中文版 Verilog HDL 简明教程

前言

Verilog HDL 是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。

Verilog HDL 语言具有下述描述能力：设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监控和设计验证方面的时延和波形产生机制。所有这些都使用同一种建模语言。此外，Verilog HDL 语言提供了编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

Verilog HDL 语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用 Verilog 仿真器进行验证。语言从 C 编程语言中继承了多种操作符和结构。Verilog HDL 提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL 语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然,，完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

第 1 章 简介

Verilog HDL 是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。

Verilog HDL 语言具有下述描述能力：设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监控和设计验证方面的时延和波形产生机制。所有这些都使用同一种建模语言。此外，Verilog HDL 语言提供了编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

Verilog HDL 语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用 Verilog 仿真器进行验证。语言从 C 编程语言中继承了多种操作符和结构。Verilog HDL 提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL 语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然,完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

历史

Verilog HDL 语言最初是于 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言。那时它只是一种专用语言。由于他们的模拟、仿真器产品的广泛使用，Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者所接受。在一次努力增加语言普及性的活动中，Verilog HDL 语言于 1990 年被推向公众领域。 Open Verilog International （OVI）是促进 Verilog 发展的国际性组织。1992 年， OVI 决定致力于推广 Verilog OVI 标准成为 IEEE 标准。这一努力最后获得成功，Verilog 语言于 1995 年成为 IEEE 标准，称为 IEEE Std 1364－1995。完整的标准在 Verilog 硬件描述语言参考手册中有详细描述。

主要能力

下面列出的是 Verilog 硬件描述语言的主要能力：

* 基本逻辑门，例如 and、or 和 nand 等都内置在语言中。
* 用户定义原语（UDP）创建的灵活性。用户定义的原语既可以是组合逻辑原语，也可以是时序逻辑原语。
* 开关级基本结构模型，例如 pmos 和 nmos 等也被内置在语言中。
* 提供显式语言结构指定设计中的端口到端口的时延及路径时延和设计的时序检查。
* 可采用三种不同方式或混合方式对设计建模。这些方式包括：行为描述方式—使用过程化结构建模；数据流方式—使用连续赋值语句方式建模；结构化方式—使用门和模块实例语句描述建模。
* Verilog HDL 中有两类数据类型：线网数据类型和寄存器数据类型。线网类型表示

构件间的物理连线，而寄存器类型表示抽象的数据存储元件。
* 能够描述层次设计，可使用模块实例结构描述任何层次。
* 设计的规模可以是任意的；语言不对设计的规模（大小）施加任何限制。
* Verilog HDL 不再是某些公司的专有语言而是 IEEE 标准。
* 人和机器都可阅读 Verilog 语言，因此它可作为 EDA 的工具和设计者之间的交互语言。
* Verilog HDL 语言的描述能力能够通过使用编程语言接口（PLI）机制进一步扩展。PLI 是允许外部函数访问 Verilog 模块内信息、允许设计者与模拟器交互的例程集合。
* 设计能够在多个层次上加以描述，从开关级、门级、寄存器传送级（RTL）到算法级，包括进程和队列级。
* 能够使用内置开关级原语在开关级对设计完整建模。
* 同一语言可用于生成模拟激励和指定测试的验证约束条件，例如输入值的指定。
* Verilog HDL 能够监控模拟验证的执行，即模拟验证执行过程中设计的值能够被监控和显示。这些值也能够用于与期望值比较，在不匹配的情况下，打印报告消息。
* 在行为级描述中，Verilog HDL 不仅能够在 RTL 级上进行设计描述，而且能够在体系结构级描述及其算法级行为上进行设计描述。
* 能够使用门和模块实例化语句在结构级进行结构描述。
* Verilog HDL 的混合方式建模能力，即在一个设计中每个模块均可以在不同设计层次上建模。
* Verilog HDL 还具有内置逻辑函数，例如&（按位与）和|（按位或）。
* 对高级编程语言结构，例如条件语句、情况语句和循环语句，语言中都可以使用。
* 可以显式地对并发和定时进行建模。
* 提供强有力的文件读写能力。
* 语言在特定情况下是非确定性的，即在不同的模拟器上模型可以产生不同的结果；例如，事件队列上的事件顺序在标准中没有定义。

习题

1. Verilog HDL 是在哪一年首次被 IEEE 标准化的？
2. Verilog HDL 支持哪三种基本描述方式？
3. 可以使用 Verilog HDL 描述一个设计的时序吗？
4. 语言中的什么特性能够用于描述参数化设计？
5. 能够使用 Verilog HDL 编写测试验证程序吗？
6. Verilog HDL 是由哪个公司最先开发的？
7. Verilog HDL 中的两类主要数据类型什么？
8. UDP 代表什么？
9. 写出两个开关级基本门的名称。
10. 写出两个基本逻辑门的名称。

第 2 章 HDL 指南

本章提供 HDL 语言的速成指南。

2.1 模块

模块是 Verilog 的基本描述单位，用于描述某个设计的功能或结构及其与其他模块通信的外部端口。一个设计的结构可使用开关级原语、门级原语和用户定义的原语方式描述; 设计的数据流行为使用连续赋值语句进行描述; 时序行为使用过程结构描述。一个模块可以在另一个模块中使用。

一个模块的基本语法如下：

```
module module_name (port_list);
Declarations:
reg, wire, parameter,
input, output, inout,
function, task, . . .
Statements:
Initial statement
Always statement
Module instantiation
Gate instantiation
UDP instantiation
Continuous assignment
endmodule
```

说明部分用于定义不同的项，例如模块描述中使用的寄存器和参数。语句定义设计的功能和结构。说明部分和语句可以散布在模块中的任何地方；但是变量、寄存器、线网和参数等的说明部分必须在使用前出现。为了使模块描述清晰和具有良好的可读性, 最好将所有的说明部分放在语句前。本书中的所有实例都遵守这一规范。

以下为建模一个半加器电路的模块的简单实例。

```
module HalfAdder (A, B, Sum, Carry);
input A, B;
output Sum, Carry;

assign #2 Sum = A ^ B;
assign #5 Carry = A & B;
endmodule
```

模块的名字是 HalfAdder。 模块有 4 个端口: 两个输入端口 A 和 B，两个输出端口 Sum 和 Carry。由于没有定义端口的位数, 所有端口大小都为 1 位；同时, 由于没有各端口的数据类型说明, 这四个端口都是线网数据类型。
模块包含两条描述半加器数据流行为的连续赋值语句。从这种意义上讲，这些语句在模块中出现的顺序无关紧要，这些语句是并发的。每条语句的执行顺序依赖于发生在变量 A 和 B 上的事件。

在模块中，可用下述方式描述一个设计：

1) 数据流方式;
2) 行为方式;
3) 结构方式;
4) 上述描述方式的混合。

下面几节通过实例讲述这些设计描述方式。不过有必要首先对 Verilog HDL 的时延作简要介绍。

2.2 时延

Verilog HDL 模型中的所有时延都根据时间单位定义。 下面是带时延的连续赋值语句实例。

```
assign #2 Sum = A ^ B;
```

#2 指 2 个时间单位。

使用编译指令将时间单位与物理时间相关联。这样的编译器指令需在模块描述前定义，如下所示:

```
` timescale 1ns /100ps
```

此语句说明时延时间单位为 1ns 并且时间精度为 100ps (时间精度是指所有的时延必须被限定在 0.1ns 内)。 如果此编译器指令所在的模块包含上面的连续赋值语句, #2 代表 2ns。

如果没有这样的编译器指令, Verilog HDL 模拟器会指定一个缺省时间单位。IEEE Verilog HDL 标准中没有规定缺省时间单位。

2.3 数据流描述方式

用数据流描述方式对一个设计建模的最基本的机制就是使用连续赋值语句。在连续赋值语句中，某个值被指派给线网变量。 连续赋值语句的语法为:

```
assign [delay] LHS_net = RHS_ expression;
```

右边表达式使用的操作数无论何时发生变化, 右边表达式都重新计算, 并且在指定的时延后变化值被赋予左边表达式的线网变量。时延定义了右边表达式操作数变化与赋值给左边表达式之间的持续时间。如果没有定义时延值, 缺省时延为 0。
下面的例子显示了使用数据流描述方式对 2-4 解码器电路的建模的实例模型。

```
`timescale 1ns/ 1ns
module Decoder2x4 (A, B, EN, Z);
input A, B, EN;
output [ 0 :3] Z;
wire Abar, Bbar;

assign #1 Abar = ~ A; / / 语句 1。
assign #1 Bbar = ~ B; / / 语句 2。
assign #2 Z[0] = ~ (Abar & Bbar & EN) ; / / 语句 3。
assign #2 Z[1] = ~ (Abar & B & EN) ; / / 语句 4。
assign #2 Z[2] = ~ (A & Bbar & EN) ; / / 语句 5。
assign #2 Z[3] = ~ (A & B & EN) ; / / 语句 6。
endmodule
```

以反引号“ ` ”开始的第一条语句是编译器指令，编译器指令`timescale 将模块中所有时延的单位设置为 1 ns，时间精度为 1 ns。例如，在连续赋值语句中时延值#1 和#2 分别对应时延 1 ns 和 2 ns。

模块 Decoder2x4 有 3 个输入端口和 1 个 4 位输出端口。线网类型说明了两个连线型变量 Abar 和 Bbar (连线类型是线网类型的一种)。此外，模块包含 6 个连续赋值语句。

当 EN 在第 5 ns 变化时,语句 3、4、5 和 6 执行。这是因为 EN 是这些连续赋值语句中右边表达式的操作数。Z[0]在第 7 ns 时被赋予新值 0。当 A 在第 15 ns 变化时，语句 1、5 和 6 执行。执行语句 5 和 6 不影响 Z[0]和 Z[1]的取值。执行语句 5 导致 Z[2]值在第 17 ns 变为 0。执行语句 1 导致 Abar 在第 16 ns 被重新赋值。由于 Abar 的改变，反过来又导致 Z[0]值在第 18 ns 变为 1。

请注意连续赋值语句是如何对电路的数据流行为建模的 ;这种建模方式是隐式而非显式的建模方式。此外,连续赋值语句是并发执行的，也就是说各语句的执行顺序与其在描述中出现的顺序无关。

2.4 行为描述方式

设计的行为功能使用下述过程语句结构描述：

1) initial 语句：此语句只执行一次。

2) always 语句：此语句总是循环执行，或者说此语句重复执行。

只有寄存器类型数据能够在这两种语句中被赋值。寄存器类型数据在被赋新值前保持原有值不变。所有的初始化语句和 always 语句在 0 时刻并发执行。

下例为 always 语句对 1 位全加器电路建模的示例。

```
module FA_Seq (A, B, Cin, Sum, Cout);
input A, B, Cin;
output Sum, Cout;
reg Sum, Cout;
reg T1, T2, T3;
always
@ ( A or B or Cin ) begin
Sum = (A ^ B) ^ Cin;
```

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

模块 FA_Seq 有三个输入和两个输出。由于 Sum、Cout、T1、T2 和 T3 在 always 语句中被赋值,它们被说明为 reg 类型(reg 是寄存器数据类型的一种)。always 语句中有一个与事件控制(紧跟在字符@ 后面的表达式)。相关联的顺序过程(begin-end 对)。这意味着只要 A、B 或 Cin 上发生事件，即 A、B 或 Cin 之一的值发生变化，顺序过程就执行。在顺序过程中的语句顺序执行，并且在顺序过程执行结束后被挂起。顺序过程执行完成后，always 语句再次等待 A、B 或 Cin 上发生的事件。

在顺序过程中出现的语句是过程赋值模块化的实例。模块化过程赋值在下一条语句执行前完成执行。过程赋值可以有一个可选的时延。

时延可以细分为两种类型:

1) 语句间时延: 这是时延语句执行的时延。

2) 语句内时延: 这是右边表达式数值计算与左边表达式赋值间的时延。

下面是语句间时延的示例：

```
Sum = (A ^ B) ^ Cin;
#4 T1 = A & Cin;
```

在第二条语句中的时延规定赋值延迟 4 个时间单位执行。就是说，在第一条语句执行后等待 4 个时间单位，然后执行第二条语句。下面是语句内时延的示例。

```
Sum = #3 (A^ B) ^ Cin;
```

这个赋值中的时延意味着首先计算右边表达式的值, 等待 3 个时间单位，然后赋值给 Sum。

如果在过程赋值中未定义时延，缺省值为 0 时延，也就是说，赋值立即发生。这种形式以及在 always 语句中指定语句的其他形式将在第 8 章中详细讨论。

下面是 initial 语句的示例：

```
`timescale 1ns / 1ns
module Test (Pop, Pid);
output Pop, Pid;
reg Pop, Pid;

initial
begin
Pop = 0; // 语句 1。
Pid = 0; // 语句 2。
Pop = #5 1; // 语句 3。
Pid = #3 1; // 语句 4。
```

```
Pop = #6 0; // 语句 5。
Pid = #2 0; // 语句 6。
end
endmodule
```

initial 语句包含一个顺序过程。这一顺序过程在 0 ns 时开始执行，并且在顺序过程中所有语句全部执行完毕后, initial 语句永远挂起。这一顺序过程包含带有定义语句内时延的分组过程赋值的实例。语句 1 和 2 在 0 ns 时执行。第三条语句也在 0 时刻执行, 导致 Pop 在第 5 ns 时被赋值。语句 4 在第 5 ns 执行,并且 Pid 在第 8 ns 被赋值。同样，Pop 在 14 ns 被赋值 0，Pid 在第 16 ns 被赋值 0。第 6 条语句执行后，initial 语句永远被挂起。

2.5 结构化描述形式

在 Verilog HDL 中可使用如下方式描述结构:

1) 内置门原语(在门级)；
2) 开关级原语(在晶体管级)；
3) 用户定义的原语(在门级)；
4) 模块实例 (创建层次结构)。

通过使用线网来相互连接。下面的结构描述形式使用内置门原语描述的全加器电路实例。

```
module FA_Str (A, B, Cin, Sum, Cout);
input A, B, Cin;
output Sum, Cout;
wire S1, T1, T2, T3;

xor
X1 (S1, A, B),
X2 (Sum, S1, Cin);

and
A1 (T3, A, B),
A2 (T2, B, Cin),
A3 (T1, A, Cin),

or
O1 (Cout, T1, T2, T3);
endmodule
```

在这一实例中，模块包含门的实例语句，也就是说包含内置门 xor、and 和 or 的实例语句。门实例由线网类型变量 S1、T1、T2 和 T3 互连。由于没有指定的顺序, 门实例语句可以以任何顺序出现；图中显示了纯结构；xor、and 和 or 是内置门原语；X1、X2、A1 等是实例名称。紧跟在每个门后的信号列表是它的互连；列表中的第一个是门输出，余下的是输入。例如，S1 与 xor 门实例 X1 的输出连接，而 A 和 B 与实例 X1

的输入连接。
4 位全加器可以使用 4 个 1 位全加器模块描述。下面是 4 位全加器的结构描述形式。

```
module FourBitFA (FA, FB, FCin, FSum, FCout );
parameter SIZE = 4;
input [SIZE:1] FA, FB;
output [SIZE:1] FSum
input FCin;
input FCout;
wire [ 1: SIZE－1] FTemp;
FA_Str
FA1( .A (FA[1]), .B(FB[1]), .Cin(FCin),
.Sum(FSum[1]), .Cout(FTemp[2])),
FA2( .A (FA[2]), .B(FB[2]), .Cin(FTemp[1]),
.Sum(FSum[2]), .Cout(FTemp[2])),
FA3(FA[3], FB[3], FTemp[2], FSum[3], FTemp[3],
FA4(FA[4], FB[4], FTemp[3], FSum[4], FCout);
endmodule
```

在这一实例中，模块实例用于建模 4 位全加器。在模块实例语句中，端口可以与名称或位置关联。前两个实例 FA1 和 FA2 使用命名关联方式，也就是说，端口的名称和它连接的线网被显式描述（每一个的形式都为“.port_name (net_name)）。最后两个实例语句，实例 FA3 和 FA4 使用位置关联方式将端口与线网关联。这里关联的顺序很重要，例如，在实例 FA4 中，第一个 FA[4]与 FA_Str 的端口 A 连接，第二个 FB[4]与 FA_Str 的端口 B 连接，余下的由此类推。

2.6 混合设计描述方式

在模块中，结构的和行为的结构可以自由混合。也就是说，模块描述中可以包含实例化的门、模块实例化语句、连续赋值语句以及 always 语句和 initial 语句的混合。它们之间可以相互包含。来自 always 语句和 initial 语句（切记只有寄存器类型数据可以在这两种语句中赋值）的值能够驱动门或开关，而来自于门或连续赋值语句（只能驱动线网）的值能够反过来用于触发 always 语句和 initial 语句。

下面是混合设计方式的 1 位全加器实例。

```
module FA_Mix (A, B, Cin, Sum, Cout);
input A,B, Cin;
output Sum, Cout;
reg Cout;
reg T1, T2, T3;
wire S1;

xor X1(S1, A, B); // 门实例语句。

always
```

```
@ ( A or B or Cin ) begin // always 语句。
T1 = A & Cin;
T2 = B & Cin;
T3 = A & B;
Cout = (T1| T2) | T3;
end

assign Sum = S1 ^ Cin; // 连续赋值语句。
endmodule
```

只要 A 或 B 上有事件发生，门实例语句即被执行。只要 A、B 或 Cin 上有事件发生，就执行 always 语句，并且只要 S1 或 Cin 上有事件发生，就执行连续赋值语句。

2.7 设计模拟

Verilog HDL 不仅提供描述设计的能力，而且提供对激励、控制、存储响应和设计验证的建模能力。激励和控制可用初始化语句产生。验证运行过程中的响应可以作为“变化时保存”或作为选通的数据存储。最后，设计验证可以通过在初始化语句中写入相应的语句自动与期望的响应值比较完成。
下面是测试模块 Top 的例子。该例子测试 2.3 节中讲到的 FA_Seq 模块。

```
‘timescale 1ns/1ns
module Top; // 一个模块可以有一个空的端口列表。
reg PA, PB, PCi;
wire PCo, PSum;

// 正在测试的实例化模块：
FA_Seq F1(PA, PB, PCi, PSum, PCo); // 定位。

initial
begin: ONLY_ONCE
reg [3:0] Pal;
//需要 4 位, Pal 才能取值 8。

for (Pal = 0; Pal < 8; Pal = Pal + 1)
begin
{PA, PB, PCi} = Pal;
#5 $display (“PA, PB, PCi = %b%b%b”, PA, PB, PCi,
“ : : : PCo, PSum=%b%b”, PCo, PSum);
end
end
endmodule
```

在测试模块描述中使用位置关联方式将模块实例语句中的信号与模块中的端口相连接。也就是说，PA 连接到模块 FA_Seq 的端口 A，PB 连接到模块 FA_Seq 的端口 B，

依此类推。注意初始化语句中使用了一个 for 循环语句，在 PA、PB 和 PCi 上产生波形。for 循环中的第一条赋值语句用于表示合并的目标。自右向左，右端各相应的位赋给左端的参数。初始化语句还包含有一个预先定义好的系统任务。系统任务$display 将输入以特定的格式打印输出。

系统任务$display 调用中的时延控制规定$display 任务在 5 个时间单位后执行。这 5 个时间单位基本上代表了逻辑处理时间。即是输入向量的加载至观察到模块在测试条件下输出之间的延迟时间。

这一模型中还有另外一个细微差别。Pal 在初始化语句内被局部定义。为完成这一功能，初始化语句中的顺序过程（begin-end）必须标记。在这种情况下, ONLY_ONCE 是顺序过程标记。如果在顺序过程内没有局部声明的变量，就不需要该标记。下面是测试模块产生的输出。

```
PA, PB, PCi = 000 ::: PCo, PSum = 00
PA, PB, PCi = 001 ::: PCo, PSum = 01
PA, PB, PCi = 010 ::: PCo, PSum = 01
PA, PB, PCi = 011 ::: PCo, PSum = 10
PA, PB, PCi = 100 ::: PCo, PSum = 01
PA, PB, PCi = 101 ::: PCo, PSum = 10
PA, PB, PCi = 110 ::: PCo, PSum = 10
PA, PB, PCi = 111 ::: PCo, PSum = 11
```

验证与非门交叉连接构成的 RS_FF 模块的测试模块如下例所示。

```
`timescale 10ns/1ns
module RS_FF (Q, Qbar, R, S);
output Q, Qbar;
input R, S;

nand #1 (Q, R, Qbar);
nand #1 (Qbar, S, Q,);
//在门实例语句中，实例名称是可选的。
endmodule

module Test;
reg TS, TR;
wire TQ, TQb;

//测试模块的实例语句：
RS_FF NSTA (.Q(TQ), .S(TS), .R(TR), .Qbar(TQb));
//采用端口名相关联的连接方式。

// 加载激励：
initial
begin:
TR = 0;
```

```
TS = 0;
#5 TS = 1;
#5 TS = 0;
TR = 1;
#5 TS = 1;
TR = 0;
#5 TS = 0;
#5 TR = 1;
end
//输出显示：
initial
$monitor ("At time %t ," , $time,
"TR = %b, TS=%b, TQ=%b, TQb= %b", TR, TS, TQ, TQb);
endmodule
```

RS_FF 模块描述了设计的结构。在门实例语句中使用门时延；例如，第一个实例语句中的门时延为 1 个时间单位。该门时延意味着如果 R 或 Qbar 假定在 T 时刻变化，Q 将在 T+1 时刻获得计算结果值。

模块 Test 是一个测试模块。测试模块中的 RS_FF 用实例语句说明其端口用端口名关联方式连接。在这一模块中有两条初始化语句。第一个初始化语句只简单地产生 TS 和 TR 上的波形。这一初始化语句包含带有语句间时延的程序块过程赋值语句。

第二条初始化语句调用系统任务$monitor。这一系统任务调用的功能是只要参数表中指定的变量值发生变化就打印指定的字符串。下面是测试模块产生的输出。请注意 `timescale 指令在时延上的影响。

```
At time 0, TR=0, TS=0, TQ=x, TQb= x
At time 10, TR=0, TS=0, TQ=1, TQb= 1
At time 50, TR=0, TS=1, TQ=1, TQb= 1
At time 60, TR=0, TS=1, TQ=1, TQb= 0
At time 100, TR=1, TS=0, TQ=1, TQb= 0
At time 110, TR=1, TS=0, TQ=1, TQb= 1
At time 120, TR=1, TS=0, TQ=0, TQb= 1
At time 150, TR=0, TS=1, TQ=0, TQb= 1
At time 160, TR=0, TS=1, TQ=1, TQb= 1
At time 170, TR=0, TS=1, TQ=1, TQb= 0
At time 200, TR=0, TS=0, TQ=1, TQb= 0
At time 210, TR=0, TS=0, TQ=1, TQb= 1
At time 250, TR=1, TS=0, TQ=1, TQb= 1
At time 260, TR=1, TS=0, TQ=0, TQb= 1
```

后面的章节将更详细地讲述这些主题。

习题

1．在数据流描述方式中使用什么语句描述一个设计？
2．使用`timescale 编译器指令的目的是什么？举出一个实例。
3．在过程赋值语句中可以定义哪两种时延？请举例详细说明。
4．initial 语句与 always 语句的关键区别是什么？
5．为 2.3 节中描述的模块 Decode2x4 编写一个测试验证程序。
6．列出你在 Verilog HDL 模型中使用的两类赋值语句。
7．在顺序过程中何时需要定义标记？
8．找出下面连续赋值语句的错误。

```
assign Reset = #2 ^ WriteBus;
```

第 3 章 Verilog 语言要素

本章介绍 Verilog HDL 的基本要素，包括标识符、注释、数值、编译程序指令、系统任务和系统函数。另外，本章还介绍了 Verilog 硬件描述语言中的两种数据类型。

3.1 标识符

Verilog HDL 中的标识符(identifier)可以是任意一组字母、数字、$符号和_(下划线)符号的组合，但标识符的第一个字符必须是字母或者下划线。另外，标识符是区分大小写的。以下是标识符的几个例子：

```
Count
COUNT //与 Count 不同。
_R1_D2
R56_68
FIVE$
```

转义标识符(escaped identifier)可以在一条标识符中包含任何可打印字符。转义标识符以\ (反斜线)符号开头，以空白结尾（空白可以是一个空格、一个制表字符或换行符）。下面例举了几个转义标识符：

```
\7400
\.*.$
\{******}
\~Q
\OutGate 与 OutGate 相同。
```

最后这个例子解释了在一条转义标识符中，反斜线和结束空格并不是转义标识符的一部分。也就是说，标识符\OutGate 和标识符 OutGate 恒等。

Verilog HDL 定义了一系列保留字，叫做关键词，它仅用于某些上下文中。 附录 A 列出了语言中的所有保留字。注意只有小写的关键词才是保留字。例如，标识符 always(这是个关键词)与标识符 ALWAYS(非关键词)是不同的。

另外，转义标识符与关键词并不完全相同。标识符\initial 与标识符 initial（这是个关键词）不同。注意这一约定与那些转义标识符不同。

3.2 注释

在 Verilog HDL 中有两种形式的注释。

```
/*第一种形式:可以扩展至
```

```
多行 */

//第二种形式:在本行结束。
```

3.3 格式

Verilog HDL 区分大小写。也就是说大小写不同的标识符是不同的。此外，Verilog HDL 是自由格式的，即结构可以跨越多行编写，也可以在一行内编写。白空（新行、制表符和空格）没有特殊意义。下面通过实例解释说明。

```
initial begin Top = 3' b001; #2 Top = 3' b011; end
```

和下面的指令一样:

```
initial
begin
Top = 3' b001;
#2 Top = 3' b011;
end
```

3.4 系统任务和函数

以$字符开始的标识符表示系统任务或系统函数。任务提供了一种封装行为的机制。这种机制可在设计的不同部分被调用。任务可以返回 0 个或多个值。函数除只能返回一个值以外与任务相同。此外，函数在 0 时刻执行，即不允许延迟，而任务可以带有延迟。

```
$display ("Hi, you have reached LT today");
/* $display 系统任务在新的一行中显示。*/
$time
//该系统任务返回当前的模拟时间。
```

系统任务和系统函数在第 10 章中详细讲解。

3.5 编译指令

以`（反引号）开始的某些标识符是编译器指令。在 Verilog 语言编译时，特定的编译器指令在整个编译过程中有效（编译过程可跨越多个文件），直到遇到其它的不同编译程序指令。完整的标准编译器指令如下:

```
* `define, `undef
* `ifdef, `else, `endif
* `default_nettype
* `include
* `resetall
```

* `timescale
* `unconnected_drive, `nounconnected_drive
* `celldefine, `endcelldefine

3.5.1 `define 和`undef

`define 指令用于文本替换，它很像 C 语言中的#define 指令，如:

```
`define MAX_BUS_SIZE 32
. . .
reg [ `MAX_BUS_SIZE - 1:0 ] AddReg;
```

一旦`define 指令被编译，其在整个编译过程中都有效。例如，通过另一个文件中的`define 指令，MAX_BUS_SIZE 能被多个文件使用。
`undef 指令取消前面定义的宏。例如:

```
`define WORD 16 //建立一个文本宏替代。
. . .
wire [ `WORD : 1] Bus;
. . .
`undef WORD
// 在`undef 编译指令后，WORD 的宏定义不再有效.
```

3.5.2 `ifdef、`else 和`endif

这些编译指令用于条件编译，如下所示：

```
`ifdef WINDOWS
parameter WORD_SIZE = 16
`else
parameter WORD_SIZE = 32
`endif
```

在编译过程中，如果已定义了名字为 WINDOWS 的文本宏，就选择第一种参数声明，否则选择第二种参数说明。

`else 程序指令对于`ifdef 指令是可选的。

3.5.3 `default_nettype

该指令用于为隐式线网指定线网类型。也就是将那些没有被说明的连线定义线网类型。

```
`default_nettype wand
```

该实例定义的缺省的线网为线与类型。因此，如果在此指令后面的任何模块中没

有说明的连线，那么该线网被假定为线与类型。

3.5.4 `include

`include 编译器指令用于嵌入内嵌文件的内容。文件既可以用相对路径名定义，也可以用全路径名定义，例如:

```
`include " . . / . . /primitives.v"
```

编译时，这一行由文件“ ../../primitives.v ” 的内容替代。

3.5.5 `resetall

该编译器指令将所有的编译指令重新设置为缺省值。

```
`resetall
```

例如，该指令使得缺省连线类型为线网类型。

3.5.6 `timescale

在 Verilog HDL 模型中，所有时延都用单位时间表述。使用`timescale 编译器指令将时间单位与实际时间相关联。该指令用于定义时延的单位和时延精度。`timescale 编译器指令格式为：

```
`timescale time_unit / time_precision
```

time_unit 和 time_precision 由值 1、10、和 100 以及单位 s、ms、us、ns、ps 和 fs 组成。例如：

```
`timescale 1ns/100ps
```

表示时延单位为 1ns，时延精度为 100ps。`timescale 编译器指令在模块说明外部出现，并且影响后面所有的时延值。例如:

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
output Z;
input A, B;

and # (5.22, 6.17 ) Al (Z, A, B);
//规定了上升及下降时延值。
endmodule
```

编译器指令定义时延以 ns 为单位，并且时延精度为 1/10 ns（100 ps）。因此，时延值 5.22 对应 5.2 ns，时延 6.17 对应 6.2 ns。如果用如下的`timescale 程序指令代替上例中的编译器指令，

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

那么 5.22 对应 52ns，6.17 对应 62ns。

在编译过程中，`timescale 指令影响这一编译器指令后面所有模块中的时延值，直至遇到另一个`timescale 指令或`resetall 指令。当一个设计中的多个模块带有自身的`timescale 编译指令时将发生什么？在这种情况下，模拟器总是定位在所有模块的最小时延精度上，并且所有时延都相应地换算为最小时延精度。例如，

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
output Z;
input A, B;

and # (5.22, 6.17 ) Al (Z, A, B);
endmodule

`timescale 10ns/ 1ns
module TB;
reg PutA, PutB;
wire Get0;

initial
begin
PutA = 0;
PutB = 0;
#5.21 PutB = 1;
#10.4 PutA = 1;
#15 PutB = 0;
end
AndFunc AF1(Get0, PutA, PutB);
endmodule
```

在这个例子中，每个模块都有自身的`timescale 编译器指令。`timescale 编译器指令第一次应用于时延。因此，在第一个模块中，5.22 对应 5.2 ns，6.17 对应 6.2 ns；在第二个模块中 5.21 对应 52 ns，10.4 对应 104 ns，15 对应 150 ns。如果仿真模块 TB，设计中的所有模块最小时间精度为 100 ps。因此，所有延迟（特别是模块 TB 中的延迟）将换算成精度为 100 ps。延迟 52 ns 现在对应 520*100 ps，104 对应 1040*100 ps，150 对应 1500*100 ps。更重要的是，仿真使用 100 ps 为时间精度。如果仿真模块 AndFunc，由于模块 TB 不是模块 AddFunc 的子模块，模块 TB 中的`timescale 程序指令将不再有效。

3.5.7 `unconnected_drive 和`nounconnected_drive

在模块实例化中，出现在这两个编译器指令间的任何未连接的输入端口或者为正偏电路状态或者为反偏电路状态。

```
`unconnected_drive pull1
. . .
/*在这两个程序指令间的所有未连接的输入端口为正偏电路状态（连接到高电平）*/
`nounconnected_drive

`unconnected_drive pull0
. . .
/*在这两个程序指令间的所有未连接的输入端口为反偏电路状态（连接到低电平）*/
`nounconnected_drive
```

3.5.8 `celldefine 和 `endcelldefine

这两个程序指令用于将模块标记为单元模块。它们表示包含模块定义，如下例所示。

```
`celldefine
module FD1S3AX (D, CK, Z) ;
. . .
endmodule
`endcelldefine
```

某些 PLI 例程使用单元模块。

3.6 值集合

Verilog HDL 有下列四种基本的值：

1) 0：逻辑 0 或“假”
2) 1：逻辑 1 或“真”
3) x：未知
4) z：高阻

注意这四种值的解释都内置于语言中。如一个为 z 的值总是意味着高阻抗，一个为 0 的值通常是指逻辑 0。

在门的输入或一个表达式中的为“z”的值通常解释成“x”。此外，x 值和 z 值都是不分大小写的，也就是说，值 0x1z 与值 0X1Z 相同。Verilog HDL 中的常量是由以上这四类基本值组成的。

Verilog HDL 中有三类常量：

1) 整型
2) 实数型
3) 字符串型

下划线符号（_）可以随意用在整数或实数中，它们就数量本身没有意义。它们能用来提高易读性；唯一的限制是下划线符号不能用作为首字符。

3.6.1 整型数

整型数可以按如下两种方式书写：

1）简单的十进制数格式
2）基数格式

1．简单的十进制格式
这种形式的整数定义为带有一个可选的“+”（一元）或“－”（一元）操作符的数字序列。下面是这种简易十进制形式整数的例子。
32 十进制数 32
－15 十进制数－15
这种形式的整数值代表一个有符号的数。负数可使用两种补码形式表示。因此 32 在 5 位的二进制形式中为 10000，在 6 位二进制形式中为 110001；－15 在 5 位二进制形式中为 10001，在 6 位二进制形式中为 110001。

2．基数表示法
这种形式的整数格式为：
[size] 'base value
size 定义以位计的常量的位长；base 为 o 或 O（表示八进制），b 或 B（表示二进制），d 或 D（表示十进制），h 或 H（表示十六进制）之一；value 是基于 base 的值的数字序列。值 x 和 z 以及十六进制中的 a 到 f 不区分大小写。
下面是一些具体实例：

5'O37 5 位八进制数
4'D2 4 位十进制数
4'B1x_01 4 位二进制数
7'Hx 7 位 x(扩展的 x)，即 xxxxxxx
4'hZ 4 位 z(扩展的 z)，即 zzzz
4'd-4 非法：数值不能为负
8'h 2 A 在位长和字符之间,以及基数和数值之间允许出现空格
3'bOO1 非法：` 和基数 b 之间不允许出现空格
(2+3)'b10 非法:位长不能够为表达式
注意，x（或 z）在十六进制值中代表 4 位 x（或 z），在八进制中代表 3 位 x（或 z），在二进制中代表 1 位 x（或 z）。
基数格式计数形式的数通常为无符号数。这种形式的整型数的长度定义是可选的。如果没有定义一个整数型的长度，数的长度为相应值中定义的位数。下面是两个例子：
'o721 9 位八进制数
'hAF 8 位十六进制数
如果定义的长度比为常量指定的长度长，通常在左边填 0 补位。但是如果数最左边一位为 x 或 z，就相应地用 x 或 z 在左边补位。例如：
10'b10 左边添 0 占位，0000000010
10'bx0x1 左边添 x 占位,xxxxxxx0x1
如果长度定义得更小，那么最左边的位相应地被截断。例如：
3'b1001_0011 与 3'b011 相等
5'H0FFF 与 5'H1F 相等
？字符在数中可以代替值 z 在值 z 被解释为不分大小写的情况下提高可读性（参见第 8 章）。

3.6.2 实数

实数可以用下列两种形式定义：

1) 十进制计数法；例如

```
2.0
5.678
11572.12
0.1
2. //非法：小数点两侧必须有 1 位数字
```

2) 科学计数法； 这种形式的实数举例如下：

```
23_5.1e2 其值为 23510.0; 忽略下划线
3.6E2 360.0 (e 与 E 相同)
5E－4 0.0005
```

Verilog 语言定义了实数如何隐式地转换为整数。实数通过四舍五入被转换为最相近的整数。

```
42.446， 42.45 转换为整数 42
92.5, 92.699 转换为整数 93
－15.62 转换为整数－16
－26.22 转换为整数－26
```

3.6.3 字符串

字符串是双引号内的字符序列。字符串不能分成多行书写。例如:

```
"INTERNAL ERROR"
"REACHED－>HERE"
```

用 8 位 ASCII 值表示的字符可看作是无符号整数。因此字符串是 8 位 ASCII 值的序列。为存储字符串“ INTERNAL ERROR ”，变量需要 8*14 位。

```
reg [1 : 8*14] Message;
. . .
Message = "INTERNAL ERROR"
```

反斜线 (\ ) 用于对确定的特殊字符转义。

```
\n 换行符
\t 制表符
\\ 字符\本身
\" 字符"
\206 八进制数 206 对应的字符
```

3.7 数据类型

Verilog HDL 有两大类数据类型。

1) 线网类型。net type 表示 Verilog 结构化元件间的物理连线。它的值由驱动元件的值决定，例如连续赋值或门的输出。如果没有驱动元件连接到线网，线网的缺省值为 z。

2) 寄存器类型。register type 表示一个抽象的数据存储单元，它只能在 always 语句和 initial 语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器类型的变量具有 x 的缺省值。

3.7.1 线网类型

线网数据类型包含下述不同种类的线网子类型。

* wire
* tri
* wor
* trior
* wand
* triand
* trireg
* tri1
* tri0
* supply0
* supply1

简单的线网类型说明语法为：

```
net_kind [msb:lsb] net1, net2, . . . , netN;
```

net_kind 是上述线网类型的一种。msb 和 lsb 是用于定义线网范围的常量表达式；范围定义是可选的；如果没有定义范围，缺省的线网类型为 1 位。下面是线网类型说明实例。

```
wire Rdy, Start; //2 个 1 位的连线。
wand [2:0] Addr; //Addr 是 3 位线与。
```

当一个线网有多个驱动器时，即对一个线网有多个赋值时，不同的线网产生不同的行为。例如，

```
wor Rde;
. . .
assign Rde = Blt & Wyl;
. . .
assign Rde = Kbl | Kip;
```

本例中，Rde 有两个驱动源，分别来自于两个连续赋值语句。由于它是线或线网，Rde 的有效值由使用驱动源的值（右边表达式的值）的线或(wor)表（参见后面线或网的有关章节）决定。

1. wire 和 tri 线网

用于连接单元的连线是最常见的线网类型。连线与三态线(tri)网语法和语义一致；三态线可以用于描述多个驱动源驱动同一根线的线网类型；并且没有其他特殊的意义。

```
wire Reset;
wire [3:2] Cla, Pla, Sla;
tri [ MSB－1 : LSB +1] Art;
```

如果多个驱动源驱动一个连线（或三态线网），线网的有效值由下表决定。

wire (或 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

下面是一个具体实例：

```
assign Cla = Pla & Sla;
. . .
assign Cla = Pla ^ Sla;
```

在这个实例中，Cla 有两个驱动源。两个驱动源的值（右侧表达式的值）用于在上表中索引，以便决定 Cla 的有效值。由于 Cla 是一个向量，每位的计算是相关的。例如，如果第一个右侧表达式的值为 01x，并且第二个右测表达式的值为 11z，那么 Cla 的有效值是 x1x (第一位 0 和 1 在表中索引到 x，第二位 1 和 1 在表中索引到 1，第三位 x 和 z 在表中索引到 x)。

2. wor 和 trior 线网

线或指如果某个驱动源为 1，那么线网的值也为 1。线或和三态线或(trior)在语法和功能上是一致的。

```
wor [MSB:LSB] Art;
trior [MAX－1: MIN－1] Rdx, Sdx, Bdx;
```

如果多个驱动源驱动这类网，网的有效值由下表决定。

wor (或 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

3. wand 和 triand 线网

线与(wand)网指如果某个驱动源为 0,那么线网的值为 0。线与和三态线与(triand)网在语法和功能上是一致的。

```
wand [-7 : 0] Dbus;
triand Reset, Clk;
```

如果这类线网存在多个驱动源，线网的有效值由下表决定。

```
wand (或 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
```

4. trireg 线网

此线网存储数值（类似于寄存器），并且用于电容节点的建模。当三态寄存器(trireg)的所有驱动源都处于高阻态，也就是说，值为 z 时，三态寄存器线网保存作用在线网上的最后一个值。此外，三态寄存器线网的缺省初始值为 x。

```
trireg [1:8] Dbus, Abus;
```

5. tri0 和 tri1 线网

这类线网可用于线逻辑的建模，即线网有多于一个驱动源。tri0（tri1）线网的特征是，若无驱动源驱动，它的值为 0（tri1 的值为 1）。

```
tri0 [－3:3] GndBus;
tri1 [0:－5] OtBus, ItBus;
```

下表显示在多个驱动源情况下 tri0 或 tri1 网的有效值。

```
tri0 (tri1) 0 1 x z
0 0 x x 0
1 x 1 x 1
x x x x x
z 0 1 x 0(1)
```

6. supply0 和 supply1 线网

supply0 用于对“地”建模，即低电平 0；supply1 网用于对电源建模，即高电平 1；例如:

```
supply0 Gnd, ClkGnd;
supply1 [2:0] Vcc;
```

3.7.2 未说明的线网

在 Verilog HDL 中，有可能不必声明某种线网类型。在这样的情况下，缺省线网类型为 1 位线网。

可以使用`default_nettype 编译器指令改变这一隐式线网说明方式。使用方法如下：

```
`default_nettype net_kind
```

例如，带有下列编译器指令：

```
`default_nettype wand
```

任何未被说明的网缺省为 1 位线与网。

3.7.3 向量和标量线网

在定义向量线网时可选用关键词 scalared 或 vectored。如果一个线网定义时使用了关键词 vectored，那么就不允许位选择和部分选择该线网。换句话说，必须对线网整体赋值（位选择和部分选择在下一章中讲解）。例如:

```
wire vectored [3:1] Grb;
//不允许位选择 Grb[2]和部分选择 Grb [3:2]
wor scalared [4:0] Best;
//与 wor [4:0] Best 相同，允许位选择 Best [2]和部分选择 Best [3:1]。
```

如果没有定义关键词，缺省值为标量。

3.7.4 寄存器类型

有 5 种不同的寄存器类型。

```
* reg
* integer
* time
* real
* realtime
```

1. reg 寄存器类型

寄存器数据类型 reg 是最常见的数据类型。reg 类型使用保留字 reg 加以说明，形式如下：

```
reg [ msb: lsb] reg1, reg2, . . . regN;
```

msb 和 lsb 定义了范围，并且均为常数值表达式。范围定义是可选的；如果没有定义范围，缺省值为 1 位寄存器。例如：

```
reg [3:0] Sat; //Sat 为 4 位寄存器。
reg Cnt; //1 位寄存器。
reg [1:32] Kisp, Pisp, Lisp;
```

寄存器可以取任意长度。寄存器中的值通常被解释为无符号数，例如：

```
reg [1:4] Comb;
. . .
Comb = －2; //Comb 的值为 14（1110），1110 是 2 的补码。
Comb = 5; //Comb 的值为 15（0101）。
```

2．存储器

存储器是一个寄存器数组。存储器使用如下方式说明：

```
reg [ msb: 1sb] memory1 [ upper1: lower1],
memory2 [upper2: lower2],. . . ;
```

例如：

```
reg [0:3 ] MyMem [0:63]
//MyMem 为 64 个 4 位寄存器的数组。
reg Bog [1:5]
//Bog 为 5 个 1 位寄存器的数组。
```

MyMem 和 Bog 都是存储器。数组的维数不能大于 2。注意存储器属于寄存器数组类型。线网数据类型没有相应的存储器类型。

单个寄存器说明既能够用于说明寄存器类型，也可以用于说明存储器类型。

```
parameter ADDR_SIZE = 16 , WORD_SIZE = 8;
reg [1: WORD_SIZE] RamPar [ ADDR_SIZE－1 : 0], DataReg;
```

RamPar 是存储器，是 16 个 8 位寄存器数组，而 DataReg 是 8 位寄存器。

在赋值语句中需要注意如下区别：存储器赋值不能在一条赋值语句中完成，但是寄存器可以。因此在存储器被赋值时，需要定义一个索引。下例说明它们之间的不同。

```
reg [1:5] Dig; //Dig 为 5 位寄存器。
. . .
Dig = 5'b11011;
```

上述赋值都是正确的，但下述赋值不正确：

```
reg BOg[1:5]; //Bog 为 5 个 1 位寄存器的存储器。
. . .
Bog = 5'b11011;
```

有一种存储器赋值的方法是分别对存储器中的每个字赋值。例如：

```
reg [0:3] Xrom [1:4]
. . .
Xrom[1] = 4'hA;
Xrom[2] = 4'h8;
Xrom[3] = 4'hF;
Xrom[4] = 4'h2;
```

为存储器赋值的另一种方法是使用系统任务：

1) $readmemb （加载二进制值）

2) $readmemb （加载十六进制值）

这些系统任务从指定的文本文件中读取数据并加载到存储器。文本文件必须包含

相应的二进制或者十六进制数。例如：

```
reg [1:4] RomB [7:1] ;
$ readmemb ("ram.patt", RomB);
```

Romb 是存储器。文件“ ram.patt ”必须包含二进制值。文件也可以包含空白空间和注释。下面是文件中可能内容的实例。

```
1101
1110
1000
0111
0000
1001
0011
```

系统任务$readmemb 促使从索引 7 即 Romb 最左边的字索引，开始读取值。如果只加载存储器的一部分，值域可以在$readmemb 方法中显式定义。例如：

```
$readmemb ("ram.patt", RomB, 5, 3);
```

在这种情况下只有 Romb[5],Romb[4]和 Romb[3]这些字从文件头开始被读取。被读取的值为 1101、1100 和 1000。
文件可以包含显式的地址形式。

```
@hex_address value
```

如下实例：

```
@5 11001
@2 11010
```

在这种情况下，值被读入存储器指定的地址。

当只定义开始值时，连续读取直至到达存储器右端索引边界。例如：

```
$readmemb ("rom.patt", RomB, 6);
//从地址 6 开始，并且持续到 1。
$readmemb ( "rom.patt", RomB, 6, 4);
//从地址 6 读到地址 4。
```

3. Integer 寄存器类型

整数寄存器包含整数值。整数寄存器可以作为普通寄存器使用，典型应用为高层次行为建模。使用整数型说明形式如下：

```
integer integer1, integer2,. . . intergerN [msb:1sb] ;
```

msb 和 lsb 是定义整数数组界限的常量表达式，数组界限的定义是可选的。注意容许

无位界限的情况。一个整数最少容纳 32 位。但是具体实现可提供更多的位。下面是整数说明的实例。

```
integer A, B, C; //三个整数型寄存器。
integer Hist [3:6]; //一组四个寄存器。
```

一个整数型寄存器可存储有符号数，并且算术操作符提供 2 的补码运算结果。
整数不能作为位向量访问。例如，对于上面的整数 B 的说明，B[6]和 B[20:10]是非法的。一种截取位值的方法是将整数赋值给一般的 reg 类型变量，然后从中选取相应的位，如下所示：

```
reg [31:0] Breg;
integer Bint;
. . .
//Bint[6]和 Bint[20:10]是不允许的。
. . .
Breg = Bint;
/*现在，Breg[6]和 Breg[20:10]是允许的，并且从整数 Bint 获取相应的位值。*/
```

上例说明了如何通过简单的赋值将整数转换为位向量。类型转换自动完成，不必使用特定的函数。从位向量到整数的转换也可以通过赋值完成。例如:

```
integer J;
reg [3:0] Bcq;

J = 6; //J 的值为 32'b0000...00110。
Bcq = J; // Bcq 的值为 4'b0110。

Bcq = 4'b0101.
J = Bcq; //J 的值为 32'b0000...00101。

J = －6; //J 的值为 32'b1111...11010。
Bcq = J; //Bcq 的值为 4'b1010。
```

注意赋值总是从最右端的位向最左边的位进行；任何多余的位被截断。如果你能够回忆起整数是作为 2 的补码位向量表示的，就很容易理解类型转换。

4. time 类型

time 类型的寄存器用于存储和处理时间。time 类型的寄存器使用下述方式加以说明。

```
time time_id1, time_id2, . . . ,time_idN [ msb:lsb];
```

msb 和 lsb 是表明范围界限的常量表达式。如果未定义界限，每个标识符存储一个至少 64 位的时间值。时间类型的寄存器只存储无符号数。例如:

```
time Events [0:31]; //时间值数组。
time CurrTime; //CurrTime 存储一个时间值。
```

5. real 和 realtime 类型

实数寄存器（或实数时间寄存器）使用如下方式说明：

```
//实数说明：
real real_reg1, real_reg2, . . ., real_regN;
//实数时间说明：
realtime realtime_reg1, realtime_reg2, . . . ,realtime_regN;
```

realtime 与 real 类型完全相同。例如:

```
real Swing, Top;
realtime CurrTime;
```

real 说明的变量的缺省值为 0。不允许对 real 声明值域、位界限或字节界限。

当将值 x 和 z 赋予 real 类型寄存器时，这些值作 0 处理。

```
real RamCnt;
. . .
RamCnt = 'b01x1Z;
```

RamCnt 在赋值后的值为'b01010。

3.8 参数

参数是一个常量。参数经常用于定义时延和变量的宽度。使用参数说明的参数只被赋值一次。参数说明形式如下：

```
parameter param1 = const_expr1, param2 = const_expr2, . . . ,
paramN = const_exprN;
```

下面为具体实例：

```
parameter LINELENGTH = 132, ALL_X_S = 16'bx;
parameter BIT = 1, BYTE = 8, PI = 3.14;
parameter STROBE_DELAY = ( BYTE + BIT) / 2;
parameter TQ_FILE = " /home/bhasker/TEST/add.tq";
```

参数值也可以在编译时被改变。改变参数值可以使用参数定义语句或通过在模块初始化语句中定义参数值（这两种机制将在第 9 章中详细讲解）。

习题

1. 下列标识符哪些合法，哪些非法？

   COunT, 1_2 Many, **1, Real?, \wait, Initial

2. 系统任务和系统函数的第一个字符标识符是什么？

3．举例说明文本替换编译指令？
4．在 Verilog HDL 中是否有布尔类型？
5．下列表达式的位模式是什么？
```
7'o44, 'Bx0, 5'bx110, 'hA0, 10'd2, 'hzF
```
6．赋值后存储在 Qpr 中的位模式是什么？
```
reg [1:8*2] Qpr;
. . .
Qpr = "ME" ;
```
7．如果线网类型变量说明后未赋值，其缺省值为多少？
8．Verilog HDL 允许没有显式说明的线网类型。如果是这样，怎样决定线网类型？
9．下面的说明错在哪里？
```
integer [0:3] Ripple;
```
10．编写一个系统任务从数据文件“memA.data”中加载 32×64 字存储器。
11．写出在编译时覆盖参数值的两种方法。

第 4 章 表 达 式

本章讲述在 Verilog HDL 中编写表达式的基础。
表达式由操作数和操作符组成。表达式可以在出现数值的任何地方使用。

4.1 操作数

操作数可以是以下类型中的一种：
1) 常数
2) 参数
3) 线网
4) 寄存器
5) 位选择
6) 部分选择
7) 存储器单元
8) 函数调用

4.1.1 常数

前面的章节已讲述了如何书写常量。下面是一些实例。

256,7 //非定长的十进制数。
4'b10_11, 8'h0A //定长的整型常量。
'b1, 'hFBA //非定长的整数常量。
90.00006 //实数型常量。
"BOND" //串常量；每个字符作为 8 位 ASCII 值存储。

表达式中的整数值可被解释为有符号数或无符号数。如果表达式中是十进制整数，例如，12 被解释为有符号数。如果整数是基数型整数（定长或非定长），那么该整数作为无符号数对待。下面举例说明。

12 是 01100 的 5 位向量形式（有符号）
-12 是 10100 的 5 位向量形式（有符号）
5'b01100 是十进制数 12（无符号）
5'b10100 是十进制数 20（无符号）
4'd12 是十进制数 12（无符号）

更为重要的是对基数表示或非基数表示的负整数处理方式不同。非基数表示形式的负整数作为有符号数处理，而基数表示形式的负整数值作为无符号数。因此-44 和 -6'o54 （十进制的 44 等于八进制的 54）在下例中处理不同。

```
integer Cone;
. . .
Cone = -44/4
Cone = -6'o54/ 4;
```

注意－44 和－6'o54 以相同的位模式求值；但是－44 作为有符号数处理，而－6'o54 作为无符号数处理。因此第一个字符中 Cone 的值为－11，而在第二个赋值中 Cone 的值为 1073741813。

4.1.2 参数

前一章中已对参数作了介绍。参数类似于常量，并且使用参数声明进行说明。下面是参数说明实例。

```
parameter LOAD = 4'd12, STORE = 4'd10;
```

LOAD 和 STORE 为参数的例子，值分别被声明为 12 和 10。

4.1.3 线网

可在表达式中使用标量线网（1 位）和向量线网（多位）。下面是线网说明实例。

```
wire [0:3] Prt; //Prt 为 4 位向量线网。
wire Bdq; //Bbq 是标量线网。
```

线网中的值被解释为无符号数。在连续赋值语句中，

```
assign Prt = -3;
```

Prt 被赋于位向量 1101，实际上为十进制的 13。在下面的连续赋值中，

```
assign Prt = 4'HA;
```

Prt 被赋于位向量 1010，即为十进制的 10。

4.1.4 寄存器

标量和向量寄存器可在表达式中使用。寄存器变量使用寄存器声明进行说明。例如:

```
integer TemA, TemB;
reg [1:5] State;
time Que [1:5];
```

整型寄存器中的值被解释为有符号的二进制补码数，而 reg 寄存器或时间寄存器中的值被解释为无符号数。实数和实数时间类型寄存器中的值被解释为有符号浮点数。

```
TemA = -10; //TemA 值为位向量 10110，是 10 的二进制补码。
TemA = 'b1011; //TemA 值为十进制数 11。
```

State = -10; //State 值为位向量 10110，即十进制数 22。
State = 'b1011; //State 值为位向量 01011，是十进制值 11。

4.1.5 位选择

位选择从向量中抽取特定的位。形式如下：

net_or_reg_vector [bit_select_expr]

下面是表达式中应用位选择的例子。

State [1] && State [4] //寄存器位选择。
Prt [0] | Bbq //线网位选择。

如果选择表达式的值为 x、z，或越界，则位选择的值为 x。例如 State [x]值为 x。

4.1.6 部分选择

在部分选择中，向量的连续序列被选择。形式如下：

net_or_reg_vector [msb_const_expr:1sb_const_expr]

其中范围表达式必须为常数表达式。例如。

State [1:4] //寄存器部分选择。
Prt [1:3] //线网部分选择。
选择范围越界或为 x、z 时，部分选择的值为 x。

4.1.7 存储器单元

存储器单元从存储器中选择一个字。形式如下：

memory [word_address]

例如：
reg [1:8] Ack, Dram [0:63];
. . .
Ack = Dram [60]; //存储器的第 60 个单元。

不允许对存储器变量值部分选择或位选择。例如，

Dram [60] [2] 不允许。
Dram [60] [2:4] 也不允许。

在存储器中读取一个位或部分选择一个字的方法如下：将存储器单元赋值给寄存器变量，然后对该寄存器变量采用部分选择或位选择操作。例如，Ack [2] 和 Ack [2:4] 是合法的表达式。

4.1.8 函数调用

表达式中可使用函数调用。函数调用可以是系统函数调用（以$字符开始）或用户定义的函数调用。例如：

```
$time + SumOfEvents (A, B)
/*$time 是系统函数，并且 SumOfEvents 是在别处定义的用户自定义函数。*/
```

4.2 操作符

Verilog HDL 中的操作符可以分为下述类型：

1) 算术操作符
2) 关系操作符
3) 相等操作符
4) 逻辑操作符
5) 按位操作符
6) 归约操作符
7) 移位操作符
8) 条件操作符
9) 连接和复制操作符

下表显示了所有操作符的优先级和名称。操作符从最高优先级（顶行）到最低优先级（底行）排列。同一行中的操作符优先级相同。

除条件操作符从右向左关联外，其余所有操作符自左向右关联。下面的表达式：

```
A + B - C
等价于：
(A + B ) - C //自左向右
而表达式：
A ? B : C ? D : F
等价于：
A ? B : (C ? D : F) //从右向左
圆扩号能够用于改变优先级的顺序，如以下表达式：
(A ? B : C) ? D : F
```

4.2.1 算术操作符

算术操作符有：

* +（一元加和二元加）

* — （一元减和二元减）
* * （乘）
* / （除）
* % （取模）

整数除法截断任何小数部分。例如：

7/4 结果为 1
取模操作符求出与第一个操作符符号相同的余数。
7%4 结果为 3
而:
- 7%4 结果为 -3

如果算术操作符中的任意操作数是 X 或 Z，那么整个结果为 X。例如：

'b10x1 + 'b01111 结果为不确定数'bxxxxx

1. 算术操作结果的长度

算术表达式结果的长度由最长的操作数决定。在赋值语句下，算术操作结果的长度由操作符左端目标长度决定。考虑如下实例：

```
reg [0:3] Arc, Bar, Crt;
reg [0:5] Frx;
. . .
Arc = Bar + Crt;
Frx = Bar + Crt;
```

第一个加的结果长度由 Bar，Crt 和 Arc 长度决定，长度为 4 位。第二个加法操作的长度同样由 Frx 的长度决定（Frx、Bat 和 Crt 中的最长长度），长度为 6 位。在第一个赋值中，加法操作的溢出部分被丢弃；而在第二个赋值中，任何溢出的位存储在结果位 Frx[1]中。

在较大的表达式中，中间结果的长度如何确定？在 Verilog HDL 中定义了如下规则：表达式中的所有中间结果应取最大操作数的长度（赋值时，此规则也包括左端目标）。考虑另一个实例：

```
wire [4:1] Box, Drt;
wire [1:5] Cfg;
wire [1:6] Peg;
wire [1:8] Adt;
. . .
assign Adt = (Box + Cfg) + (Drt + Peg);
```

表达式左端的操作数最长为 6，但是将左端包含在内时，最大长度为 8。所以所有的加操作使用 8 位进行。例如：Box 和 Cfg 相加的结果长度为 8 位。

2. 无符号数和有符号数

执行算术操作和赋值时，注意哪些操作数为无符号数、哪些操作数为有符号数非常重要。无符号数存储在：

* 线网
* 一般寄存器
* 基数格式表示形式的整数

有符号数存储在：

* 整数寄存器
* 十进制形式的整数

下面是一些赋值语句的实例：

```
reg [0:5] Bar;
integer Tab;
. . .
Bar = -4'd12; //寄存器变量 Bar 的十进制数为 52，向量值为 110100。
Tab = -4'd12; //整数 Tab 的十进制数为-12，位形式为 110100。

-4'd12 / 4 //结果是 1073741821。
-12 / 4 //结果是-3
```

因为 Bar 是普通寄存器类型变量，只存储无符号数。右端表达式的值为'b110100（12 的二进制补码）。因此在赋值后，Bar 存储十进制值 52。在第二个赋值中，右端表达式相同，值为'b110100，但此时被赋值为存储有符号数的整数寄存器。Tab 存储十进制值－12（位向量为 110100）。注意在两种情况下，位向量存储内容都相同；但是在第一种情况下，向量被解释为无符号数，而在第二种情况下，向量被解释为有符号数。

下面为具体实例：

```
Bar = - 4'd12/4;
Tab = - 4'd12 /4;

Bar = - 12/4
Tab = - 12/4
```

在第一次赋值中，Bar 被赋于十进制值 61（位向量为 111101）。而在第二个赋值中，Tab 被赋于与十进制 1073741821（位值为 0011...11101）。Bar 在第三个赋值中赋于与第一个赋值相同的值。这是因为 Bar 只存储无符号数。在第四个赋值中，Bar 被赋于十进制值－3。

下面是另一些例子：

```
Bar = 4 - 6;
Tab = 4 - 6;
```

Bar 被赋于十进制值 62（－2 的二进制补码），而 Tab 被赋于十进制值－2（位向量为 111110）。

下面为另一个实例：

Bar = -2 + (-4);
Tab = -2 + (-4);
Bar 被赋于十进制值 58（位向量为 111010），而 Tab 被赋于十进制值－6（位向量为 111010）。

4.2.2 关系操作符

关系操作符有：

* >（大于）
* <（小于）
* >=（不小于）
* <=（不大于）

关系操作符的结果为真（1）或假（0）。如果操作数中有一位为 X 或 Z，那么结果为 X。例如：

23 > 45
结果为假（0），而：
52< 8'hxFF
结果为 x。如果操作数长度不同，长度较短的操作数在最重要的位方向（左方）添 0 补齐。例如：
'b1000 > = 'b01110
等价于：
'b01000 > = 'b01110
结果为假（0）。

4.2.3 相等关系操作符

相等关系操作符有:

* = =（逻辑相等）
* !=（逻辑不等）
* = = =（全等）
* != =（非全等）

如果比较结果为假，则结果为 0；否则结果为 1。在全等比较中，值 x 和 z 严格按位比较。也就是说，不进行解释，并且结果一定可知。而在逻辑比较中，值 x 和 z 具有通常的意义，且结果可以不为 x。也就是说，在逻辑比较中，如果两个操作数之一包含 x 或 z，结果为未知的值（x）。
如下例，假定：

Data = 'b11x0;

Addr = 'b11x0;
那么：
Data = = Addr
不定，也就是说值为 x，但：
Data = = = Addr
为真，也就是说值为 1。

如果操作数的长度不相等，长度较小的操作数在左侧添 0 补位，例如：

2'b10 = = 4'b0010
与下面的表达式相同：
4'b0010 = = 4'b0010
结果为真（1）。

4.2.4 逻辑操作符

逻辑操作符有:

* && (逻辑与)
* || (逻辑或)
* ！(逻辑非)

这些操作符在逻辑值 0 或 1 上操作。逻辑操作的结构为 0 或 1。例如，假定:

Crd = 'b0; //0 为假
Dgs = 'b1; //1 为真
那么:
Crd && Dgs 结果为 0 (假)
Crd || Dgs 结果为 1 (真)
！ Dgs 结果为 0 (假)
对于向量操作，非 0 向量作为 1 处理。例如，假定:
A_Bus = 'b0110;
B_Bus = 'b0100;
那么:
A_Bus || B_Bus 结果为 1
A_Bus && B_Bus 结果为 1
并且:
! A_Bus 与! B_Bus 的结果相同。
结果为 0。
如果任意一个操作数包含 x，结果也为 x。
!x 结果为 x

4.2.5 按位操作符

按位操作符有:

* ~（一元非）
* &（二元与）
* |（二元或）
* ^（二元异或）
* ~^, ^~（二元异或非）

这些操作符在输入操作数的对应位上按位操作，并产生向量结果。下表显示对于不同操作符按步操作的结果。

例如，假定,
A = 'b0110;
B = 'b0100;
那么:
A | B 结果为 0110
A & B 结果为 0100
如果操作数长度不相等，长度较小的操作数在最左侧添 0 补位。例如,
'b0110 ^ 'b10000
与如下式的操作相同:
'b00110 ^ 'b10000
结果为'b10110。

4.2.6 归约操作符

归约操作符在单一操作数的所有位上操作，并产生 1 位结果。归约操作符有:

* & (归约与)
如果存在位值为 0，那么结果为 0；若如果存在位值为 x 或 z，结果为 x；否则结果为 1。
* ~& (归约与非)
与归约操作符&相反。
* | (归约或)
如果存在位值为 1，那么结果为 1；如果存在位 x 或 z，结果为 x；否则结果为 0。
* ~| (归约或非)
与归约操作符|相反。
* ^ (归约异或)
如果存在位值为 x 或 z，那么结果为 x；否则如果操作数中有偶数个 1，结果为 0；否则结果为 1。
* ~^ (归约异或非)
与归约操作符^正好相反。
如下所示。假定,
A = 'b0110;
B = 'b0100;
那么:
|B 结果为 1
& B 结果为 0

~ A 结果为 1
归约异或操作符用于决定向量中是否有位为 x。假定,
MyReg = 4'b01x0；
那么:
^MyReg 结果为 x
上述功能使用如下的 if 语句检测:

```
if (^MyReg = = = 1'bx)
$ display ("There is an unknown in the vector MyReg !")
```

注意逻辑相等(==)操作符不能用于比较；逻辑相等操作符比较将只会产生结果 x。全等操作符期望的结果为值 1。

4.2.7 移位操作符

移位操作符有:

* << (左移)
* >> (右移)

移位操作符左侧操作数移动右侧操作数表示的次数，它是一个逻辑移位。空闲位添 0 补位。如果右侧操作数的值为 x 或 z, 移位操作的结果为 x。假定:

```
reg [0：7] Qreg；
. . .
Qreg = 4'b0111;
```

那么:
Qreg >> 2 是 8'b0000_0001

Verilog HDL 中没有指数操作符。但是，移位操作符可用于支持部分指数操作。例如，如果要计算 ZNumBits 的值，可以使用移位操作实现，例如:

```
32'b1 << NumBits //NumBits 必须小于 32。
```

同理，可使用移位操作为 2-4 解码器建模，如

```
wire [0:3] DecodeOut = 4'b1 << Address [0:1];
```

Address[0:1] 可取值 0,1,2 和 3。与之相应，DecodeOut 可以取值 4'b0001、4'b0010、4'b0100 和 4'b1000，从而为解码器建模。

4.2.8 条件操作符

条件操作符根据条件表达式的值选择表达式，形式如下:

```
cond_expr ? expr1 : expr2
```

如果 cond_expr 为真(即值为 1)，选择 expr1；如果 cond_expr 为假(值为 0)，选择 expr2。

如果 cond_expr 为 x 或 z，结果将是按以下逻辑 expr1 和 expr2 按位操作的值: 0 与 0 得 0，1 与 1 得 1，其余情况为 x。
如下所示:

```
wire [0:2] Student = Marks > 18 ? Grade_A : Grade_C;
```

计算表达式 Marks > 18; 如果真, Grade_A 赋值为 Student; 如果 Marks < =18, Grade_C 赋值为 Student。下面为另一实例：

```
always
#5 Ctr = (Ctr != 25) ? (Ctr + 1) : 5;
```

过程赋值中的表达式表明如果 Ctr 不等于 25, 则加 1；否则如果 Ctr 值为 25 时, 将 Ctr 值重新置为 5。

4.2.9 连接和复制操作

连接操作是将小表达式合并形成大表达式的操作。形式如下:

```
{expr1, expr2, . . .， exprN}
```

实例如下所示:

```
wire [7:0] Dbus;
wire [11:0] Abus;

assign Dbus [7:4] = {Dbus [0], Dbus [1], Dbus[2], Dbus[3]};
//以反转的顺序将低端 4 位赋给高端 4 位。
assign Dbus = {Dbus [3:0], Dbus [7:4]};
//高 4 位与低 4 位交换。
```

由于非定长常数的长度未知, 不允许连接非定长常数。例如, 下列式子非法：

```
{Dbus,5} //不允许连接操作非定长常数。
```

复制通过指定重复次数来执行操作。形式如下:

```
{repetition_number {expr1, expr2, ...,exprN}}
```

以下是一些实例：

```
Abus = {3{4'b1011}}; //位向量 12'b1011_1011_1011）
Abus = {{4{Dbus[7]}}, Dbus}; /*符号扩展*/
{3{1'b1}} 结果为 111
{3{Ack}} 结果与{Ack, Ack, Ack}相同。
```

4.3 表达式种类

常量表达式是在编译时就计算出常数值的表达式。通常，常量表达式可由下列要素构成:

1) 表示常量文字, 如'b10 和 326。
2) 参数名，如 RED 的参数表明:

parameter RED = 4'b1110；

标量表达式是计算结果为 1 位的表达式。如果希望产生标量结果，但是表达式产生的结果为向量，则最终结果为向量最右侧的位值。

习题

1. 说明参数 GATE_DELAY，参数值为 5。
2. 假定长度为 64 个字的存储器，每个字 8 位，编写 Verilog 代码，按逆序交换存储器的内容。即将第 0 个字与第 63 个字交换，第 1 个字与第 62 个字交换，依此类推。
3. 假定 32 位总线 Address_Bus，编写一个表达式，计算从第 11 位到第 20 位的归约与非。
4. 假定一条总线 Control_Bus [15:0]，编写赋值语句将总线分为两条总线：Abus [0:9] 和 Bbus [6:1]。
5. 编写一个表达式，执行算术移位，将 Qparity 中包含的 8 位有符号数算术移位。
6. 使用条件操作符，编写赋值语句选择 NextState 的值。如果 CurrentState 的值为 RESET，那么 NextState 的值为 GO；如果 CurrentState 的值为 GO，则 NextState 的值为 BUSY；如果 CurrentState 的值为 BUSY；则 NextState 的值为 RESET。
7. 如何从标量变量 A，B，C 和 D 中产生总线 BusQ[0:3]？如何从两条总线 BusA [0:3] 和 BusY [20:15]形成新的总线 BusR[10:1]?

第 5 章 门电平模型化

本章讲述 Verilog HDL 为门级电路建模的能力，包括可以使用的内置基本门和如何使用它们来进行硬件描述。

5.1 内置基本门

Verilog HDL 中提供下列内置基本门：

1) 多输入门：

and, nand,or, nor,xor,xnor

2) 多输出门：

buf, not

3) 三态门：

bufif0, bufif1, notif0,notif1

4) 上拉、下拉电阻：

pullup, pulldown

5) MOS 开关：

cmos, nmos, pmos, rcmos, rnmos, rpmos

6) 双向开关：

tran,tranif0, tranif1, rtran, rtranif0, rtranif1

门级逻辑设计描述中可使用具体的门实例语句。下面是简单的门实例语句的格式。

gate_type[instance_name] (term1, term2, . . . ,termN);

注意，instance_name 是可选的；gate_type 为前面列出的某种门类型。各 term 用于表示与门的输入/输出端口相连的线网或寄存器。

同一门类型的多个实例能够在一个结构形式中定义。语法如下:

```
gate_type
[instance_name1] (term11, term12, . . .,term1N),
[instance_name2] (term21, term22, . . .,term2N),
. . .
[instance_nameM] (termM1, termM2, . . .,termMN);
```

5.2 多输入门

内置的多输入门如下:

```
and nand nor or xor xnor
```

这些逻辑门只有单个输出，1 个或多个输入。多输入门实例语句的语法如下:

```
multiple_input_gate_type
[instance_name] (OutputA, Input1, Input2, . . .,InputN);
```

第一个端口是输出，其它端口是输入。

下面是几个具体实例。

```
and A1(Out1, In1, In2);

and RBX (Sty, Rib, Bro, Qit, Fix);

xor (Bar, Bud[0],Bud[1], Bud[2]),
(Car, Cut[0], Cut[1]),
(Sar, Sut[2], Sut[1], Sut[0], Sut[3]);
```

第一个门实例语句是单元名为 A1、输出为 Out1、并带有两个输入 In1 和 In2 的两输入与门。第二个门实例语句是四输入与门，单元名为 RBX，输出为 Sty，4 个输入为 Rib、Bro、Qit 和 Fix。第三个门实例语句是异或门的具体实例，没有单元名。它的输出是 Bar，三个输入分别为 Bud[0]、Bud[1]和 Bud[2]。同时，这一个实例语句中还有两个相同类型的单元。

5.3 多输出门

多输出门有:

```
buf not
```

这些门都只有单个输入，一个或多个输出。这些门的实例语句的基本语法如下:

```
multiple_output_gate_type
[instance_name] (Out1, Out2, . . . OutN ,InputA);
```

最后的端口是输入端口，其余的所有端口为输出端口。

例如:

buf B1 （Fan [0]，Fan [1]，Fan [2]，Fan [3]，Clk）;
not N1 （PhA，PhB，Ready）;

在第一个门实例语句中，Clk 是缓冲门的输入。门 B1 有 4 个输出：Fan[0]到 Fan[3]。在第二个门实例语句中，Ready 是非门的唯一输入端口。门 N1 有两个输出：PhA 和 PhB。

5.4 三态门

三态门有:

bufif0 bufif1 notif0 notif1

这些门用于对三态驱动器建模。这些门有一个输出、一个数据输入和一个控制输入。三态门实例语句的基本语法如下：

tristate_gate[instance_name] (OutputA, InputB,ControlC);

第一个端口 OutputA 是输出端口，第二个端口 InputB 是数据输入，ControlC 是控制输入。根据控制输入，输出可被驱动到高阻状态，即值 z。对于 bufif0，若通过控制输入为 1，则输出为 z；否则数据被传输至输出端。对于 bufif1，若控制输入为 0，则输出为 z。对于 notif0，如果控制输出为 1，那么输出为 z；否则输入数据值的非传输到输出端。对于 notif1，若控制输入为 0；则输出为 z。

例如:

bufif1 BF1 （Dbus，MemData，Strobe）；
notif0 NT2 (Addr, Abus, Probe);
当 Strobe 为 0 时，bufif1 门 BF1 驱动输出 Dbus 为高阻 ;否则 MemData 被传输至 Dbus。在第 2 个实例语句中，当 Probe 为 1 时，Addr 为高阻；否则 Abus 的非传输到 Addr。

5.5 上拉、下拉电阻

上拉、下拉电阻有：

pullup pulldown

这类门设备没有输入只有输出。上拉电阻将输出置为 1。下拉电阻将输出置为 0。门实例语句形式如下：

pull_gate[instance_name] (OutputA);
门实例的端口表只包含 1 个输出。例如：
pullup PUP (Pwr);

此上拉电阻实例名为 PUP，输出 Pwr 置为高电平 1。

5.6 MOS 开关

MOS 开关有：

cmos pmos nmos rcmos rpmos rnmos

这类门用来为单向开关建模。即数据从输入流向输出，并且可以通过设置合适的控制输入关闭数据流。
pmos(p 类型 MOS 管)、nmos(n 类型 MOS 管)，rnmos(r 代表电阻)和 rpmos 开关有一个输出、一个输入和一个控制输入。实例的基本语法如下：

gate_type[instance_name] (OutputA, InputB, ControlC);
第一个端口为输出，第二个端口是输入，第三个端口是控制输入端。如果 nmos 和 rnmos 开关的控制输入为 0，pmos 和 rpmos 开关的控制为 1，那么开关关闭，即输出为 z；如果控制是 1，输入数据传输至输出；如图 5-5 所示。与 nmos 和 pmos 相比，rnmos 和 rpmos 在输入引线和输出引线之间存在高阻抗(电阻)。因此当数据从输入传输至输出时，对于 rpmos 和 rmos，存在数据信号强度衰减。信号强度将在第 10 章进行讲解。
例如：

pmos P1 (BigBus, SmallBus, GateControl);
rnmos RN1 (ControlBit, ReadyBit, Hold);

第一个实例为一个实例名为 P1 的 pmos 开关。开关的输入为 SmallBus，输出为 BigBus，控制信号为 GateControl。
这两个开关实例语句的语法形式如下：

(r)cmos [instance_name]
(OutputA, InputB, Ncontrol, PControl);
第一个端口为输出端口，第二个端口为输入端口，第三个端口为 n 通道控制输入，第四个端口为是 P 通道控制输入。cmos(rcmos)开关行为与带有公共输入、输出的 pmos（rpmos）和 nmos(rnmos)开关组合十分相似。

5.7 双向开关

双向开关有：

tran rtran tranif0 rtranif0 tranif1 rtranif1

这些开关是双向的，即数据可以双向流动，并且当数据在开关中传播时没有延时。后 4 个开关能够通过设置合适的控制信号来关闭。tran 和 rtran 开关不能被关闭。
tran 或 rtran(tran 的高阻态版本)开关实例语句的语法如下：

(r)tran [instance_name] (SignalA, SignalB);

端口表只有两个端口，并且无条件地双向流动，即从 SignalA 向 SignalB，反之亦然。

其它双向开关的实例语句的语法如下：

gate_type[instance_name] (SignalA, SignalB, ControlC);

前两个端口是双向端口，即数据从 SignalA 流向 SignalB，反之亦然。第三个端口是控制信号。如果对 tranif0 和 tranif0，ControlC 是 1；对 tranif1 和 rtranif1，Controlc 是 0；那么禁止双向数据流动。对于 rtran、rtranif0 和 rtranif1，当信号通过开关传输时，信号强度减弱。

5.8 门时延

可以使用门时延定义门从任何输入到其输出的信号传输时延。门时延可以在门自身实例语句中定义。带有时延定义的门实例语句的语法如下：

gate_type [delay][instance_name](terminal_list);

时延规定了门时延，即从门的任意输入到输出的传输时延。当没有强调门时延时，缺省的时延值为 0。

门时延由三类时延值组成：

1) 上升时延
2) 下降时延
3) 截止时延

门时延定义可以包含 0 个、1 个、2 个或 3 个时延值。下表为不同个数时延值说明条件下，各种具体的时延取值情形。

无时延 1 个时延(d) 2 个时延(d1, d2) 3 个时延 (dA, dB, dC)
上升 0 d d1 dA
下降 0 d d2 dB
to_x 0 d min① (d1, d2) min (dA, dB, dC)
截止 0 d min (d1, d2) dC
① min 是 minimum 的缩写词。

注意转换到 x 的时延(to_x)不但被显式地定义，还可以通过其它定义的值决定。

下面是一些具体实例。注意 Verilog HDL 模型中的所有时延都以单位时间表示。单位时间与实际时间的关联可以通过`timescale 编译器指令实现。在下面的实例中,

not N1 (Qbar, Q);

因为没有定义时延，门时延为 0。下面的门实例中，

nand #6 (Out, In1, In2);

所有时延均为 6，即上升时延和下降时延都是 6。因为输出决不会是高阻态，截止时延不适用于与非门。转换到 x 的时延也是 6。

and #(3,5) (Out, In1, In2, In3);

在这个实例中，上升时延被定义为 3，下降时延为 5，转换到 x 的时延是 3 和 5 中间的最小值，即 3。在下面的实例中，

notif1 #(2,8,6) (Dout, Din1, Din2);

上升时延为 2，下降时延为 8，截止时延为 6，转换到 x 的时延是 2、8 和 6 中的最小值，即 2。

对多输入门（例如与门和非门）和多输出门(缓冲门和非门)总共只能够定义 2 个时延(因为输出决不会是 z)。三态门共有 3 个时延，并且上拉、下拉电阻实例门不能有任何时延。

min:typ:max 时延形式

门延迟也可采用 min:typ:max 形式定义。形式如下：

minimum: typical: maximum

最小值、典型值和最大值必须是常数表达式。下面是在实例中使用这种形式的实例。

nand #(2:3:4, 5:6:7) (Pout, Pin1, Pin2);

选择使用哪种时延通常作为模拟运行中的一个选项。例如，如果执行最大时延模拟，与非门单元使用上升时延 4 和下降时延 7。

程序块也能够定义门时延。

5.9 实例数组

当需要重复性的实例时，在实例描述语句中能够有选择地定义范围说明(范围说明也能够在模块实例语句中使用)。这种情况的门描述语句的语法如下：

gate_type [delay]instance_name [leftbound:rightbound]
(list_of_terminal_names);
leftbound 和 rightbound 值是任意的两个常量表达式。左界不必大于右界，并且左、右界两者都不必限定为 0。示例如下。
wire [3:0] Out, InA, InB;
. . .
nand Gang [3:0] (Out, InA, InB);
带有范围说明的实例语句与下述语句等价：
nand
Gang3 (Out[3], InA[3], InB[3]),
Gang2 (Out[2], InA[2], InB[2]),

```
Gang1 (Out[1], InA[1], InB[1]),
Gang0 (Out[0], InA[0], InB[0]);
```

注意定义实例数组时，实例名称是不可选的。

5.10 隐式线网

如果在 Verilog HDL 模型中一个线网没有被特别说明，那么它被缺省声明为 1 位线网。但是

`default_nettype 编译指令能够用于取代缺省线网类型。编译指令格式如下：

```
`default_nettype net_type
```

例如：

```
`default_nettype wand
```

根据此编译指令，所有后续未说明的线网都是 wand 类型。

`default_nettype 编译指令在模块定义外出现，并且在下一个相同编译指令或`resetall 编译指令出现前一直有效。

5.11 简单示例

下面是 4-1 多路选择电路的门级描述。注意因为实例名是可选的(除用于实例数组情况外)，在门实例语句中没有指定实例名。

```
module MUX4x1 (Z,D0,D1,D2,D3,S0,S1);
output Z;
input D0,D1,D2,D3,S0,S1;

and (T0,D0,S0bar,S1bar),
(T1,D1,S0bar,S1),
(T2,D2,S0,S1bar),
(T3,D3,S0,S1),

not (S0bar,S0),
(S1bar,S1);

or (Z,T0,T1,T2,T3,);
endmodule
```

如果或门实例由下列的实例代替呢?

```
or Z (Z,T0,T1,T2,T3); //非法的 Verilog HDL 表达式。
```

注意实例名还是 Z，并且连接到实例输出的线网也是 Z。这种情况在 Verilog HDL 中是不允许的。在同一模块中，实例名不能与线网名相同。

5.12 2-4 解码器举例

2-4 解码器电路的门级描述如下：

```
module DEC2×4 (A,B,Enable,Z);
input A,B,Enable;
output [0:3] Z;
wire Abar, Bbar;

not # (1,2)
V0 (Abar,A),
V1(Bbar, B);

nand # (4,3)
N0 (Z[3], Enable, A,B),
N1 (Z[0], Enable, Abar,Bbar),
N2 (Z[1], Enable, Abar,B),
N3 (Z[2], Enable, A,Bbar),
endmodule
```

5.13 主从触发器举例

主从 D 触发器的门级描述如下：

```
module MSDFF (D,C,Q,Qbar);
input D,C;
output Q,Qbar;

not
NT1 (NotD,D),
NT2 (NotC,C),
NT3 (NotY,Y);

nand
ND1 (D1,D,C),
ND2 (D2,C,NotD),
ND3 (Y,D1,Ybar),
ND4 (Ybar,Y,D2),
ND5 (Y1,Y,NotC),
ND6 (Y2,NotY,NotC),
ND7 (Q,Qbar,Y1),
ND8 (Qbar,Y2,Q);
endmodule
```

5.14 奇偶电路

9 位奇偶发生器门级模型描述如下：

```
module Parity_9_Bit (D, Even,Odd);
```

```
input [0:8] D;
output Even, Odd;
xor # (5,4)
XE0 (E0,D[0],D[1]),
XE1 (E1,D[2],D[3]),
XE2 (E2,D[4],D[5]),
XE3 (E3,D[6],D[7]),
XF0 (F0,E0,E1),
XF1 (F1,E2,E3),
XH0 (H0,F0,F1),
XEVEN (Even, D[8], H0);
not #2
XODD (Odd, Even);
endmodule
```

习题

1. 用基本门描述图 5-11 显示的电路模型。编写一个测试验证程序用于测试电路的输出。使用所有可能的输入值对电路进行测试。
2. 使用基本门描述如图 5-12 所示的优先编码器电路模型。当所有输入为 0 时，输出 Valid 为 0，否则输出为 1。并且为验证优先编码器的模型行为编写测试验证程序。