第1章 简 介

本章介绍Verilog HDL语言的发展历史和它的主要能力。

1.1 什么是Verilog HDL？

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。

1.2 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。

1.3 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。

⊖ Gateway Design Automation公司后来被Cadence Design Systems公司收购。

- 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。
- 图1-1显示了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。

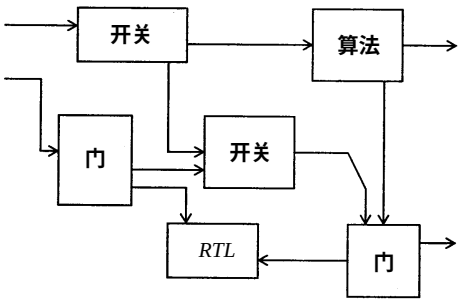


图1-1 混合设计层次建模

- 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。

图2-1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。

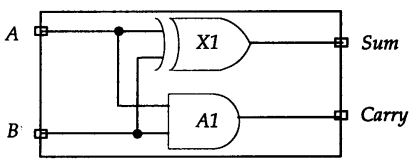


图2-1　半加器电路

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。

图2-2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。

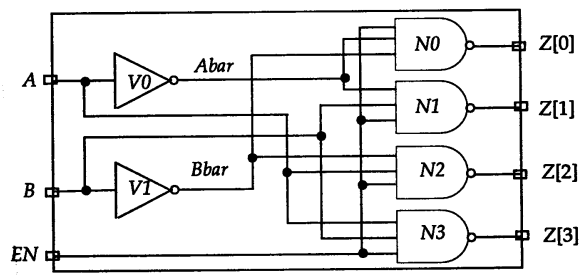


图2-2 2-4解码器电路

```
`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。

参见图2-3中的波形图。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。

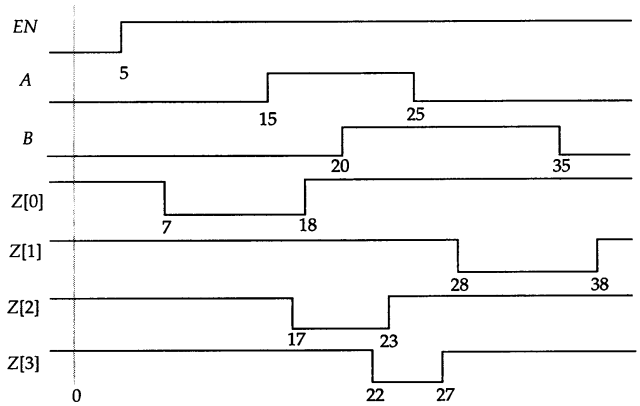


图2-3 连续赋值语句实例

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，如图2-4。

```
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。

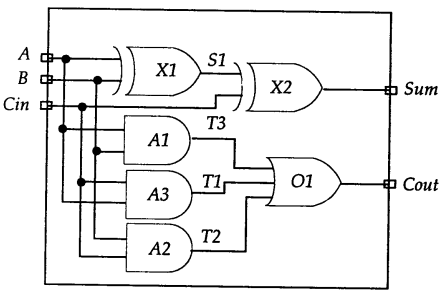


图2-4 1位全加器电路

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
```

这一模块产生如图2-5所示的波形。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。第 8 章将更详细地讲解initial语句。

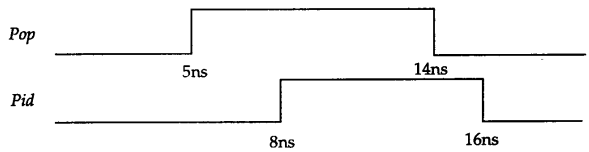


图2-5 *Test* 模块的输出波形

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。该实例基于图2-4所示的逻辑图。

```
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，描述的逻辑图如图 2-6所示。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。

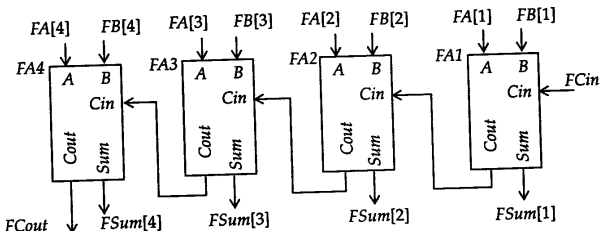


图2-6 4位全加器

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。测试模块产生的波形如图2-7显示。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
```

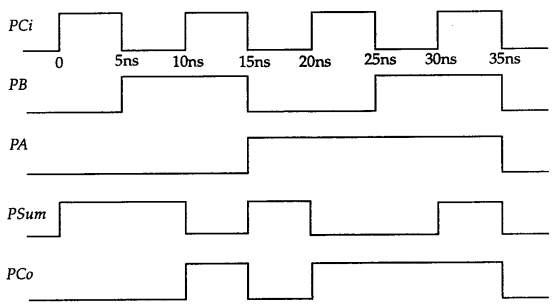


图2-7 测试模块*Top*执行产生的波形

验证与非门交叉连接构成的*RS_FF*模块的测试模块如图2-8所示。

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

  nand #1 (Q, R, Qbar);
```

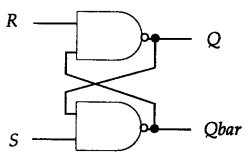


图2-8 交叉连接的与非门

```
    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。产生的相应波形如图 2-9所示。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。

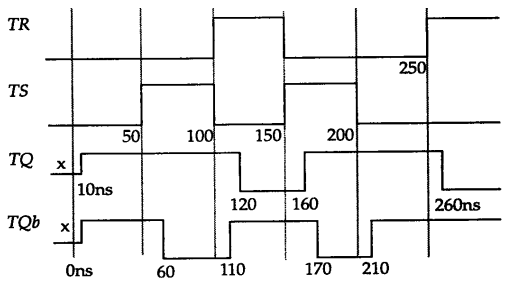


图2-9 Test模块产生的波形

```
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. 采用数据流描述方式描述图2-4中所示的1位全加器。
5. initial语句与always 语句的关键区别是什么？
6. 写出产生图2-10所示波形的变量*BullsEye*的初始化语句。

图2-10 变量BullsEye 的波形

7. 采用结构描述方式描写图2-2中所示的2-4译码器。
8. 为2.3节中描述的模块*Decode2x4*编写一个测试验证程序。
9. 列出你在Verilog HDL模型中使用的两类赋值语句。
10. 在顺序过程中何时需要定义标记？
11. 使用数据流描述方式编写图 2-11所示的异或逻辑的Verilog HDL描述，并使用规定的时延。
12. 找出下面连续赋值语句的错误。

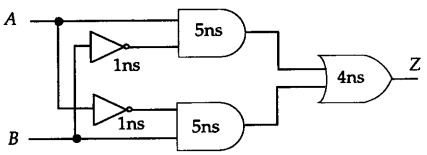


图2-11 异或逻辑

```
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 GetO;

  initial
    begin
      PutA = 0;
      PutB = 0;
      #5.21 PutB = 1;
      #10.4 PutA = 1;
      #15 PutB = 0;
    end
  AndFunc AF1(GetO, 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, Q) ;
  . . .
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          位八进制数
4'D2           位十进制数
4'B1x_01       位二进制数
7'Hx           位x(扩展的x)，即xxxxxxx
4'hZ           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'b001         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          位八进制数
'hAF           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) **$readmemh （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:1sb];
```

*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：

⊖ 因为Cone 为非定长整型变量，基数表示形式的负数在机内以补码形式出现。—— 译者注

```
net_or_reg_vector [msb_const_expr:lsb_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。*/
```

第10章将详细介绍函数。

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。

⊖ 归约操作符为一元操作符，对操作数的各位进行逻辑操作，结果为二进制数。 —— 译者

+	一元加	>>	右移
-	一元减	<	小于
!	一元逻辑非	<=	小于等于
~	一元按位求反	>	大于
&	归约与	>=	大于等于
~&	归约与非	==	逻辑相等
^	归约异或	!=	逻辑不等
^~ 或 ~^	归约异或非	===	全等
|	归约或	!==	非全等
~|	归约或非	&	按位与
*	乘	^	按位异或
/	除	^~ or ~^	按位异或非
%	取模	|	按位或
+	二元加	&&	逻辑与
_	二元减	||	逻辑或
<<	左移	?:	条件操作符

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; //为假
Dgs = 'b1; //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。

& 与	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

| 或	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

^ 异或	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

^~异或非	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

~ 非	0	1	x	z
	1	0	x	x

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$Z^{NumBits}$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. 使用单一连续赋值语句为图2-2所示的2-4解码器电路的行为建模。[提示：使用移位操作符、条件操作符和连接操作符。]
8. 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。如图5-1所示。

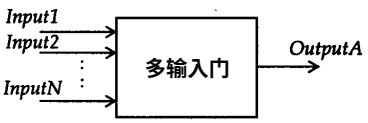


图5-1　多输入门

下面是几个具体实例。图5-2为对应的逻辑图。

```
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]);
```

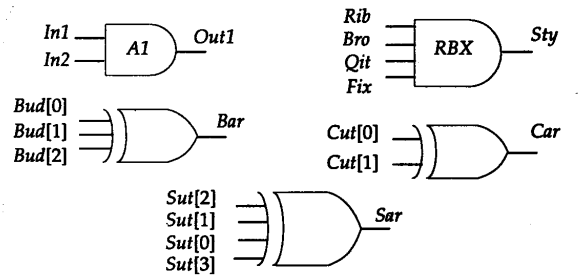


图5-2 多输入门实例

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。

下面是这些门的真值表。注意在输入端的 z与对x的处理方式相同；多输入门的输出决不能是z。

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

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。如图 5-3所示。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。

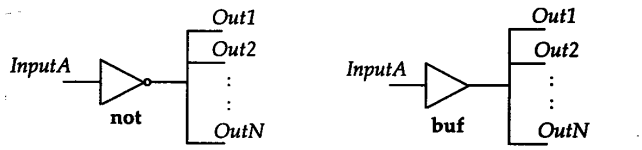


图5-3 多输出门

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*。

这些门的真值表如下：

buf	0	1	x	z
(输出)	0	1	x	x

not	0	1	x	z
(输出)	1	0	x	x

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,Control);
```

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。参见图5-4。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*。

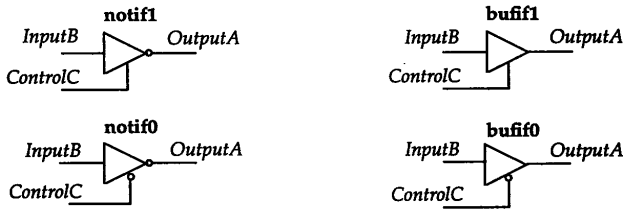


图5-4 三态门

下面是这些门的真值表。表中的某些项是可选项。例如， 0/z表明输出根据数据的信号强度和控制值既可以为0也可以为z，信号强度在第10章中讨论。

bufif0		控制			
		0	1	x	z
数据	0	0	z	0/z	0/z
	1	1	z	1/z	1/z
	x	x	z	x	x
	z	x	z	x	x

bufif1		控制			
		0	1	x	z
数据	0	z	0	0/z	0/z
	1	z	1	1/z	1/z
	x	z	x	x	x
	z	z	x	x	x

notif0		控制			
		0	1	x	z
数据	0	1	z	1/z	1/z
	1	0	z	0/z	0/z
	x	x	z	x	x
	z	x	z	x	x

notif1		控制			
		0	1	x	z
数据	0	z	1	1/z	1/z
	1	z	0	0/z	0/z
	x	z	x	x	x
	z	z	x	x	x

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, Control);
```

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。

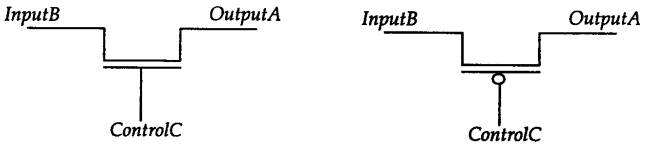


图5-5 nmos和pmos开关

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*。

这些开关的真值表如下所示。表中的某些项是可选项。例如， 1/z表明，根据输入和控制信号的强度，输出既可以为1，也可以为z。

pmos rpmos		控制			
		0	1	x	z
数据	0	0	z	0/z	0/z
	1	1	z	1/z	1/z
	x	x	z	x	x
	z	z	z	z	z

nmos rnmos		控制			
		0	1	x	z
数据	0	z	0	0/z	0/z
	1	z	1	1/z	1/z
	x	z	x	x	x
	z	z	z	z	z

cmos(mos求补)和rcmos(cmos的高阻态版本)开关有一个数据输出，一个数据输入和两个控制输入。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。参见图5-6。

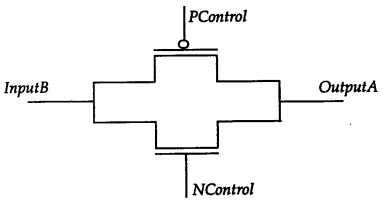


图5-6 (r)cmos开关

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, Control);
```

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。程序块的定义和说明在第10章中讨论。

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

下面是图5-7中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
```

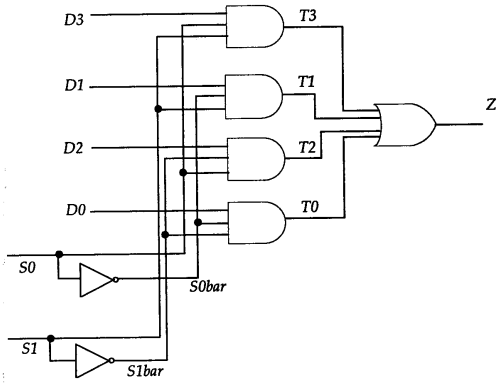


图5-7 4-1多路选择电路

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 HD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

图5-8中显示的2-4解码器电路的门级描述如下：

```
module DEC2X4 (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
```

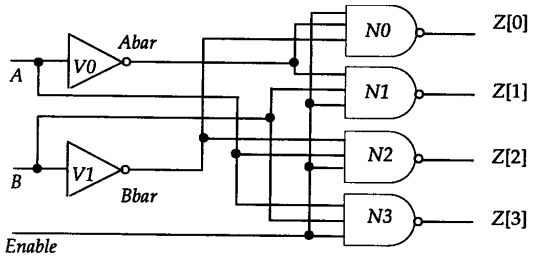


图5-8 2-4解码器电路

5.13 主从触发器举例

图5-9所示的主从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
```

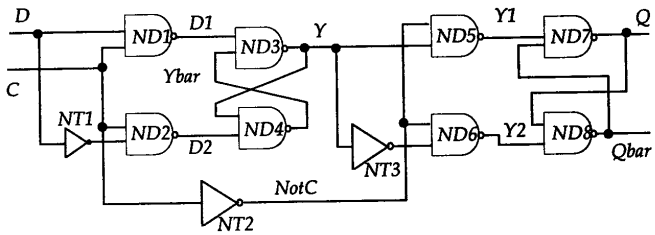


图5-9 主从触发器

5.14 奇偶电路

图5-10所示的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
```

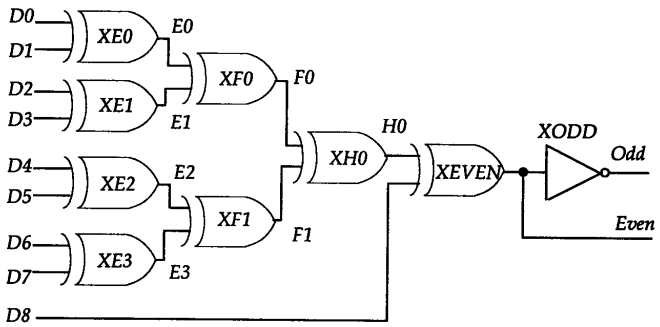


图5-10 奇偶发生器

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。

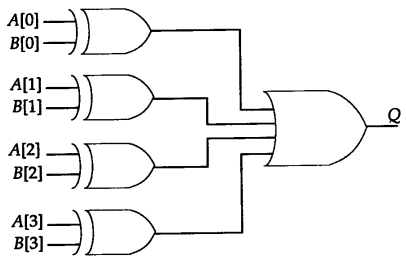


图5-11 A不等于B的逻辑

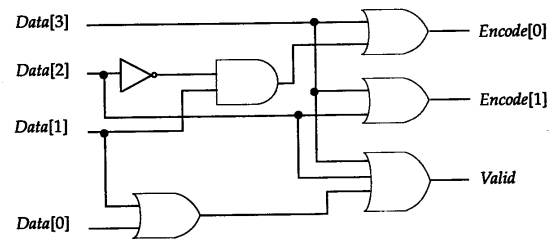


图5-12 优先编码器

第6章　用户定义的原语

在前一章中，我们介绍了Verilog HDL提供的内置基本门。本章讲述Verilog HDL指定用户定义原语UDP的能力。

UDP的实例语句与基本门的实例语句完全相同，即 UDP实例语句的语法与基本门的实例语句语法一致。

6.1 UDP的定义

使用具有如下语法的UDP说明定义UDP。

```
primitive UDP_name (OutputName, List_of_inputs);
  Output_declaration
  List_of_input_declarations
  [Reg_declaration]
  [Initial_statement]
  table
    List_of_tabel_entries
  endtable
endprimitive
```

UDP的定义不依赖于模块定义，因此出现在模块定义以外。也可以在单独的文本文件中定义UDP。

UDP只能有一个输出和一个或多个输入。第一个端口必须是输出端口。此外，输出可以取值0、1或x(不允许取z值)。输入中出现值z以x处理。UDP的行为以表的形式描述。

在UDP中可以描述下面两类行为：

1) 组合电路

2) 时序电路(边沿触发和电平触发)

6.2 组合电路UDP

在组合电路UDP中，表规定了不同的输入组合和相对应的输出值。没有指定的任意组合输出为x。下面以2-1多路选择器为例加以说明。

```
primitive MUX2x1 (Z, Hab, Bay, Sel);
  output Z;
  input Hab,Bay, Sel;

  table
    // Hab Bay Sel  : Z 注：本行仅作为注释。
     0    ?   1 : 0 ;
     1    ?   1 : 1 ;
     ?    0   0 : 0 ;
     ?    1   0 : 1 ;
     0    0   x : 0 ;
```

```
        1    1   x : 1 ;
    endtable
  endprimitive
```

字符?代表不必关心相应变量的具体值，即它可以是0、1或x。输入端口的次序必须与表中各项的次序匹配，即表中的第一列对应于原语端口队列的第一个输入(例子中为 *Hab*)，第二列是 *Bay*，第三列是 *Sel*。在多路选择器的表中没有输入组合 01x项(还有其它一些项)；在这种情况下，输出的缺省值为 x(对其它未定义的项也是如此)。

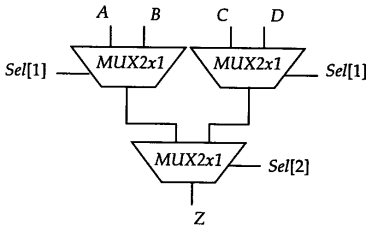


图6-1 使用UDP构造的4-1多路选择器

图6-1为使用2-1多路选择器原语组成的4-1多路选择器的示例。

```
module MUX4x1 (Z, A, B, C, D, Sel);
  input A, B, C, D;
  input [2:1] Sel;
  output Z;
  parameter tRISE = 2, tFALL = 3;

  MUX2x1 #(tRISE, tFALL)
     (TL, A, B, Sel[1]),
     (TP, C, D, Sel[1]),
     (Z, TL, TP, Sel[2]);
endmodule
```

如上例所示，在UDP实例中，总共可以指定2个时延，这是由于UDP的输出可以取值0、1或x(无截止时延)。

6.3 时序电路UDP

在时序电路UDP中，使用1位寄存器描述内部状态。该寄存器的值是时序电路UDP的输出值。共有两种不同类型的时序电路 UDP：一种模拟电平触发行为；另一种模拟边沿触发行为。时序电路UDP使用寄存器当前值和输入值决定寄存器的下一状态 (和后继的输出)。

6.3.1 初始化状态寄存器

时序电路UDP的状态初始化可以使用带有一条过程赋值语句的初始化语句实现。形式如下：

```
initial reg_name = 0,1,or x;
```

初始化语句在UDP定义中出现。

6.3.2 电平触发的时序电路UDP

下面是D锁存器建模的电平触发的时序电路 UDP示例。只要时钟为低电平 0，数据就从输入传递到输出；否则输出值被锁存。

```
primitive Latch (Q, Clk, D);
  output Q;
  reg Q;
  input Clk, D;

  table
```

```
    // Clk   D   Q(State) Q(next)
       0    1     :?     :     1 ;
       0    0     :?     :     0 ;
       1    ?     :?     :     - ;
  endtable
endprimitive
```

“-”字符表示值“无变化”。注意UDP的状态存储在寄存器D中。

6.3.3 边沿触发的时序电路UDP

下例用边沿触发时序电路UDP为D边沿触发触发器建模。初始化语句用于初始化触发器的状态。

```
primitive D_Edge_FF (Q, Clk, Data);
  output Q;
  reg Q;
  input Data, Clk;

  initial Q = 0;
  table
    // Clk    Data    (State)  Q(next)
       (01)  0  :  ?  :  0 ;
       (01)  1  :  ?  :  1 ;
       (0x)  1  :  1  :  1 ;
       (0x)  0  :  0  :  0 ;
       // 忽略时钟负边沿：
       (?0)  ?  :  ?  :  - ;
       // 忽略在稳定时钟上的数据变化：
       ?     (??): ? :  - ;
  endtable
endprimitive
```

表项(01)表示从0转换到1，表项(0x)表示从0转换到x，表项(?0)表示从任意值(0,1或x)转换到0，表项(??)表示任意转换。对任意未定义的转换，输出缺省为x。

假定*D_Edge_FF*为UDP定义，它现在就能够象基本门一样在模块中使用，如下面的4位寄存器所示。

```
module Reg4 (Clk, Din, Dout);
  input Clk;
  input [0:3] Din;
  output [0:3] Dout;

  D_Edge_FF
         DLAB0 (Dout[0],Clk, Din[0]),
         DLAB1 (Dout[1],Clk, Din[1]),
         DLAB2 (Dout[2],Clk, Din[2]),
         DLAB3 (Dout[3],Clk, Din[3]),
endmodule
```

6.3.4 边沿触发和电平触发的混合行为

在同一个表中能够混合电平触发和边沿触发项。在这种情况下，边沿变化在电平触发之

前处理，即电平触发项覆盖边沿触发项。

下例是带异步清空的D触发器的UDP描述。

```
primitive D_Async_FF(Q, Clk, Clr, Data);
  output Q;
  reg Q;
  input Clr, Data, Clk;

  table
    // Clk    Clr    Data   (State) Q(next)
       (01)   0   0   :   ?   :   0 ;
       (01)   0   1   :   ?   :   1 ;
       (0x)   0   1   :   1   :   1 ;
       (0x)   0   0   :   0   :   0 ;
       // 忽略时钟负边沿：
       (?0)   0   ?   :   ?   :   - ;
       (??)   1   ?   :   ?   :   0 ;
       ?      1   ?   :   ?   :   0;
  endtable
endprimitive
```

6.4 另一实例

下面是3位表决电路的UDP描述。如果输入向量中存在2个或更多的1，则输出为1。

```
primitive Majority3(Z, A, B, C);
  input A, B, C;

  output  Z
  table
    //A   B    C    :   Z
    0     0    ?    :   0 ;
    0     ?    0    :   0 ;
    ?     0    0    :   0 ;
    1     1    ?    :   1 ;
    1     ?    1    :   1 ;
    ?     1    1    :   1 ;
  endtable
endprimitive
```

6.5 表项汇总

出于完整性考虑，下表列出了所有能够用于 UDP原语中表项的可能值。

符　号	意　义	符　号	意　义
0	逻辑0	(AB)	由A变到B
1	逻辑1	*	与(??)相同
x	未知的值	r	上跳变沿，与(01)相同
?	0、1或x中的任一个	f	下跳变沿，与(10)相同
b	0或1中任选一个	p	(01)、(0x)和(x1)的任一种
–	输出保持	n	(10)、(1x)和(x0)的任一种

习题

1. 组合电路UDP与时序电路UDP如何区别?
2. UDP可有一个或多个输出，是否正确?
3. 初始语句可用于初始化组合电路UDP吗?
4. 为图5-12中显示的优先编码器电路编写UDP描述。使用测试激励验证描述的模型。
5. 为T触发器编写UDP描述。在T触发器中，如果数据输入为0，则输出不变化。如果数据输入是1，那么输出在每个时钟沿翻转。假定触发时钟沿是时钟下跳沿，使用测试激励验证所描述的模型。
6. 以UDP方式为上跳边沿触发的JK触发器建模。如果J和K两个输入均为0，则输出不变。如果J为0，K为1，则输出为0。如果J是1，K是0，则输出是1。如果J和K都是1，则输出翻转。使用测试激励验证描述的模型。

第7章　数据流模型化

本章讲述Verilog HDL语言中连续赋值的特征。连续赋值用于数据流行为建模；相反，过程赋值用于(下章的主题)顺序行为建模。组合逻辑电路的行为最好使用连续赋值语句建模。

7.1　连续赋值语句

连续赋值语句将值赋给线网(连续赋值不能为寄存器赋值)，它的格式如下(简单形式)：

```
assign LHS_target = RHS_expression;
```

例如，

```
wire [3:0] Z, Preset, Clear;    /线网说明
assign Z = Preset & Clear;      /连续赋值语句
```

连续赋值的目标为Z，表达式右端为“Preset & Clear”。注意连续赋值语句中的关键词assign。

连续赋值语句在什么时候执行呢？只要在右端表达式的操作数上有事件(事件为值的变化)发生时，表达式即被计算；如果结果值有变化，新结果就赋给左边的线网。

在上面的例子中，如果*Preset*或*Clear*变化，就计算右边的整个表达式。如果结果变化，那么结果即赋值到线网Z。

连续赋值的目标类型如下：

1) 标量线网

2). 向量线网

3) 向量的常数型位选择

4) 向量的常数型部分选择

5) 上述类型的任意的拼接运算结果

下面是连续赋值语句的另一些例子：

```
assign BusErr = Parity| (One & OP);
assign Z = ~ (A | B) & (C | D) & (E | F);
```

只要*A*、*B*、*C*、*D*、*E*或*F*的值变化，最后一个连续赋值语句就执行。在这种情况下，计算右边整个表达式，并将结果赋给目标Z。

在下一个例子中，目标是一个向量线网和一个标量线网的拼接结果。

```
wire Cout, Cin;
wire [3:0] Sum, A, B;
. . .
assign {Cout, Sum} = A + B + Cin;
```

因为*A*和*B*是4位宽，加操作的结果最大能够产生5位结果。左端表达式的长度指定为5位(*Cout* 1位，*Sum* 4位)。赋值语句因此促使右端表达式最右边的4位的结果赋给*Sum*，第5位(进位位)赋给*Cout*。

下例说明如何在一个连续赋值语句中编写多个赋值方式。

```
assign Mux = (S = = 0)? A : 'bz,
       Mux = (S = = 1)? B : 'bz,
       Mux = (S = = 2)? C : 'bz,
       Mux = (S = = 3)? D : 'bz;
```

这是下述4个独立的连续赋值语句的简化书写形式。

```
assign Mux = (S = = 0)? A : 'bz;
assign Mux = (S = = 1)? B : 'bz;
assign Mux = (S = = 2)? C : 'bz;
assign Mux = (S = = 3)? D : 'bz;
```

7.2 举例

下例采用数据流方式描述1位全加器。

```
module FA_Df (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;

  assign Sum = A ^B ^Cin;
  assign Cout = (A & Cin) | (B & Cin) | (A & B);
endmodule
```

在本例中，有两个连续赋值语句。这些赋值语句是并发的，与其书写的顺序无关。只要连续赋值语句右端表达式中操作数的值变化（即有事件发生），连续赋值语句即被执行。如果 A 变化，则两个连续赋值都被计算，即同时对右端表达式求值，并将结果赋给左端目标。

7.3 线网说明赋值

连续赋值可作为线网说明本身的一部分。这样的赋值被称为线网说明赋值。例如：

```
wire [3:0] Sum = 4'b0;
wire Clear = 'b1;
wire A_GT_B = A > B, B_GT_A= B > A;
```

线网说明赋值说明线网与连续赋值。说明线网然后编写连续赋值语句是一种方便的形式。参见下例。

```
wire Clear;
assign Clear = 'b1;
```

等价于线网声明赋值：

```
wire Clear = 'b1;
```

不允许在同一个线网上出现多个线网说明赋值。如果多个赋值是必需的，则必须使用连续赋值语句。

7.4 时延

如果在连续赋值语句中没有定义时延，如前面的例子，则右端表达式的值立即赋给左端表达式，时延为0。如下例所示显式定义连续赋值的时延。

```
assign #6 Ask = Quiet | | Late;
```

规定右边表达式结果的计算到其赋给左边目标需经过6个时间单位时延。例如，如果在时刻5，*Late*值发生变化，则赋值的右端表达式被计算，并且 *Ask*在时刻11(= 5 +6)被赋于新值。

图7-1举例说明了时延概念。

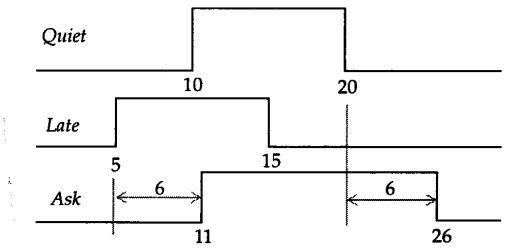


图7-1 连续赋值语句中的时延

如果右端在传输给左端之前变化，会发生什么呢？在这种情况下，应用最新的变化值。下例显示了这种行为：

```
assign #4 Cab = Drm;
```

图7-2显示了这种变化的效果。右端发生在时延间隔内的变化被滤掉。例如，在时刻 5，*Drm*的上升边沿预定在时刻9显示在*Cab*上，但是因为*Drm*在时刻8下降为0，预定在*Cab*上的值被删除。同样，*Drm*在时刻18和20之间的脉冲被滤掉。这也同样适用于惯性时延行为：即右端值变化在能够传播到左端前必须至少保持时延间隔；如果在时延间隔内右端值变化，则前面的值不能传输到输出。

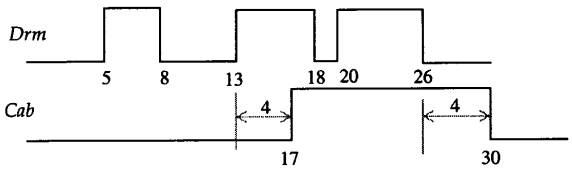


图7-2 值变化快于时延间隔

对于每个时延定义，总共能够指定三类时延值:

1) 上升时延

2) 下降时延

3) 关闭时延

这三类时延的语法如下:

```
assign # (rise, fall, turn-off) LHS_target = RHS_expression;
```

下面是当三类时延值定义为0时，如何解释时延的实例：

```
assign #4 Ask = Quiet || Late;      // One delay value.
assign # (4,8) Ask = Quick;         // Two delay values.
assign # (4,8,6) Arb = & DataBus;  // Three delay values.
assign Bus = MemAddr [7:4];         // No delay value.
```

在第一个赋值语句中，上升时延、下降时延、截止时延和传递到 x的时延相同，都为 4。在第二个语句中，上升时延为 4，下降时延为8，传递到x和z的时延相同，是4和8中的最小值，即4。在第3个赋值中，上升时延为 4，下降时延为8，截止时延为 6，传递到x的时延为4(4、8和6中的最小值)。在最后的语句中，所有的时延都为 0。

上升时延对于向量线网目标意味着什么呢？如果右端从非0向量变化到0向量，那么就使用

下降时延。如果右端值到达z，那么使用下降时延；否则使用上升时延。

7.5 线网时延

时延也可以在线网说明中定义，如下面的说明。

```
wire #5 Arb;
```

这个时延表明*Arb*驱动源值改变与线网*Arb*本身间的时延。考虑下面对线网*Arb*的连续赋值语句：

```
assign #2 Arb = Bod & Cap;
```

假定在时刻10，*Bod*上的事件促使右端表达式计算。如果结果不同，则在2个时间单位后赋值给*Arb*，即时刻12。但是因为定义了线网时延，实际对*Arb*的赋值发生在时刻17(= 10 + 2 + 5)。图7-3的波形举例说明了不同的时延。

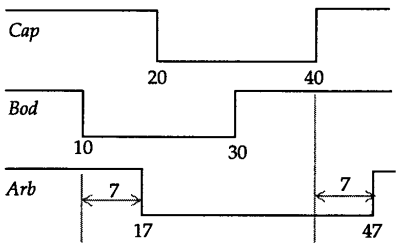


图7-3 带有赋值时延的线网时延

图7-4很好地描述了线网时延的效果。首先使用赋值时延，然后增加任意线网时延。

如果时延在线网说明赋值中出现，那么时延不是线网时延，而是赋值时延。下面是A的线网说明赋值，2个时间单位是赋值时延，而不是线网时延。

```
wire #2 A = B - C;      //赋值时延
```

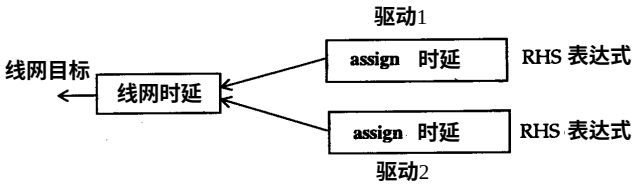


图7-4 线网时延的效果

7.6 举例

7.6.1 主从触发器

下面是图5-9所示的主从触发器的Verilog HDL模型。

```
module MSDFF_DF (D, C, Q, Qbar);
  input D, C;
  output Q, Qbar;
  wire NotC, NotD, NotY, Y, D1, D2, Ybar, Y1,;Y2

  assign NotD = ~ D;
  assign NotC = ~ C;
  assign NotY = ~ Y;

  assign D1 = ~ (D & C);
  assign D2 = ~ (C & NotD);
```

```
  assign Y = ~ (D1 & Ybar);
  assign Ybar = ~ (Y & D2);
  assign Y1 = ~ (Y & NotC);
  assign Y2 = ~ (NotY & NotQ);
  assign Q = ~ (Qbar & Y1);
  assign Qbar = ~ (Y2 & Q);
endmodule
```

7.6.2 数值比较器

下面是8位(参数定义的)数值比较器数据流模型。

```
module MagnitudeComparator(A, B, AgtB, AeqB, AltB);
parameter BUS = 8;
parameter EQ_DELAY = 5, LT_DELAY = 8, GT_DELAY = 8;
input [1 : BUS]A, B;
output AgtB, AeqB, AltB;

assign #EQ_DELAY AeqB = A = = B;
assign #GT_DELAY AgtB = A > B;
assign #LT_DELAY AltB = A < B;
endmodule
```

习题

1. 举例说明截止时延在连续赋值语句中如何使用？
2. 当对同一目标有2个或多个赋值形式时，如何决定目标有效值？
3. 写出图5-10所示的奇偶产生电路的数据流模型描述形式。只允许使用 2个赋值语句，并规定上升和下降时延。
4. 使用连续赋值语句，描述图5-12所示的优先编码器电路的行为。
5. 假定：

```
tri0 [4:0] Qbus;
assign Qbus = Sbus;
assign Qbus = Pbus;
```

如果*Pbus*和*Sbus*均为高阻态z，*Qbus*上的值是什么？

第8章 行为建模

在前几章中，我们已经介绍了使用门和 UDP实例语句的门级建模方式，以及用连续赋值语句的数据流建模方式。本章描述 Verilog HDL中的第三种建模方式，即行为建模方式。为充分使用Verilog HDL，一个模型可以包含所有上述三种建模方式。

8.1 过程结构

下述两种语句是为一个设计的行为建模的主要机制。

1) initial 语句

2) always语句

一个模块中可以包含任意多个 initial或always语句。这些语句相互并行执行，即这些语句的执行顺序与其在模块中的顺序无关。一个 initial语句或always语句的执行产生一个单独的控制流，所有的initial和always语句在0时刻开始并行执行。

8.1.1 initial 语句

initial 语句只执行一次。initial 语句在模拟开始时执行，即在 0时刻开始执行。initial 语句的语法如下：

```
initial
[timing_control] procedural_statement
```

procedural_statement是下列语句之一：

```
procedural_assignment(blocking or non-blocking)//阻塞或非阻塞性过程赋值语句//
procedural_continuous_assignment
conditional_statement
case_statement
loop_statement
wait_statement
disable_statement
event_trigger
sequential_block
parallel_block
task_enable(user or system)
```

顺序过程(begin...end)最常使用在进程语句中。这里的时序控制可以是时延控制，即等待一个确定的时间；或事件控制，即等待确定的事件发生或某一特定的条件为真。 initial语句的各个进程语句仅执行一次。注意 initial语句在模拟的0时刻开始执行。initial语句根据进程语句中出现的时间控制在以后的某个时间完成执行。

下面是initial语句实例。

```
reg Yurt;
. . .
initial
  Yurt = 2;
```

上述initial语句中包含无时延控制的过程赋值语句。 initial语句在0时刻执行，促使*Yurt*在0时刻被赋值为2。下例是一个带有时延控制的initial语句。

```
reg Curt;
. . .
initial
  #2 Curt = 1;
```

寄存器变量*Curt*在时刻2被赋值为1。initial语句在0时刻开始执行，在时刻2完成执行。

下例为带有顺序过程的initial语句。

```
parameter SIZE = 1024;
reg [7:0] RAM [0:SIZE-1];
reg RibReg;

initial
  begin: SEQ_BLK_A
    integer Index;
  RibReg = 0;
  for (Index = 0; Index < SIZE; Index = Index + 1)
    RAM [Index] = 0;
end
```

顺序过程由关键词begin...end定界，它包含顺序执行的进程语句，与 C语言等高级编程语言相似。*SEQ_BLK_A*是顺序过程的标记；如果过程中没有局部说明部分，不要求这一标记。例如，如果对*Index*的说明部分在initial语句之外，可不需要标记。整数型变量 *Index*已在过程中声明。并且，顺序过程包含 1个带循环语句的过程性赋值。这一 initial语句在执行时将所有的内存初始化为0。

下例是另一个带有顺序过程的 initial语句。在此例中，顺序过程包含时延控制的过程性赋值语句。

```
//波形生成：
parameter APPLY_DELAY = 5;
reg[0:7]port_A;
. . .
initial
  begin
    Port_A ='h20;
  #APPLY_DELAY Port_A= 'hF2;
  #APPLY_DELAY Port_A= 'h41;
  #APPLY_DELAY Port_A= 'h0A;
end
```

执行时，*Port_A*的值如图8-1所示。

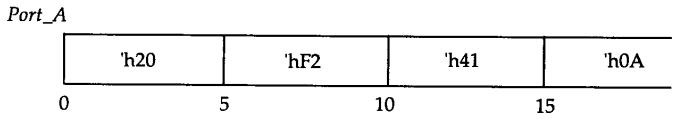


图8-1 使用initial语句产生的波形

如上面举例所示，Initial语句主要用于初始化和波形生成。

8.1.2 always语句

与initial语句相反，always语句重复执行。与initial语句类似，always语句语法如下：

```
always
  [timing_control] procedural_statement
```

过程语句和时延控制（时序控制）的描述方式与上节相同。

例如：

```
always
  Clk = ~ Clk;
//将无限循环。
```

此always语句有一个过程性赋值。因为always语句重复执行，并且在此例中没有时延控制，过程语句将在0时刻无限循环。因此， always语句的执行必须带有某种时序控制，如下例的always语句，形式上与上面的实例相同，但带有时延控制。

```
always
  #5 Clk = ~ Clk;
//产生时钟周期为10的波形。
```

此always语句执行时产生周期为10个时间单位的波形。

下例是由事件控制的顺序过程的always语句。

```
reg [0:5] InstrReg;
reg [3:0] Accum;
wire ExecuteCycle;

always
  @ (EcecuteCycle)
    begin
     case(InstrReg[0:1])
       2'b00: Store (Accum, InstrReg[2:5]);
       2'b11: Load (Accum, InstrReg[2:5]);
       2'b01: Jump (InstrReg[2:5]);
       2'b10:;
     endcase
  end
//Store、Load和Jump是 在别处定义的用户自定义的任务。
```

顺序过程(begin...end)中的语句按顺序执行。这个always语句意味着只要有事件发生，即只要发生变化，*ExecuteCycle*就执行顺序过程中的语句；顺序过程的执行意味着按顺序执行过程中的各个语句。

下例为带异步预置的负边沿触发的D触发器的行为模型。

```
module DFF(Clk, D, Set, Q, Qbar);
  input Clk, D, Set;
  output Q, Qbar;
  reg Q, Qbar;

  always
   wait (Set == 1)
    begin
      #3 Q = 1;
      #2 Qbar = 0;
```

```
      wait (Set == 0);
    end
always
  @ (negedge Clk)
    begin
     if (Set != 1)
      begin
        #5 Q = D;
        #1 Qbar = ~ Q;
      end
    end
endmodule
```

此模型中有2条always语句。第一条always语句中顺序过程的执行由电平敏感事件控制。第二条always语句中顺序过程的执行由边沿触发的事件控制。

8.1.3 两类语句在模块中的使用

一个模块可以包含多条always语句和多条initial语句。每条语句启动一个单独的控制流。各语句在0时刻开始并行执行。

下例中含有1条initial语句和2条always语句。

```
module TestXorBehavior;
  reg Sa, Sb, Zeus;

  initial
  begin
    Sa = 0;
    Sb = 0;
    #5 Sb = 1;
    #5 Sa = 1;
    #5 Sb = 0;
  end

always
  @ (Sa or Sb) Zeus = Sa ^ Sb

always
  @ (Zeus)
     $display ("At time %t, Sa = %d, Sb = %d, Zeus = %b",
               $time, Sa, Sb, Zeus);
endmodule
```

模块中的3条语句并行执行，其在模块中的书写次序并不重要。 initial语句执行时促使顺序过程中的第一条语句执行，即*Sa*赋值为0；下一条语句在0时延后立即执行。initial语句中的第3行表示“等待5个时间单位”。这样*Sb*在5个时间单位后被赋值为1，*Sa*在另外5个时间单位后被赋值为0。执行顺序过程最后一条语句后，initial语句被永远挂起。

第一条always语句等待*Sa*或*Sb*上的事件发生。只要有事件发生，就执行always语句内的语句，然后always语句重新等待发生在*Sa*或*Sb*上的事件。注意根据initial语句对*Sa*和*Sb*的赋值，always语句将在第0、5、10和15个时间单位时执行。

同样，只要有事件发生在*Zeus*上，就执行第2条always语句。在这种情况下，系统任务 $display被执行，然后always语句重新等待发生在*Zeus*上的事件。*Sa*、*Sb*和*Zeus*上产生的波形如图8-2所示。下面是模块模拟运行产生的输出。

```
在时刻          5,  Sa = 0, Sb = 1, Zeus = 1
在时刻          10, Sa = 1, Sb = 1, Zeus = 0
在时刻          15, Sa = 1, Sb = 0, Zeus = 1
```

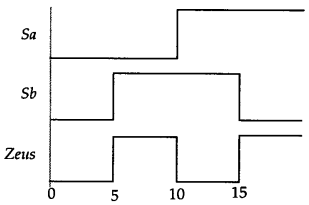


图8-2 *Sa*、*Sb*和*Zeus*上产生的波形

8.2 时序控制

时序控制与过程语句关联。有2种时序控制形式：

1) 时延控制

2) 事件控制

8.2.1 时延控制

时延控制形式如下：

```
#delay procedural_statement
```

实例如下；

```
#2 Tx = Rx-5;
```

时延控制定义为执行过程中首次遇到该语句与该语句的执行的时间间隔。时延控制表示在语句执行前的“等待时延”。上面的例子中，过程赋值语句在碰到该语句后的2个时间单位执行，然后执行赋值。

另一实例如下：

```
initial
  begin
    #3 Wave = 'b0111;
    #6 Wave = 'b1100;
    #7 Wave = 'b0000;
end
```

initial语句在0时刻执行。首先，等待3个时间单位执行第一个赋值，然后等待6个时间单位，执行第2个语句；再等待7个时间单位，执行第3个语句；然后永远挂起。

时延控制也可以用另一种形式定义：

```
#delay;
```

这一语句促使在下一条语句执行前等待给定的时延。下面是这种用法的实例。

```
parameter ON_DELAY = 3, OFF_DELAY = 5;
always
  begin
    # ON_DELAY;    //等待ON_DELAY规定的时延。
    RefClk = 0;
    # OFF_DELAY;    //等待OFF_DELAY规定的时延。
    RefClk = 1;
end
```

时延控制中的时延可以是任意表达式，即不必限定为某一常量，见下面的例子。

```
# Strobe
  Compare = TX^ask;
```

```
# (PERIOD/2)
  Clock = ~Clock
```

如果时延表达式的值为0，则称之为显式零时延。

```
#0;          /显式零时延。
```

显式零时延促发一个等待，等待所有其它在当前模拟时间被执行的事件执行完毕后，才将其唤醒；模拟时间不前进。

如果时延表达式的值为x或z,其与零时延等效。如果时延表达式计算结果为负值，那么其二进制的补码值被作为时延，这一点在使用时务请注意。

8.2.2 事件控制

在事件控制中，always的过程语句基于事件执行。有两种类型的事件控制方式：

1) 边沿触发事件控制

2) 电平敏感事件控制

1. 边沿触发事件控制

边沿触发事件控制如下：

```
@ event procedural_statement
```

如下例所示：

```
@ (posedge Clock)
Curr_State = Next_State;
```

带有事件控制的进程或过程语句的执行，须等到指定事件发生。上例中，如果*Clock*信号从低电平变为高电平（正沿），就执行赋值语句；否则进程被挂起直到*Clock*信号产生下一个正跳边沿。

下面是进一步的实例。

```
@ (negedge Reset)  Count = 0;
@Cla
Zoo = Foo;
```

在第一条语句中，赋值语句只在 *Reset*上的负沿执行。第二条语句中 ,当*Cla*上有事件发生时，*Foo*的值被赋给*Zoo*,即等待*Cla*上发生事件；当*Cla*的值发生变化时,*Foo*的值被赋给*Zoo*。

也可使用如下形式：

```
@ event ;
```

该语句促发一个等待，直到指定的事件发生。下面是确定时钟在周期的 initial语句中使用的一个例子。

```
time RiseEdge, OnDelay
initial
  begin
    //等待，直到在时钟上发生正边沿：
    @ (posedge ClockA);
    RiseEdge = $time;
    //等待，直到在时钟上发生负边沿：
    @ (negedge ClockA);
    OnDelay = $time - RiseEdge;
    $display ("The on-period of clock is %t.",OnDelay);
end
```

事件之间也能够相或以表明“ 如果有任何事件发生 ”。下例将对此进行说明。

```
@ (posedge Clear or negedge Reset)
```

```
Q = 0;
@ (Ctrl_A or Ctrl_B)
Dbus = 'bz;
```

注意关键字or并不意味着在1个表达式中的逻辑或。

在Verilog HDL中posedge和negedge是表示正沿和负沿的关键字。信号的负沿是下述转换的一种：

```
1 -> x
1 -> z
1 -> 0
x -> 0
z -> 0
```

正沿是下述转换的一种：

```
0 -> x
0 -> z
0 -> 1
x -> 1
z -> 1
```

2. 电平敏感事件控制

在电平敏感事件控制中，进程语句或进程中的过程语句一直延迟到条件变为真后才执行。电平敏感事件控制以如下形式给出：

```
wait (Condition)
  procedural_statement
```

过程语句只有在条件为真时才执行，否则过程语句一直等待到条件为真。如果执行到该语句时条件已经为真，那么过程语句立即执行。在上面的表示形式中，过程语句是可选的。

例如：

```
wait (Sum > 22)
Sum = 0;

wait (DataReady)
Data = Bus;

wait (Preset);
```

在第一条语句中，只有当 *Sum*的值大于22时，才对Sum清0。在第二条语句中，只有当*DataReady*为真，即*DataReady*值为1时，将*Bus*赋给*Data*。最后一条语句表示延迟至 *Preset*变为真（值为1）时，其后续语句方可继续执行。

8.3 语句块

语句块提供将两条或更多条语句组合成语法结构上相当于一条语句的机制。在 Verilog HDL中有两类语句块，即：

1) 顺序语句块(begin...end)：语句块中的语句按给定次序顺序执行。

2) 并行语句块(fork...join)：语句块中的语句并行执行。

语句块的标识符是可选的，如果有标识符，寄存器变量可在语句块内部声明。带标识符的语句块可被引用；例如，语句块可使用禁止语句来禁止执行。此外，语句块标识符提供唯一标识寄存器的一种方式。但是，要注意所有的寄存器均是静态的，即它们的值在整个模拟

运行中不变。

8.3.1 顺序语句块

顺序语句块中的语句按顺序方式执行。每条语句中的时延值与其前面的语句执行的模拟时间相关。一旦顺序语句块执行结束，跟随顺序语句块过程的下一条语句继续执行。顺序语句块的语法如下：

```
begin
  [:block_id{declarations}]
  procedural_statement(s)
end
```

例如：

```
//产生波形:
begin
  #2 Stream = 1;
  #5 Stream = 0;
  #3 Stream = 1;
  #4 Stream = 0;
  #2 Stream = 1;
  #5 Stream = 0;
end
```

假定顺序语句块在第10个时间单位开始执行。两个时间单位后第 1条语句执行，即第12个时间单位。此执行完成后，下 1条语句在第17个时间单位执行(延迟5个时间单位)。然后下1条语句在第20个时间单位执行，以此类推。该顺序语句块执行过程中产生的波形如图 8-3所示。

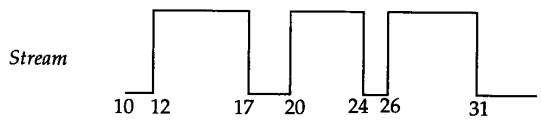


图8-3 顺序语句块中的累积时延

下面是顺序过程的另一实例。

```
begin
  Pat = Mask | Mat;
  @(negedge Clk);
   FF = & Pat
end
```

在该例中，第1条语句首先执行，然后执行第 2条语句。当然，第2条语句中的赋值只有在 *Clk*上出现负沿时才执行。下面是顺序过程的另一实例。

```
begin: SEQ_BLK
  reg[0:3] Sat;

  Sat = Mask & Data;
  FF = ^Sat;
end
```

在这一实例中，顺序语句块带有标记 *SEQ_BLK*，并且有一个局部寄存器说明。在执行时，首先执行第1条语句，然后执行第 2条语句。

8.3.2 并行语句块

并行语句块带有定界符 fork和join(顺序语句块带有定界符 begin和end)，并行语句块中的各语句并行执行。并行语句块内的各条语句指定的时延值都与语句块开始执行的时间相关。当并行语句块中最后的动作执行完成时 (最后的动作并不一定是最后的语句)，顺序语句块的语句继续执行。换一种说法就是并行语句块内的所有语句必须在控制转出语句块前完成执行。并行语句块语法如下：

```
fork
  [:block_id{declarations}]
  procedural_statement(s);
join
```

例如：

```
// 生成波形：
fork
  #2 Stream = 1;
  #7 Stream = 0;
  #10 Stream = 1;
  #14 Stream = 0;
  #16 Stream = 1;
  #21 Stream = 0;
join
```

如果并行语句块在第 10个时间单位开始执行，所有的语句并行执行并且所有的时延都是相对于时刻10的。例如，第3个赋值在第20个时间单位执行，并在第 26个时间单位执行第 5个赋值，以此类推。其产生的波形如图 8-4所示。

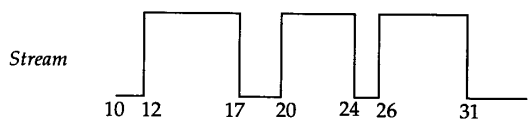


图8-4 并行语句块中的相对时延

下例混合使用了顺序语句块和并行语句块，以强调两者的不同之处。

```
always
  begin:SEQ_A
    #4 Dry = 5;              // S1

    fork: PAR_A               //S2
      #6  Cun = 7;           //P1

      begin: SEQ_B             //P2
       EXE = Box;             //S6
       #5 Jap = Exe;          //S7
      end

      #2 Dop = 3;             //P3
      #4 Gos = 2;             //P4
      #8 Pas = 4;             //P5
    join
```

```
    #8 Bax = 1;                 //S3
    #2 Zoom = 52;               //S4
    #6 $stop;                   //S5
end
```

always语句中包含顺序语句块 *SEQ_A*，并且顺序语句块内的所有语句 (S1、S2、S3、S4和S5)顺序执行。因为always语句在0时刻执行，*Dry*在第4个时间单位被赋值为5，并且并行语句块*PAR_A*在第4个时间单位开始执行。并行语句块中的所有语句 (P1、P2、P3、P4和P5)在第4个时间单位并行执行。这样 *Cun*在第10个时间单位被赋值，*Dop*在第6个时间单位被赋值，*Gos*在第8个时间单位被赋值，*Pas*在第12个时间单位被赋值。顺序语句块 *SEQ_B*在第4个时间单位开始执行，并导致该顺序块中的语句 S6、S7依次被执行；*Jap*在时间单位9被赋于新值。因为并行语句块 *PAR_A*中的所有语句在第 12个时间单位完成执行，语句 S3在第12个时间单位被执行，在第20个时间单位 *Bax*被赋值，然后语句 S4执行，在第22个时间单位 *Zoom*被赋值，然后执行下一语句。最终在第 28个时间单位执行系统任务 **$stop**。always语句执行时发生的事件如图8-5所示。

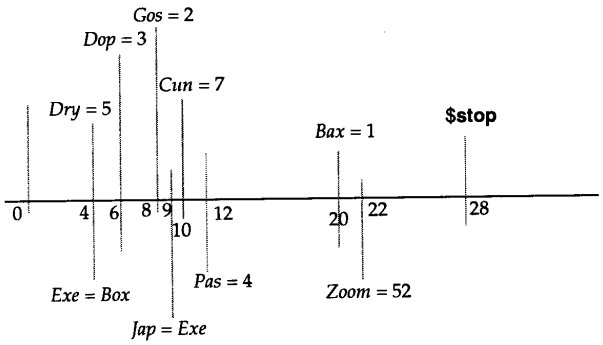


图8-5 顺序语句块和并行语句块混合使用时的时延

8.4 过程性赋值

过程性赋值是在initial语句或always语句内的赋值，它只能对寄存器数据类型的变量赋值。表达式的右端可以是任何表达式。例如：

```
reg[1:4] Enable,A,B;
...
#5 Enable = ~A ^ ~B;
```

*Enable*为寄存器。根据时延控制，赋值语句被延迟5个时间单位执行。右端表达式被计算，并赋值给*Enable*。

过程性赋值与其周围的语句顺序执行。 always语句实例如下：

```
always
  @ (A or B or C or D)
    begin:AOI
      reg Temp1,Temp2;

      Temp1 = A & B;
      Temp2 = C & D;
      Temp1 = Temp1 | Temp2;
```

```
        Z = ~Temp1;
    end
  /*可用一个语句代替上面的4条语句，例如：
      Z = ~((A&B)|(C&D));
    但是，上例的目的主要用于解释说明顺序过程语句的顺序特性 */
```

always语句内的顺序过程在信号 *A*、*B*、*C*或*D*发生变化时开始执行。 *Temp1*的赋值首先执行。然后执行第二个赋值。在以前赋值中计算的 *Temp1*和*Temp2*的值在第三条赋值语句中使用。最后一个赋值使用在第三条语句中计算的 *Temp1*的值。

过程性赋值分两类：

1) 阻塞性过程赋值

2) 非阻塞性过程赋值

在讨论这两类过程性赋值前，先简要地说明语句内部时延的概念。

8.4.1 语句内部时延

在赋值语句中表达式右端出现的时延是语句内部时延。通过语句内部时延表达式，右端的值在赋给左端目标前被延迟。例如：

```
Done = #5 'b1;
```

重要的是右端表达式在语句内部时延之前计算，随后进入时延等待，再对左端目标赋值。下例说明了语句间和语句内部时延的不同。

```
Done = #5 'b1;      /语句内部时延控制
```

与

```
begin
Temp = 'b1;
#5 Done = Temp;        /语句间时延控制
end
```

相同。而语句

```
Q = @(posedge Clk ) D;    //语句内事件控制
```

与

```
begin
Temp = D;
@(posedge Clk)          /语句间事件控制
Q = Temp;
end
```

相同。

除以上两种时序控制(时延控制和事件控制)可用于定义语句内部时延外，还有另一种重复事件控制的语句内部时延表示形式。形式如下：

```
repeat(express) @ (event_expression)
```

这种控制形式用于根据一定数量的 1个或多个事件来定义时延。例如，

```
Done = repeat(2) @ (negedge ClkA) A_REG + B_REG
```

这一语句执行时先计算右端的值，即 *A_Reg + B_Reg*的值，然后等待时钟 *ClkA*上的两个负沿，再将右端值赋给 *Done*。这一重复事件控制实例的等价形式如下：

```
begin
  Temp = A_REG + B_REG
  @ (negedge ClkA);
  @ (negedge ClkA);
  Done = Temp;
end
```

这种形式的时延控制方式在给某些边或一定数量的边的同步赋值过程（语句）中非常有用。

8.4.2 阻塞性过程赋值

赋值操作符是“ = ”的过程赋值是阻塞性过程赋值。例如，

```
RegA = 52;
```

是阻塞性过程赋值。阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。如下所示：

```
always
  @(A or B or Cin)
   begin: CARRY_OUT
    reg T1,T2,T3;

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

*T1*赋值首先发生，计算 *T1*；接着执行第二条语句， *T2*被赋值；然后执行第三条语句， *T3*被赋值；依此类推。

下例是使用语句内部时延控制的阻塞性过程赋值语句。

```
initial
begin
  Clr = #5 0;
  Clr = #4 1;
  Clr = #10 0;
end
```

第一条语句在0时刻执行，*Clr*在5个时间单位后被赋值；接着执行第二条语句，使 *Clr*在4个时间单位后被赋值为1(从0时刻开始为第9个时间单位)；然后执行第三条语句促使*Clr*在10个时间单位后被赋值为0(从0时刻开始为第19个时间单位)。图8-6显示*Clr*上产生的波形。

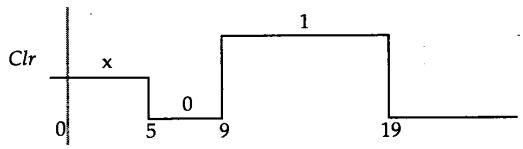


图8-6 带有语句内部时延控制的阻塞性过程赋值

另一实例如下：

```
begin
  Art = 0;
```

```
    Art = 1;
end
```

在这种情况下，*Art*被赋值为1。这是因为第一个*Art*被赋值为0，然后执行下一条语句促使*Art*在0时延后被赋值为1。因此对*Art*的0赋值被丢弃。

8.4.3 非阻塞性过程赋值

在非阻塞性过程赋值中，使用赋值符号“ <= ”。例如：

```
begin
  Load <= 32;
  RegA <= Load;
  RegB <= Store;
end
```

在非阻塞性过程赋值中，对目标的赋值是非阻塞的 (因为时延)，但可预定在将来某个时间步发生(根据时延；如果是0时延，那么在当前时间步结束)。当非阻塞性过程赋值被执行时，计算右端表达式，右端值被赋于左端目标，并继续执行下一条语句。预定的最早输出将在当前时间步结束时，这种情况发生在赋值语句中没有时延时。在当前时间步结束或任意输出被调度时，即对左端目标赋值。

在上面的例子中，我们假设顺序语句块在时刻 10执行。第一条语句促使*Load*在第10个时间单位结束时被赋值为 32；然后执行第2条语句，*Load*的值不变(注意时间还没有前进，并且第1个赋值还没有被赋新值)，*RegA*的赋值被预定为在第 10个时间步结束时。在所有的事件在第10个时间单位发生后，完成对左端目标的所有预定赋值。

下面的例子更进一步解释这种赋值特征。

```
initial
  begin
    Clr <= #5 1;
    Clr <= #4 0;
    Clr <= #10 0;
end
```

第一条语句的执行使*Clr* 在第5个时间单位被赋于值1；第二条语句的执行使*Clr*在第4个时间单位被赋值为 0(从0时刻开始的第 4个时间单位)；最终，第3条语句的执行使*Clr*在第10个时间单位被赋值为0(从0时刻开始的第10个时间单位)。注意3条语句都是在0时刻执行的。此外，在这种情况下，非阻塞性赋值执行次序变得彼此不相关。*Clr*上产生的波形如图8-7所示。

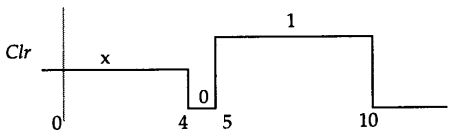


图8-7 带有语句内部时延的非阻塞性过程赋值

下面是带有0时延的例子。

```
initial
  begin
    Cbn <= 0;
    Cbn <= 1;
end
```

在initial语句执行后，因为同时对同一寄存器变量有多个赋值， *Cbn*的值变得不确定，即*Cbn* = x。Verilog HDL标准中既没有规定在这种情况下，何种事件被调度，也没有规定事件

被取消的次序。结果是根据特定的 Verilog模拟器的事件调度算法，*Cbn*将被赋值为0或1[⊖]。

下面是同时使用阻塞性和非阻塞性过程赋值的实例，注意它们的区别。

```
reg [0:2]  Q_State;

initial
  begin
   Q_State = 3'b011;
   Q_State <= 3'b100;
  $display ("Current value of Q_State is %b", Q_State);
  #5;  //等待一定的时延。
  $display ("The delayed value of Q_State is %b", Q_State);
end
```

执行initial语句产生如下结果：

```
Current value of Q_State is 011
The delayed value of Q_State is 100
```

第一个阻塞性赋值使*Q_State*被赋值为3'b011。执行第二条赋值语句(为非阻塞性赋值语句)促使*Q_State*在当前时间步(第0步)结束时被赋值为3'b100。因此当第一个$display任务被执行时，*Q_State*还保持来自第一个赋值的值，即 3'b011。当#5时延被执行后，促使被调度的*Q_State*赋值发生，*Q_State*的值被更新。延迟5个时间单位后，执行下一个$display任务，此时显示*Q_State*的更新值。

8.4.4 连续赋值与过程赋值的比较

连续赋值与过程赋值有什么不同之处？表8-1列举了它们的差异。

表8-1 过程赋值与连续赋值间的差异

过程赋值	连续赋值
在always语句或initial语句内出现	在一个模块内出现
执行与周围其它语句有关	与其它语句并行执行；在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
使用“=”或“=”赋值符号	使用“=”赋值符号
无assign关键词(在过程性连续赋值中除外，参见第8章第8节)	有assign关键词

下例进一步解释了这些差别。

```
module Procedural;
reg A,B,Z;

always
  @(B) begin
   Z = A;
   A = B;
  end
endmodule
```

⊖ 在上述情况下，使用不同模拟器对其进行模拟所产生的模拟结果可能会有所不同。—— 译者注

```
module Continuous
  wire A,B,Z;

  assign Z = A;
  assign A = B;
endmodule
```

假定*B*在10 ns时有一个事件。在过程性赋值模块中，两条过程语句被依序执行， *A*在10 ns时得到*B*的新值。*Z*没有得到*B*的值，因为赋值给*Z*发生在赋值给*A*之前。在连续性赋值语句模块中，第二个连续赋值被触发，因为这里有一个关于 *B*的事件。这引起了关于*A*的事件，*A*引发第一个连续赋值被执行，这相应引起 *Z*得到了*A*的值。*Z*的新值为*A*而不是*B*。然而，如果事件发生在*A*上，过程性模块中的always语句不执行，因为*A*不在那个always语句的实时控制事件清单中。然而连续赋值语句中的第一个连续赋值执行，并且 *Z*得到*A*的新值。

8.5 if 语句

if语句的语法如下：

```
if(condition_1)
  procedural_statement_1
{else if(condition_2)
  procedural_statement_2}
{else
  procedural_statement_3}
```

如果对*condition_1* 求值的结果为一个非零值，那么*procedural_statement_1*被执行，如果*condition_1* 的值为0、x或z，那么*procedural_statement_1*不执行。

如果存在一个else分支，那么这个分支被执行。以下是一个例子。

```
if(Sum < 60)
  begin
    Grade = C;
    Total_C = Total _c + 1;
  end
else if(Sum < 75)
  begin
    Grade = B;
    Total_B = Total_B + 1;
  end
else
  begin
    Grade = A;
    Total_A = Total_A + 1;
  end
```

注意条件表达式必须总是被括起来，如果使用 if-if-else格式，那么可能会有二义性，如下例所示：

```
if(Clk)
  if(Reset)
    Q = 0;
  else
    Q = D;
```

问题是最后一个else属于哪一个if? 它是属于第一个if的条件(*Clk*)还是属于第二个if的条件(*Reset*)? 这在Verilog HDL中已通过将else与最近的没有else的if相关联来解决。在这个例子中，else与内层if语句相关联。

以下是另一些if语句的例子。

```
if(Sum < 100)
  Sum = Sum + 10;
if(Nickel_In)
  Deposit = 5;
else if(Dime_In)
  Deposit = 10;
else if(Quarter_In)
  Deposit = 25;
else
  Deposit = ERROR;

if(Ctrl)
  begin
    if(~Ctrl2)
      Mux = 4'd2;
    else
      Mux = 4'd1;
  end
else
  begin
    if(~Ctrl2)
      Mux = 4'd8;
    else
      Mux = 4'd4;
  end
```

8.6 case语句

case语句是一个多路条件分支形式，其语法如下：

```
case(case_expr)
  case_item_expr{,case_item_expr}:procedural_statement
. . .
. . .
  [default:procedural_statement]
endcase
```

case语句首先对条件表达式 *case_expr*求值，然后依次对各分支项求值并进行比较，第一个与条件表达式值相匹配的分支中的语句被执行。可以在 1个分支中定义多个分支项；这些值不需要互斥。缺省分支覆盖所有没有被分支表达式覆盖的其他分支。

分支表达式和各分支项表达式不必都是常量表达式。在 case语句中，x和z值作为文字值进行比较。case语句如下所示：

```
parameter
    MON = 0 , TUE = 1, WED = 2,
    THU = 3, FRI = 4,
    SAT = 5, SUN = 6;
reg [0:2] Day;
```

```
integer Pocket_Money;

case (Day)
  TUE : Pocket_Money = 6;      /分支1。
  MON ,
  WED : Pocket_Money  = 2;    /分支2。
  FRI,
  SAT,
  SUN : Pocket_Money = 7;     /分支3。
  default : Pocket_Money = 0;        /分支4。
endcase
```

如果*Day*的值为*MON*或*WED*，就选择分支2。分支3覆盖了值*FRI*、*SAT*和*SUN*，而分支4覆盖了余下的所有值，即*THU*和位向量111。case语句的另一实例如下：

```
module ALU (A, B, OpCode, Z);
  input [3:0] A, B;
  input [1:2] OpCode;
  output [7:0] Z;
  reg [7:0] Z;
parameter
    ADD_INSTR = 2'b10,
    SUB_INSTR = 2'b11,
    MULT_INSTR = 2'b01,
    DIV_INSTR = 2'b00;
always
  @ (A or B or OpCode)
    case (OpCode)
      ADD_INSTR:    Z = A + B;
      SUB_INSTR:    Z = A -B;
      MULT_INSTR:   Z = A * B;
      DIV_INSTR:    Z = A / B;
    endcase
endmodule
```

如果case表达式和分支项表达式的长度不同会发生什么呢？在这种情况下，在进行任何比较前所有的case表达式都统一为这些表达式的最长长度。下例说明了这种情况。

```
case (3'b101 << 2)
  3'b100  : $display ( "First branch taken!");
  4'b0100 : $display ( "Second branch taken!");
  5'b10100: $display ( "Third branch taken!");
  default : $display ( "Default branch taken!");
endcase
```

产生：

```
Third branch taken!
```

因为第3个分支项表达式长度为5位，所有的分支项表达式和条件表达式长度统一为5。当计算3'b101<<2时，结果为5'b10100，并选择第3个分支。

case语句中的无关位

上节描述的case语句中，值x和z只从字面上解释，即作为x和z值。这里有case语句的其

它两种形式：casex和casez，这些形式对x和z值使用不同的解释。除关键字casex和casez以外，语法与case语句完全一致。

在casez语句中，出现在case表达式和任意分支项表达式中的值z被认为是无关值，即那个位被忽略(不比较)。

在casex语句中，值x和z都被认为是无关位。casez语句实例如下：

```
case(Mask)
  4'b1??? : Dbus[4] = 0;
  4'b01?? : Dbus[3] = 0;
  4'b001? : Dbus[2] = 0;
  4'b0001 : Dbus[1] = 0;
endcase
```

? 字符可用来代替字符z,表示无关位。casez语句表示如果Mask的第1位是1(忽略其它位)，那么将*Dbus*[4]赋值为0；如果*Mask*的第1位是0，并且第2位是1(忽略其它位)，那么*Dbus[3]*被赋值为0，并依此类推。

8.7 循环语句

Verilog HDL中有四类循环语句，它们是：

1) forever循环
2) repeat循环
3) while循环
4) for 循环

8.7.1 forever 循环语句

这一形式的循环语句语法如下：

```
forever
  procedural_statement
```

此循环语句连续执行过程语句。因此为跳出这样的循环，中止语句可以与过程语句共同使用。同时，在过程语句中必须使用某种形式的时序控制，否则， forever循环将在0时延后永远循环下去。

这种形式的循环实例如下：

```
initial
  begin
    Clock = 0;
    # 5 forever
      #10 Clock = ~Clock;
  end
```

这一实例产生时钟波形；时钟首先初始化为 0，并一直保持到第5个时间单位。此后每隔10个时间单位，*Clock*反相一次。

8.7.2 repeat 循环语句

repeat 循环语句形式如下：

```
repeat(loop_count)
  procedural_statement
```

这种循环语句执行指定循环次数的过程语句。如果循环计数表达式的值不确定，即为 x或z时，那么循环次数按0处理。下面是一些具体实例。

```
repeat (Count)
  Sum = Sum + 10;

repeat (ShiftBy)
  P_Reg = P_Reg << 1;
```

repeat循环语句与重复事件控制不同。例如，

```
repeat(Count)          //repea循环语句
  @ (posedge Clk) Sum = Sum + 1;
```

上例表示计数的次数，等待*Clk*的正边沿，并在*Clk*正沿发生时，对*Sum*加1。但是，

```
Sum = repeat(Count) @ (posedge Clk) Sum + 1;
  // 重复事件控制
```

该例表示首先计算Sum + 1，随后等待*Clk*上正沿计数，最后为左端赋值。

下面的表达式意味着什么？

```
repeat(NUM_OF_TIMES) @ (negedge ClockZ);
```

它表示在执行跟随在repeat语句之后的语句之前，等待*ClockZ* 的*NUM_OF_TIMES*个负沿。

8.7.3 while 循环语句

while 循环语句语法如下：

```
while(condition)
  procedural_statement
```

此循环语句循环执行过程赋值语句直到指定的条件为假。如果表达式在开始时为假，那么过程语句便永远不会执行。如果条件表达式为 x或z，它也同样按0（假）处理。例如：

```
while (BY > 0 )
  begin
    Acc = Acc << 1;
    By = By - 1;
  end
```

8.7.4 for 循环语句

for 循环语句的形式如下：

```
for(initial_assignment;condition;step_assignment)
  procedural_statement
```

一个for循环语句按照指定的次数重复执行过程赋值语句若干次。初始赋值*initial_assignment*给出循环变量的初始值。*condition*条件表达式指定循环在什么情况下必须结束。只要条件为真，循环中的语句就执行；而 *step_assignment*给出要修改的赋值，通常为增加或减少循环变量计数。

```
integer K;

for (K=0 ; K < MAX_RANGE ; K = K + 1)
  begin
    if(Abus[K] == 0)
     Abus[K] = 1;
```

```
    else if(Abus[k] == 1)
     Abus[K] = 0;
    else
     $display("Abus[K] is an x or a z");
  end
```

8.8 过程性连续赋值

过程性连续赋值是过程性赋值的一类，即它不能够在 always语句或initial语句中出现。这种赋值语句能够替换其它所有对线网或寄存器的赋值。它允许赋值中的表达式被连续驱动到寄存器或线网当中。注意，这不是一个连续赋值，连续赋值发生在 initial或always语句之外。

过程性连续赋值语句有两种类型：

1) 赋值和重新赋值过程语句：它们对寄存器进行赋值。

2) 强制和释放过程性赋值语句：虽然它们也可以用于对寄存器赋值，但主要用于对线网赋值。

赋值和强制语句在如下意义上是“连续”的：即当赋值或强制发生效用时，右端表达式中操作数的任何变化都会引起赋值语句重新执行。

过程性连续赋值的目标不能是寄存器部分选择或位选择。

8.8.1 赋值—重新赋值

一个赋值过程语句包含所有对寄存器的过程性赋值，重新赋值过程语句中止对寄存器的连续赋值。寄存器中的值被保留到其被重新赋值为止。

```
module DEF(D,Clr,Clk,Q);
  input D,Clr,Clk;
  output Q;
  reg Q;

  always
    @(Clr) begin
     if(!Clr)
      assign Q = 0;  // 对Q无效。
     else
      deassign Q;
  end

  always
    @(negedge Clk) Q = D;
endmodule
```

如果*Clr*为0，assign赋值语句使*Q*清0，而不管时钟边沿的变化情形，即 *Clk*和*D*对*Q*无效。如果*Clr*变为1，重新赋值语句被执行；这就使得强制赋值方式被取消，以后 *Clk*能够对*Q*产生影响。

如果赋值应用于一个已经被赋值的寄存器， assign赋值在进行新的过程性连续赋值前取消了原来的赋值。实例如下：

```
reg[3:0] Pest;
...
```

```
Pest = 0;
...
assign Pest = Hty ^ Mtu;
...
assign Pest = 2;    /将对Pest重新赋值，然后赋值。
...
deassign Pest;      //Pes连续地保持值为2。
...
assign Pest[2] = 1;   /错误：对寄存器的位选择不能够作为过程性连续赋值的目标 */
```

第二个赋值语句在进行下一个赋值前促使第一个赋值被重新赋值。在重新分配执行后，*Pest*的值在另一个对寄存器的赋值前保持为 2。

赋值语句在如下意义上是连续的：即在第 1个赋值执行后，第2个赋值开始执行前，*Hty*或*Mtu*上的任何变化将促使第1个赋值语句被重新计算。

8.8.2 force与release

force和release过程语句与assign和deassign非常相似，不同的是 force和release过程语句不仅能够应用于线网，也能够应用于寄存器的赋值。

当force语句应用于寄存器时，寄存器的当前值被 force语句的值覆盖；当 release语句应用于寄存器时，寄存器中的当前值保持不变，除非过程性连续赋值已经生效（在 force语句被执行时），在这种情况下，连续赋值为寄存器建立新值。

当用force过程语句对线网进行赋值时，该赋值方式为线网替换所有驱动源，直到在那个线网上执行release语句为止。

```
wire Prt;
...
or #1 (Prt,Std,Dzx);
initial
  begin
    force Prt = Dzx & Std;
    #5;                      //等待5个时间单位。
    release Prt;
  end
```

执行force语句使*Prt*的值覆盖来自于或门原语的值，直到 release语句被执行，然后或门原语的*Prt*驱动源重新生效。尽管 force赋值有效（在前5个时间单位），*Dzx*和*Std*上的任何变化都促使赋值重新执行。

另一实例如下：

```
reg[2:0] Colt;
...
Colt = 2;
force Colt = 1;
...
release Colt;          // Col保持值为1。
...
assign Colt = 5;
...
force Colt = 3;
...
```

```
release Colt;          //Colt值变为5。
...
force Colt[1:0] = 3;   /错误：寄存器的部分选择不能设为过程性连续赋值的目标 */
```

*Colt*的第1次释放促使 *Colt*的值被保持为1。这是因为在force语句被应用时没有过程性连续赋值对寄存器赋值。在后面的release语句中，*Colt*因为过程性连续赋值在*Colt*上重新生效而重新获得值5。

8.9 握手协议实例

always语句可用于描述交互进程的行为，如有限状态机的交互。这些模块内的语句用对所有always语句可见的寄存器来相互通信。在always语句间使用在一个always语句内声明的寄存器变量传递信息并不可取（这可以使用层次路径名实现，见第 10章）。

考虑下面两个交互进程的实例：*RX* ,接收器；*MP*,微处理器。*RX*进程读取串行的输入数据。并发送*Ready*信号表明数据可被读入*MP*进程。*MP*进程在将数据分配给输出后，回送一个接收信号*Ack*到*RX*进程以读取新的输入数据。两个进程的语句块流程如图 8-8所示。

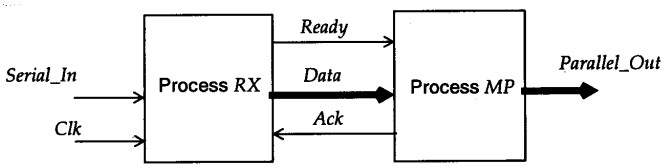


图8-8 两个交互进程

这两个交互进程的行为可用下述行为模型加以描述：

```
'timescale 1ns/100ps
module  Interacting (Serial_In,Clk,Parallel_Out)
  input Serial_In,Clk;
  output [0:7] Parallel_Out;
  reg [0:7] Parallel_Out;

  reg Ready,Ack;
  wire [0:7] data;

  'include "Read_Word.v"  /Read_Word任务在此文件中定义。
always
  begin: RX
    Read_Word(Serial_In,Clk,Data);
    //任务Read_Word在每个时钟周期读取串行数据，将其转换为并行数据并存于 Data中。Read_Word完
    成上述任务需要10ns。
    Ready = 1;
    wait(Ack);
    Ready = 0;
    #40;
  end

always
  begin: MP
    #25;
```

```
    Parallel_Out = Data;
    Ack = 1;
    #25 Ack = 0;
    wait (ready);
  end
endmodule
```

这两个进程通过寄存器*Ready*和*Ack*的交互握手协议如图8-9中的波形显示。

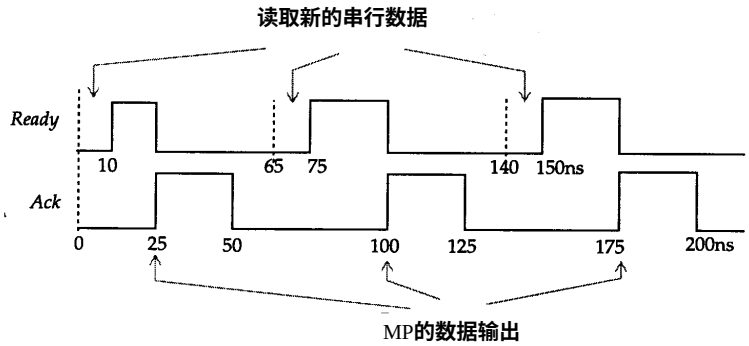


图8-9 两进程间的握手协议

习题

1. initial语句和always语句，哪一种可重复执行？
2. 顺序语句块和并行语句块的区别是什么？举例说明。顺序语句块能否出现在并行语句块中？
3. 语句块在什么时候需要标识符？
4. 在always语句中是否有必要指定时延？
5. 语句内部时延和语句间时延的区别是什么？举例说明。
6. 阻塞性赋值和非阻塞性赋值有何区别？
7. casex语句与case语句有何区别？
8. 能否在always语句中为线网类型（例如wire型线网）赋值？
9. 产生一个在5 ns时刻开始、周期为10 ns的时钟波形。
10. 用一个initial语句和1个forever循环语句替代下述always语句。

```
always
  @(Expected or Observed)
   if (Expected !== Observed) begin
    $display ("MISMATCH: Expected = %b,Observed = %b"
             Expected, Observed);
    $stop;
   end
```

11. 按如下条件，两个always语句中*NextStateA*和*NextStateB*上的值是多少：*ClockP*在5 ns时有1个正沿；*CurrentState*在时钟边沿前值为5，并且在时钟沿3 ns后改变为7？

```
always
  @ (posedge ClockP)
    #7 NextStateA = CurrentState;
```

```
always
  @(posedge ClockP)
    NextState = #7 CurrentState;
```

12. **使用行为建模方式写出如下有限状态机模型的行为描述。**

Inp(Gak)	*PresentState*	*NextState*	*Output(Zuk)*
0	NO_ONE	NO_ONE	0
1	NO_ONE	ONE_ONE	0
0	ONE_ONE	ONE_ONE	0
1	ONE_ONE	TWO_ONE	0
0	TWO_ONE	NO_ONE	0
1	TWO_ONE	THREE_ONE	1
0	THREE_ONE	NO_ONE	0
1	THREE_ONE	THREE_ONE1	

13. **使用**always**语句描述**JK**触发器的行为功能。**
14. **描述如下电路行为：该电路在每一个时钟下跳沿（负沿）检查输入数据，当输入数据** *Usg* **为**1011**时，输出***Asm***被置为**1**。**
15. **描述多数逻辑电路行为。输入为** 12**位的向量。如果其中** 1**的数量超过**0**的数量，输出设置为** 1**。当***Data_Ready***为**1**时，才对输入数据进行检查。**

第9章 结构建模

本章讲述Verilog HDL中的结构建模方式。结构建模方式用以下三种实例语句描述：

- Gate实例语句
- UDP实例语句
- Module实例语句

第5章和第6章已经讨论了门级建模方式和UDP建模方式，本章讲述模块实例语句。

9.1 模块

Verilog HDL中，基本单元定义成模块形式，如下所示：

```
module module_name(port_list);
  Declarations_and_Statements
endmodule
```

端口队列*port_list*列出了该模块通过哪些端口与外部模块通信。

9.2 端口

模块的端口可以是输入端口、输出端口或双向端口。缺省的端口类型为线网类型（即wire类型）。但是，端口可被显式地指定为线网。输出或输入输出端口能够被重新声明为reg型寄存器。无论是在线网说明还是寄存器说明中，线网或寄存器必须与端口说明中指定的长度相同。下面是一些端口说明实例。

```
module Micro (PC, Instr, NextAddr);
  //端口说明
  input [3:1] PC;
  output [1:8] Instr;
  inout [16:1] NextAddr;

  //重新说明端口类型：
  wire [16:1] NextAddr; //该说明是可选的，但如果指定了，就必须与它的端口说明保持相同长度。

  reg [1:8] Instr;
  //Instr已被重新说明为reg类型，因此它能在always 语句或在initial语句中赋值。
  ...
endmodule
```

9.3 模块实例语句

一个模块能够在另外一个模块中被引用，这样就建立了描述的层次。模块实例语句形式如下：

```
module_name instance_name(port_associations);
```

信号端口可以通过位置或名称关联；但是关联方式不能够混合使用。端口关联形式如下：

```
port_expr                    //通过位置。
.PortName (port_expr)        //通过名称。
```

*port_expr*可以是以下的任何类型：

1) 标识符（reg或net）

2) 位选择

3) 部分选择

4) 上述类型的合并

5) 表达式（只适用于输入端口）

在位置关联中，端口表达式按指定的顺序与模块中的端口关联。在通过名称实现的关联中，模块端口和端口表达式的关联被显式地指定，因此端口的关联顺序并不重要。下例使用两个半加器模块构造全加器；逻辑图如图 9-1所示。

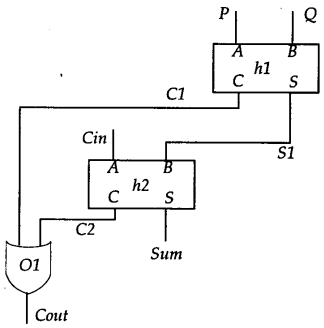


图9-1 使用两个半加器模块构造的全加器

```
module HA(A,B,S,C);
  input A,B;
  output S, C;
  parameter AND_DELAY = 1, XOR_DELAY = 2;

  assign #XOR_DELAY  S = A ^ B;
  assign #AND_DELAY  C = A & B;
endmodule

module FA(P, Q, Cin, Sum, Cout);
  input P, Q, Cin;
  output Sum, Cout;
  parameter OR_DELAY = 1;
  wire S1, C1, C2;

  //两个模块实例语句
  HA h1 (P, Q, S1, C1);  //通过位置关联。
  HA h2 (.A(Cin), .S(Sum), .B(S1), .C(C2)); //通过端口与信号的名字关联。
  //门实例语句：
  or #OR_DELAY O1 (Cout, C1, C2);
endmodule
```

在第一个模块实例语句中，*HA*是模块的名字，*h1*是实例名称，并且端口按位置关联，即信号*P*与模块（*HA*）的端口*A*连接，信号*Q*与端口*B*连接，*S1*与*S*连接，*C1*与模块端口*C*连接。在第二个实例中，端口按名称关联，即模块（ *HA*）和端口表达式间的连接是显示地定义的。

下例是使用不同端口表达式形式的模块实例语句。

```
Micro M1 (UdIn[3:0], {WrN, RdN}, Status[0], Status[1],
         & UdOut [0:7], TxData);
```

这个实例语句表示端口表达式可以是标识符（ *TxData*）、位选择（*Status*[0]）、部分位选择（*UdIn*[3:0]）、合并（{*WrN*,*RdN*}）或一个表达式（& *udOut*[0:7]）；表达式只能够连接到输入端口。

9.3.1 悬空端口

在实例语句中，悬空端口可通过将端口表达式表示为空白来指定为悬空端口，例如：

```
DFF d1 (.Q(QS), .Qbar(), .Data(D),
        .Preset(), .Clock(CK));   /名称对应方式。

DFF d2 (QS, , D, , CK);//位置对应方式。
//输出端口Qbar悬空。
//输入端口Preset打开，其值设定为z。
```

在这两个实例语句中，端口*Qbar*和*Preset*悬空。

模块的输入端悬空，值为高阻态z。模块的输出端口悬空，表示该输出端口废弃不用。

9.3.2 不同的端口长度

当端口和局部端口表达式的长度不同时，端口通过无符号数的右对齐或截断方式进行匹配。例如：

```
module Child(Pba, Ppy);
  input [5:0] Pba;
  output [2:0] Ppy;
  ...
endmodule

module Top;
  wire [1:2] Bdl;
  wire [2:6] Mpr;

  Child C1(Bdl, Mpr);
endmodule
```

在对*Child*模块的实例中，*Bdl*[2]连接到*Pba*[0]，*Bdl*[1] 连接到*Pba*[1]，余下的输入端口*Pba*[5]、*Pba*[4]和*Pba*[3]悬空，因此为高阻态z。与之相似，*Mpr*[6]连接到*Ppy*[0]，*Mpr*[5]连接到*Ppy*[1]，*Mpr*[4] 连接到*Ppy*[2]。参见图9-2。

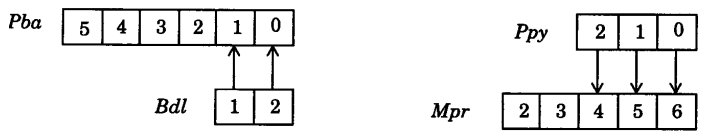


图9-2 端口匹配

9.3.3 模块参数值

当某个模块在另一个模块内被引用时，高层模块能够改变低层模块的参数值。模块参数值的改变可采用下述两种方式：

1) 参数定义语句（defparam）；

2) 带参数值的模块引用。

1. 参数定义语句

参数定义语句形式如下：

```
defparam hier_path_name1= value1,
         hier_path_name2= value2, ...;
```

较低层模块中的层次路径名参数可以使用如下语句显式定义（层次路径名在下一章中讲

述）。下面是一个例。模块*FA*和*HA*已在本节前面描述过。

```
module TOP (NewA,NewB,NewS,NewC);
  input NewA,NewB;
  output NewS,NewC;
  defparam  Ha1.XOR_DELAY = 5,
              //实例Ha1中的参数XOR_DELAY。
            Ha1.AND_DELAY = 2;
              //实例Ha1中参数的AND_DELAY。

HA Hal (NewA, NewB, NewS,NewC);
endmodule
module TOP2 (NewP, NewQ, NewCin,NewSum,NewCout);
  input NewP, NewQ, NewCin;
  output NewSum,NewCout;
  defparam  Fa1.h1.XOR_DELAY = 2,
              //实例Fa1的实例h1中的参数XOR_DELAY。
            Fa1.h1.AND_DELAY = 3,
              //实例Fa1的实例h1中的参数AND_DELAY。
            Fa1.OR_DELAY = 3;
              //实例Fa1中的参数OR_DELAY)
  FA Fa1 (NewP, NewQ, NewCin,NewSum,NewCout);
endmodule
```

2. 带参数值的模块引用

在这种方法中，模块实例语句自身包含有新的参数值。下面的例子在前几节中也出现过，本例中采用带参数的模块引用方式。

```
module TOP3 (NewA,NewB,NewS,NewC);
  input NewA,NewB;
  output NewS,NewC;

  HA #(5,2) Ha1(NewA,NewB,NewS,NewC);
    //第1个值5赋给参数AND_DELAY，该参数在模块HA中说明。
    //第2个值2赋给参数 XOR_DELAY，该参数在模块HA中说明。
endmodule

module TOP4(NewP,NewQ,NewCin,NewCout);
  input NewP,NewQ,NewCin;
  output NewSum,NewCount;
  defparam Fa1.h1.XOR_DELAY = 2,    //实例Fa1中实例h1的参数XOR_DELAY。
           Fa1.h1.AND_DELAY = 3;   //实例Fa1中实例h1的参数AND_DELAY。

  FA #(3) Fa1(NewP,NewQ,NewCin,NewCout);
    //值3是参数OR_DELAY的新值。
endmodule
```

模块实例语句中参数值的顺序必须与较低层被引用的模块中说明的参数顺序匹配。在模块*TOP3*中，*AND_DELAY*已被设置为5，*XOR_DELAY*已被设置为2。

模块*TOP3*和*TOP4*解释说明了带参数的模块引用只能用于将参数值向下传递一个层次（例如，*OR_DELAY*），但是参数定义语句能够用于替换层次中任意一层的参数值。

应注意到：在带参数的模块引用中，参数的指定方式与门级实例语句中时延的定义方式

相似；但由于对复杂模块的引用时，其实例语句不能像对门实例语句那样指定时延，故此处不会导致混淆。

参数值还可以表示长度。下面是通用的M×N乘法器建模的实例。

```
module Multiplier(Opd_1,Opd_2,Result);
  parameter EM = 4,EN = 2;  //默认值
  input [EM:1] Opd_1;
  input [EN:1] Opd_2;
  output [EM+EN:1] Result;

  assign Result = Opd_1 * Opd_2;
endmodule
```

这个带参数的乘法器可在另一个设计中使用，下面是8×6乘法器模块的带参数引用方式：

```
wire [1:8] Pipe_Reg;
wire [1:6] Dbus;
wire [1:14] Addr_Counter;
...
Multiplier #(8,6) M1(Pipe_Reg,Dbus,Addr_Counter);
```

第1个值8指定了参数*EM*的新值，第2个值6指定了参数*EN*的新值。

9.4 外部端口

在迄今为止所见到的模块定义中，端口表列举出了模块外部可见的端口。例如，

```
module Scram_A(Arb,Ctrl,Mem_Blk,Byte);
  input[0:3] Arb;
  input Ctrl;
  input [8:0] Mem_Blk;
  output [0:3] Byte;
  ...
endmodule
```

Arb、*Ctrl*、*Mem_Blk*和*Byte*为模块端口。这些端口同时也是外部端口，即在实例中，当采用名称关联方式时，外部端口名称用于指定相互连接。下面是模块*Scram_A*的实例。

```
Scram_A SX(.Byte(B1),.Mem_Blk(M1),.Ctrl(C1),.Arb(A1));
```

在模块*Scram_A*中，外部端口名称隐式地指定。Verilog HDL中提供显式方式指定外部端口名称。这可以通过按如下形式指定一个端口来完成：

```
.external_port_name(internal_port_name)
```

下面是同一个例子，只不过是显式地指定外部端口。

```
module Scram_B (.Data(Arb),.Control(Ctrl),
                .Mem_Word(Mem_Blk),.Addr(Byte));
  input [0:3] Arb;
  input Ctrl;
  input [8:0]Mem_Blk;
  output [0:3] Byte;
  ...
endmodule
```

模块*Scram_B*在此实例中指定的外部端口是*Data*、*Control*、*Mem_Word*和*Addr*。端口表显式地表明了外部端口和内部端口之间的连接。注意外部端口无需声明，但是模块的内部端口

却必须声明。外部端口在模块内不可见，但是却要在模块实例语句中使用，而内部端口因为在模块中可见，所以必须在模块中说明。在模块实例语句中，外部端口的使用如下所示：

```
Scram_B S1(.Addr(A1),.Data(D1),.Control(C1),
           .Mem_Word(M1));
```

在模块定义的端口表中，这两种概念不能混淆，即在模块定义中所有端口必须指定显式的端口名称，或者没有一个端口带有显式的端口名称。

如果模块端口通过位置连接，则模块实例语句中不能使用外部端口名称。

内部端口名称可以是标识符，也可以是下述类型的表达式：

- 位选择；
- 部分选择；
- 位选择、部分选择和标识符的合并。

例如，

```
module Scram_C (Arb[0:2],Ctrl,
                { Mem_Blk[0],Mem_Blk[1]},Byte[3]);
  input [0:3] Arb;
  input Ctrl;
  input [8:0] Mem_Blk;
  output [0:3] Byte;
  ...
endmodule
```

在*Scram_C*的模块定义中，端口表包括部分选择（ *Arb*[0:2]）、标识符（*Ctrl*）、合并（{*Mem_Blk*[0]，*Mem_Blk*[1]}）和位选择（*Byte*[3]）。在内部端口是位选择、部分选择或合并的情况下，没有隐式地指定外部端口名。因此，在这样的模块实例语句中，模块端口必须通过位置关联相互连接。例如，

```
Scram_C SYA (L1[4:6],CL,MMY[1:0],BT);
```

在这个实例语句中，端口通过位置关联相连接，因此 *L1*[4：6]连接到*Arb*[0：2]，*CL*连接到*Ctrl*，*MMY*[1]连接到*Mem_Blk*[0],*MMY*[0]连接到*Mem_Blk*[1]，*BT*连接到*Byte*[3]。

若使用端口名称关联（即当内部端口不是标识符时），必须对模块中的端口指定外部端口名。如下面的*Scram_D*模块定义所示。

```
module Scram_D (.Data(Arb[0:2]),.Control(Ctrl),
                .Mem_Word({Mem_Blk[0],Mem_Blk[1]}),
                .Addr(Byte[3]));
  input [0:3] Arb;
  input Ctrl;
  input [8:0] Mem_Blk;
  output [0:3] Byte;
  ...
endmodule
```

在*Scram_D*模块实例语句中，端口既能够使用位置连接，也能够使用名称连接，但是不能混合使用。下例的模块实例语句端口通过名称连接。

```
Scram_D SZ(.Data(L1[4:6],.Control(CL),
           .Mem_Word(MMY[1:0], .Addr(BT));
```

模块中可以只有外部端口而没有内部端口。即模块在引用中其外部端口可以悬空，不与内部信号相连。例如：

```
module Scram_E (.Data(), .Control (Ctrl),
                .Mem_Word ({ Mem_Blk[0], Mem_Blk [1]}),
                .Addr());
  input Ctrl;
  input [8:0] Mem_Blk;
  ...
endmodule
```

模块*Scram_E*有两个外部端口*Data*和*Addr*，这两个端口在使用时被悬空。

一个内部端口是否能与多个外部端口连接？Verilog HDL允许这样连接。例如，

```
module FanOut (.A(CtrlIn),.B(CondOut),.C(CondOut));
  input CtrlIn;
  output CondOut;

  assign CondOut = CtrlIn;
endmodule
```

内部端口*CondOut*与两个外部端口*B*和*C*连接，所以*CondOut*的值在*B*和*C*上都出现。

9.5 举例

下例采用结构模型描述十进制计数器。十进制计数器的逻辑图如图 9-3所示。

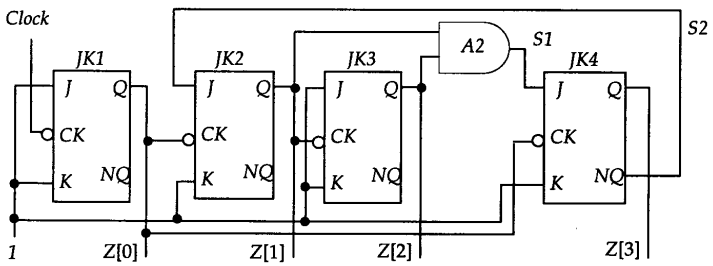


图9-3 十进制计数器

```
module Decade_Ctr (Clock,Z);
  input Clock;
  output [0:3] Z;
  wire S1,S2;

  and A1 (S1,Z[2],Z[1]); //基本门实例语句。

  // 4个模块实例语句：
  JK_FF JK1(.J(1'b1),.K(1'b1),.CK(Clock),.Q(Z[0]),.NQ()),
        JK2(.J(S2),.K(1'b1),.CK(Z[0]),.Q(Z[1]),.NQ()),
        JK3(.J(1'b1),.K(1'b1),.CK(Z[1]),.Q(Z[2]),.NQ()),
        JK4(.J(S1),.K(1'b1),.CK(Z[0]),.Q(Z[3]),.NQ(S2));
endmodule
```

注意常数作为输入端口信号的用法，以及悬空端口。

下面是另一个例子，3位可逆计数器的逻辑结构如图9-4所示，其结构描述如下：

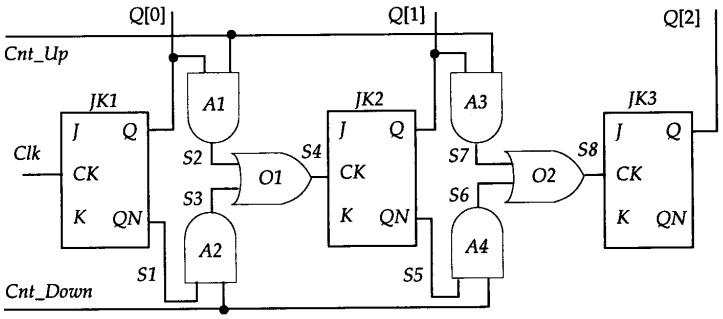
```
module Up_Down(Clk,Cnt_Up,Cnt_Down,Q);
  input Clk,Cnt_Up,Cnt_Down;
  output [0:2] Q;
```

```
  wire S1,S2,S3,S4,S5,S6,S7,S8

  JK_FF JK1(1'b1,1'b1,Clk,Q[0],S1),
        JK2(1'b1,1'b1,S4,Q[1],S5),
        JK3(1'b1,1'b1,S8,Q[2],);

  and A1(S2,Cnt_Up,Q[0]),
      A2(S3,S1,Cnt_Down),
      A3(S7,Q[1],Cnt_Up),
      A4(S6,S5,Cnt_Down);

  or O1(S4,S2,S3),
     O2(S8,S7,S6);
endmodule
```



所有触发器的J、 K输入端口与高电平相连

图9-4 位可逆计数器

习题

1.模块实例语句与门实例语句的区别是什么？

2.当端口悬空时，即端口没有被连接时，端口的值是什么？

3.对于9.3节中的模块*FA*，*OR_DELAY*值为4，*XOR_DELAY*值为7，AND_DELAY值为5，写出其结构描述形式。

4.用本章讲述的模块*FA*编写执行加法和减法的4位ALU的结构模型。

5.用5.11节中描述的*MUX4x1*模块编写16-1多路选择器的结构化模型。

6.用异步低电平复位描述通用 *N*位计数器。将通用计数器在实例语句中用作 5位计数器用测试验证程序验证这个5位计数器。

第10章 其他论题

本章讲述诸如函数、任务、层次结构、值变转储文件和编译程序指令等多种论题。

10.1 任务

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

10.1.1 任务定义

任务定义的形式如下：

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

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

```
module Has_Task;
  parameter MAXBITS = 8;

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

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

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

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

  begin
```

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

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

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

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

10.1.2 任务调用

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

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

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

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

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

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

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

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

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

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

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

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

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

存储器*RamQ*的地址被作为参数传递，而存储器本身在任务内直接引用。

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

```
module TaskWait;
  reg NoClock;

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

  initial
    GenarateWaveform (NoClock);
endmodule
```

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

10.2 函数

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

10.2.1 函数说明部分

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

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

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

```
module Function_Example
  parameter  MAXBITS = 8;

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

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

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

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

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

    Parity = Ret % 2;
  end
endfunction
```

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

10.2.2 函数调用

函数调用是表达式的一部分。形式如下：

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

以下是函数调用的例子：

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

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

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

函数的多个调用之间保持它们的值。

10.3 系统任务和系统函数

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

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

PLA建模任务和随机建模任务不在本书的讨论范围内。

10.3.1 显示任务

显示系统任务用于信息显示和输出。这些系统任务进一步分为：

- 显示和写入任务
- 探测监控任务
- 连续监控任务

1. 显示和写入任务

语法如下：

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

*task_name*是如下编译指令的一种：

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

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

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

如果没有特定的参数格式说明，缺省值如下：

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

可以用如下代码序列输出特殊字符：

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

例如：

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

上述语句输出$time、R、S、Q和QB**等值的执行结果如下：**

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

2. 探测任务

探测任务有：

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

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

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

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

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

其格式定义与显示和写入任务相同。

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

```
integer  Cool;

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

产生的输出为：

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

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

3. 监控任务

监控任务有：

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

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

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

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

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

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

可以用如下两个系统任务打开和关闭监控。

```
$monitoroff;  //禁止所有监控任务。

$monitoron;  //使能所有监控任务。
```

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

10.3.2 文件输入/输出任务

1. 文件的打开和关闭

系统函数$fopen用于打开一个文件。

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

而下面的系统任务可用于关闭一个文件：

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

这是一个关于它的用法的例子。

```
integer  Tq_File;

initial
```

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

2. 输出到文件

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

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

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

```
integer  Vec_File;

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

等到$fdisplay任务执行时，文件“div.vec”中出现下列语句：

```
The simulation time is          0
```

3. 从文件中读取数据

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

```
$readmemb      $readmemh
```

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

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

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

显式的地址可以在系统任务调用中可选地指定，例如：

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

也可以在文本文件中显式地给出地址。形式如下：

```
@address_in_hexadecimal
```

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

10.3.3 时间标度任务

系统任务

```
$printtimescale
```

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

```
$printtimescale;

$printtimescale(hier_path_to_module);
```

下面是这些系统被调用时输出的样本。

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

系统任务

```
$timeformat
```

指定%t格式定义如何报告时间信息，该任务形式如下：

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

其中*units_number*为：

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

系统任务调用

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

将显示$display任务中%t说明符的值，如下：

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

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

10.3.4 模拟控制任务

系统任务

```
$finish;
```

使模拟器退出，并将控制返回到操作系统。

系统任务

```
$stop
```

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

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

500个时间单位后，模拟停止。

10.3.5 定时校验任务

系统任务：

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

如果

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

则报告时序冲突（timing violation）；

系统调用实例如下：

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

系统任务：

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

如果

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

则报数据保持时间时序冲突。

例如：

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

系统任务$setuphold是$setup和$hold任务的结合：

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

而系统任务：

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

则检查信号的脉冲宽度限制，如果

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

则报告信号上出现脉冲宽度不够宽的时序错误。

数据事件来源于基准事件：它是带有相反边沿的基准事件，例如：

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

系统任务：

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

检查信号的周期，若

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

则报告时序错误。

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

系统任务：

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

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

```
time_of_data_event - time_of_reference_event > limit
```

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

系统任务:

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

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

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

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

系统任务：

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

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

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

如果在*Clear*为低时*Preset*发生变化，将报告时序冲突错误。

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

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

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

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

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

10.3.6 模拟时间函数

下列系统函数返回模拟时间。

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

例如

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

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

该例产生的输出如下：

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

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

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

10.3.7 变换函数

下列系统函数是数字类型变换的功能函数：

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

10.3.8 概率分布函数

函数：

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

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

例如，

```
integer Seed, Rnum;
wire Clk;

initial Seed = 12;

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

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

如果数字在取值范围内，下述模运算符可产生－ 10～+10之间的数字。

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

下面是没有显式指定种子的例子。

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

注意数字产生的顺序是伪随机排序的，即对于一个初始种子值产生相同的数字序列。

表达式：

```
{random} % 11
```

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

下列函数根据在函数名中指定的概率函数产生伪随机数。

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

$dist_normal ( seed,mean,standard_deviation,upper)

$dist_exponential ( seed,mean)

$dust_poisson ( seed,mean)

$dist_chi_square ( seed,degree_of_freedom)

$dist_t ( seed,degree_of_freedom)

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

这些函数的所有参数都必须是整数。

10.4 禁止语句

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

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

在禁止语句执行后，继续执行任务调用或被禁止的程序块的下一条语句。

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

语句3和语句4从未被执行。在禁止语句被执行后，执行语句5。又如：

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

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

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

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

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

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

```
disable Example；
```

那么在禁止语句开始执行后， *Count*的值不确定。

10.5 命名事件

考虑下述两个always语句。

```
reg Ready ,Done;

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

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

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

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

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

```
event Ready, Done;
```

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

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

命名事件上的事件能够同变量上的事件一样被监控，即使用 @机制，例如：

```
@ (Done) <动作语句>
```

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

可以用命名事件重写两类always 语句的简例：

```
event  Ready, Done

initial
  ->Done;

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

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

也可以使用事件描述状态机。下面是一个异步状态机实例：

```
event State1, State2, State3;

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

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

```
  end

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

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

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

10.6 结构描述方式和行为描述方式的混合使用

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

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

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

下面的实例采用不同的风格进行硬件建模：

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

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

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

模块包含内置逻辑门（结构化组件）和连续赋值（数据流方式）的混合描述形式。

10.7 层次路径名

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

1) 模块实例化
2) 任务定义
3) 函数定义
4) 命名程序块

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

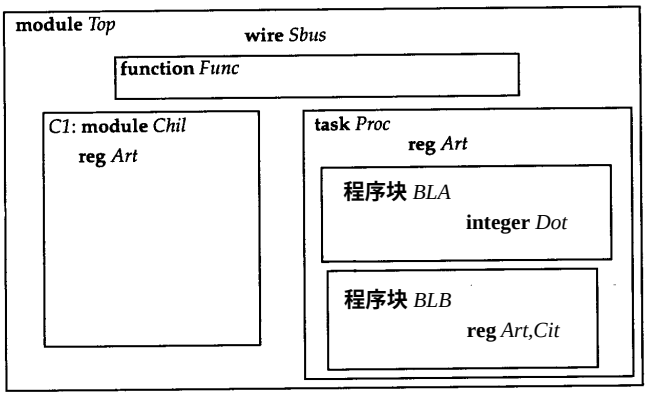


图10-1 模块层次

```
module Top;
  wire  Sbus;

  function Func...
    ...
  endfunction

  task Proc
   ...
   reg Art;

   begin : BLA
     integer  Dot;
```

```
      ...
    end

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

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

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

本例中的层次名为：

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

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

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

```
module_instance_name.variable_name
```

对于向下路径引用，模块实例必须与较低层模块在同一层。例如：

```
module Top;
  wire Sbus;

  Chil Cl (. . .);        //一个模块实例。

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

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

10.8 共享任务和函数

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

```
function SignedPlus;
```

```
  . . .
endfunction

function SignedMinus;
 . . .
endfunction

task PresetClear;
  . . .
endtask
```

下面是在模块中使用文件的方式：

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

  //包含共享函数的定义。
  `include "share.h"

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

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

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

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

  function SignedMinus;
    . . .
  endfunction

  task PresetClear;
    . . .
  endtask
endmodule
```

下面是在其它模块中引用共享函数的方法。

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

```
  reg [0:3] Z;

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

10.9 值变转储文件

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

下面的系统任务用于创建和将信息导入 VCD文件。

1) $dumpfile：本系统任务指定转储文件名。

例如：

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

2) $dumpvars：本系统任务指定哪些变量值变化时转储进转储文件。

```
$dumpvars;
  //无参数，它指定在设计中转储所有变量。

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

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

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

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

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

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

3) $dumpoff：本系统任务促使转储任务被挂起。

```
$dumpoff;
```

4) $dumpon：本系统任务促使所有转储任务被挂起。语法如下：

```
$dumpon;
```

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

```
$dumpall
```

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

例如，

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

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

```
$dumpflush;
```

10.9.1 举例

下面是在5～12之间计数的可逆计数器的例子：

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

  initial Count = 'd5;

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

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

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

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

    CountUpDown C1(Clock, Cnt_Out, UpDn);

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

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

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

10.9.2 VCD文件格式

VCD文件是ASCII文件。VCD文件包含如下信息：

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

VCD文件实例如图10-2所示。

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

图10-2 VCD文件

10.10 指定程序块

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

1) 指定源和目的之间的通路。

2) 为这些通路分配时延。

3) 对模块进行时序校验。

指定程序块出现在模块说明内。形式如下：

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

specparam(或指定参数)说明指定程序块内使用的参数。实例如下：

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

在指定程序块内能够描述下述三类模块通路：

- 简单通路
- 边沿敏感通路
- 与状态有关的通路

可以使用如下两种形式说明简单通路。

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

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

以下是一些实例。

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

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

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

在边沿敏感通路中，通路描述与源的边沿相关。例如，

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

与状态有关的通路在某些条件为真时指定通路时延。例如，

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

下面是可在指定程序块内使用的时序验证系统任务。

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

下面是指定程序块的实例。

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

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

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

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

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

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

也可以按如下形式使用：

```
PATHPULSE$input_terminal$output_terminal
```

PATHPULSE $参数为特殊通路指定脉冲界限。

下面是指定程序块的实例。

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

10.11 强度

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

10.11.1 驱动强度

驱动强度可以在如下情况中指定：

1) 在线网说明中带赋值的线网。

2) 原语门实例中的输出端口。

3) 在连续赋值语句中。

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

```
(strength_for_1, strength_for_0)
```

值的顺序并不重要。对于值1的赋值，只允许如下的信号驱动强度：

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

对于值0的赋值，允许如下的信号驱动强度：

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

缺省的信号驱动强度定义为(strong0, strong1)。例如：

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


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


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

线网的驱动强度可以在显示任务中用 %v格式定义输出。例如：

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

结果为：

```
Prq is Wel
```

10.11.2 电荷强度

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

- 小型
- 中型（如果没有特别强调，则为缺省值）
- 大型

此外，三态寄存器线网存储的电荷衰退时间也可被指定。实例如下：

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

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

10.12 竞争状态

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

下面用一个简例解释说明使用非阻塞性赋值的零时延情况。

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

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

下面的另一例子显示由于事件排序产生的竞争状态。

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

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

  end

assign Cot = Pal;
```

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

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

因此，在处理零时延赋值时要格外注意。下面是竞争状态的另一实例。

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

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

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

的取值(第1条always语句中的赋值)。

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

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

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

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

习题

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

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

    -GetBaseName：返回文件名(即*fileA*)

    假定路径名中的最大字符数为512个字符。

第11章 验 证

本章介绍了如何编写测试验证程序 (test bench)。测试验证程序用于测试和验证设计的正确性。Verilog HDL提供强有力的结构来说明测试验证程序。

11.1 编写测试验证程序

测试验证程序有三个主要目的：

1) 产生模拟激励(波形)。

2) 将输入激励加入到测试模块并收集其输出响应；

3) 将响应输出与期望值进行比较。

Verilog HDL提供了大量的方法以编写测试验证程序。在本章中，我们将对其中的某些方法进行探讨。典型的测试验证程序形式如下：

```
module Test_Bench;
  //通常测试验证程序没有输入和输出端口。
  Local_reg_and_net_declarations
  Generate_waveforms_using_initial_&_always_statements
  Instantiate_module_under_test
  Monitor_output_and_compare_with_expected_values
endmodule
```

测试中，通过在测试验证程序中进行实例化，激励自动加载于测试模块。

11.2 波形产生

有两种产生激励值的主要方法：

1) 产生波形，并在确定的离散时间间隔加载激励。

2) 根据模块状态产生激励，即根据模块的输出响应产生激励。

通常需要两类波形。一类是具有重复模式的波形，例如时钟波形，另一类是一组指定的值确定的波形。

11.2.1 值序列

产生值序列的最佳方法是使用initial语句。例如:

```
initial
  begin
    Reset = 0;
    #100 Reset = 1;
    #80 Reset = 0;
    #30 Reset = 1;
end
```

产生的波形如图11-1所示。Initial语句中的赋值语句用时延控制产生波形。此外，语句内时延也能够按如下实例所示产生波形。

```
initial
  begin
    Reset = 0;
    Reset = #100 1;
    Reset = #80 0;
    Reset = #30 1;
end
```

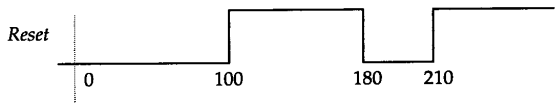


图11-1 使用initial语句产生的波形

因为使用的是阻塞性过程赋值，上面语句中的时延是相对时延。如果使用绝对时延，可用带有语句内时延的非阻塞性过程性赋值，例如，

```
initial
  begin
    Reset <= 0;
    Reset <= #100 1;
    Reset <= #180 0;
    Reset <= #210 1;
end
```

这三个initial语句产生的波形与图11-1中所示的波形一致。

为重复产生一个值序列，可以使用always语句替代initial语句，这是因为initial语句只执行一次而always语句会重复执行。下例的always语句所产生的波形如图11-2所示。

```
parameter REPEAT_DELAY= 35;
integer CoinValue;

always
  begin
    CoinValue = 0;
    #7 CoinValue = 25;
    #2 CoinValue = 5;
    #8 CoinValue = 10;
    #6 CoinValue = 5;
    #REPEAT_DELAY;
end
```

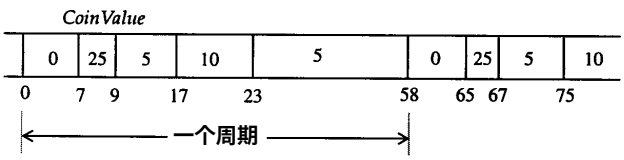


图11-2 使用always语句产生的重复序列

11.2.2 重复模式

重复模式的生成通过使用如下形式的连续赋值形式加以简化：

```
assign # (PERIOD/2)   Clock = ~ Clock;
```

但是这种做法并不完全正确。问题在于 *Clock*是一个线网(只有线网能够在连续赋值中被赋值)，它的初始值是z，并且，z等于x，~x等于x。因此*Clock*的值永远固定为值x。

现在需要一种初始化*Clock*的方法。可以用initial语句实现。

```
initial
  Clock = 0;
```

但是现在Clock必须是寄存器数据类型(因为只有寄存器数据类型能够在 initial语句中被赋值)，因此连续赋值语句需要被变换为 always语句。下面是一个完整的时钟产生器模块。

```
module Gen_Clk_A (Clk_A);
  output Clk_A;
  reg Clk_A;
  parameter tPERIOD = 10;

  initial
    Clk_A = 0;

  always
    # (tPERIOD/2)  Clk_A = ~ Clk_A;
endmodule
```

图11-3显示了该模块产生的时钟波形。

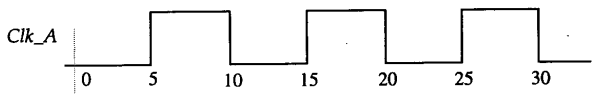


图11-3 周期性的时钟波形

下面给出了产生周期性时钟波形的另一种可选方式。

```
module Gen_Clk_B (Clk_B);
  output Clk_B;
  reg Start;

  initial
    begin
      Start = 1;
      #5 Start = 0;
    end

nor #2 (Clk_B, Start, Clk_B);
endmodule
//产生一个高、低电平宽度均为2的时钟。
```

initial语句将*Start*置为1，这促使或非门的输出为 0(从x值中获得)。5个时间单位后，在*Start*变为0时，或非门反转产生带有周期为 4个时间单位的时钟波形。产生的波形如图 11-4所示。

如果要产生高低电平持续时间不同的时钟波形，可用 always语句建立模型，如下所示：

```
module Gen_Clk_C (Clk_C);
  parameter tON = 5, tOFF = 10;
  output Clk_C;
```

```
  reg Clk_C;

  always
    begin
      # tON    Clk_C= 0;
      # tOFF   Clk_C= 1;
    end
endmodule
```

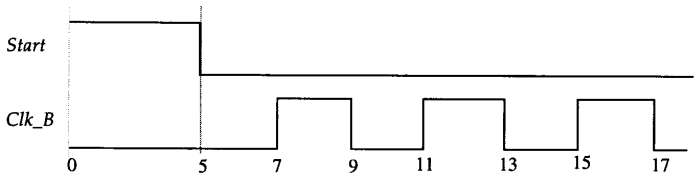


图11-4 受控时钟

因为值0和1被显式地赋值，在这种情况下不必使用 initial语句。图11-5显示了这一模块生成的波形。

为在初始时延后产生高低电平持续时间不同的时钟，可以在 initial语句中使用forever循环语句。

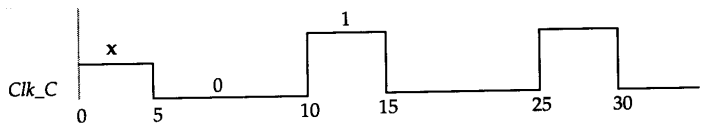


图11-5 高低电平持续时间不同的时钟

```
module Gen_Clk_D (Clk_D);
  output Clk_D;
  reg Clk_D;
  parameter START_DELAY = 5, LOW_TIME = 2, HIGH_TIME = 3;

  initial
    begin
      Clk_D = 0;
      # START_DELAY ;

      forever
        begin
          # LOW_TIME ;
          Clk_D = 1;
          # HIGH_TIME;
          Clk_D = 0;
        end
    end
endmodule
```

上面模块所产生的波形如图11-6所示。

为产生确定数目的时钟脉冲，可以使用 repeat循环语句。下面带参数的时钟模块产生一定数目的时钟脉冲数列，时钟脉冲的高低电平持续时间也是用参数表示的。

```
module Gen_Clk_E(Clk_E);
  output Clk_E;
  reg Clk_E;
  parameter Tburst = 10, Ton = 2, Toff = 5;

  initial
    begin
      Clk_E = 1'b0;

      repeat(Tburst)
        begin
        # Toff Clk_E= 1'b1;
        # Ton Clk_E= 1'b0;
        end
    end
endmodule
```

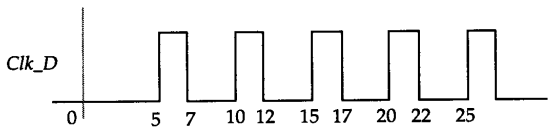


图11-6 带有初始时延的时钟

模块*Gen_Clk_E*在具体应用时，参数*Tburst*、*Ton*和*Toff*可带不同的值。

```
module Test;
  wire Clk_Ea, Clk_Eb, Clk_Ec;

  Gen_Clk_E G1(Clk_Ea);
    //产生10个时钟脉冲，高、低电平持续时间分别为2个和5个时间单位。

  Gen_Clk_E  # (5, 1, 3)  C2(Clk_Eb);
    //产生5个时钟脉冲，高、低电平持续时间分别为1个和3个时间单位。

  Gen_Clk_E  # (25, 8, 10)  C3(Clk_Ec);
    //产生25个时钟脉冲，高、低电平持续时间分别为8个和10个时间单位。
endmodule
```

*Clk_Eb*的波形如图11-7所示。

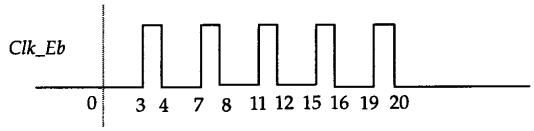


图11-7 确定数目的时钟脉冲

可用连续赋值产生一个时钟的相移时钟。下述模块产生的两个时钟波形如图11-8所示。一个时钟是另一个时钟的相移时钟。

```
module Phase (Master_Clk, Slave_Clk);
  output Master_Clk, Slave_Clk;
  reg Master_Clk;
```

```
  wire Slave_Clk;
  parameter tON = 2, tOFF = 3, tPHASE_DELAY= 1;

  always
    begin
      #tON Master_Clk= 0;
      #tOFF Master_Clk= 1;
  end

  assign #tPHASE_DELAY Slave_Clk= Master_Clk;
endmodule
```

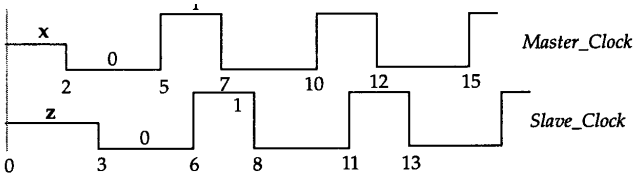


图11-8 相移时钟

11.3 测试验证程序实例

11.3.1 解码器

下面是2-4解码器和它的测试验证程序。任何时候只要输入或输出信号的值发生变化，输出信号的值都会被显示输出。

```
`timescale 1ns / 1ns
module Dec2x4 (A, B, Enable, Z);
  input A, B, Enable;
  output [0:3] Z;
  wire Abar, Bbar;

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

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

module Dec_Test;
  reg Da, Db, Dena;
  wire [0:3] Dz;

  //被测试的模块：
  Dec2x4 D1(Da, Db, Dena, Dz);
```

```
  //产生输入激励：
  initial
    begin
      Dena = 0;
      Da = 0;
      Db = 0;
      #10 Dena = 1;
      #10 Da = 1;
      #10 Db = 1;
      #10 Da = 0;
      #10 Db = 0;
      #10 $stop;
    end

  //输出模拟结果：
  always
    @ (Dena or Da or Db or Dz)
      $display ("At time %t, input is %b%b%b, output is %b"
            $time, Da, Db, Dena, Dz);
endmodule
```

下面是测试模块执行时产生的输出。

```
At time                    4, input is 000, output is 1111
At time                   10, input is 001, output is 1111
At time                   13, input is 001, output is 0111
At time                   20, input is 101, output is 0111
At time                   23, input is 101, output is 0101
At time                   26, input is 101, output is 1101
At time                   30, input is 111, output is 1101
At time                   33, input is 111, output is 1100
At time                   36, input is 111, output is 1110
At time                   40, input is 011, output is 1110
At time                   44, input is 011, output is 1011
At time                   50, input is 001, output is 1011
At time                   54, input is 001, output is 0111
```

11.3.2 触发器

下例是主从D触发器及其测试模块。

```
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, NotQ),
    ND6 (Y2, NotY, NotQ),
    ND7 (Q, Qbar, Y1),
    ND8 (Qbar, Y2, Q);
endmodule

module Test;
  reg D, C;
  wire Q, Qb;

  MSDFF M1(D, C, Q, Qb);

  always
    #5 C = ~C;

  initial
    begin
      D = 0;
      C = 0;
      #40 D = 1;
      #40 D = 0;
      #40 D = 1;
      #40 D = 0;
      $stop;
    end

  initial
    $ monitor ("Time = %t ::", $time,"C=%b, D=%b, Q=%b,
            Qb=%b", C,D, Q, Qb );
endmodule
```

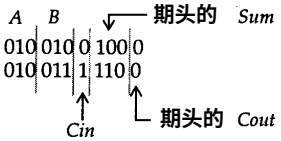
在此测试验证模块中，触发器的两个输入和两个输出结果均设置了监控，故只要其中任何值发生变化就输出指定变量的值。下面是执行产生的输出结果。

```
Time=                  0:: C=0, D=0, Q=x, Qb=x
Time=                  5:: C=1, D=0, Q=x, Qb=x
Time=                 10:: C=0, D=0, Q=0, Qb=1
Time=                 15:: C=1, D=0, Q=0, Qb=1
Time=                 20:: C=0, D=0, Q=0, Qb=1
Time=                 25:: C=1, D=0, Q=0, Qb=1
Time=                 30:: C=0, D=0, Q=0, Qb=1
Time=                 35:: C=1, D=0, Q=0, Qb=1
Time=                 40:: C=0, D=1, Q=0, Qb=1
Time=                 45:: C=1, D=1, Q=0, Qb=1
Time=                 50:: C=0, D=1, Q=1, Qb=0
Time=                 55:: C=1, D=1, Q=1, Qb=0
Time=                 60:: C=0, D=1, Q=1, Qb=0
Time=                 65:: C=1, D=1, Q=1, Qb=0
Time=                 70:: C=0, D=1, Q=1, Qb=0
Time=                 75:: C=1, D=1, Q=1, Qb=0
Time=                 80:: C=0, D=0, Q=1, Qb=0
```

```
Time=                 85:: C=1, D=0, Q=1, Qb=0
Time=                 90:: C=0, D=0, Q=0, Qb=1
Time=                 95:: C=1, D=0, Q=0, Qb=1
Time=                100:: C=0, D=0, Q=0, Qb=1
Time=                105:: C=1, D=0, Q=0, Qb=1
Time=                110:: C=0, D=0, Q=0, Qb=1
Time=                115:: C=1, D=0, Q=0, Qb=1
Time=                120:: C=0, D=1, Q=0, Qb=1
Time=                125:: C=1, D=1, Q=0, Qb=1
Time=                130:: C=0, D=1, Q=1, Qb=0
Time=                135:: C=1, D=1, Q=1, Qb=0
Time=                140:: C=0, D=1, Q=1, Qb=0
Time=                145:: C=1, D=1, Q=1, Qb=0
Time=                150:: C=0, D=1, Q=1, Qb=0
Time=                155:: C=1, D=1, Q=1, Qb=0
```

11.4 从文本文件中读取向量

可用$readmemb系统任务从文本文件中读取向量(可能包含输入激励和输出期望值)。下面为测试3位全加器电路的例子。假定文件“test.vec”包含如下两个向量。



向量的前三位对应于输入A，接下来的三位对应于输入B，再接下来的位是进位，八到十位是期望的求和结果，最后一位是期望进位值的输出结果。下面是全加器模块和相应的测试验证程序。

```
module Adder1Bit (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;

  assign Sum = (A ^ B ) ^ Cin;
  assign Cout = (A ^ B )| (A & Cin) | (B & Cin);
endmodule

module Adder3Bit(First, Second, Carry_In,Sum_Out, Carry_Out);
  input [0:2] First, Second;
  input Carry_In;
  output [0:2] Sum_Out;
  output Carry_Out;
  wire [0:1]Car;

  Adder1Bit
    A1 (First[2], Second[2], Carry_In,Sum_Out[2], Car[1]),
    A2 (First[1], Second[1], Car[1], Sum_Out [1],Car[0]),
    A3 (First[0], Second[0], Car[0], Sum_Out [0], Carry_Out);
endmodule

module TestBench;
```

```
    parameter BITS = 11, WORDS= 2;
    reg [1:BITS]Vmem [1:WORDS];
    reg[0:2]A,B,Sum_Ex;
    reg Cin, Cout_Ex;
    integer J;
    wire [0:2] Sum;
    wire Cout;

    //被测试验证的模块实例。
    Adder3Bit   F1 (A, B, Cin, Sum, Cout);

    initial
      begin
        $readmemb ("test.vec", Vmem);

        for (J = 1; J <= WORDS; J = J + 1)
          begin
            {A, B, Cin, Sum_Ex, Cout_Ex} = Vmem [J];
            #5;  //延迟5个时间单位等待电路稳定。

            if ((Sum ! = = Sum_Ex) | | (Cout ! = = Cout_Ex))
              $display ("****Mismatch on vector %b ****", Vmem [J]);
            else
              $display ("No mismatch on vector %b", Vmem [J]);
          end
      end
  endmodule
```

测试模块中首先定义存储器 *Vmem*，字长对应于每个向量的位数，存储器字数对应于文件中的向量数。系统任务 $**readmemb**从文件“ test.vec ”中将向量读入存储器 *Vmem*中。for循环通过存储器中的每个字，即每个向量，将这些向量应用于待测试的模块，等待模块稳定并探测模块输出。条件语句用于比较期望输出值和监测到的输出值。如果发生不匹配的情况，则输出不匹配消息。下面是以上测试验证模块模拟执行时产生的输出。因为模型中不存在错误，因此没有报告不匹配情形。

```
No mismatch on vector 01001001000
No mismatch on vector 01001111100
```

11.5 向文本文件中写入向量

在上节的模拟验证模块实例中，我们看到值如何被打印输出。设计中的信号值也能通过如$**fdisplay**、$fmonitor和$fstrobe等具有写文件功能的系统任务输出到文件中。下面是与前一节中相同的测试验证模块实例，本例中的验证模块将所有输入向量和观察到的输出结果输出到文件“ mon.Out ”中。

```
  module F_Test_Bench;
    parameter BITS = 11, WORDS= 2;
    reg [1:BITS] Vmem [1:WORDS];
    reg [0:2] A, B, Sum_Ex;
    reg Cin,Cout_Ex
```

```
  integer J;
  wire [0:2] Sum;
  wire Cout;

  //待测试验证模块的实例。
  Adder3Bit   F1 (A, B, Cin, Sum, Cout);

  initial
    begin: INIT_LABLE
      integer Mon_Out_File;

      Mon_Out_File= $fopen ("mon.out");
      $readmemb ("test.vec", Vmem);

      for (J = 1; J <= WORDS; J = J + 1)
        begin
          {A, B, Cin, Sum_Ex, Cout_Ex} = Vmem [J];
          #5;  //延迟5个时间单位，等待电路稳定。

          if ((Sum ! = = Sum_Ex) | | (Cout ! = = Cout_Ex))
            $display ("****Mismatch on vector %b ****", Vmem [J]);
          else
            $display ("No mismatch on vector %b", Vmem [J]);

          //将输入向量和输出结果输入到文件：
          $fdisplay (Mon_Out_File,"Input = %b%b%b, Output = %b%b",
                  A, B, Cin, Sum, Cout);
        end
      $fclose (Mon_Out_File);
    end
endmodule
```

下面是模拟执行后文件“ mon.out ”包含的内容。

```
Input = 0100100, Output = 1000
Input = 0100111, Output = 1100
```

11.6 其他实例

11.6.1 时钟分频器

下面是应用波形方法的完整测试验证程序。待测试的模块名为 *Div*。输出响应写入文件以便于以后进行比较。

```
module Div (Ck, Reset, TestN, Ena);
  input Ck, Reset, TestN;
  output Ena;
  reg [0:3] Counter;

  always
    @ (posedge Ck) begin
      if (~Reset)
```

```
        Counter = 0;
      else
        begin
          if (~ TestN)
            Counter = 15;
          else
            Counter = Counter + 1
        end
      end
  assign Ena = (Counter = = 15) ? 1: 0;
endmodule

module Div_TB;
  integer Out_File;
  reg Clock, Reset, TestN;
  wire Enable;

  initial
    Out_File = $fopen ("out.vec");

  always
    begin
      #5 Clock = 0;
      #3 Clock = 1;
    end

  Div D1 (Clock, Reset, TestN, Enable);

  initial
    begin
      Reset = 0;
      #50 Reset = 1;
    end

initial
  begin
    TestN = 0;
    #100 TestN = 1;
    #50 TestN = 0;
    #50 $fclose (Out_File);
    $finish;          /模拟结束。
  end
//将使能输出信号上的每个事件写入文件。

  initial
    $fmonitor (Out_File,"Enable changed to %b at time %t", Enable, $time);
endmodule
```

模拟执行后，文件“ out.vec **”所包含的输出结果如下：**

```
Enable changed to x at time            0
Enable changed to 0 at time            8
```

```
Enable changed to 1 at time          56
Enable changed to 0 at time         104
Enable changed to 1 at time         152
```

11.6.2 阶乘设计

本例介绍产生输入激励的另一种方式，在该方式中，根据待测模块的状态产生相应的输出激励。该输出激励产生方式对有穷状态自动机（ FSM）的模拟验证非常有效，因为状态机的模拟验证需根据各个不同的状态产生不同的输入激励。设想一个用于计算输入数据阶乘(factorial)的设计。待测试模块与测试验证模块之间的握手机制如图 11-9所示。

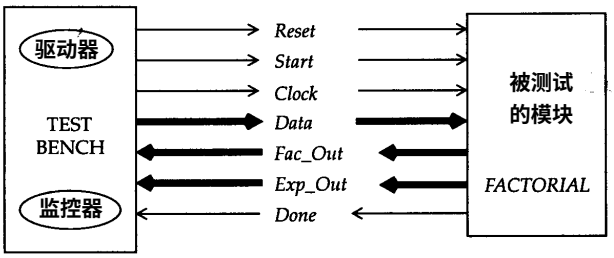


图11-9 测试验证模块与待测试模块间的握手机制

模块的输入信号 *Reset*将阶乘模型复位到初始状态，在加载输入数据 *Data*后，*Start*信号被置位；计算完成时，对输出 *Done*置位，表明计算结果出现在输出 *Fac_Out*和*Exp_Out*上。阶乘结果值为*Fat_Out* $*2^{Exp\_Out}$，测试验证模块在 *Data*上提供从值1开始递增到20的输入数据。测试验证模块加载数据，对 *Start*信号置位并等待*Done*信号有效，然后加载于下一输入数据。若输出结果不正确，即打印错误信息。阶乘模块及其测试验证模块描述如下：

```
`timescale 1ns / 1ns
module FACTORIAL(Reset, StartSig, Clk, Data, Done,
                 FacOut, ExpOut);
  input Reset, StartSig, Clk;
  input [4:0] Data;
  output Done;
  output [7:0] FacOut, ExpOut;

  reg Stop;
  reg [4 : 0] InLatch;
  reg [7:0] Exponent, Result;
  integer I;

  initial Stop = 1;

  always
    @ (posedge Clk) begin
      if (( StartSig = = 1) && (Stop = = 1) && (Reset = = 1))
        begin
          Result = 1;
          Exponent = 0;
          InLatch = Data;
```

```
            Stop = 0;
          end
        else
          begin
            if (( InLatch > 1) && (Stop = = 0)
              begin
                Result = Result * InLatch;
                InLatch = InLatch- 1;
              end

            if (InLatch < 1)
              Stop = 1;
        //标准化：
          for (I = 1; I <= 5; I = I + 1)
            if (Result >256)
              begin
                Result = Result / 2;
                Exponent = Exponent+ 1;
              end
          end
    end

    assign Done = Stop;
    assign FacOut = Result;
    assign ExpOut = Exponent;
  endmodule

  module FAC_TB;
    parameter IN_MAX = 5, OUT_MAX = 8;
    parameter RESET_ST = 0, START_ST = 1, APPL_DATA_ST = 2,
              WAIT_RESULT_ST = 3;
    reg Clk, Reset, Start;
    wire Done;
    reg [IN_MAX-1 : 0] Fac_Out, Exp_Out;
    integer Next_State;
    parameter MAX_APPLY = 20;
    integer Num_Applied;

    initial
      Num_Applied= 1;

    always
      begin: CLK_P
        #6 Clk = 1;
        #4 Clk = 0;
      end

    always
      @ (negedge Clk)   //时钟下跳边沿触发
        case (Next_State)
```

```
        RESET_ST:
          begin
            Reset = 1;
            Start = 0;
          Next_State = APPL_DATA_ST;
          end
        APPL_DATA_ST:
          begin
            Data = Num_Applied;
            Next_State = START_ST;
          end
        START_ST:
          begin
            Start = 1;
            Next_State = WAIT_RESULT_ST;
          end
        WAIT_RESULT_ST:
          begin
            Reset = 0;
            Start = 0;
            wait (Done = = 1);

            if (Num_Applied = =
                        Fac_Out *('h0001 <<Exp_Out))
              $display ("Incorrect result from factorial",
                        "model for input value %d", Data);

            Num_Applied = Num_Applied + 1;

            if (Num_Applied < MAX_APPLY)
              Next_State = APPL_DATA_ST;
            else
              begin
                $display ("Test completed successfully");
                $finish;  //模拟结束。
              end
            end
          default:
            Next_State = START_ST;
        endcase

      //将输入激励加载到待测试模块：
      FACRORIAL F1(Reset, Start, Clk, Data, Done,
                   Fac_Out, Exp_Out);
    endmodule
```

11.6.3 时序检测器

下面是时序检测器的模型。模型用于检测数据线上连续三个 1的序列。在时钟的每个下沿检查数据。图11-10列出了相应的状态图。带有测试验证模块的模型描述如下：

```
module Count3_ls(Data, Clock, Detect3_ls);
  input Data, Clock;
  output Detect3_ls;
  integer Count;
  reg Detect3_1s;

  initial
    begin
      Count = 0;
      Detect3_ls= 0;
    end

  always
    @ (negedge Clock) begin
      if ( Data = = 1)
        Count = Count + 1;
      else
        Count = 0;

      if (Count >= 3)
        Detect3_ls = 1;
      else
        Detect3_ls= 0;
    end
  endmodule

  module Top;
    reg Data, Clock;
    integer Out_File;

    //待测试模块的应用实例。
    Count3_ls   F1(Data, Clock, Detect);

    initial
      begin
    Clock = 0;

      forever
        #5 Clock = ~ Clock;
    end

  initial
    begin
     Data = 0;
      #5 Data = 1;
      #40 Data = 0;
      #10 Data = 1;
      #40 Data = 0;
      #20 $stop;   // 模拟结束。
    end
```

```
    initial
      begin
        //在文件中保存监控信息。
        Out_File = $fopen ("results.vectors");
        $fmonitor (Out_File,"Clock = %b, Data = %b, Detect = %b",
                  Clock, Data, Detect);
      end
  endmodule
```

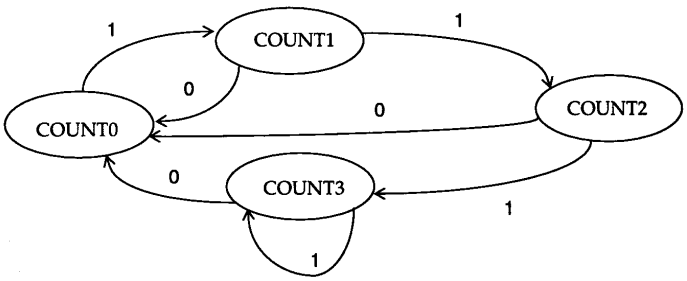


图11-10 时序检测器

习题

1. 产生一个高电平持续时间和低电平持续时间分别为 3 ns和10 ns的时钟。
2. 编写一个产生图11-11所示波形的Verilog HDL模型。

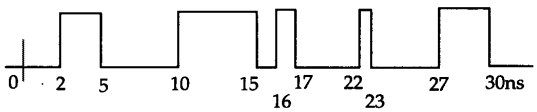


图11-11 波形

3. 产生一个时钟*ClockV*，该时钟是模块*Gen_Clk_D*中描述的时钟*Clk_D*(如图11-6所示)的相移时钟，相位延迟为15 ns。[提示：用连续赋值语句可能会不合适。]
4. 编写测试时序检测器的测试验证程序。时序列检测器按模式 10010在每个时钟正沿检查输入数据流。如果找到该模式，将输出置为 1；否则输出置为0。
5. 编写一个模块生成两个时钟， *ClockA*和*ClockB*。*ClockA* 延迟10 ns后有效，*ClockB*延迟40 ns后有效。两个时钟有相同的高、低电平持续时间，高电平持续时间为 1 ns，低电平持续时间为2ns。*ClockB*与时钟*ClockA*边沿同步，但极性相反。
6. 描述4位加法/减法器的行为模型。用测试验证模块测试该模型。在测试验证模块内描述所有输入激励及其期望的输出值。将输入激励、期望的输出结果和监控输出结果转储到文本文件中。
7. 描述在两个4位操作数上执行所有关系操作(<, <=, >, >=)的ALU。编写一个从文本文件中读取测试模式和期望结果的测试验证模块。
8. 编写一个对输入向量作算术移位操作的模块。指定输入长度用参数表示，缺省值为 32。同时指定移位次数用参数表示，缺省值为 1。编写一个模拟、测试模块以验证对 12位向量进行

8次移位算术操作的正确性。

9. 编写*N*倍时钟倍频器模型。输入是频率未知的参考时钟。输出时钟的倍数与参考时钟的每个正沿同步。[提示：确定参考时钟的时钟周期。
10. 编写一个模型，显示输入时钟每次由0转换到1的时间。
11. 编写一个计数器模型，该计数器在*Count_Flag*为1期间对时钟脉冲(正沿)计数。如果计数超过*MAX_COUNT*，*OverFlow*被置位，并且计数值停留在*MAX_COUNT*界限上。*Count_Flag*的上沿促使计数器复位到0，并重新开始计数。编写测试验证模块，并测试该模型的正确性。
12. 编写参数化的格雷（Gray）码计数器，其缺省长度是3。当变量*Reset*为0时，计数器被异步复位。计数器在每个时钟负沿计数。然后在模拟验证模块中对 4位格雷码计数器进行测试验证。
13. 编写一个带异步复位的T触发器的行为模型。如果开关为1，输出在0和1之间反复。如果开关为0，输出停留在以前状态。接下来用*specify*块指定T触发器的数据建立时间为2 ns，保持时间为3 ns。编写模拟测试模块以测试该模型。

第12章 建模实例

本章给出了一些用Verilog HDL编写的硬件建模实例。

12.1 简单元件建模

连线是一种最基本的硬件单元。连线在 Verilog HDL中可被建模为线网数据类型。考虑 4 位与门，其行为描述如下：

```
`timescale 1ns/1ns
module And4 (A, B, C);
  input [3:0] B, C;
  output [3:0] A;

  assign #5 A = B & C;
endmodule
```

&(与)逻辑的时延定义为5 ns。这个模型代表的硬件如图 12-1所示。

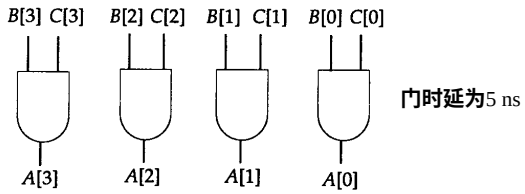


图12-1 一个4位与门

本实例和下面的实例表明布尔等式如何在连续赋值语句中被建模为表达式。连线单元能被建模为线网数据类型。例如，在下面的描述中， F表示将~ (非)操作符的输出连接到^ (异或)操作符输入的线网。该模块表示的电路如图 12-2所示。

```
module Boolean_Ex (D, G, E, );
  input G, E;
  output D;
  wire F;

  assign F = ~E;
  assign D = F ^ G;
endmodule
```

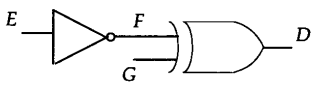


图12-2 组合电路

考虑下列行为和如图 12-3所示相应硬件的表示。

```
module Asynchronous;
  wire A, B, C, D;

  assign C = A | D
  assign A = ~ (B & C);
endmodule
```

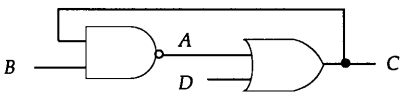


图12-3 异步反馈环路

该电路带有一个异步反馈环路。如果模型用特定的值集 (*B*=1, *D*=0)仿真，仿真时间将由于仿真器总在两个赋值语句间迭代而永远停滞不前。迭代时间是两个零时延。因此，在使用带有零时延的连续赋值语句对线网赋值，以及在表达式中使用相同的线网值时，必须格外小心。

在特定的情况下，有时需要使用这样的异步反馈。下面将演示一个这样的异步反馈；语句代表一个周期为20 ns的周期性波形。其硬件表示如图 12-4所示。注意这样的always 语句需要一个initial 语句将寄存器初始化为0或1，否则寄存器的值将固定在值x上。

```
reg Ace;

initial
  Ace = 0;

always
  #10 Ace = ~ Ace;
```

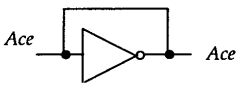


图12-4 时钟发生器

向量线网或向量寄存器型元件能被访问，既可以访问称为位选择的单个元素，也可以访问称为部分选择的片段。例如，

```
reg A;
reg [0:4] C;
reg [5:0] B, D;

always
  begin
    . . .
    D [4:0] = B [5:1] | C; //D [4:0]和B[5:1]都是部分选择。
    D [5] = A & B [5];  //D [5]和B[5]都是位选择。
  end
```

第一个过程性赋值语句暗示着：

```
D [4] = B [5] | C [0];
D [3] = B [4] | C [1];
. . .
```

位选择、部分选择和向量可以并置，形成更大的向量。如，

```
wire [7:0] C, CC;
wire CX;
. . .
assign C = {CX, CC [6:0]};
```

也可以引用索引值在运行时才可计算的向量元素。如：

```
Adf = Plb[K];
```

意味着解码器的输出为*Adf*，并且*K*指定选择地址。*Plb*是向量；本语句对解码器的行为建模。

可以使用预定义的移位操作符执行移位操作。移位操作可以用合并操作符建模。例如，

```
wire [0:7] A, Z;
. . .
assign Z = {A [1:7], A [0]};          /循环左移。
assign Z = {A [7], A [0:6]};          /循环右移。
assign Z = {A [1:7], 1'b0};           左移操作。
```

向量的子域称为部分选择，也能够在表达式中使用。例如， 32位指令寄存器*Instr_Reg*中前16位表示地址，接下来8位表示操作码，余下的8位表示索引。给出如下说明：

```
reg [31:0] Memory [0: 1023 ];
wire [31:0] Instr_Reg;
wire [15:0] Address;
wire [7:0] Op_Code, Index;
wire [0:9] Prog_Ctr;
wire Read_Ctl;
```

从*Instr_Reg*读取子域信息的一种方法是使用三个连续赋值语句。指令寄存器的部份选择被赋值给指定的线网。

```
assign INstr_Reg =  Memory[Prog_Ctr];

assign Address = Instr_Reg[31:16];
assign Op_Code = Instr_Reg[15:8];
assign Index = Instr_Reg[7:0];
...
always
  @(posedge Read_Ctl)
     Task_Call( Address, Op_Code, Index);
```

可用连续赋值语句为三态门的行为建模，如：

```
wire Triout = Enable ? TriIN 1'bz;
```

当*Enable*为1时，*TriOut*获得*TriIn*的值。当*Enable*为0时，*TriOut*为高阻值。

12.2 建模的不同方式

本节给出了Verilog HDL语言提供的三种不同建模方式的实例：数据流方式、行为方式和结构方式。参看图12-5所示的电路，该电路将输入*A*的值存入寄存器，然后与输入*C*相乘。

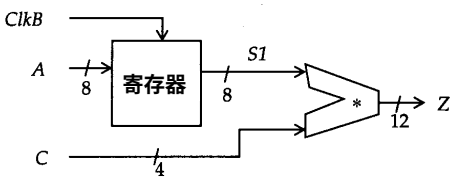


图12-5 带缓冲的乘法器

第一种建模方式是数据流方式，它使用连续赋值语句对电路建模。

```
module Save_Mult_Df(A,C,ClkB,Z);
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
  output [0:11] Z;
  wire S1;

  assign Z = S1 * C;
  assign S1 = ClkB ? A : S1;
endmodule
```

这种表示方法不直接蕴含任何结构，却隐式地描述了结构。但是，它的功能性非常清晰。寄存器已使用时钟控制方式建模。

第二种描述电路的方法是使用带有顺序程序块的 always 语句，将电路建模为顺序程序描述。

```
module Save_Mult_Seq (A,C,ClkB,Z);
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
```

```
  output [0:11] Z;
  reg [0:11] Z;

  always
    @(A or C or ClkB)
      begin: SEQ
        //由于标记了程序块，可以说明一个局部寄存器变量S1。
        reg [0:7] S1;

        if (ClkB)
           S1 =  A;

        Z = S1 * C;
      end
endmodule
```

这种模型也描述了电路的行为，但是没有蕴含任何结构，无论是隐式还是显示的。在这种方式中，寄存器用*if*语句建模。

第三种描述*Save_Mult*电路的方法是假定已存在8位寄存器和8位乘法器，将电路建模为线网表。

```
module Save_Mult_Netlist (A,C,ClkB,Z)
  input [0:7] A;
  input [0:3] C;
  input  ClkB;
  output [0:11] Z;
  wire [0:7] S1,S3;
  Wire [0:15] S2;

  Reg8   R1 (.Din(A),.Clk(ClkB),.Dout(S1));
  Mult8  M1 (.A(S1),.B({4'1b0000,C}),.Z(Z));
endmodule
```

这种描述方式显式地描述了电路结构，但其行为是未知的。这是因为 *Reg8*和*Mult8*的模块名是任意的，并且它们可以有与其相关的各种行为。

12.3 时延建模

考虑3输入或非门。它的行为可以使用连续赋值语句建模，如

```
assign #12 Gate_Out = ~(A | B | C);
```

这条语句对带有12个时间单位时延的或非门建模。这一时延表示从信号A、B或C上事件发生到结果值出现在信号*Gate_Out*上的时间。一个事件可以是任何值的变化，如x -> z、 x - > 0, 或者1- > 0。

如果要显式地对上升和下降时间建模，则在连续赋值语句中使用两个时延，例如：

```
assign #(12,14) Zoom = ~(A | B | C);
  /*12是上升时延，14是下降时延，min(12,14) = 1是转换到x的时延*/
```

在能够对高阻值Z赋值的逻辑中，也能够定义第三种时延，即关断时延。例如：

```
assign #(12,14,10)Zoom = A > B ? C : 1bz;
  //上升时延为12，14是下降时延，min(12,14，10) = 1是转换到x的时延，关断时延为10。
```

每个时延值都能够用*min:typ:max*表示，例如，

```
assign #(9:10:11,11:12:13,13:14:15) Zoom = A > B ? C : 1bz;
```

时延值通常可以是表达式。

基本门和UDP中的时延可以通过在实例中指定时延值建模。下面是5输入基本与门。

```
and #(2,3) A1(Ot,In1,In2,In3,In4,In5);
```

已指定输出的上升时延为2个时间单位，下降时延为3个时间单位。

模型中位于端口边界的时延可使用指定程序块来定义。如下面的半加器模块的例子。

```
module  Half_Adder(A,B,S,C);
  input A,B;
  output S,C;

  specify
    (A => S) = (1.2,0.8);
    (B => S) = (1.0,0.6);
    (A => C) = (1.2,1.0);
    (B => C) = (1.2,0.6);
  endspecify

  assign S = A ^ B;
  assign C = A | B;
endmodule
```

除在连续赋值语句中对时延建模外，时延还可以用指定程序块建模。是否存在一种从模块外部指定时延的方法？一种选择是使用SDF[⊖]（标准时延格式）和Verilog仿真器可能提供的反标机制。如果需要在Verilog HDL模型中显式地指定这一信息，一种方法是在Half-Adder模块上创建两个虚模块，每个模块带有不同的时延集。

```
module  Half_Adder(A,B,S,C);
  input A,B;
  output S,C;

  assign S = A ^ B;
  assign C = A | B;
endmodule

module  Ha_Opt(A,B,S,C);
  input A,B;
  output S,C;

  specify
    (A => S) = (1.2,0.8);
    (B => S) = (1.0,0.6);
    (A => C) = (1.2,1.0);
    (B => C) = (1.2,0.6);
  endspecify

  Half_Adder  H1(A,B,S,C);
endmodule

module  Half_Pess(A,B,S,C);
```

⊖ 见参考文献

```
  input A,B;
  output S,C;

  specify
    (A => S) = (0.6,0.4);
    (B => S) = (0.5,0.3);
    (A => C) = (0.6,0.5);
    (B => C) = (0.6,0.3);
  endspecify

  Half_Adder  H1(A,B,S,C);
endmodule
```

有了这两个模块，*Half_Adder*模块独立于任何时延，并且依赖于你所选用的时延方式，即模拟相应的高层模块*Ha_Opt*或*Ha_Pess*。

传输时延

在连续赋值语句和门级原语模型中指定的时延为惯性时延。传输时延能够用带有语句内时延的非阻塞性赋值语句建模。实例如下，

```
module Transport (WaveA,DelayedWave);
  parameter TRANSPORT_DELAY = 500;
  input WaveA;
  output DelayedWave;
  reg DelayedWave;

  always
    @(WaveA) DelayedWave <= #TRANSPORT_DELAY WaveA;
endmodule
```

always语句中包含带有语句内时延的非阻塞性赋值。 *WaveA*上的任何变化以后都会使*DelayedWave*在延迟*TRANSPORT_DELAY*时获得调度。结果在*WaveA*上出现的波形在被延迟*TRANSPORT_DELAY*后出现在*DelayedWave*上；这种时延波形的实例如图12-6所示。

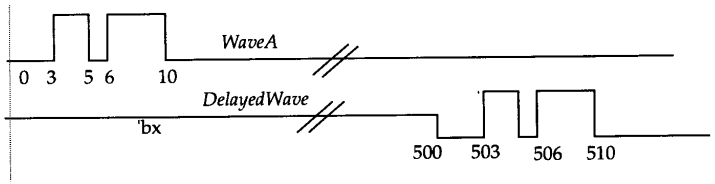


图12-6 传输时延实例

12.4 条件操作建模

在特定条件下发生的操作可以使用带有条件操作符的连续赋值语句，或在 always语句中用if语句或case语句建模。请看一个算术逻辑电路。它的行为可用如下所示的连续赋值语句建模。

```
module  Simple_ALU(A,B,C, PM, ALU);
  input [0:3] A,B, C;
  input PM;
```

```
  output [0:3] ALU;

  assign ALU = PM ? A + B: A - B;
endmodule
```

多路选择开关也能够使用 always 语句建模。首先确定选线的值，然后， case语句根据这个值选择相应的输入赋值到输出。

```
`timescale 1ns/1ns
module  Multiplexer(Sel, A,B,C,D, Mux_Out);
  input [0:1] Sel;
  input A, B,C, D;
  output Mux_Out;
  reg Mux_Out;
  reg Temp;
  parameter MUX_DELAY= 15;

  always
    @ (Sel or A or B or C or D)
      begin: P1
        case (Sel)
          0: Temp = A;
          1: Temp = B;
          2: Temp = C;
          3: Temp = D;
         endcase
        Mux_Out = # MUX_DELAY Temp;
    end
endmodule
```

多路选择开关也可以用如下形式的连续赋值语句建模：

```
assign #MUX_DELAY Mux_Out= (Sel == 0)? A : (Sel == 1)? B :
                 (Sel == 2)? C : (Sel == 3)? D : 1'bx;
```

12.5 同步时序逻辑建模

到目前为止，本章中的绝大多数实例都是组合逻辑。对于同步时序逻辑建模，语言中已提供寄存器数据类型对寄存器和存储器建模。但是并不意味着每种寄存器数据类型都可对同步时序逻辑建模。通过控制赋值方式对同步时序逻辑建模是一种常见的方法。

参见如下实例，它表明了如何通过控制寄存器对同步边沿触发的 D型触发器建模。

```
`timescale 1ns/1ns
module  D_Flip_Flop (D, Clock, Q);
  input D, Clock;
  output Q;
  reg Q;

always
  @ (posedge Clock)
     Q = #5 D;
endmodule
```

always语句表明当 *Clock*上出现上升沿时， *Q*会在5 ns后被赋值为 *D*；否则*Q*不发生变化（寄存器在被赋新值前保持原值）。always语句中的行为表达了*D*型触发器的语义。提供这一模

型后，8位寄存器可按如下方式进行建模。

```
module Register8(D, Q, Clock);
  parameter START = 0, STOP = 7;
  input [START : STOP] D;
  input Clock;
  output [START : STOP] Q;
  wire [START : STOP] Cak;

  D_Flip_Flop  DFF0
              [START : STOP] (. D (D), . Clock (Cak), .Q (Q));

  buf  B1      (Cak [0], Cak [1], Cak [2], Cak [3], Cak [4],
                Cak [5], Cak [6], Cak [7], Clock);
endmodule
```

考虑如图12-7所示的门级交叉耦合锁存器电路及其数据流模型。

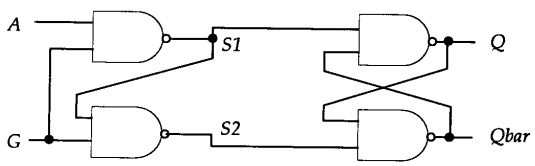


图12-7 门级锁存器

```
module Gated_FF (A, G, Q, Qbar);
  input A, G;
  output Q, Qbar;
  wire S1, S2;

  assign S1 = ~ (A & G);
  assign S2 = ~ (S1 & G);
  assign Q = ~ (Qbar & S1);
  assign Qbar = ~ (Q & S2);
endmodule
```

在此例中，连续赋值语句的语义蕴含了锁存器结构。

存储器可用寄存器数组建模。实例如下，其中*ASIZE*是地址端口的位数，*DSIZE*是RAM数据端口的位数。

```
module RAM_Generic(Address, Data_In, Data_out, RW);
  parameter ASIZE = 6, DSIZE = 4;
  input [ASIZE-1 :0] Address;
  input [DSIZE-1 :0] Data_In;
  input RW;
  output [DSIZE-1 :0] Data_Out;
  reg [DSIZE－1:0] Data_Out;
  reg [0: DSIZE－1] Mem_FF [0:63];

  always
    @ (RW)
      if (RW)          //从RAM中读取数据。
        Data_Out = Mem_FF [Address];
```

```
        else          //向RAM写入数据。
          Mem_FF [Address] = Data_In;
    endmodule
```

同步时序逻辑也可以用电平敏感或边沿触发控制方式建模。电平敏感类型锁存器建模实例如下。

```
    module Level_Sens_FF(Strobe, D, Q, Qbar);
      input Strobe, D;
      output Q, Qbar;
      reg Q, Qbar;

      always
        begin
          wait (Strobe = = 1);
            Q = D;
            Qbar = ~ D;
        end
    endmodule
```

当*strobe*为1时，*D*上的任何事件都被传输到*Q*；当*Strobe*变成0时，*Q*和*Qbar*中的值保持不变，并且输入*D*上的任何变化都不再影响*Q*和*Qbar*上的值。

理解过程性赋值语句的语义对决定同步时序逻辑的行为功能非常重要。考虑模块 *Body1*和*Body2*之间的不同之处。

```
    module Body1;
      reg A;

    initial A = 0;

      always A = ~ A;
    endmodule

    module Body2;
      wire Clock;
      reg A;

      initial A = 0;

      always
        @ (Clock )
          if (~ Clock)
            A = ~ A;
    endmodule
```

模块*Body1*蕴含的电路结构如图12-8所示，而模块*Body2*蕴含的电路结构如图12-9所示。

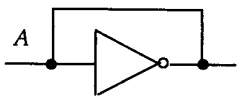


图12-8 不蕴含触发器

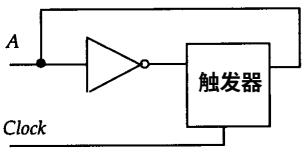


图12-9 蕴含触发器

如果*Body1*按如上所述进行模拟，模拟将由于零时延异步反馈而陷入死循环（模拟时间停止不前）。在模块*Body 2*中，*A*的值只有在*Clock*信号下降沿时锁存，在其他情形下（*Clock*不处在上升沿时），*A*上的任何变化（触发器的输入）都不会影响触发器的输出。

12.6 通用移位寄存器

通用串行输入、串行输出移位寄存器能够使用 always语句块内的for循环语句建模。寄存器的数量被定义为参数，这样通用移位寄存器的数量在其他设计中被引用时，可以修改。

```
module Shift_Reg (D, Clock, Z);
  input D, Clock;
  output Z;
  parameter NUM_REG = 6,
  reg [1: NUM_REG ] Q;
  integer P;

  always
    @ (negedge Clock) begin
      //寄存器右移一位：
      for (P = 1; P< NUM_REG; P = P + 1)
        Q[P+1] = Q[P];

      //加载串行数据：
      Q[1] = D;
    end

  //从最右端寄存器获取输出：
  assign Z = Q [NUM_REG];
endmodule
```

可以通过用不同的参数值引用模块 *Shift_Reg*获取不同长度的移位寄存器。

```
module Dummy;
  wire Data, Clk, Za, Zb, Zc;

  //6位移位寄存器：
  Shift_Reg  SRA(Data, Clk, Za);

  //4位移位寄存器：
  Shift_Reg #4 SRB (Data, Clk, Zb);

  //10位移位寄存器：
  Shift_Reg #10 SRC (Data, Clk, Zc);
endmodule
```

12.7 状态机建模

状态机通常可使用带有 always语句的case语句建模。状态信息存储在寄存器中。 case语句的多个分支包含每个状态的行为。下面是表示状态机简单乘法算法的实例。当 *Reset*信号为高时，累加器*Acc*和计数器 *Count*被初始化。当 *Reset*变为低时，乘法开始运算。如果乘数 *Mplr*在*Count*位的值为1，被乘数加到累加器上。然后，被乘数左移1位且计数器加1。如果*Count*是16，乘法运算完成，并且 *Done*信号被置为高。如若不然，检查乘数 *Mplr*的*Count*位，并重复 always语句。状态图如图12-10所示，其后是相应的状态机模型。

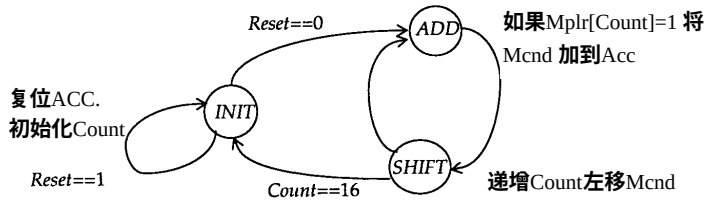


图12-10 乘法器的状态图

```
module Multiply (Mplr, Mcnd, Clock, Reset, Done, Acc);
       //Mplr是乘数，Mcnd是被乘数。
  input [15:0] Mplr, Mcnd;
  input Clock, Reset;
  output Done;
  reg Done;
  output [31:0] Acc;
  reg [31:0] Acc;
  parameter INIT = 0, ADD = 1, SHIFT = 2;
  reg [0:1] Mpy_State;
  reg [31:0] Mcnd_Temp;

  initial Mpy_State = INIT;      //初始状态为INIT。

  always
    @ (negedge Clock) begin: PROCESS
      integer Count;

      case (Mpy_State)
        INIT:
          if (Reset)
           Mpy_State = INIT;
           /*由于Mpy_State将保持原值，上面的语句并不需要*/。
          else
            begin
              Acc = 0;
              Count = 0;
              Mpy_State = ADD;
              Done = 0;
              Mcnd_Temp [15:0] =Mcnd;
              Mcnd_Temp [31:16] = 16'd0;
            end

        ADD:
          begin
            if (Mplr [Count])
              Acc = Acc + Mcnd_Temp;

              Mpy_State = SHIFT;
          end

        SHIFT:
```

```
        begin
          //Mcnd_Temp左移:
          Mcnd_Temp = {Mcnd_Temp[30:0], 1'b0};
          Count = Count+ 1;

          if (Count == 16)
            begin
              Mpy_State = INIT;
              Done = 1;
            end
          else
            Mpy_State = ADD;
        end
      endcase  //对Mpy_State的case语句结束。
    end  //顺序程序块PROCESS。
  endmodule
```

寄存器*Mpy_State*保存状态机模型的状态。最初，状态机模型处于 *INIT*状态，并且只要*Reset*为真，模型就停留在这一状态。当*Reset*为假时，累加器*Acc*被清空。计数器*Count*被复位，被乘数*Mcnd*被加载到临时变量 *Mcnd_Temp*中，然后模型状态前进到状态 *ADD*。当模型处于*ADD*状态时，只有当乘数在*Count*位置的位为1时，*Mcnd_Temp*中的被乘数才被加到*Acc*上，然后模型状态前进到 *SHIFT*状态。在这一状态，乘法器再一次左移，计数器加 1；并且如果计数器值为16，*Done*置为真，模型返回到*INIT*状态。此时，*acc*包含乘法运算的结果。如果计数器值小于16，模型本身反复通过*ADD*和*SHIFT*状态直到计数器值变为16。

状态转换发生在时钟的各个下跳沿；这通过使用@ (**negedge** *Clock*)时序控制来指定。

12.8 交互状态机

交互状态机能够使用通过公共寄存器通信的独立的 always语句进行描述。考虑图 12-11所示的两个交互进程的状态图， *TX*是一个发送器， *MP*是一个微处理器。如果进程 *TX*不忙，进程*MP*将要发送的数据放置在数据总线上，然后向进程 *TX*发送信号*Load_TX*，通知其装载数据并开始发送数据。进程 *TX*在数据传送期间设置*TX_Busy*表明其处于忙状态，不能从进程 *MP*接收任何进一步的数据。

下面显示了这两个交互进程的框架模型，图中仅显示了控制信号和状态转换，没有描述操作数据的代码。

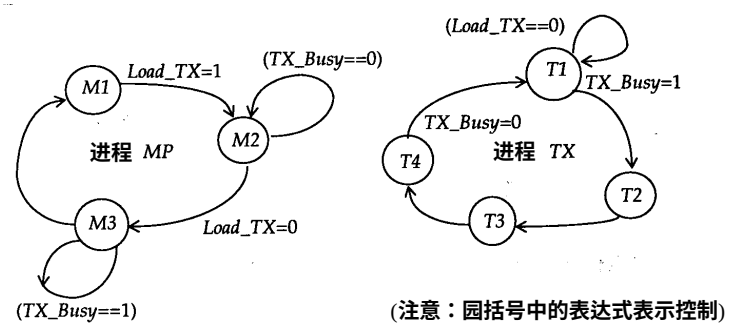


图12-11 两个交互进程的状态图

```
module Interacting_FSM(Clock);
  input Clock;

  parameter M1 = 0, M2 = 1, M3 = 2;
  parameter T1 = 0, T2 = 1; T3 = 2, T4 = 3;
  reg [0:1] MP_State;
  reg [0:1] TX_State;
  reg Load_TX, TX_Busy

  always
    @ (negedge Clock) begin: MP
      case (MP_State)
        M1:   //向数据总线装载数据。
          begin
            Load_TX = 1;
            MP_State = M2;
          end

        M2:   //等待确认信号。
          if (TX_Busy)
            begin
              MP_State = M3;
              Load_MX = 0;
            end

        M3:           /等待进程TX结束。
            if (~TX_Busy)
              MP_State = M1;
        endcase
  end   //顺序块MP结束。

  always
    @ (negedge Clock) begin: TX
      case (TX_State)
        T1:             /等待装载的数据。
          if (Load_TX)
            begin
              TX_State = T2;
              TX_Busy = 1;//从数据总线中读取数据。
            end

        T2:          /发送开始标志。
          TX_State = T3;

        T3:         //传送数据。
          TX_State = T4;

        T4:   //发送跟踪标志以结束传送。
          begin
            TX_Busy = 0;
            TX_State = T1;
```

```
        end
      endcase
    end                    //顺序块TX结束。
endmodule
```

此交互有限状态机的时序行为关系如图 12-12所示。

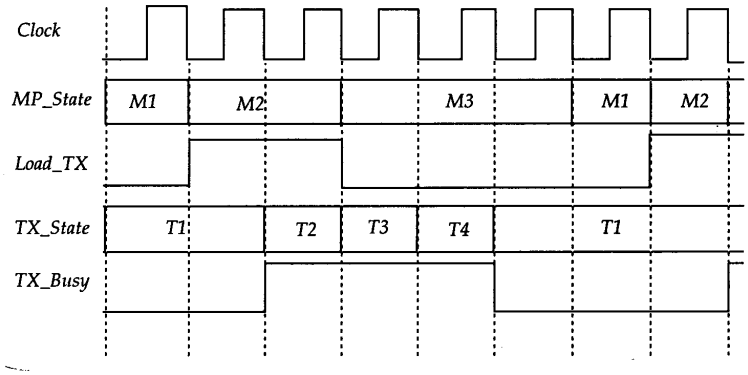


图12-12 两个交互进程的时序行为

考虑两个交互进程的另外一个实例，时钟分频器 *DIV*和接收器*RX*。在这种情况下，进程*DIV*产生一个新时钟，并且进程状态变换（转换）序列与新时钟同步。状态图如图 12-13所示。

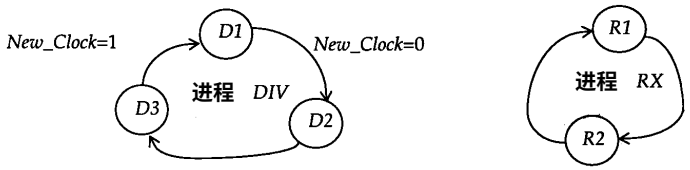


图12-13 *DIV*产生*RX*的时钟

```
module Another_Example_FSM2 (Clock);
  input Clock;

  parameter D1 = 1, D2 = 2, D3 = 3;
  parameter R1 = 1, R2 = 2;
  reg [0:1] Div_State, RX_State;
  reg New_Clock;

  always
    @ (posedge Clock) begin: DIV
      case (Div_State)
        D1:
          begin
            Div_State = D2;
            New_Clock = 0;
          end

        D2:
          Div_State = D3;
```

```
        D3:
          begin
            New_Clock = 1;
            Div_State = D1;
          end
        endcase
      end          //顺序块DIV结束。

    always
      @ (negedge New_Clock) begin: RX
        case (RX_State)
          R1: RX_State= R2;
          R2: RX_State= R1;
        endcase
    end    //顺序块结束。
  endmodule
```

顺序块*DIV*在其状态序列转换过程中产生新时钟。这一进程的状态转换发生在时钟的上升沿。顺序块在*New_Clock*的每个下降沿执行。图12-14显示了这些交互状态机的波形序列。

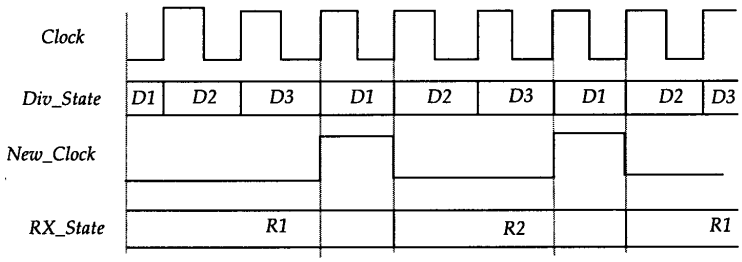


图12-14 进程*RX*和*DIV*之间的交互

12.9 Moore有限状态机建模

Moore有限状态机（FSM）的输出只依赖于状态而不依赖其输入。这种类型有限状态机的行为能够通过使用带有在状态值上转换的case语句的always语句建模。图12-15显示了Moore有限状态机的状态转换图实例，接着是Moore有限状态机对应的行为模型。

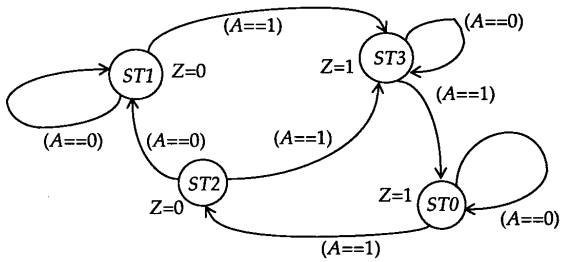


图12-15 Moore机的状态图

```
module Moore_FSM(A, Clock, Z);
  input A, Clock;
```

```
    output Z;
    reg Z;

    parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
    reg [0:1] Moore_State;

    always
      @ (negedge  Clock)
        case (Moore_State)
          ST0:
          begin
            Z = 1;
            if (A)
              Moore_State = ST2;
          end

        ST1:
          begin
            Z = 0;
            if (A)
              Moore_State = ST3;
           end

          ST2:
            begin
              Z = 0;
              if (~A)
                Moore_State = ST1;
              else
                Moore_State= ST3;
            end

          ST3:
            begin
              Z = 1;
              if (A)
                Moore_State = ST0;
            end
        endcase
endmodule
```

12.10 Mealy型有限状态机建模

在Mealy型有限状态机中，输出不仅依赖机器的状态而且依赖于它的输入。这种类型的有限状态机能够使用与 Moore FSM相似的形式建模。即使用 always语句。为了说明语言的多样性，使用不同的方式描述 Mealy机。这一次，我们用两条 always语句，一条对有限状态机的同步时序行为建模，一条对有限状态机的组合部分建模。图 12-16给出了状态转换表的一个实例，接着是相应的行为模型。

当前状态 \ 输入A	0	1
ST0	ST0 / 0	ST3 / 1
ST1	ST1 / 1	ST0 / 0
ST2	ST2 / 0	ST1 / 1
ST3	ST2 / 0	ST1 / 0

表中的项为次态和输出Z

图12-16 Mealy机状态转换表

```
module Mealy_FSM (A, Clock, Z);
  input A, Clock;
  output Z;
  reg Z;

  parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
  reg [1:2] P_State, N_State;

  always
    @ (negedge Clock)   //同步时序逻辑部分。
      P_State = N_State;
  always
    @ (P_State or A) begin: COMB_PART
      case (P_State)
        ST0:
          if (A)
            begin
              Z = 1;
              N_State = ST3;
             end
            else
              Z = 0;

        ST1:
          if (A)
            begin
              Z = 0;
              N_State = ST0;
             end
            else
              Z = 1;

        ST2:
          if (~A)
              Z = 0;
            else
              begin
                Z = 1;
                N_State = ST1;
```

```
                end

          ST3:
            begin
              Z = 0;
              if (~A)
                N_State = ST2;
              else
                N_State = ST1;
            end
        endcase
    end      //顺序块COMB_PART结束。
endmodule
```

在这种类型的有限状态机中，因为状态机的输出可以直接依赖独立于时钟的输入，将输入信号放入组合的部分时序程序块的事件列表中是非常重要的。因为 Moore有限状态机的输出只依赖于状态，并且状态转换在时钟上同步发生，在 Moore有限状态机中不会发生这种情况。

12.11 简化的21点程序

本节介绍简化的21点程序的状态机描述。玩21点程序需要一幅扑克牌。从2到10的牌取值与面值相同，牌A的值可以为1或者为11。游戏的目标是接收一定数量随机产生的牌，总分（所有牌值的总和）尽可能接近21而又不超过21。

当插入新牌时，*Card_Rdy*为真，并且*Card_Value*为牌的值。*Request_Card*表明程序何时就绪准备接收新牌。如果接收牌的序列总和超过 21，*Lost*置为真，以表明牌序列已经输了；否则*Won*置为真以表明游戏已获胜。状态排序由时钟控制。图 12-17显示了21点程序模块的输入和输出。

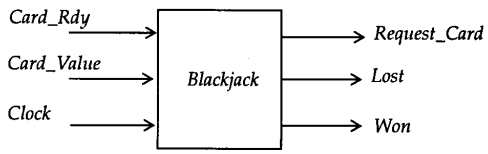


图12-17 21点程序模块的外部视图

程序的行为在如下的模块说明中描述。程序在总得分不大于 17前一直接收新牌。得分不超过21时，第一个A按11取值；得分超过21时，牌A按减10处理，即取值为1。三个寄存器用于存储程序的值：*Total*保存总和，*Current_Card_Value*保存读取的牌值（取值从 1到10），*Aec_As_11*用于记忆牌A是否取值为11而不是1。21点程序的状态存储在寄存器*BJ_State*中。

```
module Blackjack(Card_Rdy, Card_Value,
                 Request_Card, Won, Lost, Clock);
  input Card_Rdy, Clock;
  input [0:3] Card_Value;
  output Request_Card, Lost, Won;
  reg Request_Card, Lost, Won;

  Parameter INITIAL_ST = 0,GETCARD_ST= 1,
            REMCARD_ST= 2, ADD_ST = 3, CHECK_ST = 4,
```

```
          WIN_ST = 5 , BACKUP_ST = 6, LOSE_ST = 7;
reg [0:2] BJ_State;
reg [0:3] Current_Card_Value;
reg [0:4] Total;
reg Ace_As_11;

always
  @ (negedge Clock)
    case (BJ_State)
      INITIAL_ST:
       begin
         Total = 0;
         Ace_As_11 = 0;
         Won = 0;
         Lost = 0;
         BJ_State = GETCARD_ST
       end

      GETCATD_ST:
       begin
         Request_Card= 1;

         if (Card_Rdy)
           begin
             Current_Card_Value = Card_Value;
             BJ_State = REMCATD_ST;
           end          //否则停留在状态GETCARD_ST上不变。
        end

      REMCARD_ST :   //等待牌被移走。
       if (Card_Rdy)
         Request_Card=0;
       else
         BJ_State = ADD_ST;

      ADD_ST:
        begin
          if (~Ace_As_11 && Current_Card_Value)
            begin
              Current_Card_Value= 11;
              Ace_As_11 = 1;
            end

          Total = Total + Current_Card_Value;
          BJ_State = CHECK_ST;
       end

      CHECK_ST:
        if (Total < 17)
          BJ_State = GETCARD_ST;
        else
```

```
            begin
              if (Total <22)
                BJ_State = WIN_ST;
              else
                BJ_State = BACKUP_ST;
            end

          BACKUP_ST:
            if (Ace_As_11)
              begin
                Total = Total -10;
                Ace_As_11 = 0;
                BJ_State = CHECK_ST;
               end
             else
             BJ_State = LOSE_ST;

          LOSE_ST:
            begin
              Lost = 1;
              Request_Card= 1;
              if (Card_Rdy)
                BJ_State = INITIAL_ST;
                //否则停留在这个状态上不变。
             end

          WIN_ST:
            begin
              Won = 1;
              Request_Card= 1;

              if (Card_Rdy)
                BJ_State = INITIAL_ST;
                //否则停留在这个状态上不变。
              end
          endcase
    endmodule       //21点程序结束。
```

习题

1. 为芒果汁饮料机编写一个 Verilog HDL模型。机器分发价值 15美分一听的芒果汁。只接收 5分镍币和一角硬币。任何变化必须被返回。使用测试验证程序测试该模型。
2. 编写一个模型，描述带有同步预置和清空的触发器模型行为。
3. 编写带有串行数据输入、并行数据输入、时钟和并行数据输出的 4位移位寄存器模型。使用测试验证程序测试该模型。
4. 使用行为构造方式描述D型触发器。然后使用这一模块，编写一个 8位寄存器模型。
5. 编写测试12.11节描述的21点游戏模型的测试验证程序。
6. 使用移位操作符，描述解码器模块，然后使用测试验证程序测试该模块。解码器的输入数量指定为：

```
`define NUM_INPUTS 4
```

[提示：使用移动操作符计算解码器输出的数量。]

7. 编写并行到串行的8位转换器模型。输入为8位向量，在时钟上升沿从左位开始一次发送出一位。只有在前面的输入向量的所有位都发送出去以后，才读入下一个输入。
8. 编写与练习7行为相反的串行到并行的8位转换器。为体现传输时延，在时钟上升沿经过一小段时延后对输入流采样。使用高层模块将本练中编写的模型与练习7编写的模型连接，并使用测试验证程序测试其输出。
9. 编写具有保持控制的*N*位计数器模型。如果保持为1，计数器保持它的值，如果保持变为0，计数器被重置为0，并再次启动计数器。编写测试验证程序测试该模型。
10. 编写通用队列模型，队列中的字长为 *N*，字数为*M*。*InputData* 是被写入队列的字；当*Addword*为1时，向队列中添加字。从队列中读出的字存储在 *Output Data*中；当*Read Word*为1时，从队列中读取字。设置相应的标志 *Empty* 和*Full*。所有的事务发生在时钟*ClockA*的下降沿。编写测试验证程序测试该模型。
11. 编写可参数化的时钟分频器模型。输出时钟的周期是输入时钟周期的 2**N*倍，输出时钟与输入时钟的上升沿同步。编写测试验证程序测试该模型的正确性。

附录 语法参考

本附录提供了Verilog HDL语言的所有语法。⊖

关键词

以下是Verilog HDL硬件描述语言的关键词。注意，只有小写的名字才是关键词。

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

⊖ 源自IEEE Std 1364-1995，

语法规范

下列规范应用于语法描述，规则采用巴科斯—诺尔范式(BNF)书写：

1) 语法规则按自左向右非终结字符的字母序组织。

2) 保留字、操作符和标点标记是语法的组成部分，以粗体字表示。

3) 非终结名字前的斜体名字的语义表示与非终结名字相关联。

4) 非粗体的垂直符号（|）用于分离可替换的选项。

5) 非粗体的方括号（[...]）表示可选项。

6) 非粗体的大括号（{...}）表明某项可以重复0次或多次。

7) 以粗体出现的方括号、圆括号、大括号（[...], (...){...},）以及其他符号（如；）表示符号是语法的组成部分。

8) 起始的非终结名字为“源文本（source_text）”

9) 此语法中使用的终结名字以大写形式出现。

文法

```
always_construct ::=
    always
      statement
binary_base ::=
    'b | 'B
binary_digit ::=
    x | X | z | Z | 0 | 1
binary_number ::=
    [ size ] binary_base binary_digit { _ | binary_digit }
binary_operator ::=
        + | - | * | / | %
    | == | != | === | !== | && | | | | < | <= | > | >=
    | & | | | ^ | ^~ | ~^ | >> | <<
block_item_declaration ::=
        parameter_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
blocking_assignment ::=
    reg_lvalue = [ delay_or_event_control ] expression
case_item ::=
        expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
case_statement ::=
        case ( expression ) case_item { case_item } endcase
    | casez ( expression ) case_item { case_item } endcase
    | casex ( expression ) case_item { case_item } endcase
charge_strength ::=
```

```
        ( small )
    |   ( medium )
    |   ( large )
cmos_switch_instance ::=
    [ name_of_gate_instance ] ( output_terminal , input_terminal ,
    ncontrol_terminal , pcontrol_terminal )
cmos_switchtype ::=
    cmos | rcmos
combinational_body ::=
    table
      combinational_entry { combinational_entry }
    endtable
combinational_entry ::=
    level_input_list : output_symbol ;
comment ::=
        short_comment
    |   long_comment
comment_text ::=
    { ANY_ASCII_CHARACTER }
concatenation ::=
    { expression { , expression } }
conditional_statement ::=
    if ( expression ) statement_or_null [ else statement_or_null ]
constant_expression ::=
        constant_primary
    |   unary_operator constant_primary
    |   constant_expression binary_operator constant_expression
    |   constant_expression ? constant_expression : constant_expression
    |   string
constant_mintypmax_expression ::=
        constant_expression
    |   constant_expression : constant_expression : constant_expression
constant_primary ::=
        number
    |   parameter_identifier
    |   constant_concatenation
    |   constant_multiple_concatenation
continuous_assign ::=
    assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor
    [ &&& timing_check_condition ]
current_state ::=
    level_symbol
data_source_expression ::=
    expression
decimal_base ::=
    'd | 'D
decimal_digit ::=
```

```
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_number ::=
        [ sign ] unsigned_number
    |   [ size ] decimal_base unsigned_number
delay2 ::=
        # delay_value
    |   # ( delay_value [ , delay_value ] )
delay3 ::=
        # delay_value
    |   # ( delay_value [ , delay_value [ , delay_value ] ] )
delay_control ::=
        # delay_value
    |   # ( mintypmax_expression )
delay_or_event_control ::=
        delay_control
    |   event_control
    |   repeat ( expression ) event_control
delay_value ::=
        unsigned_number
    |   parameter_identifier
    |   constant_mintypmax_expression
description ::=
        module_declaration
    |   udp_declaration
disable_statement ::=
        disable task_identifier ;
    |   disable block_identifier ;
drive_strength ::=
        ( strength0 , strength1 )
    |   ( strength1 , strength0 )
    |   ( strength0 , highz1 )
    |   ( strength1 , highz0 )
    |   ( highz1 , strength0 )
    |   ( highz0 , strength1 )
edge_control_specifier ::=
    edge [ edge_descriptor [ , edge_descriptor ] ]
edge_descriptor ::=
        01
    |   10
    |   0x
    |   x1
    |   1x
    |   x0
edge_identifier ::=
    posedge | negedge
edge_indicator ::=
        ( level_symbol level_symbol )
    |   edge_symbol
edge_input_list ::=
    { level_symbol } edge_indicator { level_symbol }
```

```
edge_sensitive_path_declaration ::=
        parallel_edge_sensitive_path_description = path_delay_value
    |   full_edge_sensitive_path_description = path_delay_value

edge_symbol ::=
    r | R | f | F | p | P | n | N | *

enable_gate_instance ::=
    [ name_of_gate_instance ] ( output_terminal , input_terminal ,
    enable_terminal )

enable_gate_type ::=
    bufif0 | bufif1 | notif0 | notif1

enable_terminal ::=
    scalar_expression

escaped_identifier ::=
    \ { ANY_ASCII_CHARACTER_EXCEPT_WHITE_SPACE } white_space

event_control ::=
        @ event_identifier
    |   @ ( event_expression )

event_declaration ::=
    event event_identifier { , event_identifier } ;

event_expression ::=
        expression
    |   event_identifier
    |   posedge expression
    |   negedge expression
    |   event_expression or event_expression

event_trigger ::=
    -> event_identifier ;

expression ::=
        primary
    |   unary_operator primary
    |   expression binary_operator expression
    |   expression ? expression : expression
    |   string

full_edge_sensitive_path_description ::=
    ( [ edge_identifier ] list_of_path_inputs *> list_of_path_outputs
    [ polarity_operator ] : data_source_expression )

full_path_description ::=
    ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )

function_call ::=
        function_identifier ( expression { , expression } )
    |   name_of_system_function [ ( expression { , expression } ) ]

function_declaration ::=
    function [ range_or_type ] function_identifier ;
        function_item_declaration { function_item_declaration }
        statement
    endfunction

function_item_declaration ::=
        block_item_declaration
    |   input_declaration
```

```
gate_instantiation ::=
        n_input_gatetype [ drive_strength ] [ delay2 ] n_input_gate_instance
          { , n_input_gate_instance } ;
    | n_output_gatetype [ drive_strength ] [ delay2 ] n_output_gate_instance
          { , n_output_gate_instance } ;
    | enable_gatetype [ drive_strength ] [ delay3 ] enable_gate_instance
          { , enable_gate_instance } ;
    | mos_switchtype [ delay3 ] mos_switch_instance
          { , mos_switch_instance } ;
    | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
    | pass_en_switchtype [ delay3 ] pass_en_switch_instance
          { , pass_en_switch_instance } ;
    | cmos_switchtype [ delay3 ] cmos_switch_instance
          { , cmos_switch_instance } ;
    | pullup [ pullup_strength ] pull_gate_instance { , pull_gate_instance } ;
    | pulldown [ pulldown_strength ] pull_gate_instance
          { , pull_gate_instance } ;

hex_base ::=
    'h | 'H

hex_digit ::=
        x | X | z | Z
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    | a | b | c | d | e | f | A | B | C | D | E | F

hex_number ::=
    [ size ] hex_base hex_digit { _ | hex_digit }

identifier :=
    IDENTIFIER [ { . IDENTIFIER } ]
    /* The period may not be followed or preceded by a space */

IDENTIFIER ::=
        simple_identifier
    | escaped_identifier

init_val ::=
    1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0

initial_construct ::=
    initial
        statement

inout_declaration ::=
    inout [ range ] list_of_port_identifiers ;

inout_terminal ::=
        terminal_identifier
    | terminal_identifier [ constant_expression ]

input_declaration ::=
    input [ range ] list_of_port_identifiers ;

input_identifier ::=
        input_port_identifier
    | inout_port_identifier

input_terminal ::=
    scalar_expression

integer_declaration ::=
    integer list_of_register_identifiers ;
```

level_input_list ::=
    level_symbol { level_symbol }

level_symbol ::=
    0 | 1 | x | X | ? | b | B

limit_value ::=
    constant_mintypmax_expression

list_of_module_connections ::=
      ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }

list_of_net_assignments ::=
    net_assignment { , net_assignment }

list_of_net_decl_assignments ::=
    net_decl_assignment { , net_decl_assignment }

list_of_net_identifiers ::=
    *net*_identifier { , *net*_identifier }

list_of_param_assignments ::=
    param_assignment { , param_assignment }

list_of_path_delay_expressions ::=
      *t*_path_delay_expression
    | *trise*_path_delay_expression , *tfall*_path_delay_expression
    | *trise*_path_delay_expression , *tfall*_path_delay_expression ,
        *tz*_path_delay_expression
    | *t01*_path_delay_expression , *t10*_path_delay_expression ,
        *t0z*_path_delay_expression , *tz1*_path_delay_expression ,
        *t1z*_path_delay_expression , *tz0*_path_delay_expression
    | *t01*_path_delay_expression , *t10*_path_delay_expression ,
        *t0z*_path_delay_expression , *tz1*_path_delay_expression ,
        *t1z*_path_delay_expression , *tz0*_path_delay_expression ,
        *t0x*_path_delay_expression , *tx1*_path_delay_expression ,
        *t1x*_path_delay_expression , *tx0*_path_delay_expression ,
        *txz*_path_delay_expression , *tzx*_path_delay_expression

list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }

list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }

list_of_port_identifiers ::=
    *port*_identifer { , *port*_identifier }

list_of_ports ::=
    (port { , port })

list_of_real_identifiers ::=
    *real*_identifier { , *real*_identifier }

list_of_register_identifiers ::=
    register_name { , register_name }

list_of_specparam_assignments ::=
    specparam_assignment { , specparam_assignment }

long_comment ::=
    /* comment_text */

loop_statement ::=
      forever statement

```
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( reg_assignment ; expression ; reg_assignment ) statement

mintypmax_expression ::=
      expression
    | expression : expression : expression

module_declaration ::=
    module_keyword module_identifier [ list_of_ports ] ;
      { module_item }
    endmodule

module_instance ::=
    name_of_instance ( [ list_of_module_connections ] )

module_instantiation ::=
    module_identifier [ parameter_value_assignment ] module_instance
      { , module_instance } ;

module_item ::=
      module_item_declaration
    | parameter_override
    | continuous_assign
    | gate_instantiation
    | udp_instantiation
    | module_instantiation
    | specify_block
    | initial_construct
    | always_construct

module_item_declaration ::=
      parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | task_declaration
    | function_declaration

module_keyword ::=
    module | macromodule

mos_switch_instance ::=
    [ name_of_gate_instance ] ( output_terminal , input_terminal ,
      enable_terminal )

mos_switchtype ::=
    nmos | pmos | rnmos | rpmos

multiple_concatenation ::=
    { expression { expression { , expression } } }

n_input_gate_instance ::=
    [ name_of_gate_instance ] ( output_terminal , input_terminal
      { , input_terminal } )
```

```
n_input_gatetype ::=
    and | nand | or | nor | xor | xnor

n_output_gate_instance ::=
    [ name_of_gate_instance ] ( output_terminal { , output_terminal } ,
      input_terminal )

n_output_gatetype ::=
    buf | not

name_of_gate_instance ::=
    gate_instance_identifier [ range ]

name_of_instance ::=
    module_instance_identifier [ range ]

name_of_system_function ::=
    $identifier

name_of_udp_instance ::=
    udp_instance_identifier [ range ]

named_port_connection ::=
    . port_identifier ( [ expression ] )

ncontrol_terminal ::=
    scalar_expression

net_assignment ::=
    net_lvalue = expression

net_decl_assignment ::=
    net_identifier = expression

net_declaration ::=
      net_type [ vectored | scalared ] [ range ] [ delay3 ] list_of_net_identifiers ;
    | trireg [ vectored | scalared ] [ charge_strength ] [ range ] [ delay3 ]
        list_of_net_identifiers ;
    | net_type [ vectored | scalared ] [ drive_strength ] [ range ] [ delay3 ]
        list_of_net_decl_assignments ;

net_lvalue ::=
      net_identifier
    | net_identifier [ expression ]
    | net_identifier [ msb_constant_expression : lsb_constant_expression ]
    | net_concatenation

net_type ::=
    wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior

next_state ::=
    output_symbol | -

non_blocking_assignment ::=
    reg_lvalue <= [ delay_or_event_control ] expression

notify_register ::=
    register_identifier

number ::=
      decimal_number
    | octal_number
    | binary_number
    | hex_number
    | real_number

octal_base ::=
```

```
    'o | 'O
octal_digit ::=
    x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
octal_number ::=
    [ size ] octal_base octal_digit { _ | octal_digit }
ordered_port_connection ::=
    [ expression ]
output_declaration ::=
    output [ range ] list_of_port_identifiers ;
output_identifier ::=
      output_port_identifier
    | inout_port_identifier
output_symbol ::=
    0 | 1 | x | X
output_terminal ::=
      terminal_identifier
    | terminal_identifier [ constant_expression ]
par_block ::=
    fork
      [ : block_identifier
        { block_item_declaration } ]
      { statement }
    join
parallel_edge_sensitive_path_description ::=
    ( [ edge_identifier ] specify_input_terminal_descriptor =>
      specify_output_terminal_descriptor [ polarity_operator ] :
      data_source_expression )
parallel_path_description ::=
    ( specify_input_terminal_descriptor [ polarity_operator ] =>
      specify_output_terminal_descriptor )
param_assignment ::=
    parameter_identifier = constant_expression
parameter_declaration ::=
    parameter list_of_param_assignments ;
parameter_override ::=
    defparam list_of_param_assignments ;
parameter_value_assignment ::=
    # ( expression { , expression } )
pass_en_switchtype ::=
    tranif0 | tranif1 | rtranif1 | rtranif0
pass_en_switch_instance ::=
    [ name_of_gate_instance ] ( inout_terminal , inout_terminal ,
      enable_terminal )
pass_switch_instance ::=
    [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
pass_switchtype ::=
    tran | rtran
path_declaration ::=
```

```
        simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;

path_delay_expression ::=
    constant_mintypmax_expression

path_delay_value ::=
        list_of_path_delay_expressions
    | ( list_of_path_delay_expressions )

pcontrol_terminal ::=
    scalar_expression

polarity_operator ::=
    + | -

port ::=
        [ port_expression ]
    | .port_identifier ( [ port_expression ] )

port_expression ::=
        port_reference
    | { port_reference { , port_reference } }

port_reference ::=
        port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ msb_constant_expression : lsb_constant_expression ]

primary ::=
        number
    | identifier
    | identifier [ expression ]
    | identifier [ msb_constant_expression : lsb_constant_expression ]
    | concatenation
    | multiple_concatenation
    | function_call
    | ( mintypmax_expression )

procedural_continuous_assignment ::=
        assign reg_assignment ;
    | deassign reg_lvalue ;
    | force reg_assignment ;
    | force net_assignment ;
    | release reg_lvalue ;
    | release net_lvalue ;

procedural_timing_control_statement ::=
    delay_or_event_control statement_or_null

pull_gate_instance ::=
    [ name_of_gate_instance ] ( output_terminal )

pulldown_strength ::=
        ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 )

pullup_strength ::=
        ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength1 )
```

```
pulse_control_specparam ::=
        PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] );
    |   PATHPULSE$specify_input_terminal_descriptor /*no space; continue*/
        $specify_output_terminal_descriptor = ( reject_limit_value
        [ , error_limit_value ] );[1]

range ::=
    [ msb_constant_expression : lsb_constant_expression ]

range_or_type ::=
    range | integer | real | realtime | time

real_declaration ::=
    real list_of_real_identifiers ;

real_number ::=
        [ sign ] unsigned_number . unsigned_number
    |   [ sign ] unsigned_number [ . unsigned_number ] e [ sign ]
          unsigned_number
    |   [ sign ] unsigned_number [ . unsigned_number ] E [ sign ]
          unsigned_number

realtime_declaration ::=
    realtime list_of_real_identifiers ;

reg_assignment ::=
    reg_lvalue = expression

reg_declaration ::=
    reg [ range ] list_of_register_identifiers ;

reg_lvalue ::=
        reg_identifier
    |   reg_identifier [ expression ]
    |   reg_identifier [ msb_constant_expression : lsb_constant_expression ]
    |   reg_concatenation

register_name ::=
        register_identifier
    |   memory_identifier [ upper_limit_constant_expression :
          lower_limit_constant_expression ]

scalar_constant ::=
    1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

scalar_timing_check_condition ::=
        expression
    |   ~ expression
    |   expression == scalar_constant
    |   expression === scalar_constant
    |   expression != scalar_constant
    |   expression !== scalar_constant

seq_block ::=
    begin
      [ : block_identifier
          { block_item_declaration } ]
      { statement }
    end

seq_input_list ::=
    level_input_list | edge_input_list

sequential_body ::=
```

```
    [ udp_initial_statement ]
    table
      sequential_entry
      { sequential_entry }
    endtable

sequential_entry ::=
    seq_input_list : current_state : next_state ;

short_comment ::=
    // comment_text \n

sign ::=
    + | -

simple_identifier ::=
    [a-zA-Z][a-zA-Z_$0-9]

simple_path_declaration ::=
      parallel_path_description = path_delay_value
    | full_path_description = path_delay_value

size ::=
    unsigned_number

source_text ::=
    { description }

specify_block ::=
    specify
      { specify_item }
    endspecify

specify_input_terminal_descriptor ::=
      input_identifier
    | input_identifier [ constant_expression ]
    | input_identifier [ msb_constant_expression : lsb_constant_expression ]

specify_item ::=
      specparam_declaration
    | path_declaration
    | system_timing_check

specify_output_terminal_descriptor ::=
      output_identifier
    | output_identifier [ constant_expression ]
    | output_identifier [ msb_constant_expression : lsb_constant_expression ]

specify_terminal_descriptor ::=
      specify_input_terminal_descriptor
    | specify_output_terminal_descriptor

specparam_assignment ::=
      specparam_identifier = constant_expression
    | pulse_control_specparam

specparam_declaration ::=
    specparam list_of_specparam_assignments ;

state_dependent_path_declaration ::=
      if ( conditional_expression ) simple_path_declaration
    | if ( conditional_expression ) edge_sensitive_path_declaration
    | ifnone simple_path_declaration

statement ::=
```

```
        blocking_assignment ;
    | non_blocking_assignment ;
    | procedural_continuous_assignment ;
    | procedural_timing_control_statement
    | conditional_statement
    | case_statement
    | loop_statement
    | wait_statement
    | disable_statement
    | event_trigger
    | seq_block
    | par_block
    | task_enable
    | system_task_enable
statement_or_null ::=
    statement | ;
strength0 ::=
    supply0 | strong0 | pull0 | weak0
strength1 ::=
    supply1 | strong1 | pull1 | weak1
string ::=
    " { ANY_ASCII_CHARACTERS_EXCEPT_NEWLINE } "

                    | path_declaration
system_task_name ::=
    $identifier
    /* The $ cannot be followed by a space */
system_timing_check ::=
        $setup ( timing_check_event , timing_check_event , timing_check_limit
          [ , notify_register ] ) ;
    | $hold ( timing_check_event , timing_check_event , timing_check_limit
          [ , notify_register ] ) ;
    | $period ( controlled_timing_check_event , timing_check_limit
          [ , notify_register ] ) ;
    | $width ( controlled_timing_check_event , timing_check_limit ,
          constant_expression [ , notify_register ] ) ;
    | $skew ( timing_check_event , timing_check_event , timing_check_limit
          [ , notify_register ] ) ;
    | $recovery ( controlled_timing_check_event , timing_check_event ,
          timing_check_limit [ , notify_tegister ] ) ;
    | $setuphold ( timing_check_event , timing_check_event ,
          timing_check_limit , timing_check_limit [ , notify_register ] ) ;
task_declaration ::=
    task task_identifier ;
      { task_item_declaration }
      statement_or_null
    endtask
task_enable ::=
    task_identifier [ ( expression { , expression } ) ] ;
task_item_declaration ::=
        block_item_declaration
    | input_declaration
```

```
      | output_declaration
      | inout_declaration
time_declaration ::=
    time list_of_register_identifiers ;
timing_check_condition ::=
        scalar_timing_check_condition
    | ( scalar_timing_check_condition )
timing_check_event ::=
    [ timing_check_event_control ] specify_terminal_descriptor
      [ &&& timing_check_condition ]
timing_check_event_control ::=
        posedge
    | negedge
    | edge_control_specifier
timing_check_limit ::=
    expression
udp_body ::=
        combinational_body
    | sequential_body
udp_declaration ::=
    primitive udp_identifier ( udp_port_list ) ;
      udp_port_declaration
      { udp_port_declaration }
      udp_body
    endprimitive
udp_initial_statement ::=
    initial udp_output_port_identifier = init_val ;
udp_instance ::=
    [ name_of_udp_instance ] ( output_port_connection , input_port_connection
      { , input_port_connection } )
udp_instantiation ::=
    udp_identifier [ drive_strength ] [ delay2 ] udp_instance { , udp_instance } ;
udp_port_declaration ::=
        output_declaration
    | input_declaration
    | reg_declaration
udp_port_list ::=
    output_port_identifier , input_port_identifier { , input_port_identifier }
unary_operator ::=
    + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
unsigned_number ::=
    decimal_digit { _ | decimal_digit }
wait_statement ::=
    wait ( expression ) statement_or_null
white_space ::=
    space | tab | newline
```

参考文献

Prentice Hall, NJ, 1998.

2. Bhasker J., *Verilog HDL Synthesis: A Practical Primer*, Star Galaxy Pu lishing, PA, 1998, ISBN 0-9650391-5-3.
3. Lee J., *Verilog Quickstart*, Kluwer Academic, MA, 1997.
4. Palnitkar S., *Verilog HDL: A Guide to Digital Design and Synthes* Prentice Hall, NJ, 1996, ISBN 0-13-451675-3.
5. Sagdeo V., *Complete Verilog Book*, Kluwer Academic, MA, 1998.
6. Smith D., *HDL Chip Design*, Doone Publications, 1996.
7. Sternheim E., R. Singh and Y. Trivedi, *Digital Design and Synthesis w Verilog HDL*, Automata Publishing Company, CA, 1993.
8. Thomas D. and P. Moorby, *The Verilog Hardware Description Langua* Kluwer Academic, MA, 1991, ISBN 0-7923-9126-8.
10. Open Verilog International, *OVI Standard Delay File (SDF) Format Manual*.