Verilog HDL 的基础知识

第一节　Verilog HDL 的基础语言知识

1．1　　综述

硬件描述语言(HDL)概述

硬件描述语言（Hardware Description Language）是硬件设计人员和电子设计自动化（EDA）工具之间的界面。其主要目的是用来编写设计文件，建立电子系统行为级的仿真模型。即利用计算机的巨大能力对用 Verilog HDL 或 VHDL 建模的复杂数字逻辑进行仿真，然后再自动综合以生成符合要求且在电路结构上可以实现的数字逻辑网表（Netlist），根据网表和某种工艺的器件自动生成具体电路，然后生成该工艺条件下这种具体电路的延时模型。仿真验证无误后，用于制造 ASIC 芯片或写入 EPLD 和 FPGA 器件中。

在 EDA 技术领域中把用 HDL 语言建立的数字模型称为软核（Soft Core），把用 HDL 建模和综合后生成的网表称为固核（Hade Core），对这些模块的重复利用缩短了开发时间，提高了产品开发率，提高了设计效率。

随着 PC 平台上的 EDA 工具的发展，PC 平台上的 Verilog HDL 和 VHDL 仿真综合性能已相当优越，这就为大规模普及这种新技术铺平了道路。目前国内只有少数重点设计单位和高校有一些工作站平台上的 EDA 工具，而且大多数只是做一些线路图和版图级的仿真与设计，只有个别单位展开了利用 Verilog HDL 和 VHDL 模型（包括可综合和不可综合的）进行复杂的数字逻辑系统的设计。随着电子系统向集成化、大规模、高速度的方向发展，HDL 语言将成为电子系统硬件设计人员必须掌握的语言。

为何使用硬件描述语言

传统的用原理图设计电路的方法已逐渐消失，取而代之，HDL 语言正被人们广泛接受，出现这种情况有以下几点原因:

电路设计将继续保持向大规模和高复杂度发展的趋势。90 年代,设计的规模将达到百万门的数量级。作为科学技术大幅度提高的产物,芯片的集成度和设计的复杂度都大大增加，芯片的集成密度已达到一百万个晶体管以上，为使如此复杂的芯片变得易于人脑的理解，用一种高级语言来表达其功能性而隐藏具体实现的细节是很必要的。这也就是在大系统程序编写中高级程序设计语言代替汇编语言的原因。工程人员将不得不使用 HDL 进行设计，而把具体实现留给逻辑综合工具去完成。

电子领域的竞争越来越激烈。刚刚涉入电子市场的成员要面对巨大的压力：提高逻辑设计的效率，降低设计成本，更重要的是缩短设计周期。多方位的仿真可以在设计完成之前检测到其错误，这样能够减少设计重复的次数。因此，有效的 HDL 语言和主计算机仿真系统在将设计错误的数目减少到最低限方面起到不可估量的作用，并使第一次投片便能成功地实现芯片的功能成为可能。

探测各种设计方案将变成一件很容易，很便利的事情，因为只需要对描述语言进行修改，这比更改电路原理图原型要容易实现得多。

硬件描述语言的发展历史

早期的集成电路设计实际上就是掩模设计；电路的规模是非常小的，电路的复杂度也很低，工作方式则主要依靠手工作业和个体劳动。40 年后的今天，超大规模集成电路（VLSI）的电路规模都在百万门量极；由于集成电路大规模、高密度、高速度的需求，使电子设计愈来愈复杂，为了完成 10 万门以上的设计，需要制定一套新的方法。就是采用硬件描述语言设计数字电路。HDL（Hardware Description Language）于 1992 年由 Iverson 提出，随后

许多高等学校、科研单位、大型计算机厂商都相继推出了各自的 HDL，但最终成为 IEEE 技术标准的仅有两个，即 VHDL 和 Verilog HDL。Verilog HDL 语言提供非常简洁，可读性很强的句法，使用 Verilog 语言已经成功地设计了许多大规模的硬件。

Verilog HDL 是在 1983 年，由 GDA（Gate Way Design Automation）公司的 Phil Moorby 首创的。Phil Mooby 后来成为 Verilog-XL 的主要设计者和 Cadence 公司（Cadence Design System）的第一个合伙人。在 1984-1985 年 Moorby 设计出第一个关于 Verilog-XL 的仿真器，1986 年他对 Verilog HDL 的发展又作出另一个巨大贡献，提出了用于快速门级仿真的 XL 算法。

随着 Verilog-XL 算法的成功，Verilog HDL 语言得到迅速发展。1989 年，Cadence 公司收购了 GDA 公司，Verilog HDL 语言成为 Cadence 公司的私有财产。1990 年，Cadence 公司公开了 Verilog HDL 语言，成立了 OVI（Open Verilog Intemational）组织来负责 Verilog HDL 的发展。IEEE 于 1995 年制定了 Verilog HDL 的 IEEE 标准，即 Verilog HDL 1364-1995。

1987 年，IEEE 接受 VHDL（VHSIC Hadeware Description Language）为标准 HDL，即 IEEE1076-87 标准，1993 年进一步修订，定为 ANSI/IEEE1076-93 标准。现在很多 EDA 供应商都把 VHDL 作为其 EDA 软件输入/输出的标准。例如，Cadence、Synopsys、Viewlogic、Mentor Graphic 等厂商都提供了对 VHDL 的支持。

HDL 语言的主要特征

- HDL 语言既包含一些高层程序设计语言的结构形式，同时也兼顾描述硬件线路连接的具体构件
- 通过使用结构级或行为级描述，可以在不同的抽象层次描述设计。HDL 语言采用自顶向下的数字电路设计方法，主要包括三个领域五个抽象层次，如表 1 所示:

内容 / 领域 / 抽象层次	行为领域	结构领域	物理领域
系统级	性能描述	部件及它们之间的逻辑连接方式	芯片、模块、电路板和物理划分的子系统
算法级	I/O 应答算法级	硬件模块数据结构	部件之间的物理连接、电路板、底盘等
寄存器传输级	并行操作寄存器传输、状态表	算术运算部件、多路选择器、寄存器总线、微定序器、微存储器之间的物理连接方式	芯片、宏单元
逻辑级	用布尔方程叙述	门电路、触发器、锁存器	标准单元布图
电路级	微分方程表达	晶体管、电阻、电容、电感元件	晶体管布图

表 1：HDL 抽象层次描述表

- HDL 语言是并发的，即具有在同一时刻执行多任务的能力。一般来讲，编程语言是非并行的，但在实际硬件中许多操作都是在同一时刻发生的，所以 HDL 语言具有并发的特征。
- HDL 语言有时序的概念。一般来讲，编程语言是没有时序概念的，但在硬件电路中从输入到输出总是有延迟存在的，为描述这些特征 HDL 语言需要建立时序的概念。因此，使用 HDL 除了可以描述硬件电路的功能外，还可以描述其时序要求。

Verilog HDL 与 VHDL 的比较

由于 Verilog HDL 早在 1983 年就已推出，至今已有十三年的历史，因而 Verilog HDL 拥有广泛的设计群体，成熟的资源比 VHDL 丰富。而 Verilog HDL 与 VHDL 相比最大的优点是：它是一种非常容易掌握的硬件描述语言，而掌握 VHDL 设计技术就比较困难。

目前版本的 Verilog HDL 和 VHDL 在行为级抽象建模的覆盖范围方面也有所不同。一般认为 Verilog HDL 在系统抽象方面比 VHDL 强一些。Verilog HDL 较为适合算法级（Alogrithem）、寄存器传输级（RTL）、逻辑级（Logic）、门级（Gate）、设计。而 VHDL 更为适合特大型的系统级（System）设计。

Verilog HDL 设计流程及设计方法简介

1、设计流程 （见图 1）

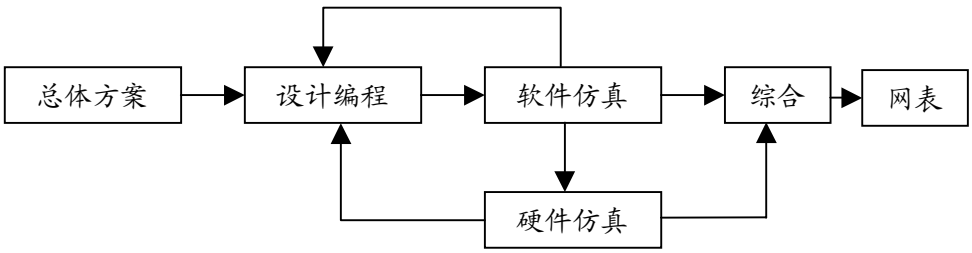


图 1：Verilog HDL 设计流程

注：1、总体方案是芯片级的。

2、软件仿真用来检测程序上的逻辑错误。

3、硬件仿真要根据需要搭成硬件电路，检查逻辑和时序上的错误。使用 FPGA（现场可编程门阵列）速度比正常慢 10 倍以上，而且只能检查逻辑错误，不能检查时序错误。

2、设计方法

I. 自下而上的设计方法

自下而上的设计是一种传统的设计方法，对设计进行逐次划分的过程是从存在的基本单元出发的，设计树最末枝上的单元要么是已经制造出的单元，要么是其他项目已开发好的单元或者是可外购得到的单元。这种设计方法与只用硬件在模拟实验板上建立一个系统的步骤有密切联系。

优点：

·设计人员对于用这种方法进行设计比较熟悉。

·实现各个子块电路所需的时间短。

缺点：

·一般来讲，对系统的整体功能把握不足。

·实现整个系统的功能所需的时间长，因为必须先将各个小模块完成。使用这种方法对设计人员之间相互进行协作有比较高的要求。

II. 自上而下（Top-Down）的设计方法

自上而下的设计是从系统级开始，把系统划分为基本单元，然后再把每个基本单元划分为下一层次的基本单元，一直这样做下去，直到可以直接用 EDA 元件库中的元件来实现为止。

优点：

·在设计周期伊始就做好了系统分析。

·由于设计的主要仿真和调试过程是在高层次完成的，所以能够早期发现结构设计上

的错误，避免设计工作的浪费，同时也减少了逻辑仿真的工作量。

·自顶向下的设计方法方便了从系统划分和管理整个项目，使得几十万门甚至几百万门规模的复杂数字电路的设计成为可能。并可减少设计人员，避免不必要的重复设计，提高了设计的一次成功率。

缺点:

·得到的最小单元不标准。

·制造成本高。

III. 综合设计方法

复杂数字逻辑电路和系统的设计过程通常是以上两种设计方法的结合。设计时需要考虑多个目标的综合平衡。在高层系统用自上而下的设计方法来实现，而在低层系统使用自下而上的方法从库元件或数据库中调用已有的单元设计。

这种方法兼有两种设计方法的优点，而且可以使用矢量测试库进行测试。

硬件描述语言新的发展

当前 EDA 工具所需解决的问题是如何大幅度提高设计能力，为此出现了一系列对 HDL 语言的扩展。

OO-VHDL（Object-Orieted VHDL），即面向对象的 VHDL。主要是引入了新的语言对象 EntityObject.此外，OO-VHDL 中的 Entity 和 Architecture 具备了继承机制，不同的 EntityObject 之间可以用消息来通信。因而 OO-VHDL 通过引入 EntityObject 作为抽象、封装和模块性的基本单元解决了 VHDL 在抽象性的不足和在封装性上能力不强等问题，也通过其继承机制解决了实际设计中的一些问题。且由于 OO-VHDL 模型的代码比 VHDL 模型短 30%—50%，开发时间缩短，提高了设计效率。

杜克大学发展的 DE-VHDL（Duke Extended VHDL）通过增加 3 条语句，使设计者可以在 VHDL 描述中调用不可综合的子系统（包括连接该子系统和激活相应功能）。杜克大学用 DE-VHDL 进行一些多芯片系统的设计，极大地提高了设计能力。

1998 年将通过得 Verilog HDL 新标准，将把 Verilog HDL-A 并入 Verilog HDL 设计中，使其不仅支持数字逻辑电路的描述，还支持模拟电路的描述，因而在混合信号电路设计中，将会得到广泛的应用。在亚微米和深亚微米 ASIC 及高密度 FPGA 中，Verilog HDL 的发展前景很大。

1．2　程序结构

作为高级语言的一种，Verilog 语言以模块集合的形式来描述数字系统，其中每一个模块都有接口部分，用来描述与其它模块之间的连接。一般说来，一个文件就是一个模块，但并不绝对如此。这些模块是并行运行的，但通常用一个高层模块来定义一个封闭的系统，包括测试数据和硬件描述。这一高层模块将调用其它模块的实例。

模块代表硬件上的逻辑实体，其范围可以从简单的门到整个大的系统，比如，一个计数器、一个存储子系统、一个微处理器等。模块可以根据描述方法的不同定义成行为型或结构型（或者是二者的组合）。行为型模块通过传统的编程语言结构定义数字系统（模块）的状态，如使用 if 条件语句、赋值语句等。结构型模块将数字系统（模块）的状态表达为具有层次概念的互相连接的子模块。其最底层的元件必须是基元或已定义过的行为型模块。Verilog 的基元包括门电路，如与非门， 和传输二极管（开关）。

模块的结构如下:

module　<模块名>（<端口列表>);

<定义>

<模块条目>

endmodule

其中，<模块名>是模块唯一性的标识符；<端口列表>是输入、输出和双向端口的列表，这些端口用来与其它模块进行连接；<定义>一段程序用来指定数据对象为寄存器型、存储器型、线型以及过程块，诸如函数块和任务块；而<模块条目>可以是 initial 结构、always 结构、连续赋值或模块实例。

下面是一个 NAND 与非模块的行为型描述,输出 out 是输入 in1 和 in2 相与后求反的结果。

```
//与非门的行为型描述
module NAND(in1, in2, out);
        input in1, in2;
        output out;
//连续赋值语句
    assign out = ~(in1 & in2);
endmodule
```

in1、in2 和 out 端口指定为线型的。assign 连续赋值语句不间断地监视等式右端变量，一旦其发生变化，右端表达式被重新赋值后结果传给等式左端进行输出。

连续赋值语句用来描述组合电路，一旦其输入发生变动，输出也随之而改变。

下例所示是一与门模块的结构型描述，这一与门是通过将一个 NAND 的输出连到另一 NAND 的两个输入上得到的。

```
//由两个 NAND 生成的与门的结构型描述
module AND(in1, in2, out);
    input in1, in2;
    output out;
    wire w1;
//两个 NAND 模块实例
    NAND NAND1(in1, in2, w1);
    NAND NAND2(w1, w1, out);
endmodule
```

这个模块含有两个 NAND 模块实例，分别是 NAND1 和 NAND2，通过内部连线 w1 连接起来。

调用模块实例的一般形式为:

<模块名> <参数列表> <实例名> (<端口列表>);

在此，<参数列表>是传输到模块实例的参数值。参数传递的典型应用是定义门级延迟，在 1．7．1 小节将有详细的说明。

下面是一个高层模块的例子，在这一模块中设置了测试数据并对变量进行监测。

```
//测试以上两个模块的高层模块
```

```
module test_AND;
reg a, b;
wire out1, out2;

initial begin    //测试数据
    a = 0; b = 0;
    #1 a = 1;
    #1 b = 1;
    #1 a = 0;
end

initial begin    //设置监测功能
    $monitor ("Time=%0d a=%b b=%b out1=%b out2=%b", $time, a, b, out1, out2);
end
//模块 AND 和 NAND 实例
AND gate1(a, b, out2);
NAND gate2(a, b, out1);

endmodule
```

应注意的是 a 和 b 的值要保持一定的时间，因此，使用了 1 位的寄存器。寄存器型变量存储过程赋值的最终结果(同传统的命令式编程语言相类似)。线型变量则没有存储能力，它们需要被持续驱动，比如用连续赋值语句或由一个模块的输出进行驱动，若线型输入的左端悬空，其值为未知的 x。

连续赋值使用关键词 assign，而过程赋值的形式是<寄存器变量> = <表达式>，其中<寄存器变量>必须是寄存器型或存储器型变量。过程赋值只允许出现在 initial 和 always 结构块中。

前者 initial 结构块中的语句顺序执行，一些语句设定了延迟#1，表示一个仿真时间单位的延迟。always 结构块与 initial 结构块功能相同，但它是无限循环的过程(直到仿真停止)。initial 和 always 结构多用来描述时序逻辑(即有限状态自动控制)

Verilog 语言中，过程赋值和 assign 连续赋值之间区别很大。过程赋值改变一个寄存器的状态，即时序逻辑；而连续赋值用来描述组合逻辑。连续赋值语句驱动线型变量，输入操作数的值一发生变化，就重新计算并更新它所驱动的变量。掌握这一区别很有必要。

将这三个模块放到一个文件中，仿真后将产生如下的结果:

Time=0 a=0 b=0 out1=1 out2=0

Time=1 a=1 b=0 out1=1 out2=0

Time=2 a=1 b=1 out1=0 out2=1

Time=3 a=0 b=1 out1=1 out2=0

在此无循环操作，仿真器执行所有的事件后自行停止，因此不需要指定仿真结束时间。

Verilog 语言的三种描述方法

- 结构型描述

是通过实例进行描述的方法。将 Verilog 预定义的基元实例嵌入到语言中，监控实例的输入，一旦其中任何一个发生变化，便重新运算并输出。

- 数据流型描述

是一种描述组合功能的方法，用 assign 连续赋值语句来实现。连续赋值语句完成如下的组合功能：等式右边的所有变量受持续监控，每当这些变量中有任何一个发生变化，整个表达式被重新赋值并送给等式左端。这种描述方法只能用来实现组合功能。

- 行为型描述

是一种使用高级语言的方法，它和用软件编程语言描述没有什么不同。具有很强的通用性和有效性。它是通过行为实例来实现的，关键词是 always，其含义是，一旦赋值给定，仿真器便等待变量的下一次变化，有无限循环之意。

使用 Verilog 进行简单设计的实例

程序如下：

```
module MUX2_1 (out, a, b, sel);   //端口定义
    output out;
    input a, b, sel;              //输入输出列表
    not (sel_, sel);
    and (a1, a, sel_);
    and (b1, b, sel);
    or (out, a1, b1);             //结构描述
endmodule
```

对应硬件电路如图 2 所示。

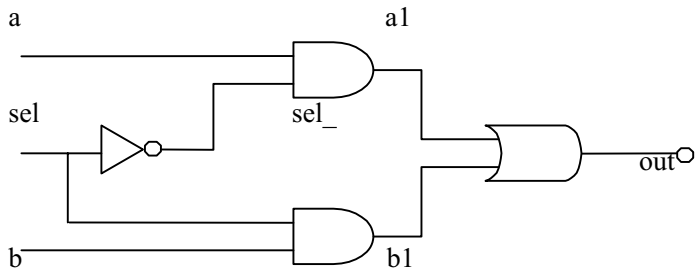


图 2：MUX2_1 模块电路示意图

a, b 和 sel 是设备的输入端口，out 是输出端口，所有信号都从这些端口流入和输出。and，or，not 是 Verilog 中预定义好的基元实例，在结构型描述中使用。关键词 module 和 endmodule 之间包含完整的二选一多路选择器的设计实现。当在其它模块中用到这一多路选择器的模块时只需使用其模块名和所定义的端口名，不需要知道其内部的具体实现。这是至上而下设计方法的一个主要特征，因为一个模块的实现方法可以从行为级转换到门级，而对使用它的高层模块不产生任何影响。

1．3　词法习俗

Verilog HDL 的源文本文件是由一串词法标识符构成的，一个词法标识符包含一个或若干个字符。源文件中这些标识符的排放格式很自由，也就是说，在句法上间隔和换行只是将这些标识符分隔开来，并不具有重要意义，转意（escaped）标识符(见后面的详细说明)

除外。

Verilog 语言中词法标识符的类型有以下几种:

——间隔符

——注释符

——算子

——数值

——字符串

——标识符

——关键词

接下来对这些标识符一一进行说明。

1、 间隔符

间隔符包括空格字符、制表符、换行以及换页符，这些字符除了起到与其它词法标识符相分隔的作用外可以被忽略，但是在字符串中空白和制表符会被认为是有意义的字符。

2、 注释

Verilog HDL 有两种注释形式，单行注释和段注释（多行）。单行注释以两个字符 “//” 起始，以新的一行作为结束；而段注释则是以/*起始，以*/结束。段注释不允许嵌套。在段注释中单行注释标识符//没有任何特殊意义。

3、 算子

算子是由单个、两个或三个字符组成的序列串，它用在表达式中，在第节讲述了表达式中算子的使用。

一元算子放在操作数的左侧，二元算子的位置在两个操作数之间，条件算子有两个算子字符分隔三个操作数。

4、 数值

Verilog HDL 的数值集合由以下四个基本的值组成:

0—代表逻辑 0 或假状态

1—代表逻辑 1 或真状态

x—逻辑不定态

z—高阻态

常数按照其数值类型可以划分为整数和实数两种。

Verilog HDL 的整数可以是十进制、十六进制、八进制或二进制的，格式为:

<位宽>' <基数> <数值>

- 位宽 ：描述常量所含位数的十进制整数，是可选项，如果没有这一项，可以从常量的值推断出
- 基数 ：可选项，可以是 b，B，d，D，o，O，h 或 H，分别表示二进制、八进制、十进制和十六进制。基数缺省默认为十进制数。
- 数值: 是由基数所决定的表示常量真实值的一串 ASC Ⅱ码。如果基数定义为 b 或 B，数值可以是 0，1，x，X，z 或 Z。若基数是 o 或 O，数值还可以是 2，3，4，5，6，7；若基数是 h 或 H，数值还可以是 8，9，a，A，b，B，c，C，d，D，

e，E，f，F。对于基数为 d 或 D 的情况，数值符可以是任何的十进制数，0 到 9，但不可以是 X 或 Z。举例如下:

15　　（十进制 15）

'h15　　（十进制 21，十六进制 15）

5'b10011　（十进制 19，二进制 10011）

12'h01F　（十进制 31，十六进制 01F）

'b01x　　（无十进制值，二进制 01x）

注:

a、数值常量中的下划线"_"是为了增加可读性，可以忽略，如: 8'b1100_0001 是 8 位二进制数。

b、在给寄存器型数据赋值时，有大小的负数并不使用符号扩展的方法生成。

c、数值常量中的“？”表示高阻状态，如: 2'B1?表示 2 位的二进制数，其中的一位是高阻状态。

Verilog 中实数用双精度浮点型数据来描述。实数既可以用小数（如 12.79）也可以用科学计数法的方式（如 24e7，表示 24 乘以 10 的 7 次方）来表达。带小数点的实数在小数点两侧都必须至少有一位数字。例如:

1.2

0.5

128.7496

1.7E8（指数符号可以是 e 或 E）

57.6e-3

0.1e-0

123.374_286_e-9　（下划线忽略）

下面的几个例子是无效的格式:

.25

3.

7.E3

.8e-2

实数可以转化为整数，根据四舍五入的原则，而不是截断原则。当将实数赋给一个整数时，这种转化会自行发生。例如，在转化成整数时，实数 25.5 和 25.8 都变成 26，而 25.2 则变成 25。

5、字符串

字符串常量是一行上写在双引号之间的字符序列串，在表达式和赋值语句中字符串用作算子，且要转换成无符号整型常量，用一串 8 位二进制 ASCⅡ码的形式表示，每一个 8 位二进制 ASCⅡ码代表一个字符。例如: 字符串“ab”等价于 16'h5758。

字符串变量是寄存器型变量，它具有与字符串的字符数乘以 8 相等的位宽。

例如:

存储 12 个字符的字符串“Hello China!”需要 8*12，即 96 位宽的寄存器。

reg [8*12:1] str1;

```
initial begin
   str = “Hello China!”
end
```

使用 Verilog HDL 的操作符可以对字符串进行处理，被操作符处理的数据是 8 位 ASCⅡ码的顺序。

Verilog 支持 C 语言中的转意符，如\t, \n, \\, \"和%%等。

6、 标识符、关键字和系统名称

标识符是赋给对象的唯一的名字，用这个标识符来提及相应的对象。标识符可以是字母、数字、$符和下划线（_）字符的任意组合序列，但它必须以字母（大小写）或下划线开头，不能是数字或$符。标识符是区分大小写的。例如: atack_del, clk_in1, _shift3，o$284 等。非法命名如下：34net，a*b_net。

逃逸标识符（Escaped identifiers）以反斜杠"\"开始，以空格结束，这种命名可以包含任何可印刷的 ASCⅡ字符，反斜杠和空格不属于名称的一部分。如：\~#@sel ，\{A,B} ，\busa+index 等。

关键字是预先定义的非逃逸标识符，用来定义语言结构，所有的关键字都是用小写方式定义的，附录 A 给出了所有已定义的关键字列表。

系统任务标识符：$<identifier>，其中$表示引入一个语言结构，其后所跟的标识符是系统任务或系统函数的名称。$<identifier>系统任务或系统函数标识符可以在三处进行定义:

——$<identifier>系统任务和函数的标准集合

——使用 PLI（Programming Language Interface）定义附加的<identifier>系统任务和函数

——通过软件工具定义附加的<identifier>系统任务和函数

系统功能可以执行不同的操作:

——实时显示当前仿真时间（$time）

——显示/监视信号的值（$display, $monitor）

——暂停仿真（$stop）

——结束仿真（$finish）

例：$monitor($time, "a = %b, b = %h", a, b);

每次 a 或 b 信号的值发生变化，这一系统任务的调用负责显示当前仿真时间、二进制格式的 a 信号和十六进制格式的 b 信号。

1．4　数据类型

Verilog HDL 的数据类型集合表示在硬件数字电路中数据进行存储和传输的要素。

Verilog 语言支持抽象数据类型，如整型、实型等；同时也支持物理数据类型，可代表真实的硬件。

1．4．1　按物理数据类型分

Verilog 中变量的物理数据类型分为线型和寄存器型两种。这两种类型的变量在定义时要设置位宽，缺省值为一位。变量的每一位可以是 0，1，X 或 Z。X 代表一个未被预置初始状态的变量或是由于两个或更多个驱动装置试图将之设定为不同的值而引起的冲突型线

型变量，Z 代表高阻状态或浮空量。

线型数据包括：wire，wand，wor 等几种类型。在被一个以上激励源驱动时，不同的线型数据有各自决定其最终值的分辨办法。

寄存器型数据与线型数据的区别在于：寄存器型数据保持最后一次的赋值，而线型数据需要有持续的驱动。

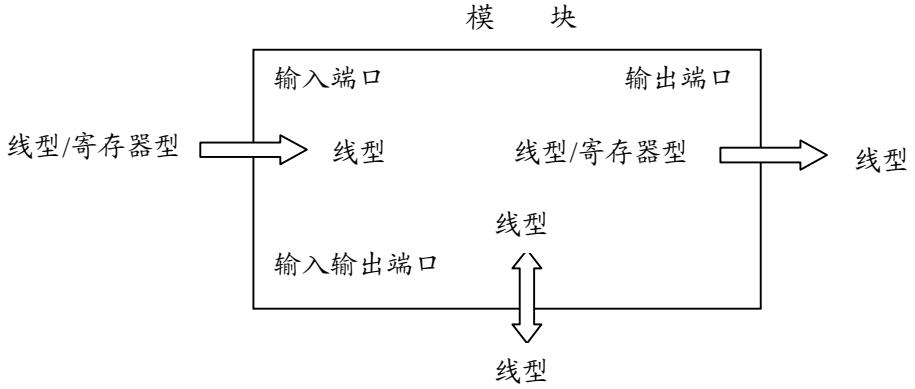


图 3：不同数据类型驱动规则图

注：1)、只能由线型/寄存器型驱动线型

2)、使用 assign 语句被连续赋值的变量必须是线型的；在 always 或 initial 程序块（行为模块）中用“=”赋值的变量必须是寄存器型的。

1．4．2　　按抽象数据类型分

分以下几种:

- integer 整型

在算术运算中整型数据被视为二进制补码形式的有符号数，而寄存器型数据当作无符号数来处理，除此以外整型数据与 32 位寄存器型数据在实际意义上相同。

- time 时间型

时间型变量与整型相类似，只是它是 64 位的无符号数。

- real 实型

实型数据在机器码表示法中是浮点型数值。Verilog 提供了将实型数据转换成位矢量以及相反过程的系统功能。

- event 事件型

它是一种特殊的变量类型，不具有任何值，作用是使模块不同部分的事件在时间上同步。

- parameter 参数型

参数型数据是被命名的常量，在仿真开始前对其赋值，在整个仿真过程中保持其置不变，数据的具体类型是由所赋的值来决定的。可以用它来定义变量的位宽以及延迟时间等。

1．5　　运算符和表达式

Verilog 语言参考了 C 语言中大多数算符的语义和句法，一个例外，Verilog 中没有增 1++

和减 1——运算符。

1．5．1 算术运算符

在 Verilog HDL 语言中，算术运算符又称为二进制运算符，共有下面几种:

1　+（加法运算符，或正值运算符，如 a + b，+3);

2　–（减法运算符，或负值运算符，如 a – 3，-3);

3　*（乘法运算符，如 a*3);

4　/（除法运算符，如 5/3);

5　%（模运算符，或称为求余运算符，要求%两侧均为整型数据，如 7%3 的值为 1)。

在进行整数除法运算时，结果值要略去小数部分，只取整数部分；而进行取模运算时结果的符号位采用模运算式里第一个操作数的符号位。如:

模运算表达式	结　果	说　明
11%3	2	余数为 2
12%3	0	整除，即余数为 0，
-10%3	-1	结果取第一个操作数的符号位，所以余数为-1

注意：在进行算术运算操作时，如果某一操作数有不确定的值 X，则运算结果也是不定值 X。%算子回送第一个操作数除以第二个操作数的余数。

下面是算术运算符应用的一个例子:

```
module arithmetic(a, b, out1, out2, out3, out4, out5)
    input [2:0] a, b;
    output [3:0] out1;
    output [4:0] out3;
    output [2:0] out2, out4, out5;
    reg [3:0] out1;
    reg [4:0] out3;
    reg [2:0] out2, out4, out5;
  always @(a or b)
  begin
    out1 = a+b;          //加运算
    out2 = a−b;          //减运算
    out3 = a*b;          //乘法运算
    out4 = a/b;          //除法运算
    out5 = a%b;          //取模运算
  end
endmodule
```

1．5．2 符号运算符

这类运算符只是将正号（+）和负号（-）赋给单个的操作数，通常操作数在定义时是没有符号的，在这种情况下，默认它为正值。

使用符号运算符的例子如下:

```
module sign(a, b, out1, out2, out3);
    input [2:0] a, b;
    output [3:0] out1, out2, out3;
    reg [3:0] out1, out2, out3;

  always @ (a or b)
  begin
    out1 = +a / -b;
    out2 = -a + -b;
    out3 = a * -b;
  end
endmodule
```

1．5．3 关系运算符

关系运算符共有以下 4 种:

1　>　　大于;

2　>=　　大于等于;

3　<　　小于;

4　<=　　小于等于;

5　==　　逻辑相等;

6　!=　　逻辑不相等;

7　===　　实例相等;

8　!==　　实例不相等。

在进行关系运算时，如果操作数之间的关系成立，返回值为 1；反之，关系不成立，则返回值为 0；若某一个操作数的值不定，则关系是模糊的，返回值是不定值 X。

关系运算符的使用方法见下例:

```
module relation(a, b, out1, out2, out3, out4);
    input [2:0] a, b;
    output out1, out2, out3, out4;
    reg out1, out2, out3, out4;

  always @ (a or b)
    begin
      out1 = a < b;          //小于运算
      out2 = a <= b;         //小于等于运算
```

```
        out3 = a > b;          //大于运算
        if (a >= b);           //大于等于运算
          out4 = 1;
        else
          out4 = 0;
      end
endmodule
```

在 Verilog 语言种有 4 种等式运算符，即 5~8。前二者为逻辑等式运算符，后二者为实例等式运算符，它们对其操作数进行比较，然后置一位标志位。二者区别在于，若操作数含有一位 X 或 Z，逻辑算子置位为 X，而实例算子可以比较含有 X 和 Z 的操作数。举例如下：

```
module  equequ;
  initial begin
  $display (″ 'bx == 'bx is %b″ , 'bx == 'bx);
  $display (″ 'bx === 'bx is %b″ , 'bx === 'bx);
$display (″ 'bz != 'bx is %b″ , 'bz != 'bx);
$display (″ 'bz !== 'bx is %b″ , 'bz !== 'bx);
  end
endmodule
```

模块 equequ 的执行会产生如下结果：

```
'bx == 'bx is x
'bx === 'bx is 1
'bz != 'bx is x
'bz !== 'bx is 1
```

所有的关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符的优先级别。

1．5．4 逻辑运算符

在 Verilog HDL 语言中有 3 种逻辑运算符：

1　&&　　逻辑与

2　||　　逻辑或

3　!　　逻辑非

“&&”和“||”是二目运算符，要求有两个操作数，如（a＞b）&&（b ＞ c），（a ＜ b）||（b ＜ c）。而“!”是单目运算符，只要求一个操作数，如!（a ＞ b）。下表为逻辑运算的真值表，它表示当 a 和 b 的值为不同的组合时，各种逻辑运算所得到的结果。

a	b	！a	！b	a&&b	A‖b
1	1	0	0	1	1
1	0	0	1	0	1

0	1	1	0	0	1
0	0	1	1	0	0

表 2：逻辑运算符真值表

逻辑运算符中“&&”和“||”的优先级别低于关系运算符，“!”高于算术运算符。逻辑运算符与其它高级语言的用法基本相似，在此不再举例说明。

1．5．5 位逻辑运算符

在 Verilog 语言中有 7 种位逻辑运算符:

1 ~ （非）

2 & （与）

3 | （或）

4 ^ （异或）

5 ^~ （同或）

6 ~& （与非）

7 ~| （或非）

位逻辑运算符对其自变量的每一位进行操作，例如，表达式 A & B 的结果是 A 和 B 的对应位相与的值。对具有不定值的位进行操作，视情况而定会得到不同的结果。例如：x 和 FALSE 相与得结果 x，x 和 TRUE 相或得结果 TURE。如果操作数的长度不相等，较短的操作数将用 0 来补位，逐位运算将返回一个与两个操作数中位宽较大的一个等宽的值。

在此需要注意的是，不要将逻辑运算符和位运算符相混淆，比如,！是逻辑非，而~是位操作的非，即按位取反，例如：对于前者！（5＝＝6）结果是 TRUE，后者对位进行操作，~{1，0，1，1}＝0100。

1．5．6 一元约简运算符

约简运算符是单目运算符，也有与、或、非运算。其与、或、非运算规则类似于位运算符的与、或、非运算规则，但其运算过程不同。位运算是对操作数的相应位进行与、或、非运算，操作数是几位数则运算结果也是几位数。而约简运算则不同，约简运算是对单个操作数进行与、或、非递推运算，最后的运算结果是 1 位的二进制数。约简运算的具体运算过程是：1° 先将操作数的第 1 位与第 2 位进行与、或、非运算；2° 将运算结果与第 3 位进行与、或、非运算，依次类推，直至最后一位。

例如:

```
reg [3:0]   B;
reg   C;
C = &B;
```

相当于:

```
C = ( (B[0]&B[1]) & B[2] ) & B[3];
```

一完整的模块举例如下:

```
module reduction(a, out1, out2, out3, out4, out5, out6);
    input [3:0] a;
```

```
    output out1, out2, out3, out4, out5, out6;
    reg out1, out2, out3, out4, out5, out6;

  always @ (a)
  begin
    out1 = & a;          //与约简运算
    out2 = | a;          //或约简运算
    out3 = ~& a;         //与非约简运算
    out4 = ~| a;         //或非约简运算
    out5 = ^ a;          //异或约简运算
    out6 = ~^ a;         //同或约简运算
  end

endmodule
```

1．5．7 其它运算符

1　　移位运算符

在 Verilog HDL 语言中有两种移位运算符：“<<”（左移位运算符）和“>>”（右移位运算符），方法是将第一个操作数向左（右）移，所移动的位数由第二个操作数来决定。这两种移位运算都用 0 来填补移出的空位。举例如下：

```
module    shift;
    reg[3:0]   a, b;
    initial
    begin
        a = 1;          //a 设为 0001
        b = (a<<2);     //移位后，a 的值为 0100，赋给 b
    end
endmodule
```

从此例可以看出，b 在移过两位后，用 0 来填补空出的位。

进行移位运算时应注意移位前后变量的位数，下面给出几个例子：

```
4’b1001<<1 = 5’b10010;      4’b1001<<2 = 6’b100100;
1<<6 = 32’b1000000;         4’b1001>> 1= 4’b0100;       4’b1001>>4 = 4’b0000;
```

2　　条件运算符

条件运算符（:?）有三个操作数，第一个操作数是 TRUE，算子返回第二个操作数，否则返回第三个操作数，条件算子可以用来实现一个选择器，例如：

```
    module conditional(time, y);
        input [2:0] time;
        output [2:0] y;
```

```
reg [2:0] y;
parameter zero = 3'b000;
parameter timeout = 3'b111;

always @(time)
   y = (time != timeout) ? time +1 : zero;
endmodule
```

嵌套的条件算子可用来实现多路选择。如:

```
wire  [1:0]  absval;
assign  absval = (a>0) ? 1 : (a<0) ? 2 : 0;
```

3　　并接运算符

Verilog HDL 语言中有一个特殊的运算符：并接运算符{}。这一运算符可以将两个或更多个信号的某些位并接起来进行运算操作。其使用方法是把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号，即:

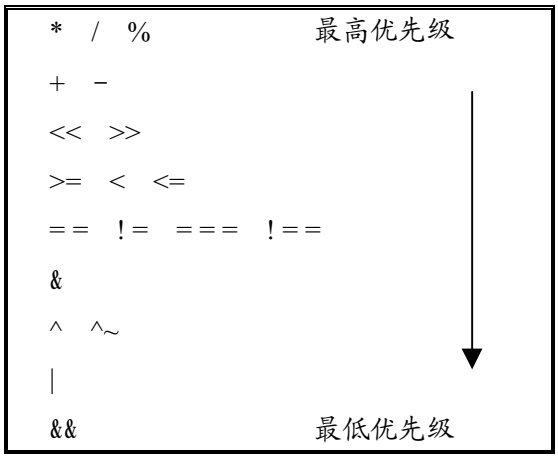
{信号 1 的某几位，信号 2 的某几位，……，信号 n 的某几位}

例如，{a[3],b,c[2:0],4'b0100}，{2'b1x，4'h7} = = = 6'b1x0111。

此外，在 Verilog 语言中还有一种重复操作符{{}}，即将一个表达式放入双重花括号中，复制因子放在第一层括号中。它为复制一个常量或变量提供一种简便记法。例如，{3{2'b01}} === 6'b010101。

1．5．8 运算符优先级排序

运算符优先级顺序如下所示:



将操作数和算子通过正常的优先级规则组合起来便生成表达式，该结构将会产生一个结果，这一结果是操作数的值和算子的语法含义的函数。一个合法的操作数不加任何运算符也可以被认为是一个表达式，例如一个线型的位选操作数。表达式用来计算数值，但不仅仅单独用来求值，而是构成语句或其它结构的一部分。表达式中的操作数可以是变量名（a），变量的某一位（a[i]），变量的连续数位（a[3:0]），或者是功能块调用。

1．6　　控制结构

Verilog HDL 语言含有丰富的控制语句，可以选择不同的控制语句生成程序代码的过程

块，比如在 initial 和 always 结构块中。大部分控制语句与传统的编程语言，如 C 语言相似。Verilog HDL 语言与 C 语言之间最大的区别在于，C 语言中的括弧{}，在 Verilog 中用 begin 和 end 代替，而在 Verilog 中括弧{}用来完成字符的位并接功能。由于大部分使用者对 C 语言都很熟悉，接下来的各小结对每一种结构进行举例介绍。

1．6．1 选择结构

选择结构包括 if 和 case 语句

1　if 语句

Verilog HDL 语言中的 if 语句与 C 语言的十分相似，使用起来也很简单。通过下例进行说明：

```
if  (a < 0 )
    begin
      b = 1;
    end
else
    begin
      b = 0;
    end
```

2　case 语句

与 C 语言的 case 语句不同，Verilog 语言中，选择第一个与<表达式>的值相匹配的<数值>，并执行相关的语句，然后控制指针将转移到 endcase 语句之后，也就是说，与 C 语言不同的是，它不需要 break 语句。其形式如下：

```
case  (<表达式>)
  <数值>：<语句>
  <数值>：<语句>
  default：<语句>
endcase
```

下面的例子检查了 1 位信号的值。

```
case  (sig)
  1'bz:  $display("Signal is floating");
  1'bx:  $display("Signal is unknown");
  default:  $display("Signal is %b", sig);
endcase
```

另举例如下：

```
module  case_statement;
  integer  i;
  initial  i = 0;
  always  begin
```

```
        $display  (″ i = %0d″ , i );
        case  (i)
            0: i = i + 2;
            1: i = i + 7;
            2: i = i − 1;
            default : $stop;
        endcase
        end
    endmodule
```

该模块的运行结果如下:

```
i = 0
i = 2
i = 1
i = 8
```

选择表达式要与 case 表达式逐位相对照，若没有事件相匹配则执行 default 事件，若 default 事件不存在，便执行 case 语句后的下一条语句。

1．6．2 重复结构

重复结构包括 for 循环语句、while 循环语句、repeat 重复语句和 forever 循环语句。

1 for 循环结构

for 循环语句与 C 语言的 for 循环语句非常相似，只是 Verilog 中没有增 1++和减 1−−运算符，因此，要使用 i = i + 1 的形式。

如下所示为 for_loop 的应用的简单例子:

```
    module  for_loop;
        integer  i;
        initial
        for  (i = 0; i < 4; i = i + 1)  begin
          $display  (″ i = %0d (%b binary) ″ , i, i);
        end
    endmodule
```

执行结果如下:

```
i = 0  (0 binary)
i = 1  (1 binary)
i = 2  (10 binary)
i = 3  (11 binary)
```

2 while 循环结构

用 while 语句同样可以实现上例的功能，得到相同的执行结果。

```
    module while_loop;
```

```
    integer   i;
    initial   begin
        i = 0;
    while   (i < 4)   begin
    $display   (″ i = %d (%b binary) ″ , i, i); );
    i = i + 1;
    end
  end
endmodule
```

3　repeat（重复）循环结构

Verilog 有两个其它编程语言中不常用的结构；repeat 和下面要介绍的 forever 结构，下面描述的是一个等待 5 个时钟周期然后停止仿真的 repeat 循环。

```
module   repeat_loop   (clock);
    input   clock;
    initial   begin
      repeat   (5)
      @(posedge   clock);
      $stop;
    end
endmodule
```

4　forever 循环结构

forever 循环用来监控一些条件，当条件发生时显示一条信息。如下例所示：

```
    module   forever_statement(a, b, c);
      input   a, b, c;
      initial   forever   begin
          @(a or b or c)
          if   (a + b = = c)   begin
              $display   (″ a(%d) + b(%d) = c(%d) ″ , a, b, c);
              $stop;
              end
          end
    endmodule
```

虽然 repeat 和 forever 循环语句都可以通过其它控制语句来实现（如 for 循环语句），但它们使用起来十分简便，尤其是在通过键盘发布交互式命令时，还有不需要事先定义任何变量的好处。

1．7　其它语句

在 Verilog 语言中，除了上述的重复结构外还有参数语句、赋值语句等其它结构，下面对这几种结构一一进行说明。

1．7．1 参数语句

参数语句允许设计者给常量起一个名字，其典型应用是定义寄存器和延迟的宽度，在程序中任何可以使用字母之处都可以使用参数。参数定义的句法为:

参数 <赋值列表>

在此，<赋值列表>是用逗号隔开的参数列表以及它们的值。

下面的设计是使用参数进行定义的一个例子。

```
module   mod1(out, in1, in2);
    …
    parameter   p1 = 8,
                real_constant = 1.079,
                x_word = 16'bx,
                file = "/net/usr/design/mem_file.dat";
    …
    wire   [p1:0]   w1;       //用参数进行定义的线型变量
    …
endmodule
```

Verilog 语言允许用户在编译时对参数重新赋值而改变其值，方法有两种:

1．使用 defparam 语句

2．在同一模块实例中使用 "#" 符号

在上例的基础上通过以下两个例子分别说明这两种方法。

```
module   p_value;
    …
    mod1   I1(out, in1, in2);
    defparam
        I1.p1 = 6,
        I1.file = "../my_mem.dat";
    …
endmodule
```

可以看到，使用 defparam 语句进行重新赋值时必须参照原参数的名字生成分级参数名。

```
module   top;
…
mod1 # (5, 3.0, 16'bx, "../my_mem.dat") I1(out, in1, in2);
…
endmodule
```

这种方法与基元实例的延迟定义相似。参数赋值的顺序必须与原始模块中进行参数定义的顺序相同，并不是一定要给所有的参数都赋予新值，但不允许跳过任何一个参数，即使是保持不变的值也要写在相应的位置。

1．7．2 连续赋值语句

连续赋值语句用来驱动线型变量，这一线型变量必须已经事先定义过。只要输入端操作数的值发生变化，该语句就重新计算并刷新赋值结果。我们可以使用连续赋值语句来描述组合逻辑，而不需要用门电路和互连线。在前面一节中已经对连续赋值做了介绍，关键词 assign 用来区分连续赋值语句和过程赋值语句，下面一条语句将线型端口 in1 和 in2 相与,并用这一结果去驱动 out 信号:

```
wire    out;
assign out = a & b;


wire #10 inv = ~in;


wire    eq;
assign    eq = (a = = b);
```

下面是连续赋值语句中线型变量使用强度定义的例子:

```
    wire    (strong1, weak0)    [7:0]    net1 = net2 & net3;
```

连续赋值语句中还可以使用延迟定义，

```
    tri    #10    xor_net = a ^ b;
```

Verilog 由于允许在一次定义中进行多路赋值，

```
    wire    and_net = a1 & a2,
    or_net = a1 | a2;
```

还可以在连续赋值语句的左端设置并接操作，

```
    assign    {carry_out, sum} = ina + inb + carry_in;
```

下面是延迟为 5 的多路连续赋值语句，

```
    assign    #5    c = a[0], d = {r1, r2, r3}, f[3:2] = {r3, r4};
```

1．7．3 阻塞和无阻塞过程赋值

简单的阻塞过程赋值语句有如下三种形式:

```
    lhs_expression = expression;
    lhs_expression = #delay expression;
    lhs_expression = @event expression;
```

lhs（左端）可以是一个变量名、变量的某一特定位、变量的指定几位或是对变量的并置。形式 1 中，仿真器先对右端表达式进行计算然后立即将结果赋给左端；形式 2，仿真器计算右端表达式后要等待 delay（延迟）时间，再将值赋到左端；形式 3 下，仿真器要等到 event 事件发生才把右端的值赋给左端，这三种赋值形式都要等到赋值操作后才能执行下一条语句。

无阻塞（Unblock）过程赋值语句在句法上与阻塞（block）过程赋值语句相似，只是用"<="代替了"="。

```
    lhs_expression <= expression;
    lhs_expression <= #delay expression;
    lhs_expression <= @event expression;
```

二者的差别在于无阻塞赋值语句右端计算好后并不立即赋给左端，在要赋值的同时控制下一条语句的继续执行。无阻塞赋值语句对描述数据流很简便，例如，要模拟一个移位寄存器，语句的顺序很重要:

```
stage1 = (#1)    stage2;
stage2 = (#1)    stage3;
stage3 = (#1)    stage4;
stage4 = (#1)    stage5;
```

而如下的编写方法便可以不考虑其顺序:

```
stage1 <= (#1)    stage2;
stage2 <= (#1)    stage3;
stage3 <= (#1)    stage4;
stage4 <= (#1)    stage5;
```

1．8　任务和函数结构

Verilog 语言中一种最有效的仿真方法就是将一段代码封闭起来形成任务（task）或函数（function）结构。

任务和函数结构之间有几点差异:

1) 一个任务块可以含有时间控制结构，而函数块则没有，也就是说函数块从零仿真时刻开始运行，结束后立即返回（实质上是组合功能）。而任务块在继续下面的运行过程之前其初始化代码必须保持到任务全部执行结束或是失效。

2) 一个任务块可以有输入和输出，而一个函数块必须有至少一个输入，没有任何输出，函数结构通过自身的名字返回结果。

3) 任务块的引发是通过一条语句，而函数块只有当它被引用在一个表达式中时才会生效。例如:

```
tsk (out, in1, in2);   //调用了任务结构，名为 tsk
```

而

```
i = func(a, b, c);
assign x = func(Y); //调用了一个函数，名为 func
```

如下所示是一个任务模块的例子:

```
task    tsk;
    input i1, i2;
    output o1, o2;
    $display("Task tsk, i1 = %b, i2 = %b", i1, i2);
    #1 o1 = i1 & i2;
    #1 o2 = i1    |    i2;
endmodule
```

函数块举例如下:

```
function [7:0] func;
    input i1;
```

```
    integer i1;
    reg [7:0] rg;
    begin
        rg = 1;
        for (i = 1; i <=i1; i = i + 1)
        rg = rg+1;
        func = rg;
    end
endfunction
```

函数块在编组代码以及增强其可读性和可维护性方面是一种十分重要的工具。

1．9　时序控制

在执行仿真进程语句之前，Verilog 语言提供了两种类型的显式时序控制，一种是延迟控制，在这种类型的时序控制中通过表达式定义了开始遇到这一语句和真正执行这一语句之间的延迟时间。另一种为事件控制，这种时序控制是通过事件表达式来完成的，只有当某一事件发生时才允许语句继续向下执行。在第 3 小节中讲述了 wait 等待语句，其原理是使仿真进程处于等待状态直到某一特定的变量发生变化。

Verilog 具有离散事件时间仿真器的特性，也就是说，在离散的时间点预先安排好各个事件并将它们按照时间顺序排成事件等待队列，最先发生的事件排在等待队列的最前面，而较迟发生的事件依次放在其后。仿真器为当前仿真时间移动整个事件队列并启动相应的进程。在运行的过程中，有可能为后续进程生成更多的事件，放置在队列中适当的位置。只有当前时刻所有的事件都运行结束后仿真器才将仿真时间向前推进，去运行排在事件队列最前面的下一个事件。

如果没有时间控制，仿真时间将不会前进。仿真时间只能被下列形式中的一种来推进:

1. 定义过的门级或线传输延迟
2. 由#符号引入的延迟控制
3. 由@符号引入的事件控制
4. 等待语句

第 1 种形式是由门级器件来决定的，在此无须讨论，下面对 2，3，4 三种形式以及路径延迟的定义分别进行讲述。

1．9．1 延迟控制（#）

在 Verilog 语言中延迟控制的格式为:

```
#  expression
```

它是将程序的执行过程中断一定时间，时间的长度由 expression 的值来确定。

延迟控制结构的应用举例如下:

```
module  delay;
  reg [1:0]  r;
  initial  #70  $stop;
  initial  begin  :  b1
  #10  r = 1;
  #20  r = 1;
```

```
    #30   r = 1;
    end
    initial   begin   :   b2
    #5    r = 2;
    #20   r = 2;
    #30   r = 2;
    end
    always   @r   begin
        $display   (″ r = %0d at time   %0d″, r, $time);
    end
  endmodule
```

delay 模块的执行产生如下结果:

```
r = 2   at time 5
r = 1   at time 10
r = 2   at time 25
r = 1   at time 30
r = 2   at time 55
r = 1   at time 60
```

1．9．2 事件

一个事件可以通过运行表达式: —>event 变量来被激发。用事件变量来控制在同一仿真时刻运行的三个 initial 块的执行顺序的例子如下:

```
  module   event_control;
    event   e1, e2;
    initial   @e1   begin
        $display   (″ I am in the middle.″ );
        −>e2;
    end
    initial   @e2
        $display   (″ I am supposed to execute last.″ );
    initial   begin
        $display   (″ I am the first.″ );
        −>e1;
    end
  endmodule
```

even_control 模块的执行过程将产生如下结果:

```
I am the first.
I am in the middle.
I am supposed to execute last.
```

时间和事件控制结构的一种特殊形式是他们在赋值语句中的使用。赋值语句：current_state = #clock_period　next_state;等价于

temp = next_state;

#clock_period　current_state = temp;

类似地，current_state = @(posedge　clock)　next_state;等价于

temp = next_state;

@(posedge　clock)　current_state = temp;

1．9．3 等待语句

一段 Verilog 程序（如：initial 或 always 块）可以通过以下两种形式来实现等待的功能，可以重新排定自身的执行顺序：

@ event_expression

wait　(expression)

其中，形式 1 是中断执行过程直到特定事件发生。在这两种情况下都是程序的调度控制当前运行事件指针从当前仿真时刻的事件列表上移走，放到某个未运行的事件列表上。形式 2 的情况是，如果等待的表达式为假则中断运行直到（通过其它程序语句的执行）它变为真。

这两种结构以及延迟控制结构或是他们的组合都能加在任何语句之前作为一个必须满足的先决条件。例如，表达式：

@ (posedge clk)　#5　out = in

表示等到时钟上升沿到来后再等 5 个时间单位，然后将 in 赋给 out。

@ event_expression 的控制时间格式要等待事件发生才继续执行程序块的其它语句，这个事件可以是以下几种形式之一：

a)　变量<或变量>……

b)　位变量的上升沿

c)　位变量的下降沿

d)　事件变量

在格式 a 的情况下，执行过程要延迟到任何一个变量发生变化。在形式 b 和 c 中，执行过程延迟直到变量从 0，X 或 Z 变到 1，或从 1，X 或 Z 变到 0。对于形式 d，程序的执行过程被中断直到事件发生。

1．9．4 延迟定义块

Verilog HDL 语言可以对模块中某一指定的路径进行延迟定义，这一路径连接模块的输入端口（或双向端口）与输出端口（或双向端口）。延迟定义块在一个独立的块结构中定义模块的时序部分，这样功能验证就可以与时序验证相独立。这是时序驱动设计的关键部分，因为包含时序信息的这部分程序在不同的抽象层次上可以保持不变。

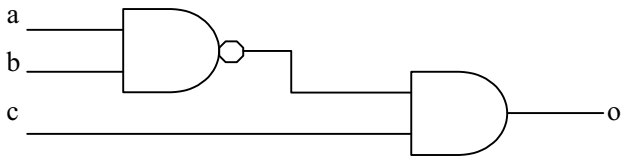
在延迟定义块中要完成的典型任务有：

——描述模块中的不同路径并给这些路径赋值

——描述时序核对，以确认硬件设备的时序约束是否能得到满足

延迟定义块的内容要放在关键字 specify 和 endspecify 之间，而且必须在某一模块内部。在定义块中还可以使用 specparam 关键字定义参数。举例说明如下：

电路图如图 4 所示。



进行路径延迟定义如下:

```
module noror (o, a, b, c);
    output o;
    input a, b, c;
      nor n1 (net1, a, b);
      or o1 (o, c, net1);
    specify
      (a => o) = 2;
      (b => o) = 3;
      (c => o) = 1;
    endspecify
endmodule
```

对于这一简单电路的延迟定义可以采用将所有的延迟集中在最后一个或门上定义的方法，简单但不精确；另一种方法就是如上述模块所做，把延迟分布在每个门上，即定义了从 a 点到 o 点的延迟为 2，从 b 点到 o 点的延迟为 3，从 c 点到 o 点延迟时间为 1。这种做法比前者精确，但要同时满足一系列等式，工作量大。

第二节 Verilog-XL 仿真

从

测试模块由以下几部分组成:

a. 数据类型的定义

b. 模块实例

c. 施加激励

d. 显示结果

上例的测试模块如下:

```
module testfixture;
      reg a, b, sel;                    //定义数据类型
      MUX2_1 mux(out, a, b, sel);       //MUX 实例
      initial
```

```
begin
    a = 0; b = 1; sel = 0;
    #5 b = 0;
    #5 b = 1; sel = 1;
    #5 a = 1;
    #5 $finish;
end                          //加激励
initial
$monitor($time, "out = %b a = %b b = %b sel = %b", out, a, b, sel);        //显示结果
endmodule
```

编译指令

编译指令用重音符(`)表示，这些指令将引起使 Verilog 编译器产生特殊的动作。编译指令将一直保持有效，除非被其他编译指令所覆盖或失效。

1) 文本交换（`define）

功能：`define 指令提供了一个简洁的文本交换的功能。

指令格式：`define <name> <macro_text>

作用：增强描述的可读性。

定义全局参数，如延迟、位宽等，好处是当需要改变配置时，秩序对一处进行修改即可。

对 Verilog 的某些命令做缩记。

应用举例:

```
`define not_delay #1
`defineand_delay #2
`define or_delay #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
  not `not_delay not1(sel_, sel);
  and `and_delay and1(a1, a, sel_);
  and `and_delay and2(b1, b, sel_);
  or `or_delay or1(out, a1, b1);
endmodule
```

注:

- 可以将一列`define 指令放到一个文件中，将这一文件与其它设计文件一起编译即可，例如:

```
verilog definitions.v mux.v
```

- +define+命令具有重新进行宏定义的功能，例：

```
verilog   +define+gate=  “or” test.v
    module test;
    `define gate and
    reg a, b;
    `gate (c, a, b);
    initial
   begin
   a=0; b=1;
   $monitor ($time, , c, a, b);
   $finish;
   end
   endmodule
```

2) 文本包含（`include）

功能：`include 编译指令用来插入一个完整文件的内容。

指令格式：`include "<file_name>"

作用：

可以把全局或经常使用的一些定义单独放到一个文件中，然后包含这个文件

可以包含一些 task 任务块，但这些任务块中不能有压缩的重复代码，这会使维护代码很困难。

例：`include "global.v"

`include "parts/count.v"

注：`include 指令最多可以嵌套八层

3) 时间尺度（`timescale）

功能：`timescale 编译指令用来定义时间单位和精度。

指令格式：`timescale <time_unit> / <time_precision>

time_unit 延迟和时间的计量单位

time_precision 时间精度，在仿真之前确定如何驱动时间

二者都是由整数和分别代表大小和单位的字符串组成，有效的整数是 1，10 和 100，字符串可以是 s，ms，ns，us，ps，fs

注：

- 如果时间单位和精度相差太大，仿真速度会大大降低，因为时间轮是由精度的倍数来驱动的。例：有一`timescale 1s / 1ps，时间轮要扫描 $10^{12}$ 次事件队列才能推进一秒，而对于`timescale 1s / 1ms，只需要扫描这些队列 10^{3} 次。
- 所有时间尺度指令的最小精度作为整个仿真的时间单位，因为仿真器需要保证设计中定义的最小时间精度，并以它的倍数来推进时间

轮。

例如：

```
`timescale 1ns/10ps
module1( ··· );
 ···
endmodule

`timescale 100ns/1ns
module2( ··· );
 ···
endmodule

`timescale 1ps/100fs
module3( ··· );
 ···
endmodule
```

仿真的时间单位是 100fs，即时间轮总是以 100fs 为倍数向前推进。第一个时间尺度表明模块 1 以 1ns 为单位，精度是 10ps，也就是设置了最小的仿真时间步长为 10ps；第二个时间尺度表示模块 2 的时间单位是 100ns，而它的精度为 1ns，因为 1ns 大于 10ps，故最小仿真补偿仍为 10ps；第三个时间尺度表示模块 3 的时间单位是 1ps，精度为 100fs，由于 100fs 小于 10ps，故最小仿真补偿变为 100fs。