001 标准中增加了一种机制,可以在 HDL 源代码中指定对象、语句或语句组的属性。这些属性称为 attribute,可以翻译为属性。属性可以由包括仿真器在内的不同工具使用,控制工具的行为和操作。属性包含在两个符号 \* 之间。属性可以应用于对象的所有实例调用,也可以只应用于对象的某一个实例调用。属性可以指定为数值(包括字符串),某对象的每个实例调用可以重新定义其属性值。

Verilog-2001 没有定义任何标准的属性,属性的名字和数值由工具厂商或其他标准定义。

下面是综合工具如何使用属性的例子:

```
(* parallel case *) case (1'b1)      //1 位独热码的有限状态机(FSM)
```

```

state[0]: ...
state[1]: ...
state[2]: ...
endcase

```

#### 4. 提高了 ASIC/FPGA 应用的正确性

Verilog 语言创建于 2  $\mu\text{m}$  或 5  $\mu\text{m}$  设计的时代, 随着硅工艺水平的进步和设计方法学的演进, Verilog 语言也在发展。Verilog - 2001 继续了这一发展, 特别针对现在和未来的深亚微米设计作了改进。

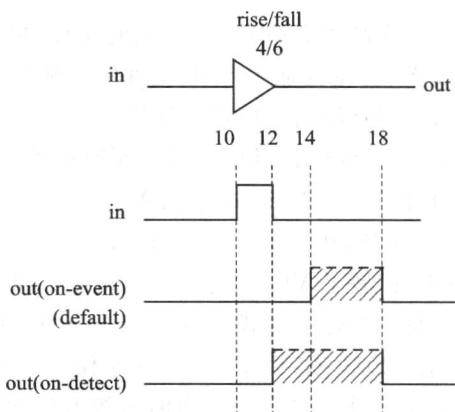
(1) 检测脉冲的传播错误 Verilog - 1995 标准可以对脚对脚(pin-to-pin)的路径延时提供根据事件(on-event)的脉冲传播错误模型。脉冲是模型路径输入端上的扰动, 其维持的时间比它在路径上的延时小。如果使用了 on-event 的脉冲传播模型, 输入脉冲的上升沿和下降沿传播到路径的输出端就变成了不定态(即逻辑值 X), 其波形(即起始和维持时间内的不定值)看起来像是输入脉冲已经传播到输出。

Verilog - 2001 增加了 on-detect 脉冲传播错误模型。当输入信号为干扰脉冲时, 设置 on-detect 是一种更保险的将输出设置为不定态的方法。与 on-event 模型相比, on-detect 也是将脉冲从前沿开始变成不定态, 到脉冲后沿结束, 不同处在于一旦检测到脉冲, 前沿立刻变为不定态。在 Verilog 的 specify 块中可以使用两个新的关键字 pulsestyle\_onevent 和 pulsestyle\_ondetect, 来显式地指定脉冲错误传播方式是 On-event 还是 on-detect。On-event 是脉冲传播错误的默认类型。举例说明如下:

```

specify
  pulsestyle_ondetect out;
  (in => out) = (4,6);
endspecify

```



(2) 负脉冲检测 当脉冲的下降沿比上升沿先到即为负脉冲。由于路径延时大于脉冲维持时间, 输入负脉冲也可能得到不定态(逻辑值为 X)的扰动输出。Verilog - 1995 中, 负脉冲通常被忽略。Verilog - 2001 提供了一种机制, 可将不定态传播到输出, 表明负脉冲已经产生, 增加了两个关键字 showcancelled 和 noshowcancelled, 在 specify 块定义时, 可以用它们让负脉冲传播功能启动或取消。举例说明如下:

```

specify
  showcancelled out;
  (a => out) = (2,3);
  (b => out) = (4,5);
endspecify

```

(3) 新的时序约束检查 Verilog - 2001 增加了几个新的时序约束检查, 可以建立更精确的深亚微米时序模型。这些新的时序检查系统任务包括 \$removal、\$recrem、\$timeskew 和 \$fullskew。本文不全面介绍和探讨这些时序检查, 对细节感兴趣的读者请参阅 IEEE 1364 - 2001 Verilog 标准。

(4) 负时序约束 Verilog - 2001 给 \$setuphold 时序约束系统任务增加了 4 个参数, 用这些参数可以精确地指定负电平的建立和保持时间。这两个时间参数用于定义沿冲突窗口(相对于参考信号沿)的位置, 在这段时间中, 数据必须是稳定的。建立和保持时间为正表示这一窗口跨过其参考的信号沿。相反, 建立和保持时间为负则意味着时间冲突窗口位移到参考信号沿的前面或后面。这种情形在真实器件中是很可能发生的, 因为器件内部时钟和数据信号路径延时的不一致总是有可能存在的。\$setuphold 的新参数可加在 Verilog - 1995 所定义的 \$setuphold 参数表的后面。新的参数是可选的。如果不指定这些参数, 也没有关系, \$setuphold 的语法仍旧与 Verilog - 1995 的规定兼容。

新的时序检查系统任务 \$recrem 是 \$recovery 和 \$removal 两个系统任务的结合, 可以处理负值, 其语法与 \$setuphold 类似。若想了解如何使用这些时序检查来指定期序约束为负值的细节, 可参考 IEEE 1364 - 2001 Verilog 标准。

(5) 提高了对 SDF(标准延时文件)的支持 IEEE 1364 - 2001 Verilog 语言参考手册增加了一节, 详细描述了 SDF 文件中的延时如何与 Verilog 语言中的延时相对应。其内容是建立在最新的 SDF 标准(即 IEEE Std 1497 - 1999 [3])基础之上的。

最新的 SDF 标准中包括了延时标签, 可以为 Verilog 程序提供延时标注的手段。为了支持 SDF 标签, 对 Verilog 语法作了一点改动。Verilog - 1995 中, specparam 常数只能在 specify 块(指定块)中定义。而 Verilog - 2001 允许在模块层次声明和使用 specparam 常数。

(6) 扩展了 VCD 文件 Verilog - 1995 标准中定义了一个标准的 4 状态逻辑 VCD(数值变化存储)的文件格式。\$dumpvars 和其他相关的系统任务用于创建和控制 VCD 文件。Verilog - 2001 对 VCD 文件格式作了一些扩展, 在 Verilog 端口变化、线路连接强度(net strength)变化以及仿真结束时间等方面增加了更多的细节。为此, Verilog - 2001 中定义了一些新的系统函数, 它们可以用来创建和控制扩展的 VCD 文件, 它们是:

\$dumpports、\$dumpportsall、\$dumpportsoff、\$dumpportson、\$dumpportslimit 和 \$dumpportsflush。

## 5. 编程语言接口(PLI)方面的改进

Verilog - 2001 对原 Verilog 编程语言接口(PLI)部分作了大量改进。这些改进可分为三部分:

- (1) 新 PLI 增加了许多新的功能;
- (2) 新的 PLI 能支持 Verilog - 2001 对 Verilog - 1995 的每个改进;
- (3) 对 Verilog - 1995 PLI 标准作了全面的清理。

Verilog PLI 标准包括 3 个 C 功能库, 即 TF、ACC 以及 VPI。TF 和 ACC 库是 Verilog PLI 的早期版本; 新标准为了兼容旧标准(即 IEEE 1364 - 1995 Verilog)保留了这两个库。VPI 库是最新版 PLI 标准所用的库, 与旧库比较有许多优点。

Verilog - 2001 清理和更正了旧的 TF 和 ACC 库中的许多定义, 但并没有给 TF 和 ACC 库加入任何新的功能。对 Verilog PLI 的所有改进都体现在 VPI 库中。其中包括支持 Veril-

og 语言的许多新特性,以及提供了 6 个 VPI 新子程序:vpi\_control()、vpi\_get\_data()、vpi\_put\_data()、vpi\_get\_userdata()、vpi\_put\_userdata() 和 vpi\_flush()。对这些 VPI 新子程序细节感兴趣的读者,可参考 IEEE 1364-2001 Verilog 标准。

## 6. 总 结

IEEE 1364-2001 Verilog 标准已经编写完毕,并且于 2001 年 3 月得到了 IEEE 官方的正式批准。Verilog-2001 对 Verilog 语言作了许多重要的改进,提供了强大的结构,可以编写可重复使用的、可升级的模型以及 IP 模型,并可给出精确的深亚微米电路的时序特性。用 Verilog 进行设计的工程师如果使用支持 Verilog-2001 版本的综合和仿真工具将从中得到极大的便利。

## 参考文献

- [1] Samir Palnitkar. Verilog HDL A Guide to Digital Design and Synthesis 2th Edition. SunSoft Press A Prentice Hall Title 2003.
- [2] Stephen Brown & Zvonko Vranesic 《Fundamentals of Digital Logic with Verilog Design》 2th Edition McBraw - Hill 2008.
- [3] Stuart Sutherland, 《Verilog 2001——A Guide to the New Features of Verilog Hardware Description Language 》Kluwer Academic Publishers, 1998.
- [4] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Computer Society, IEEE Std 1364 - 1995.
- [5] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Computer Society, IEEE Std 1364 - 2001.
- [6] Philips Semiconductors, The I<sup>2</sup>C - BUS SPECIFICATION Version2. 1.
- [7] Quick Works Version 7. 1 User's Guide with SpDE Reference, 1998.
- [8] CADENCE Version 9502 Verilog - XL Reference, 1997.
- [9] Synplify - Lite Synthesis Users' Guide, 1997.
- [10] IEEE P1364. 1 Draft Standard For Verilog Register Transfer Level Synthesis.
- [11] 刘宝琴. 数字电路与系统[M]. 北京：北京清华大学出版社,1993.
- [12] 夏宇闻. 复杂数字电路与系统的 Verilog HDL 设计技术[M]. 北京：北京航空航天大学出版社,1998.
- [13] 夏宇闻. Verilog 数字设计教程[M]. 北京：北京航空航天大学出版社,2003.