Verilog HDL

硬件描述语言概述

硬件描述语言，即 Hardware Description Language（一般简写为 HDL）。顾名思义，是指对硬件电路进行行为描述、寄存器传输描述或者结构化描述的一种语言。HDL 最初只是用来对电路进行描述以方便设计者之间进行交流的。但随着其语言功能的丰富，逐步成为了电路设计的主要方式，但“硬件描述语言”这一名称却沿用至今。

HDL 最早是由 Iverson 公司与 1962 年提出的，迄今为止已经出现过多种 HDL，其中绝大多数是专有产品，如 Silvar-lisco 公司的 HHDL、Zycad 公司的 ISP、Mentor Graphics 公司的 BLM 等。一些高校和科研机构也开发过各自的 HDL，如 AHPL、MIMOLA、SCHOLAR 等。

目前主流的 HDL 主要有两种：VHDL 和 Verilog HDL。VHDL 诞生于 1982 年。在 1987 年底，VHDL 被 IEEE 和美国国防部确认为标准硬件描述语言。自 IEEE 公布了 VHDL 的标准版本，IEEE- 1076（简称 87 版)之后，各 EDA 公司相继推出了自己的 VHDL 设计环境，或宣布自己的设计工具可以和 VHDL 接口。此后 VHDL 在电子设计领域得到了广泛的接受，并逐步取代了原有的非标准的硬件描述语言。

Verilog HDL 是由 GDA(Gateway Design Automation)公司的 PhilMoorby 在 1983 年末首创的，最初只设计了一个仿真与验证工具，之后又陆续开发了相关的故障模拟与时序分析工具。1985 年 Moorby 推出它的第三个商用仿真器 Verilog-XL,获得了巨大的成功，从而使得 Verilog HDL 迅速得到推广应用。1989 年 CADENCE 公司收购了 GDA 公司，使得 VerilogHDL 成为了该公司的独家专利。1990 年 CADENCE 公司公开发表了 Verilog HDL,并成立 OVI（Open Verilog International）组织以促进 Verilog HDL 成为 IEEE 标准，即 IEEE Standard 1364-1995。后来随着 Verilog 的进一步完善，又陆续制定了 IEEE Standard 1364-2001、IEEE Standard 1364-2005 标准。

1983	Verilog 最初由 Gateway Design Automation 公司（GDA）的 Phil Moorby 创建，作为内部仿真器的语言，主要用于逻辑建模和仿真验证，被广泛使用。
1989	GDA 公司被 Cadence 公司收购，Verilog 语言成为 Cadence 公司的私有财产。
1990	Cadence 公司成立 OVI（Open Verilog International）组织，公开 Verilog 语言，促进 Verilog 向公众领域发展。
1992	OVI 决定致力于将 Verilog OVI 标准推广为 IEEE（The Institute of Electrical and Electronics Engineers）标准。
1995	OVI 的努力获得成功，IEEE 制定了 Verilog HDL 的第一个国际标准，即 IEEE Std 1364-1995，也称之为 Verilog 1.0。
2001	IEEE 发布 Verilog 第二个标准（Verilog 2.0），即 IEEE Std 1364-2001，简称为 Verilog-2001 标准。由于 Cadence 在集成电路设计领域的影响力及 Verilog 语言的简洁易用性，Verilog 成为电路设计中最流行的硬件描述语言。
2005	IEEE Standard 1364-2005 标准发布

目前国内外高校、科研机构以及企业都普遍使用 Verilog 语言进行 FPGA 开发和集成电路设计，VHDL 的用户占比在逐步下降，本书也选择 Verilog 进行介绍。

Verilog HDL 是一门标准硬件设计语言，可以适用于电子系统设计的各个阶段，如：开发、验证、综合、测试。该语言是从 C 语言发展而来，因此在语法上和 C 语言有诸多相似，便于开发人员阅读，降低了维护、修改和生成最终电路的难度。Verilog HDL 已经被绝大多数的电路设计者所采用，称为目前最流行的一种硬件设计语言。

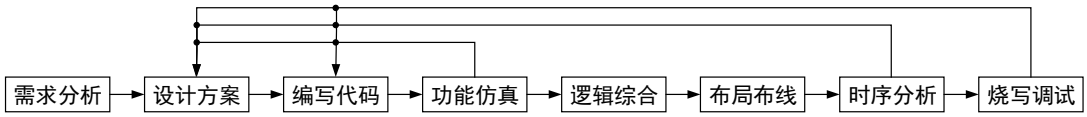
Verilog HDL 与 C 语言在语法上有许多相似之处，但是 Verilog HDL 是一门硬件描述语言，是用来描述硬件电路行为的，不能把 C 语言中对程序执行过程的理解套用到 Verilog HDL 程序上面。尤其需要区分两种语言的运行逻辑，C 语言是一种串行语言，每一条语句都会被转换成二进制的机器指令在处理器上顺序执行，而 Verilog 最终会被转换成硬件电路，电路中包含了多个功能单元，这些功能单元是可以并行运作的。例如:

```
a = b + c;
d = e + f;
```

如果对计算机工作原理有所了解，很容易知道在 C 语言中从执行顺序上来说，需要先执行完第一条语句，才能执行第二条，但是在 Verilog HDL 是用来设计硬件电路的，这种情况则会生成两个加法器，同时对这两条语句进行加法运算。

开发流程

采用 Verilog HDL 进行 FPGA 开发或者集成电路设计，一般都会遵照如下流程：需求分析、设计方案、编写代码、功能仿真、综合布线、时序约束、烧写调试，下面以 FPGA 开发为例进行介绍。



- 需求分析：分析电路需求，提炼电路功能点，一般会编写一份需求分析文档；
- 设计方案：确定电路的整体架构、功能划分和接口信号等，此阶段通常会编写一份设计方案文档。如电路功能非常简单，则可以不用撰写需求分析文档和方案设计文档；
- 编写代码：根据设计方案编写 verilog 代码，verilog 代码一般存在在后缀为.v 的文档中，该文档可以使用任何文本编辑器进行编辑，也可以使用专用的 HDL 编辑环境，推荐用户使用 notepad++编辑，不仅支持 verilog 语法高亮，此外还具有列编辑模式，可以提高代码的编写效率。在编写或修改代码过程中，如果发现设计方案存在问题，则需要返回上一步进行修改；
- 功能仿真：verilog 代码编写完成后，需要为 verilog 源文件设计仿真文件，并使用专用的仿真软件进行功能仿真，检查逻辑功能是否正确。仿真文件一般使用 verilog HDL 编写，也可以使用其它语言编写（如 system verilog），对于非常简单的设计，也可以跳过这一步骤；
- 逻辑综合：该步骤由综合工具完成，综合工具会对 verilog 源程序进行分析，最终转换成包含布尔表达式和信号连接关系的标准文件，该文件后缀为.edf，一般称为网表文件；
- 布局布线：此步骤中，通过 FPGA 厂家提供的 EDA 工具，将标准的网表文件映射到对应的 FPGA 芯片中；
- 时序分析：结合设计的时序要求和电路在 FPGA 芯片的映射情况，对电路的时序进行分析，看是否能满足时序要求，如有问题，则需要修改 verilog 代码，甚至修改设计方案；
- 烧写调试：上述所有步骤完成后，EDA 工具会生成一个可以烧写到 FPGA 芯片中的电路文件，将该文件烧写到 FPGA 芯片中，观察电路运行结果，如存在问题，则返回前面的

步骤进行修改，直至电路正常工作。

Verilog 特性

Verilog 标准

Verilog HDL 到目前为止一共发布了 3 个标准：Verilog-1995、Verilog-2001 和 Verilog-2005。Verilog-2001 在 Verilog-1995 的基础上加入了一些比较有用的特性，可以提高开发者的编码和验证效率

1	增加 generate 语句，提高模块多次实例化或选择实例化的效率
2	增强对多维数组的支持
3	增强文件 IO 的相关操作
4	增加对 task 和 function 重入的支持
5	增加 always@(*)
6	增加 part-select（+：和-：）
7	增加 localparam
8	增加新的端口声明方式
9	支持常数函数

Verilog-2005 相对于 Verilog-2001 来说并没有增加多少新特性，主要包括：把寄存器（register）类型改名为变量（variable）类型，增加了 uwire 类型，外加修正了一些书写错误。
本书涉及到的 Verilog 语法是 Verilog-2001 的一个子集，在 Verilog-2005 标准下也可正常工作，但在 Verilog-1995 标准下可能需要做少量修改。

Verilog 代码示例

```
`timescale 1ns/1ps
module 模块名(
端口声明列表
);
    //声明部分
    参数声明
    wire型变量声明
    reg型变量声明

    //赋值语句部分
    连续赋值语句(assign语句，经常使用)
    过程赋值语句(always语句，经常使用)

    //实例化语句部分
    gate实例化语句，基本不用
    UDP实例化语句，基本不用
    模块实例化语句，经常使用
endmodule
```

模块是 Verilog 的基本描述单位，用来描述一个电路单元的功能、结构以及与其他模块通信的端口。

任何一个模块从结构上都可以分为三大部分：模块接口（interface）、声明（declaration）和语句（statements），其中：

接口部分位于模块代码的开头和结尾，使用 module/endmodule 两个关键词作为模块开始和结束的标识。module 后面是模块名，可由设计者自行定义，模块名后面的括号内是该模块的端口列表；

声明部分用来定义不同的项，包括 reg/wire 型内部信号、输入输出端口、模块参数、任务、函数等。

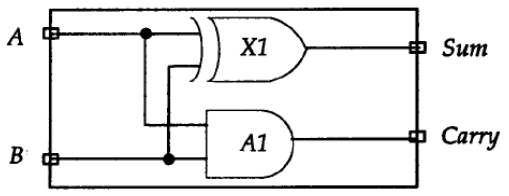
语句部分用来定义电路的功能和结构，包括初始化语句、always 过程块语句、模块实例化语句、门实例化语句、用户定于原语（UDP）实例化语句、连续赋值语句等。

声明和语句可以分散在模块中的任何位置，但寄存器、变量及参数的声明必须位于使用位置之前，因此建议将所有的声明放在语句部分之前。

一个模块可以在另一个模块中使用，以下是一个简单的例子：

模块接口部分
——以关键字module/endmodule标识
——定义模块名称
——定义端口(方向、位宽、类型、名称)
 	
声明部分
——声明参数
——声明wire型内部变量
——声明reg型内部变量
 	
语句部分
——连续赋值语句
——过程赋值语句
 	
——模块实例化语句
	`timescale 1ns/1ps
	//注释，增加代码可读性
	module 模块名(
	输入端口定义
	输出端口定义
	
	双向端口定义
	);
	//声明
	参数定义
	wire型内部变量定义；
	
	reg型内部变量定义；
	//语句
	连续赋值语句；
	过程赋值语句；
	
	模块实例化语句;
	
	endmodule
		`timescale 1ns/1ps
		//注释，增加代码可读性
		module full_adder(
		input a,b,c_in,
		output sum,
		output reg c_out
		//inout none //无双向端口
		);
		//声明
		//parameter WIDTH = 1,没有用到参数
		wire sum1,carry1;
		wire a2, sum2,carry2;
		//无reg型内部变量；
		//赋值语句
		assign sum = sum2; //连续赋值语句
		always@(*) //过程赋值语句,always过程块
		begin
		c_out = carry1 | carry2;
		end
		//模块实例化语句;
		//基于名称的端口映射，推荐使用
		half_adder ha1(
		.a (a),
		.b (b),
		.sum (sum1),
		.carry (carry1)
		);
		//基于位置的端口映射，合法但不推荐
		half_adder ha2(sum1,c_in,sum2,carry2);
		endmodule

下面以下图中的电路为例，对 Verilog 模块进行初步的了解




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

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

该模块名称叫做 HalfAdder，从名称可以看出，模块描述的是一个半加器电路。模块有 4 个端口信号，分别为：A、B、Sum、Carry，从声明部分可以看出 A、B 为输入端口，Sum、Carry 为输出端口，语句部分包含了两条连续赋值语句，实现了半加器的逻辑功能

抽象级别

Verilog 可以在三个抽象层次上对电路进行建模：行为及模型、寄存器传输级模型（Register Transfer Level，简称 RTL）和门级模型。
行为级模型是一种比较高的抽象层次，因此多用与仿真测试文件 testbench 的编写，其特点如下：

- 着重于系统行为和算法描述，不在于系统的电路实现；
- 多采用不可综合的 Verilog 语法，不能综合处电路文件；
- 功能描述主要采用高级语言结构，如 fork/join、task、function、for、repeat、while、wait、event 等

寄存器传输级（RTL）模型是采用 Verilog 设计电路时最常采用的一种模型，其特点如下：

- 是一种比较底层的模型，主要用于 FPGA 开发和集成电路设计；
- 着重于描述功能块内部或功能块之间的数据流和控制信号，重点在于描述数据或信号在寄存器之间的流动和传输；
- 采用可综合的 Verilog 语法，可以综合出门级电路模型。

门级模型是最底层的电路模型，多用于集成电路设计时后端的物理实现，其特点如下：

- 是实际电路的逻辑实现；
- 通常由 EDA 工具通过 RTL 级模型综合得出；
- 主要包含逻辑门（gate 和 switch）、用户定于原语（UDP）、模块、连线等要素。

由于开发者在实际的数字电路设计中，尤其是 FPGA 开发中多使用寄存器传输级（RTL）模型，因此 Verilog 程序常被称为 RTL 代码。
下面以半加器为例对比 RTL 级和门级模型的区别。

半加器 RTL 级模型	半加器门级模型

module half_adder(
input a,b,
output sum, carry);
assign sum = a ^ b; //连续赋值语句
assign carry = a & b; //连续赋值语句
endmodule
	module half_adder(
	input a,b,
	output sum, carry);
	xor x1(sum,a,b); //gate 实例化语句
	and a1(carry,a,b); //gate 实例化语句
	endmodule

Verilog HDL 编码风格

Verilog 代码是开发者之间以及开发者和 EDA 工具之间沟通的桥梁。我们编写的 Verilog 代码不仅要使 EDA 工具能够解析，还要方便开发者进行阅读。良好的编码风格不但可以增加代码的可读性，还可以避免电路中出现一些不易察觉到的错误。下面列举几条常见的编码规范。

- 模块划分：Verilog HDL 通过模块描述电路功能和层次关系，为增加代码的可读性，建议每个 verilog 源代码文件只包含一个模块，模块名称与文件名称保持一致，命名时做到望文生义，方便开发者通过模块名称了解到模块功能。在模块划分时，尽量按功能进行划分，模块之间不易存在过多的耦合关系。单个模块的功能不易太复杂，代码量尽量控制在 500 行以内。
- 代码编辑：编辑 Verilog 代码时对编辑器没有特殊要求，但为了确保相同代码在不同编辑器中显示同样的效果，建议使用空格进行缩进和对齐，而不要使用 tab 键。notpad++ 等编辑器支持自动将 tab 转换为空格，可提高编码效率。编辑代码时，合理使用缩进，

方便开发者快速了解代码结构。每行代码不超过 80 个字符

- 信号命名:信号命名时需要做到望文生义,使开发者从信号名称便可推断出信号的作用。Verilog HDL 对于大小写是敏感的，不建议通过大小写来区分信号，如 Clk、clk 代表两个不同的信号，但阅读时容易弄混。对于一些常见信号，有些约定俗称的命名方式，如时钟信号一般包含 clk、clock 等字符串，复位信号一般包含 rst、reset，低电平有效信号一般带有“_n”后缀。
- 注释：每个文件有一个文件头，文件头中注明文件名、功能描述、引用模块、设计者、设计时间、修改信息及版权信息等。对信号、参量、引脚、模块、函数及进程等加以说明，便于阅读与维护，如信号的作用、频率、占空比、高低电平宽度等。用“//”做小于 1 行的注释，用“/* */”做多于 1 行的注释。

第 1 部分：命令规则	每个文件只包含一个 module，module 名要小写，并且与文件名保持一致
	除 parameter 外，信号名全部小写，名字中的两个词之间用下划线连接
	由 parameter 定义的常量要求全部字母大写，自己定义的参数、类型用大写标识
	推荐用 parameter 来定义有实际意义的常数，包括单位延时、版本号、板类型、单板在位信息、LED 亮灯状态、电源状态、电扇状态等
	信号名长度不超过 20 字符
	避免使用 Verilog 和 VHDL 保留字命令
	建议给信号名添加有意义的前缀或后缀，命名符合常用命名规范（_clk 或 clk_表示时钟，n 表示低电平有效，z 表示三态信号，en 表示使能控制，rst 表示复位
	保持缩写意义在模块中的一致性
	同一信号在不同层次应该保持一致性
第 2 部分：注释	每个文件有一个文件头，文件头中注明文件名、功能描述、引用模块、设计者、设计时间、修改信息及版权信息等
	对信号、参量、引脚、模块、函数及进程等加以说明，便于阅读与维护，如信号的作用、频率、占空比、高低电平宽度等
	用“//”做小于 1 行的注释，用“/* */”做多于 1 行的注释
	更新的内容要做注释，记录修改原因，修改日期和修改人
第 3 部分：模块	module 例化名用 u_xx_x 标示
	建议每个模块加 timescale
	不要书写空的模块，即一个模块至少要有一个输入和一个输出
	为了保持代码的清晰、美观和层次感，一条语句占用一行，每行限制在 80 个字符以
	内，如果较长（超出 80 个字符）则要换行
	namebased)orderbased)name_basedorder_based
	模块的接口信号按输入、双向、输出顺序定义
	使用降序列定义向量有效位顺序，最低位是 0
	管脚和信号说明部分，一个管脚和一组总线占用一行，说明清晰

	不要采用向量的方式定义一组始终信号逻辑内部不对 input 进行驱动，在 module 内不存在没有驱动源的信号，更不能在模
	块端口存在没有驱动的输出信号，避免在 elabarate 和 compile 时产生 warning，干
	在顶层模块中，除了内部的互连和 module 的例化外，避免在做其他逻辑
	出于层次设计和同步设计的考虑，子模块输出信号建议用寄存器
	内部模块端口避免 inout，尽量在最顶层模块处理双向总线三态逻辑可以在顶层模块使用，子模块中禁止使用三态。如果能确保该信号不会被
	其它子模块使用，而是直接通过顶层模块输出要 I/O 口，可以在子模块中使用三态
	没有未连接的端口
	为逻辑升级保留的无用端口以及信号要注释
	建议采用层次化设计，模块之间相对独立
	对于层次化设计的逻辑，在升级中采用增量编译
第 4 部分：线网和寄存器	不允许锁存器和触发器在不同的 always 块中赋值，造成多重驱动出于功能仿真考虑，非阻塞赋值应该增加单位延时，尤其是对于寄存器类型的变量
	赋值时；阻塞赋值不允许使用单位延时 always 语句实现时序逻辑采用非阻塞赋值；always 语句实现的组合逻辑和 assign 语
	句块中使用阻塞赋值
	同一信号赋值不能同时使用阻塞和非阻塞方式
	不允许出现定义的 parameter/wire/reg 没有使用
	建议不使用 integer 类型寄存器
	寄存器类型的信号要初始化
	除移位寄存器外，每个 always 语句只对一个变量赋值，尽量避免在一个 always 语句，出现多个变量进行运算或赋值
第 5 部分：表达式	在表达式内使用括号表示运算的优先级
	除仿真外，不要使用 语句
	tranif1/pull gate
	logic)区分开
	一行中不能出现多个表达式
	不要给信号赋“x”态，以免 x 值传递
	设计中使用到的 0，1，z 等常数采用基数表示法书写（即表示为 1'b0,1'b1,1'bz 或十六进制）
	端口申明、比较、赋值等操作时，数据位宽要匹配
第 6 部分：条件语句	if 都有 else 和它对应
	变量在 if-else 或 case 语句中所有变量在所有分支中都赋值。如果用到 case 语句，写上 default 项
	禁止使用 casex

	case 语句 item 必须使用常数
	不允许使用常数作为 if 语句的条件表达式
	条件表达式必须是 1bit value。如异步复位，高电平有效使用“if(asynch_reset==1'b1)”，低电平有效用“if(asynch_reset==1'b0)”，不要写成“if(！asynch_reset)”或者“if(asynch_reset==0)”
	不推荐嵌套使用 5 级以上 if…else if…结构
第 7 部分：可综合的	不要使用 include 语句
	不要使用 disable、initial 等综合工具不支持的电路，而应采用复位方式进行初时化。但在 testbench 电路中可以使用
	不要使用 specify 模块
	不要使用===、！==等不可综合的操作符
	除仿真外，不要使用 fork-join 语句
	除仿真外，不要使用 while 语句
	repeat
	除仿真外，不要使用 forever 语句
	除仿真外，不要使用系统任务($)
	除仿真外，不要使用 deassign 语句
	除仿真外，不要使用 force,release 语句
	除仿真外，不要使用 named events 语句
	不要在连续赋值语句中引入驱动强度和延时
	禁止使用 trireg 型线网
	制止使用 tri1、tri0、triand 和 trior 型的连接
	不要位驱动 supply0 和 supply1 型的线网赋值
	设计中不使用 macro_module 不要在 RTL 代码中实例门级单元，尤其下列单元：CMOS/RCOMS/NMOS/PMOS/RNMOS/RPMOS/trans/rtrans/tranif0/tranif1/rtranif0/rtranif1/pull gate
第 8 部分：可重用的	考虑未使用的输入信号 power_down，避免传入不稳定态
	接口信号尽量少，接口时序尽量简单
	将状态机（FSM）电路与其它电路分开，便于综合和后端约束
	将异步电路和同步电路区分开，便于综合和后端约束
	将相关的逻辑放在一个模块内
	合理划分设计的功能模块，保证模块功能的独立性
	合理划分模块的大小，避免模块过大在设计的顶层（top）模块，将 I/O 口、Boundary scan 电路、以及设计逻辑（corelogic)区分开
第 9 部分：同步设计	在一个 module 中，在时钟信号的同一个沿动作。如果必须使用时钟上升沿和时钟下
	降沿，要分两个 module 设计在顶层模块中，时钟信号必须可见，不要在模块内部生成时钟信号，而是使用
	DCM/PLL 产生的时钟信号

	避免使用门控时钟和门控复位
	同步复位电路，建议在同一时钟域使用单一的全局同步复位电路；异步复位电路，建议使用单一的全局异步复位电路
	不要在时钟路径上添加任何 buffer
	不要在复位路径上添加任何 buffer
	避免使用 latch
	寄存器的异步复位和异步置位信号不能同时有效
	避免使用组合反馈电路
	always 有且仅有一个的敏感事件列表，敏感事件列表要完整，否则可能会造成前后仿真的结果不一致。异步复位情况下需要异步复位信号和时钟沿做敏感量，同步复
	位情况下只需要时钟沿做敏感量时钟事件的表达式要用“negedge<clk_name>”或“posedge<clock_name>”的形式
	复杂电路将组合逻辑和时序逻辑电路分成独立的 always 描述
第 10 部分：循环语句	在设计中不推荐使用循环语句。在非常有必要使用的循环语句时，可以使用 for 语句
第 11 部分：约束	对所有时钟频率和占空比都进行了约束
	对全局时钟 skew 进行了约束
	对于时序要求的路径需要针对特殊要求进行约束，如锁相环鉴相信号
	需要根据输出管脚驱动要求进行约束，包括驱动电流和信号边沿特性
	需要根据输入和输出信号的特性进行管脚上下拉约束针对关键 I/O 是否约束了输入信号和输入时钟的相位关系，控制输入信号在 CLK 信号
	之后或之前多少 ns 到达输入 pad
	fanout30fanout30 需要使用输入输出模块中的寄存器，如 Xinlinx 公司的 IOB，map properties 选项 pack I/O register/latches into IOBsactor 需要设置成为“for input and
	output”，这样可以控制管脚到内部触发器的延时时间
	布局布线报告中 IOB、LUTs、RAM 等资源利用率应小于百分之八十
	对于逻辑芯片对外输入接口，进行 tsu/th 约束；对于逻辑芯片对外输出接口，进行 t 约束
第 12 部分：PLL/DCM	如果使用 FPGA 内部 DCM 和 PLL 时，应该保证输入时钟的抖动小于 300ps，防止 DCM/PLL 失锁；如果输入时钟瞬断后必须复位 PLL/DCM 对于所有厂家的 FPGA，其片内锁相环只能使用同频率的时钟信号进行锁相。如果特殊情况下需要使用不同频率的信号进行锁相，需要得到厂家的认可，以避免出时钟
第 13 部分：	由于不同编辑器处理不同，对齐代码使用空格，而不是 tab 键

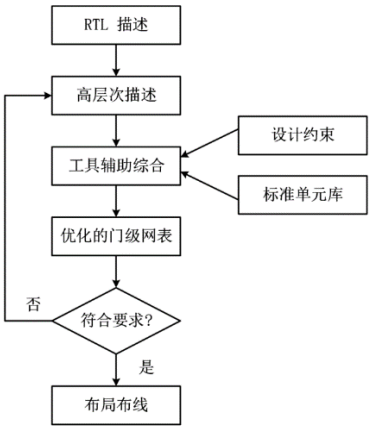
代码编辑	

Verilog 设计

Verilog HDL 的语法内容很多，Verilog-2001 标准手册共有 791 多页，是学习 Verilog 语法最权威文献，但学习起来难度也很大，本书从数字电路设计实践的角度出发，只介绍 Verilog 的一些常用语法，以帮助读者快速使用 Verilog 进行数字逻辑电路开发。

Verilog 语法可以分成可综合语法和不可综合语法两大类。综合是电路设计中的一个专业术语，就是在标准单元库和特定的设计约束基础上，把数字设计的高层次描述转换为优化的门级网表的过程。标准单元库对应工艺库，可以包含简单的与门、非门等基本逻辑门单元，也可以包含特殊的宏单元，例如乘法器、特殊的时钟触发器等。设计约束一般包括时序、负载、面积、功耗等方面的约束。

无论是数字芯片设计，还是 FPGA 开发，综合过程基本都是借用计算机辅助逻辑综合工具，自动的将高层次描述转换为逻辑门电路。设计人员可以将精力集中在系统结构方案、高层次描述、设计约束和标准工艺库等方面，而不用去关心高层次的描述如何转换为门级电路。综合工具在内部反复进行逻辑转换、优化，最终生成优化后的门级电路。该过程如下所示



可综合语法指的是综合后可以生成对应硬件电路门级网表的 Verilog 语法，不可综合语法则是指综合后不能生成对应硬件电路门级网表的 Verilog 语法。

所有综合工具都支持的结构

结构类型	关键字	描述
端口信号	inout，input，output	端口信号只有 3 种
参数	parameter, localparam	---
信号变量	wire, reg, tri, integer	---
模块	module	---
门级原语	and，nand，or，nor，xor，xnor，buf，not，bufif0，bufif1，notif0，notif1，supply0，supply1	直接调用例化即可
例化	---	支持模块例化、门级原语例化等
函数与任务	function, task	支持不含时序结构的表述
连续赋值	assign	不支持带有延迟的表述
过程赋值	always, begin, end	可设计时序逻辑或组合逻辑
条件语句	if, case, default	条件中不能包含"z"或"x"的比较
循环语句	for, while, forever	while, forever 必须包含 @(posedge clk) 或 @(negedge clk), 避免组合逻辑回路
边沿触发	negedge，posedge	---
操作符	---	支持除 "===" 与 "!==" 以外的所有操作符

所有综合工具都不支持的结构

结构类型	关键字	描述
变量类型	time	仿真时使用的时间型变量
系统任务	---	大多数系统任务都是辅助仿真，不能综合为实际电路 例如 $display, $fopen, $finish 等。
过程结构	initial	initial 常用作仿真时信号赋初值操作 或控制激励信号的时序
并行语句	fork, join	常用作仿真时并行结构的描述 always @(posedge clk) 描述的并行结构可综合
延迟语句	#	所有带延迟标志"#"的表述均不可综合 但仿真时电路中会有延时，综合时也不会报错
电平敏感触发	wait	多用于仿真中信号的检测启动
强制赋值和释放	force, release	多用于仿真中阻断其他驱动源，对信号进行强制赋值

综合工具可能支持的结构

结构类型	关键字	描述
x/z 条件语句	casex, casez	有些综合工具能识别该语句中的非"x/z"比较逻辑
不同强度的线网	wand，triand，wor，trior	当信号有多个驱动源时需要使用 但现在数字设计基本摒弃了这些变量类型
实数变量	real	往往用于仿真时的精确计算
过程终止	disable	终止过程块执行，大多数综合工具不支持该命令
循环语句	repeat, while, forever	repeat 常用作仿真中语句循环执行固定次数 while, forever 循环次数为常量时也可能可综合
用户自定义原语	UDP	其实目前大多数综合工具都支持 UDP 只是某些古老的综合工具不会识别
过程连续赋值	assign, deassign	工具大多不支持该操作下 reg 数据类型的综合 支持该操作下 wire 数据类型的综合

使用 Verilog 进行数字设计时，需要遵循以下原则：可综合的结构可大胆使用，不可综合的结构仿真中使用，有些综合工具支持有些不支持的结构尽量不使用。

除非某些特殊设计，例如 clkgate，时钟切换等电路等，否则设计中不要编写潜在的会被综合成 Latch 的逻辑。

变量声明时不要学 C 语言格式对寄存器变量进行赋初值操作。仿真时变量会有设置的初值，综合后寄存器初值是不确定的。如果信号初值会影响逻辑功能，则仿真过程可能会因验证不充分而错过查找出逻辑错误的机会。

所有内部寄存器都应该使用复位进行赋初值操作，以确保系统复位时各寄存器都有稳定的状态。因为没有复位端，综合后电路中寄存器的初始值不能确定，可能会导致功能错误。

组合逻辑使用阻塞赋值，时序逻辑使用非阻塞赋值。组合逻辑一般使用连续赋值语句 assign 描述，always 描述的电路也能被综合成组合逻辑，例如"与逻辑"可以描述如下：

```
reg       F ; //注意一定要是 reg 型变量
always @ (*)  begin
      F = A & B ;
end
```

always 语句中被赋值的信号一定要声明为 reg 型，在组合逻辑中该 reg 变量会被综合成线网，在时序逻辑中该 reg 变量会被综合成触发器。

同一个变量不能受多个时钟（或 always 块）控制，也不能受时钟的双边沿控制。此类描述也是不可综合的。

避免设计中出现 X 或 Z 值，因为综合工具只能识别 0 或 1 的逻辑值。

下面是全加器电路的 Verilog 代码，该电路由两个 Verilog 文件构成，其中 full_adder.v 是全加器顶层电路的代码描述，该电路通过实例化两个半加器来实现，半加器电路的代码则放在了 half_adder.v 文件中。我们从这个例子入手学习 Verilog 电路设计相关的语言要素。

half_adder.v

```
`timescale 1ns/1ps
//注释，增加代码可读性
module half_adder(
  input a,b,
  output sum,
  output carry
  //inout none  //无双向端口
);
//声明
  //无内部变量；
//赋值语句
  assign sum   = a ^ b; //连续赋值语句
  assign carry  = a & b; //连续赋值语句

//门级建模，功能与上述语句相同
//xor x1(sum,a,b);
//and a1(carry,a,b);

endmodule
```

full_adder.v

```
`timescale 1ns/1ps
//注释，增加代码可读性
module full_adder(
  input a,b,c_in,
  output sum,
  output reg c_out
  //inout none  //无双向端口
);
//声明
  //parameter WIDTH = 1,没有用到参数
  wire sum1,carry1;
  wire a2, sum2,carry2;
  //无reg型内部变量；
//赋值语句
  assign sum = sum2; //连续赋值语句
  always@(*)  //过程赋值语句,always过程块
  begin
    c_out = carry1 | carry2;
  end
//模块实例化语句;
  //基于名称的端口映射，推荐使用
  half_adder ha1(
  .a     (a),
  .b     (b),
  .sum  (sum1),
  .carry (carry1)
  );
  //基于位置的端口映射，不推荐使用
  half_adder ha2(sum1,c_in,sum2,carry2);
endmodule
```

时间精度
注释
标识符
关键字
端口
变量
连续赋值语句
过程赋值语句
运算符
表达式
实例化语句

从上面的代码实例中可以看出

标识符

Verilog HDL 中标识符（Identifier）是用于索引的名称，可用于声明数据，变量，端口，例化名等。又可分为普通标识符、转义标识符、生成标识符和关键字四类。转义标识符和生成标识符使用较少，此处不做详细解释，关键字将在后面单独介绍。

普通标识符是由任意字母，数字，美元符号$或下划线构成的序列，但需要注意首字符不可为数字或$，但可以是字母和下划线，并且区分大小写。普通标识符常用于定义常数、变量、信号、端口、模块例化名或参数名等，例如：input a，这里的 a 就是普通标识符，表示一个输入端口的名称。标识符是大小写敏感的，例如 input a 和 input A 中，a 与 A 表示的是两个不同的标识符。

合法标识符举例	In, in, a, _mem, abc123, money$, we_n,
非法标识符举例	123abc, $money, delay#, abc%123

注释

Verilog 和 C 语言的注释语句完全一样，提供了两种注释方式：行注释（//）和段注释（/* */），注释不作为代码的有效部分，只是起到注释的作用，提高程序的可读性。编译器在编译时自动忽略注释部分。

行注释语句是一种由双斜杠"//"构成的注释语句，只注释一行，即从 //开始到本行末都是注释部分。行注释常用来说明该行代码的含义，意图及提示等信息，也可以注释一行代码。如：

```
wire signed [3:0] a; //定义有符号 wire 类型向量 a。
```

```
wire signed [3:0] b;
//wire signed [3:0] c;
```

上面三条语句中第三条被注释了，因此编译器在编译时自动忽略该条语句。

段注释语句可以注释一段内容。例如：

```
wire signed [3:0] a; //  定义有符号 wire 类型向量 a。
/* wire signed [3:0] b;
wire signed [3:0] c;*/
```

上面三条语句中第二、第三条被注释了，因此编译器在编译时自动忽略这两条语句。

下面是 Vivado 自动生成的 Verilog 代码的文件头注释示例。一般来说在每个 Verilog 文件的开头都会有由注释语句组成的一段说明，指明文件创建日期，修改日期，作者，版权，及该文件简短说明等，称为文件头。工程师在编写文件时应尽量完善文件头信息，便于后期代码维护。可以看出，在编写 Verilog 代码时，推荐优先使用行注释，尽量不使用段注释。

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/03/24 10:33:43
// Design Name:
// Module Name: top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

关键字

关键字是 Verilog 语法保留下来用于端口定义、数据类型定义、赋值标识、进程处理等操作的特殊标识符，例如 input a 中的 input 就是一个关键字，该关键字将信号 a 定义为输入端口。关键字都是由小写字母构成，Verilog2001 标准中共定义了 123 个关键字，如下表所示：

always	defparam	for	instance	notif0	realtime	strong1	use
and	design	force	integer	notif1	reg	supply0	vectored
assign	disable	forever	join	or	release	supply1	wait
automatic	edge	fork	large	output	repeat	table	wand
begin	else	function	liblist	parameter	rnmos	task	weak0
buf	end	generate	library	pmos	rpmos	time	weak1
bufif0	endcase	genvar	localparam	posedge	rtran	tran	while
bufif1	endconfig	highz0	macromodule	primitive	rtranif0	tranif0	wire
case	endfunction	highz1	medium	pull0	rtranif1	tranif1	wor
casex	endgenerate	if	module	pull1	scalared	tri	xnor
casez	endmodule	ifnone	nand	pulldown	showcancelled	tri0	xor
cell	endprimitive	incdir	negedge	pullup	signed	tri1	
cmos	endspecify	include	nmos	pulsestyle_onevent	small	triand	
config	endtable	initial	nor	pulsestyle_ondetect	specify	trior	
deassign	endtask	inout	noshowcancelled	rcmos	specparam	trireg	
default	event	input	not	real	strong0	unsigned	

Verilog2001 标准中的关键字很多，但常用的实际上很少，尤其是可综合的常用关键字屈指可数。初学者只需要掌握其中的 20 个关键字，便可以完成绝大多数的电路设计。

always	default	endmodule	module	parameter
assign	else	if	negedge	posedge
begin	end	inout	or	reg
case	endcase	input	output	wire

下面是对关键字的说明和示例

关键字	说明	代码示例

module
endmodule
	这两个关键字用于表示模块的开始和结束，必须成对出现	module test();
		//code
		endmodule

input	表明端口类型为输入信号，该关键字一般用在模块的端口定义部分	
output	表明端口类型为输出信号，该关键字一般用在模块的端口定义部分	module test(input a,b,clk, output o); //code endmodule
wire	表明数据类型为线型，该关键字用在端口或内部信号的定义部分。一般来说，凡是通过 assign 关键字进行赋值的信号，都应该定义成 wire 类型。wire 是 Verilog 的默认数据类型，对于没有显式声明的信号，一律默认为 wire 类型	
reg	表明数据类型为寄存器类型，这是与 wire 相对应的一种数据类型，该关键字用在端口或内部信号的定义部分。一般来说，凡是在 always 语句语句内部被赋值的信号，都应该被定义成 reg 类型。在 Verilog 语法内，除了 wire 和 reg，还支持多种其它的数据类型，但用的都不多，读者现在只需掌握这两种数据类型即可	module test(input wire a,b,clk, output reg o); wire s; //code endmodule
assign	连续赋值语句关键字，此类语句将逻辑表达式的值赋给线网类型信号。一个模块内部可以有任意多个 assign 语句，但每个 assign 语句只能包含一个连续赋值表达式	

always	过程赋值语句关键字，次类语句将逻辑表达式的值赋给寄存器类型信号。一个模块内部可以有任意多个 always 语句，其一般格式为：always (时序控制) 过程语句
	always 语句是永远在循环执行的，其中过程语句一般为一条表达式赋值语句，或者 begin/end 关键字构成的顺序过程语句块。例如"always clk = ~clk；"always 后的语句重复执行，由于没有时序控制语句，将在 0 时刻无线循环，这种写法没有语法错误，但没有实际意义，因此 always 语句的执行必须带有时序控制，对于上述语句，可以改成"always #5 clk = ~clk；"该语句将产生周期为 10 个时间单位的波形（#n 表示 n 个时间单位的延时）
		module test(
		input wire a,b,
		output reg o);
		wire s;
		assign s = a & b;
		always@(*) o = s;
		/*表示任意时序控制*/
		endmodule

begin end	顺序过程语句，必须成对出现，在 begin/end 中间可以有多条语句，这些语句是顺序执行的，该关键字允许嵌套，即 begin/end 内部可以包含其他的 begin/end 语句块	
posedge	该关键字后面跟信号名，表示该信号的上升沿这一事件。一般用在 always 语句的时序控制部分	

negedge	该关键字后面跟信号名，表示该信号的下降沿这一事件。一般用在 always 语句的时序控制部分	module test(
		input wire a,b,clk,
		output reg o);
		wire s;
		assign s = a & b;
		always@(posedge clk) o <= s;
		endmodule

if	条件语句，其后跟条件判别语句，以及过程语句，有时候会与 else 语句配合使用	

else	if 语句的分支，if、else 用于实现带有优先级的条件分支，一般出现在 always 语句的过程语句部分，而不能直接在模块内部单独出现，一般用法如下：
	if(条件) 过程语句
	else 过程语句
		module test(
		input wire a,b,clk,
		output reg o);
		always@(posedge clk)
		begin
		if(a) o <= a;
		else o <= b;
		end
		endmodule

case endcase	具有相同优先级的多路条件分支，两个关键字必须成对出现。一般出现在 always 的过程语句部分，而不能在模块内部单独出现，一般用法如下： case（case 表达式） case 条目表达式 1：过程语句 case 条目表达式 2：过程语句 ... [default:过程语句] endcase	module test(input wire a,b,clk, output reg o); always@(posedge clk) case({a,b}) 2'b00: o <= 1'b0; 2'b01: o <= 1'b0; 2'b10: o <= 1'b0; 2'b11: o <= 1'b1; endcase endmodule

逻辑值

在二进制计数中，单比特的逻辑值只有“0”、“1”两种状态，但是在 Verilog 语言中，为了更加准确的对电路建模，定义了四种逻辑值，如下所示

逻辑值	说明
0	逻辑 0 或者假状态
1	逻辑 1 或者真状态
x 或 X	未知状态，不区分大小写
z 或 Z	高阻状态，不区分大小写

其中“x”只在电路仿真中存在，当触发器不满足采样时序要求，或者信号未被赋初始值的时候便会出现这种未知状态。

而高阻状态“z”却是在电路系统中真实存在的，例如三态门的使能信号为 0 时，其输出端口便处于高阻状态。

常数

Verilog 中的数据类型最终转换成逻辑电路之后都是使用特定位宽的二进制数据位来表示的。常数是指在 Verilog 代码中不变的数值，Verilog 中的常数有 3 种类型：整数型、实数型和字符串型。

整数是 Verilog 中使用最频繁的常数类型，有以下三种表示方式：

- <位宽><进制><数字>
- <进制><数字>
- <数字>

<位宽><进制><数字>：这是 Verilog 中最标准最全面的表示方式，指定了数据的位宽、进制和数值。位宽使用十进制数字表示，为该数据在电路中所占用的 bit 数；进制支持二进制（b 或 B）、八进制（o 或 O）、十进制（d 或 D）和十六进制（h 或 H）四种；数值则是该数据在相应进制下的表示，一般为非负数，如需表示负数，则采用二进制补码方式表示。

二进制数据	二进制	八进制	十进制	十六进制	说明
1101	4’b1101	4’o15	4’d13	4’hD	可以表示正数 13，也可以表示负数-3
00_1101	6’b00_1101 或 6’b1101	6’o015 或 6’o15	6’d13	6’h0D 或 6’hD	表示正数 13，下划线是为了增加可读性，没有实际意义

<进制><数字>：这种表示方式省略了数据位宽，默认数据位宽由 EDA 工具决定，一般为 32bit。例如“’H33”等价于 32’H33，对应的二进制数据为：

0000_0000_0000_0000_0000_0000_0011_0011

<数字>：这种表示方式省略了数据位宽和进制，默认采用十进制表示，默认数据位宽一般为 32bit。与我们平时的书写习惯相同，通过在数据前添加“+、-”号表示数据的正负，如：34，-15 等。

可以看出，Verilog 中的整数常量包含位宽、进制、数值三个要素，其中位宽和进制省略后会使用默认的缺省值代替，但这样会降低代码的可读性，推荐使用完整的表示方式。

实数

Verilog 中的实数是不可综合的，多用于电路仿真，实数型常量可以采用十进制和科学计数法两种格式来表示，如采用十进制格式，则必须含小数点，且小数点两侧都至少有一个数字。以下是实数型常量的实例

十进制格式	1.0, 0.2, 3.1415
科学计数法格式	1.2E12, 2e3, 0.1e-2

字符串

字符串是用双引号（“ ”）括起来的一串单行字符（不可以多行书写）。字符串在表达式或在赋值语句中被用作操作数，是一串 8bit 的 ASCII 值所组成的无符号整数常量，其中每一个 ASCII 值对应一个字符。字符串变量可以采用 reg 数据类型定义，变量宽度为字符串中字符的个数乘以 8。例如，字符串"Hello world!"共包含有 12 个字符，每个字符需要 8bit 寄存器，总共就需要 8 * 12 = 96 bit 寄存器。

```
reg [8*12:1] stringvar;
initial begin
    stringvar = "Hello world!";
end
```

字符串可以使用 Verilog HDL 运算符来运算，运算过程中所操作对象是 8bit 的 ASCII 值的序列。将一个字符串赋值给变量时，字符串 ASCII 值的序列将按照“右对齐”的方式写入变量。如果字符串 ASCII 值的序列宽度超过了变量宽度，则将字符串 ASCII 值的序列左边超出部分截取后，再按照“右对齐”的方式写入变量。

```
    module string_test;
    reg [8*14:1] stringvar;
    initial begin
        stringvar = "Hello world";
        $display("%s is stored as %h", stringvar,stringvar);
        stringvar = {stringvar,"!!!"};
        $display("%s is stored as %h", stringvar,stringvar);
    end
    endmodule
```

说明：以上代码不可综合，在仿真器中运行的结果为：

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

字符串中的某些特殊的字符必须用转义字符表示。转义字符（Escape character）是由“\”开始，以空格结束的一种特殊编程语言结构，这种结构可以用来表示那些与系统语言结构相同的内容。例如，" （半角双引号）在系统中被用来表示字符串，如果字符串本身的内容包含一个与之形式相同的半角双引号，那么就必须使用转义字符）。下面是常用的几种转义字符。

\n	\t	\"	\\
换行	制表符	双引号	反斜杠

此外，在“\\”之后加上三位八进制数 ddd，也相当于这个八进制数 ddd 所对应的 ASCII 字符。需要注意的是，三位八进制数 ddd 不能大于 377(O)，否则会发出错误。

参数

参数是一个常量，通过使用一个带有明确含义的参数名来代替常量，可以增加程序的可读性。

对于程序中有多个地方用到了同一常数的情况，使用参数来代替，只需要在参数定义处进行一次修改，便实现了对所有数值的修改，为程序的调试和维护带来极大方便。参数不可以在运行时修改数值，但是可以在编译时改变，通过参数定义语句或者模块实例化时的参数传递功能可以修改参数值。参数经常用于定义时延大小或变量宽度，通过使用参数可以允许一个模块以不同的规格重复使用。参数声明形式及实例如下：

参数声明形式	parameter 参数名 1 = 表达式 1，参数名 2 = 表达式 2;
参数实例	parameter DELAY = 5; parameter BIT = 1, BYTE = 8, WORD = BYTE * 4;

变量

变量是程序运行过程中其值可以改变的量，Verilog HDL 的变量也定义多种不同类型的变量，在 Verilog-2001 标准中定义了 net 型和 variable 型两大类数据类型。

net 型表示结构实体之间的物理连接，net 型变量不能储存值，而且它必需连接驱动，如果没有驱动连接到 net 型变量上，则该变量就是高阻，其值为 Z。

net 型数据类型定义电路中各种物理连接，类似于电子线路中使用的导线，其特点是输出值紧跟输入值的变化而变化。net 型数据的数值可以使用连续赋值（continuous assignment）方式，由赋值符右侧连接的驱动决定。

net 型数据在初始化之前的值为 X（trireg 例外，它相当于能够储存电荷的电容器），如果未连接驱动，则该数据的当前数值为 Z，即高阻态。

net 类型有 wire、tri、wor、trior、wand、triand、tri0、tri1、supply0、supply1、trireg 共 11 种，其中只有 wire、tri、supply0、supply1 是“可综合”的，可以用于硬件实现和系统仿真，其余 6 种只能用于系统仿真。

在 11 种数据类型中，wire 是最常使用的 net 型数据类型，在模块中，wire 型变量可以作为电路连线使用。在进行模块端口声明时，如果没有明确指出端口数据类型，那么这个端口将会被默认为 wire 型数据。

variable 型用于始终保持其状态的值，该值将保留直到被新的值覆盖。variable 型仅在声明时，或者在 initial 和 always 过程中，才可以给它们赋值。

variable 型数据可以保存当前的数值，直到另一个数值给它赋值，在保持当前数值的过程中，不需要驱动。未赋值 variable 数据类型的初始值为 X。variable 类型的变量必须使用过程赋值语句（initial 过程或 always 过程）赋值，且变量必须在使用前预先定义。

variable 类型的变量有以下几种：reg、integer、time、real，其中只有 reg 和 integer 是可综合的，可以用于硬件实现和系统仿真，time 和 real 只能用于系统仿真。在 4 种数据类型中，reg 是最常使用的 register 型数据类型。

	net 型变量	variable 型变量
可综合	**wire**、tri、supply0、supply1	**reg**、integer
不可综合	wor、trior、wand、triand、tri0、tri1	time、real

关于 Verilog 中两种最常用的数据类型：wire（线网类型）和 reg（寄存器类型），在使用时只需要遵循以下规则即可：

- 凡是通过连续赋值语句 assign 赋值的，一定是组合逻辑赋值信号，都应定义成 wire 类型；
- 凡是通过过程赋值语句 initial 或 always 赋值的信号，可能是组合逻辑赋值信号、也可能是时序逻辑赋值信号，都应定义成 reg 类型。

wire 型数据通常使用持续赋值语句 assign 对其进行赋值，也可以作为表达式的输入赋值给其他变量。wire 型的变量使用前，需要在模块中使用关键字 wire 预先定义变量（端口默认为 wire 型变量，可以不用预先定义变量），然后在电路功能部分使用 assign 语句对该变量进行赋值，或连接实例元件。wire 型变量的定义如下所示：

wire [n-1:0] 数据名 1,数据名 2,…数据名 i;
wire [n:1] 数据名 1,数据名 2,…数据名 i;

每一行首的 wire 是数据的确认符，[n-1:0] 和 [n:1] 代表该数据的位宽，即该数据有几位。最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。下面是 wire 型变量的几个例子。

wire n0; wire [7:0] n1; wire [4:1] n2,n3,n4;	n0 被定义为一个 1 bit 位宽的 wire 型变量 n1 被定义为一个 8 bit 位宽的 wire 型变量 n2,n3,n4 被定义为 3 个 4 bit 位宽的 wire 型变量

reg 型变量通常使用过程赋值语句 initial 和 always 对其进行赋值。reg 型的变量使用前，必须在模块中使用关键字 reg 进行声明，否则 EDA 工具在对代码检查时会报错。reg 型变量的定义如下所示：

reg [n-1:0] 数据名 1,数据名 2,…数据名 i;
reg [n:1] 数据名 1,数据名 2,…数据名 i;

每一行首的 reg 是数据的确认符，[n-1:0] 和 [n:1] 代表该数据的位宽，即该数据有几位，对于 nbit 位宽的信号，习惯上采用[n-1:0]来定义位宽。最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。reg 类型的变量则为无符号数，如需定义有符号变量，则需要使用关键字 signed，下面是 reg 型变量的几个例子。

reg r0;	r0 被定义为一个 1 bit 位宽的 reg 型变量，默认为无符号变量
reg signed [7:0] r1;	r1 被定义为一个 8 bit 位宽的 reg 型有符号变量
reg unsigned [4:1] r2,r3,r4;	r2,r3,r4 被定义为 3 个 4 bit 位宽的 reg 型无符号变量

integer 数据类型所定义的变量为有符号数，位宽为宿主机的字的位数，一般为 32 位，用 integer 的变量都可以用 reg 定义，只是使用 integer 定义计数更方便而已。

wire 用法总结：

- wire 可以在模块中表示任意宽度的节点（node）或总线 （bus）
- wire 可以用于模块的输入和输出端口以及一些其他元素并在实际模块声明中
- wire 类型的数据不能保存当前的数值，并且不能在 initial 和 always 过程赋值块内对其赋值，即赋值号左侧不可以是 wire 类型的变量。
- wire 类型的数据只能够使用 assign 语句赋值，即赋值号左侧必须是 wire 类型的变量
- wire 只能用于组合逻辑

reg 用法总结：

- reg 可以在模块中表示任意宽度的节点（node）或总线 （bus）
- reg 可以用于模块的输出端口，但不能用作输入
- reg 类型的数据可以保存当前的数值，必须在 initial 和 always 过程赋值块内对其赋值，即赋值号左侧必须是 reg 类型的变量。
- reg 类型的数据不能使用 assign 语句赋值，即赋值号左侧不能是 reg 类型的变量
- reg 类型的变量与@(posedge clock)一起使用时，将被综合出寄存器
- reg 既可以用于组合逻辑电路，也可以用于时序逻辑电路

定义模块时

- input 必须是 wire
- output 可以是 wire 也可以是 reg
- inout 必须是 wire

例化模块时

- 外部连接 input 端口的可以是 wire 也可以是 reg
- 外部连接 output 端口的必须是 wire
- 外部连接 inout 端口的必须是 wire

运算符

Verilog2001 中共定义了 36 个运算符可以分为 10 类，如下所示：

运算符种类	运算符符号列表	示例（假设 a=4'b1001,b=4'b0011）
算数运算符	+、-、*、/、%、**	a + b = 4'b1100
按位运算符	~、&、|、^、^~(~^)	a & b = 4'b0001
缩位运算符	&、~&、|、~|、^、^~(~^)	&a = 1'b0
逻辑运算符	!、&&、||	a&&b = 1'b1
等式运算符	==、!=、===、!==	(a==b) = 1'b0
关系运算符	<、<=、>、>=	(a > b) = 1'b1
移位运算符	<<、>>、<<<、>>>	(a << 2) = 6'b100100
条件运算符	? :	((a > b) ? a : b) = a
拼位运算符	{}、{{}}	{a,b}=8'b10010011、{3{a[1:0]}}= 6'b101010
事件运算符	or	always@(a or b) …，a 或 b 发生变化时触发 always 过程块

算数运算符（Arithmetic operators）都属于双目运算，Verilog-2001 标准中定义了 6 种算术运算符，具体说明如下。

运算符	含义	说明
+	加	2 个操作数相加
-	减	2 个操作数相减或取 1 个操作数的负数（二进制补码表示）
*	乘	2 个操作数相乘
/	除	2 个操作数相除，结果取商，余数舍弃
%	求模	2 个操作数求模，前一个操作数为被除数，后一个操作数为除数，结果取余数
**	求幂	2 个操作数求幂，前一个操作数为底数，后一个操作数为指数

算数运算符举例

代码实例	结果说明（假设 A=21、B=4）

_Sum = A + B;
_Difference = A - B;
_Product = A * B;
_Quotient = A / B;
_Remainder = A % B;
_Power = A ** B;
	_Sum = A + B = 21 + 4 = 25
	_Difference = A - B = 21 - 4 = 17
	_Product = A * B = 21 * 4 = 84
	_Quotient = A / B = 21 / 4 = 5
	_Remainder = A % B = 21 % 4 = 1
	_Power = A ** B = 21 ** 4 = 194481

按位运算符（Bitwise Operators）

按位运算符即操作数按照对应的位分别进行逻辑运算。按位运算符共定义了 5 种。

运算符	功能	说明
~	按位取反	1 个多位操作数按位取反
&	按位与	2 个多位操作数按位进行与运算，各位运算的结果按顺序组成一个新的多位数
|	按位或	2 个多位操作数按位进行或运算，各位运算的结果按顺序组成一个新的多位数
^	按位异或	2 个多位操作数按位进行异或运算，各位运算的结果按顺序组成一个新的多位数
^~ 或 ~^	按位同或	2 个多位操作数按位进行同或运算，各位运算的结果按顺序组成一个新的多位数

按位运算符举例及说明

代码实例	结果说明（假设 A=4'b1001、B=4'b1011）

_Not = ~A;
_And = A & B;
_Or = A | B;
_Xor = A ^ B;
_Xnor = A ~^ B;
	_Not = ~A = 4'b0110
	_And = A & B = 4'b1001
	_Or = A | B = 4'b1011
	_Xor = A ^ B = 4'b0010
	_Xnor = A ~^ B = 4'1101

缩位运算符（Reduction operators）

缩位运算符(归约运算符)都属于单目运算,缩位运算的逻辑运算法则与按位运算法则一致。缩位运算对单个操作数进行位运算，最后返回 1bit 数值，是将一个矢量缩减为一个标量。其运算是将操作数的每一个 bit 依次进行位运算，直至最后一个 bit 结束。缩位运算符共定义了 6 种。

&	缩位与	对一个多位操作数进行缩位与操作。从最高位依次进行位运算，直到最低位
~&	缩位与非	对一个多位操作数进行缩位与非操作。从最高位依次进行位运算，直到最低位
|	缩位或	对一个多位操作数进行缩位或操作。从最高位依次进行位运算，直到最低位
~ |	缩位或非	对一个多位操作数进行缩位或非操作。从最高位依次进行位运算，直到最低位
^	缩位异或	对一个多位操作数进行缩位异或操作。从最高位依次进行位运算，直到最低位
^~ 或 ~^	缩位同或	对一个多位操作数进行缩位同或非操作。从最高位依次进行位运算，直到最低位

缩位运算符举例

	A=4'b1001;

_ReduAnd = &A;
_ReduNotAnd = ~&A;
_ReduOr = |A;
_ReduNotOr = ~|A;
_ReduXor = ^A;
_ReduXnor = ~^A;
	_ReduAnd = &A = 1'b0
	_ReduNotAnd = ~&A = 1'b1
	_ReduOr = |A = 1'b1
	_ReduNotOr = ~|A = 1'b0
	_ReduXor = ^A = 1'b0
	_ReduXnor = ~^A = 1'b1

缩位与运算通常用于判断变量是否为“1”，操作数的所有 bit 全部为 1 时，输出结果为 1。
缩位或运算通常用于判断变量是否为“0”，操作数的所有 bit 全部为 0 时，输出结果为 0。
缩位异或运算通常用于判断变量奇偶性，操作数包含有奇数个 1-bit 时，输出结果为 1。

逻辑运算符（Logical operators）
逻辑运算符包括逻辑与 &&、逻辑或 ||、逻辑非 ！三种。逻辑运算对单个或两个操作数进行逻辑关系的运算，最后返回 1bit 的逻辑值。如果一个操作数的值不为 0，则这个操作数等价于逻辑 1（真）；如果一个操作数的值为 0，则这个操作数等价于逻辑 0（假）；如果一个操作数的任意某一 bit 为 X 或 Z，则这个操作数等价于 X（不确定）。仿真器通常会将 X 作为逻辑 0（假）来处理。

&&	逻辑与	对二个操作数进行逻辑与：如果这两个这个操作数同为 0，则运算结果为 0；如果这两个这个操作数同不为 0，则运算结果为 1
||	逻辑或	对二个操作数进行逻辑或：如果这两个这个操作数同为 0，则运算结果为 0；如果这两个这个操作数不同为 0，则运算结果为 1
!	逻辑非	对一个操作数进行逻辑取反：如果这个操作数值为 0，则运算结果为 1；如果这个操作数值不为 0，则运算结果为 0

逻辑运算符举例

	A=4'b1001;　　B=4'b0000;

_LogicAnd = A && B;
_LogicOr = A || B;
_LogicNot = !A;
	_LogicAnd = A && B = 1'b0 （结果为“假”）
	_LogicOr = A || B = 1'b1（结果为“真”）
	_LogicNot = !A = 1'b0（结果为“假”）

等式运算符（Equality operators）
等式预算符包括逻辑相等（==）、逻辑不等（!=）、case 相等（===）和 case 不等（!==）四种。这四种运算符都是双目运算符，得到的结果是 1bit 的逻辑值。如果得到的结果为 1，说明声明的关系为真；如果得到的结果为 0，说明声明的关系为假。
需要注意的是，逻辑相等（==）和 case 相等（===）是不同的。对于逻辑相等操作符，如果操作数的某位为 X 或 Z，则结果为 X；对于 case 相等操作符则需要对包括 X 和 Z 的 bit 进行逐位的精确比较，只有在两者完全相同的情况下，结果才会为 1，否则结果为 0。因此，case 相等操作符产生的结果一定不会为 X。

==	逻辑相等	二个操作数比较，如果各 bit 均相等，则结果为真，否则为假。如果其中任何一个操作数中含有 X 或 Z，则结果为 X
!=	逻辑不等	二个操作数比较，如果各 bit 不完全相等，则结果为真，否则为假。如果其中任何一个操作数中含有 X 或 Z，则结果为 X
===	case 相等	二个操作数比较，如果各 bit（包括 X 和 Z bit）均相等，则结果为真，否则为假。
!==	case 不等	二个操作数比较，如果各 bit（包括 X 和 Z bit）不完全相等，则结果为真，否则为假。

关系运算符（Relational operators）
关系运算符包括小于（<）、小于或等于（<=）、大于（>）、大于或等于（>=）四种。这四种运算符都是双目运算符，得到的结果是 1bit 的逻辑值。如果得到的结果为 1，说明声明的

关系为真；如果得到的结果为 0，说明声明的关系为假。 如果任何一个操作数包含 X 或 Z，则运算结果为不确定，返回值为 X 。

<	小于	二个操作数比较，如果前者小于后者，结果为真
<=	小于或等于	二个操作数比较，如果前者小于或等于后者，结果为真
>	大于	二个操作数比较，如果前者大于后者，结果为真
>=	大于或等于	二个操作数比较，如果前者大于或等于后者，结果为真

移位运算符（Shift operators）

移位运算符都属于双目运算，逻辑左移运算符（<<）、逻辑右移运算符（>>）、算术左移（<<<）和算术右移运算符（>>>）共 4 种。

<<	逻辑左移	第一个操作数向左移位，移位次数由第二个操作数确定，产生的空位用 0 填充
>>	逻辑右移	第一个操作数向右移位，移位次数由第二个操作数确定，产生的空位用 0 填充
<<<	算术左移	第一个操作数向左移位，移位次数由第二个操作数确定，产生的空位用 0 填充
>>>	算术右移	第一个操作数向右移位，移位次数由第二个操作数确定。如果第一个操作数为无符号数，则产生的空位用 0 填充；如果第一个操作数为有符号数，则符号位用原符号位填充，其余空位用 0 填充

移位运算符举例

A=8'b1001_0110;
_LogicShiftLeft = A << 2;
_LogicShiftRight = A >> 2;
_ArithShiftLeft = A <<< 2;
_ArithShiftRight = A >>> 2;
	_LogicShiftLeft = A << 2 = 8'b0101_1000
	_LogicShiftRight = A >> 2 = 8'b0010_0101
	_ArithShiftLeft = A <<< 2 = 8'b0101_1000
	_ArithShiftLeft = A >>> 2 = 8'b1000_0101

条件运算符（Conditional operators）

条件运算符是唯一的三目运算，其表达式为：

信号 = 条件 ？ 表达式 1 ： 表达式 2 ;

当“条件”成立时，“信号”被赋值为“表达式 1”的值；当“条件”不成立时，“信号”被赋值为“表达式 2”的值。

条件运算符举例

_Out1 = select ? in1 : in2;	当 select 为 1'b1 时，条件为“真”，_Out1 赋值为 in1 的值 当 select 为 1'b0 时，条件为“假”，_Out1 赋值为 in2 的值
_Out2 = (select == 0) ? in1 : in2;	当 select 等于 0 时，条件为“真”，_Out2 赋值为 in1 的值 当 select 不等于 0 时，条件为“假”，_Out2 赋值为 in2 的值

拼位运算符（Concatenations operators）

拼位运算符是将多个信号按顺序并列拼接起来的运算，其操作符为“ { } ”。拼位运算可以用来进行位扩展运算，还可以通过嵌套使用实现重复操作。

运算符举例

module adder (A,B,Sum,Carry); input [3:0]A,B; output [3:0]Sum; output Carry; assign {Carry,Sum} = A + B; endmodule	A、B 作为模块的两个输入，位宽为 4bit Sum 作为输出，位宽为 4bit Carry 作为输出，位宽为 1bit 使用拼位运算可以将 Carry 和 Sum 拼接为 5bit 位宽
A=4'b1001; _Replication = {3{A}};	_Replication = {3{A}} = 12'b1001_1001_1001

事件运算符
该运算符对事件进行或操作，

表达式

表达式由操作数和操作符组成，其目的是根据操作符的意义得到一个计算结果，任何出现数值的地方都可以使用表达式。
操作数包括常数、参数、变量、函数等几种类型。
操作符也称运算符，从功能上可以分为算术运算符、按位运算符、缩位运算符、逻辑运算符、等式运算符、关系运算符、移位运算符、条件运算符、连接和复制运算符等

赋值操作

赋值是将数值放入 wire 和 reg 变量的基本机制。
赋值操作由两部分组成，由“=”或“<=”符号分隔，符号的右边可以是任何求值的表达式，符号的左侧为被赋值的变量，变量的类型取决于赋值类型。赋值操作的基本格式如下：

变量 = 表达式;	一般用于组合逻辑信号的赋值操作
变量 <=表达式;	一般用于时序逻辑信号的赋值操作

赋值操作可以出现在连续赋值（Continuous assignment）和过程赋值（Procedural assignment）两种语句中。
连续赋值（Continuous assignment）语句必须以关键字 assign 开始，每个 assign 关键字后跟一条赋值操作，如需对多个 wire 型信号赋值，则需要使用多条连续赋值语句。

格式	说明
assign 变量 = 表达式;	连续赋值语句只能对 wire 型信号赋值
assign 变量 1 = 表达式 1; assign 变量 2 = 表达式 2;	每条连续赋值语句只能对一个 wire 型信号赋值 不允许在多个连续赋值语句中对同一 wire 型信号赋值

过程赋值（Procedural assignment）适用于 reg 类型变量。过程赋值将值放入变量中，变量将保留该值，直到下一次对该变量赋值为止。过程赋值发生在 always、initial、task 和 function 之类的过程（Procedure）中，并且需要“触发”（Triggered）条件。
过程赋值语句中只有 always 语句是可综合的，initial、task、function 等语句是不可综合的，一般只用来进行电路仿真。
在 FPGA 逻辑设计中，always 语句一般和触发条件、过程块语句配合使用。

连续赋值语句

Verilog 电路设计中可通过连续赋值语句或者过程赋值语句对变量进行赋值。
连续赋值（Continuous assignment）适用于 wire 型变量，是组合逻辑电路中对变量赋值的重要方式。通常，数字系统中如果仅仅只含有组合逻辑时可以使用连续赋值描述。
wire 类型变量不能够像 reg 类型变量那样储存当前数值，而是需要驱动提供信号，而且这种驱动必须是连续不断的，因此被称为连续赋值。在 Verilog HDL 中，连续赋值语句使用的关键字为 assign。

wire out;　　wire in1, in2; assign out = in1 & in2;	wire 型变量 out 被赋值为两个 wire 型变量 in1 和 in2 逻辑“与”的结果

在一个模块中可以有多个连续赋值语句，每条语句只对一个信号进行赋值，且同一个信号只能在一条语句中被赋值，如果在多个赋值语句中对同一个信号进行赋值，则相当于同一个信号有多个驱动源，会引发冲突，一般编译工具会进行报错处理。

过程赋值语句

过程赋值语句又包含了初始化过程语句（initial statement）和 always 过程语句两种，其中初始化过程语句是不可综合的，一般用于电路仿真，以关键字 initial 开始，只能执行一次，其用法如下表所示：

initial begin 　　inputs = 'b000000; 　　#10 inputs = 'b011001; 　　#10 inputs = 'b011011; 　　#10 inputs = 'b011000; 　　#10 inputs = 'b001000; end	在进行仿真时，变量 inputs 按照顺序依次被赋值。赋值过程只执行一次，通常用于为被仿真电路提供激励

always 过程语句以关键字 always 开始，可以连续多次运行，当该过程的最后一行代码执行完成后，再次从第一行代码开始执行。
由于其循环的性质，always 结构需要与时序控制结合使用才有用。否则，会创建仿真死锁条件，例如，以下代码会创建“零延迟”无限循环。

```
always areg = ~areg;
```

通常，需要将以上代码中增加延迟时序控制才可能产生有用的描述。

```
always #half_period areg = ~areg;
```

initial 结构语句和 always 结构语句一般都会和块语句（Block statements）配合使用。
块语句（Block statements）是将语句分组在一起的一种方法，因此块语句在句法上的作用相当于单个语句，Verilog 支持两种块语句：顺序块（Sequential block）、并行块（Parallel block）。
顺序块（Sequential block），也称为 begin-end 块，顺序块的特点有：位于 begin-end 之间的语句应当依次执行；每个语句的延迟值都参考于前一条语句的执行时间；最后一条语句执行后，控制权应从块中传出。

begin	在进行仿真时，两行赋值语句将按照顺序依次

areg = breg;
creg = areg;
end
	执行

begin
areg = breg;
@(posedge clock) creg = areg;
end
	在进行仿真时，第二行赋值语句将在时钟上升沿时被赋值

并行块（Parallel block），也称为 fork-join 块，并行块是不可综合的，其特点有：语句应同时执行；每个语句的延迟值应相参考进入块的时间；可以使用延迟时序控制（delay control）实现时序语句执行；当最后一个按时间顺序执行的语句执行时，控制权应从块中移出。

fork #50 r = 'h35; #100 r = 'hE2; #150 r = 'h00; #200 r = 'hF7; #250 -> end_wave; join	在进行仿真时，5 行赋值语句应该同时进行，但是由于加入了延迟时序控制语句，所以变量 r 的赋值过程将按照延时顺序依次执行

在 FPGA 开发中，一般使用 always 结构和顺序块的结合方式，其它用法很少出现，在电路仿真中则经常使用 initial 结构和顺序块的结合方式。

所有使用 assign 实现的连续赋值语句都可以写成 always 结构的过程赋值语句，如下所示

wire out;　　wire in1, in2;
assign out = in1 & in2;
	reg out;　　wire in1, in2;
	always@(*)
	begin
	　　out = in1 & in2;
	end

可以看出，通过 always 过程语句描述组合逻辑电路时，被赋值的变量需定义成 reg 类型，一个模块内部可以有任意多个 always 语句，其一般格式为：

always (时序控制) 过程语句

always 语句是永远在循环执行的，其中过程语句一般为一条表达式赋值语句，或者 begin/end 关键字构成的顺序过程语句块。例如“always clk = ~clk；”always 后的语句重复执行，由于没有时序控制语句，将在 0 时刻无限循环，这种写法没有语法错误，但没有实际意义，因此 always 语句的执行必须带有时序控制，对于上述语句，可以改成“always #5 clk = ~clk；”该语句将产生周期为 10 个时间单位的波形（#n 表示 n 个时间单位的延时）。一般来说 Verilog 语法中的 always 语句写成如下格式：

always@（敏感变量列表） 　　过程语句

只有当敏感变量列表中的事件发生时，才会执行过程语句。always 语句既可以用来实现组合逻辑赋值，也可以用来实现时序逻辑赋值，但在同一 always 语句块内两者不可同时出现，另外敏感变量列表中的事件也不一样，实现组合逻辑赋值应是电平敏感事件，时序逻辑则应该是边沿敏感事件，电平敏感事件和边沿敏感事件不能同时出现在敏感变量列表中。用 always 实现组合逻辑，可统一采用“always@(*)”的写法，而不用一一列举敏感事件。如下列代码所示：

```
module test(
input wire     a,b,
output reg     o);
wire     s;
assign s = a & b;
always@( * ) o = s;// *表示任意时序控制
endmodule
```

时序控制

在 Verilog 中，事件是指某一个 reg 或 wire 型变量发生了值的变化。事件控制用符号 @ 表示。语句执行的条件是信号的值发生特定的变化。信号值的变化可分为从 0 变为 1 和从 1 变为 0 两种情况。

关键字 posedge 指信号发生边沿正向跳变，negedge 指信号发生负向边沿跳变，未指明跳变方向时，则两种情况的边沿变化都会触发相关事件。例如：

```
//信号 clk 只要发生变化，就执行 q<=d，双边沿 D 触发器模型
always @(clk) q <= d ;
//在信号 clk 上升沿时刻，执行 q<=d，正边沿 D 触发器模型
always @(posedge clk) q <= d ;
//在信号 clk 下降沿时刻，执行 q<=d，负边沿 D 触发器模型
always @(negedge clk) q <= d ;
//立刻计算 d 的值，并在 clk 上升沿时刻赋值给 q，不推荐这种写法
q = @(posedge clk) d ;
```

当多个信号或事件中任意一个发生变化都能够触发语句的执行时，Verilog 中使用"或"表达式来描述这种情况，用关键字 or 连接多个事件或信号。这些事件或信号组成的列表称为"敏感列表"。当然，or 也可以用逗号 ，来代替。例如：

```
//带有低有效复位端的 D 触发器模型
always @(posedge clk or negedge rstn)       begin
//always @(posedge clk , negedge rstn)       begin
//也可以使用逗号陈列多个事件触发
      if（! rstn）begin
            q <= 1'b ;
      end
      else begin
            q <= d ;
      end
end
```

当组合逻辑输入变量很多时，那么编写敏感列表会很繁琐。此时，更为简洁的写法是 @* 或 @(*)，表示对语句块中的所有输入变量的变化都是敏感的。例如：

```
always @(*) begin
//always @(a, b, c, d, e, f, g, h, i, j, k, l, m) begin
//两种写法等价
      assign s = a? b+c : d ? e+f : g ? h+i : j ? k+l : m ;
```

```
end
```

分支控制

if…else…、case

条件语句（或 if-else 语句）用于根据条件表达式的真假判断运行的模块， 判断条件通常是表达式或 1bit 的变量。如果表达式的结果为 0、X、Z 则按照“假”处理；如果表达式的结果为 1，则按“真”处理。if-else 语句可以多重嵌套使用，为了保证代码的可读性，需要在代码中空白符。

因为 if-else 语句的 判断条件是表达式的值是否为零，所以判断条件可以简写。 例如，以下两个语句表达的逻辑是相同的。

```
if (expression)
if (expression != 0)
```

在 if-else 语句在嵌套使用时，需要特别注意 else 部分缺省的情况，可能会造成逻辑混乱。如果出现这种情况，通常将 else 与最接近的前一个 if 部分关联。 在下面的代码中，else 是与内部的 if（第二个 if）关联的。

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else
        result = regb;
```

如果需要将 else 与第一个 if 部分关联，则必须要使用一个 begin-end 块语句来强制改变这个关联，如下面的代码所示。

```
if (index > 0) begin
    if (rega > regb)
        result = rega;
    end
else
    result = regb;
```

If-else-if 结构编写多路选择器的通用方法。仿真过程中，条件表达式按顺序依次求值，如果有任何一个条件表达式的值为真，则执行与其关联的语句（ 每个语句可以是单个语句或语句块），通知整个判断流程终止。

If-else-if 结构的最后 else 部分将处理其他条件都不满足的情况，或称为默认情况。 在某些情况下，如果没有针对默认情况的操作的语句，则可以缺省结尾的 else 部分，或者，将其用于错误检查以捕获不可能的条件。

// declare regs and parameters reg [31:0] instruction, segment_area[255:0]; reg [7:0] index; reg [5:0] modify_seg1, modify_seg2, modify_seg3; parameter	代码使用 If-else-if 结构判断变量 index 的值，以确定是否必须将三个 segment_area 变量之一赋值给变量 instruction，以及将哪个增量 modify_seg 添加

segment1 = 0, inc_seg1 = 1,
segment2 = 20, inc_seg2 = 2,
segment3 = 64, inc_seg3 = 4,
data = 128;
 	
// test the index variable
if (index < segment2) begin
instruction = segment_area [index + modify_seg1];
index = index + inc_seg1;
end
else if (index < segment3) begin
instruction = segment_area [index + modify_seg2];
index = index + inc_seg2;
end
else if (index < data) begin
instruction = segment_area [index + modify_seg3];
index = index + inc_seg3;
end
else
instruction = segment_area [index];
	到 index 变量

Case 语句是一种多分支语句，用于测试表达式是否与多个分支项之一匹配，如果匹配则进入相应地分支语句。Case 语句会匹配 X 和 Z 的值，并提供确定的结果。

```
case(表达式)
        值 1:分支语句 1;
        值 2:分支语句 2;
        值 3:分支语句 3;
            ……
        值 n:分支语句 n;
        default:分支语句;
endcase
```

当表达式的值和分支的值相等时，就可以执行分支后面的语句；如果所有分支的值没有与表达式的值相匹配，就执行 default 语句。default 语句可缺省，在一个 case 语句中只能使用一个 default 语句。

reg [15:0] rega; reg [9:0] result; …… case (rega) 16'd0: result = 10'b0111111111; 16'd1: result = 10'b1011111111; 16'd2: result = 10'b1101111111; 16'd3: result = 10'b1110111111; 16'd4: result = 10'b1111011111;	示例是一个译码器 括号中给出的 case 表达式 raga 与 case 分支项的值相比较，如果某一个 case 分支项的值与 raga 的值匹配，则进入该分支项语句，并且不再与其他分支项的值比较。 如果 case 分支项的值没有与 raga 的值相匹配，并且存在 default 项，则进入 default 分支项语句；如果不存在 default 项，则执行任何分支项语句。

16'd5: result = 10'b1111101111; 16'd6: result = 10'b1111110111; 16'd7: result = 10'b1111111011; 16'd8: result = 10'b1111111101; 16'd9: result = 10'b1111111110; default result = 'bx; endcase	

case 语句与 if-else-if 结构的区别:

- if-else-if 结构中的表达式比 case 语句的表达式更灵活通用;
- 当表达式的值包含有 x 和 z 值时，case 语句提供确定的结果;

在 case 表达式在做比较时，只有两个表达式中每个 bit 的值（0、1、X、Z）完全匹配，比较才成功。因此，case 语句中表达式值的位宽必须相等，这样才可以进行精确的按位匹配。

让 case 语句能够精确的处理 X 和 Z 的原因是，在设计数字逻辑电路时可能出现的 X 和 Z 的情况，这种情况需要有效的检测并排除。通过使用 case 语句就可以在仿真中发现并定位问题的原因，从而优化电路的设计。

此外还有 casez 和 casex 语句，可以实现对 X、Z 更加灵活的判定，因为在 FPGA 开发中使用较少，此处不再介绍

层次结构

Verilog 通过三种实例化语句来实现结构建模:

- Gate 实例化语句
- UDP 实例化语句
- Module 实例化语句

Verilog HDL 中提供以下内置基本门

多输入门	and,nand,or,nor,xor,xnor
多输出门	buf,not
三态门	bufif0, bufif1,notif0,notif1
上拉、下拉电阻	pullup,pulldown
MOS 开关	cmos,nmos,pmos,rcmos,rpmos
双向开关	tran,tranif0,tranif1,rtran,rtranif0,rtranif1

在 Verilog 代码中可以使用具体的 gate 实例化语句，其格式如下:

gate_type [instance_name] (out,in_1,in_2,…,in_n);	instance_nameke 可以省略
gate_type [instance_name1] (out,in_11,in_12,…,in_1n); [instance_name1] (out,in_21,in_22,…,in_2n); … [instance_name1] (out,in_m1,in_m2,…,in_mn);	同一类型的多个实例能够在一个结构形式中定义

Verilog 还提供了用户定义源语（UDP，User Define Primitive）的功能，UDP 声明格式如下:

```
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，需要在模块外部进行定义，也可以在单独的文本文件中进行定义。UDP 只能有一个输出和一个及以上的输入，且第一个端口必须是输出端口，UDP 可以实现对组合逻辑电路、电平触发的时序逻辑电路、边沿触发的时序逻辑电路的描述。以下是 UDP 的示例代码:

组合逻辑 UDP	primitive nand_my(out, a, b);
	output out ;
	input a, b ;
	table
	//a b : out ;
	0 0 : 1 ;
	0 1 : 1 ;
	1 0 : 1 ;
	1 1 : 0 ;
	endtable
	endprimitive

电平触发 UDP（D 锁存器，q 表示当前状态，q+表示下一个状态）	primitive d_latch2(
	output reg q = 0,
	input clear, en, d);
	initial
	q = 0 ;
	table
	//clear en d :q :q+ ;
	1 ? ? :? :0 ; //clear
	0 0 ? :? :- ; //"-" means stable
	0 1 0 :? :0 ; //q = d
	0 1 1 :? :1 ;
	endtable
	endprimitive

边沿触发 UDP（D 触发器的时序逻辑）	primitive D_TRI(
	output reg Q = 0,
	input RST, CP, D);
	table
	//RST CP D :Q :Q+ ;
	//(1) 清零
	1 ? ? :? :0 ; //RST=1 时清零
	(??) ? ? :? :- ; //忽略 RST 边沿变化
	//(2) 时钟下降沿采集
	0 (10) 0 :? :0 ; //时钟下降沿采集信号

```
        0       (10)    1       :?      :1 ;
        //possible negedge
        0       (1x)    ?       :?      :- ;    //可能是时钟下降沿时保持
        0       (x0)    ?       :?      :- ;
        //(3) 时钟上升沿保持
        0       (0?)    ?       :?      :- ;    //时钟上升沿时保持
        //possible posedge
        0       (x1)    ?       :?      :- ;    //可能是时钟上升沿时保持
        //(4) 非时钟沿变化时，即便数据有跳变，输出仍然保持
        0       ?       (??)    :?      :- ;
    endtable
endprimitive // D_TRI
```

UDP 定义好之后，我们可以像调用内置门一样去调用 UDP。

不论是 UDP 还是内置门，在 FPGA 开发过程中都很少用到，使用最多的是模块实例化语句。

Verilog 中所有的电路单元都被定义成了模块的形式，其基本格式如下：

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

端口

模块的每一个端口都应指明其方向、类型、位宽、名称等属性，

端口方向可以是输入（input）、输出（output）或者双向（input）端口，该属性没有缺省值，必须显示指定。

端口类型一般包括 wire 和 reg 两种类型，其中输入端口必须为 wire 类型，输出端口根据需要，既可以是 wire 也可以是 reg 类型。缺省时默认为 wire 类型。

端口位宽表明该端口共有多少个 bit，例如：“input [31:0] pc”，说明 pc 是一个 32bit 的信号，缺省时默认为 1bit 位宽。

端口名称是一种标识符，在内部逻辑实现及模块实例化时会用到，可自行命名，但需要满足标识符的命名规则。

一个模块可以在另一个模块中被引用，这称为模块实例化（module instantiation），通过模块实例化可以建立起电路的层次化描述，模块实例化语句格式为：

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

模块实例化时需要对其端口进行关联连接，Verilog 只支持名称和位置关联两种实例化方式，可以任选其一，在同一个实例化语句中两者不能混用。

名称关联实例化方式需要实例化模块端口与外部信号按照其名字进行连接，端口顺序随意，可以与引用 module 的声明端口顺序不一致，只要保证端口名字与外部信号匹配即可。下面是例化一次 1bit 全加器的例子：

```
full_adder1   u_adder0(
      .Ai       (a[0]),
      .Bi       (b[0]),
      .Ci       (c==1'b1 ? 1'b0 : 1'b1),
      .So        (so_bit0),
      .Co        (co_temp[0]));
```

如果某些输出端口并不需要在外部连接，例化时 可以悬空不连接，甚至删除。一般来说，input 端口在例化时不能删除，否则编译报错，output 端口在例化时可以删除。例如：

```
//output 端口 Co 悬空
full_adder1   u_adder0(
      .Ai       (a[0]),
      .Bi       (b[0]),
      .Ci       (c==1'b1 ? 1'b0 : 1'b1),
      .So        (so_bit0),
      .Co        ());
```

```
//output 端口 Co 删除
full_adder1   u_adder0(
      .Ai       (a[0]),
      .Bi       (b[0]),
      .Ci       (c==1'b1 ? 1'b0 : 1'b1),
      .So        (so_bit0));
```

位置关联实例化方式将需要实例化的模块端口按照模块声明时端口的顺序与外部信号进行匹配连接，位置要严格保持一致。例如 1bit 全加器的实例化代码可以改为：

```
full_adder1   u_adder1(
      a[1], b[1], co_temp[0], so_bit1, co_temp[1]);
```

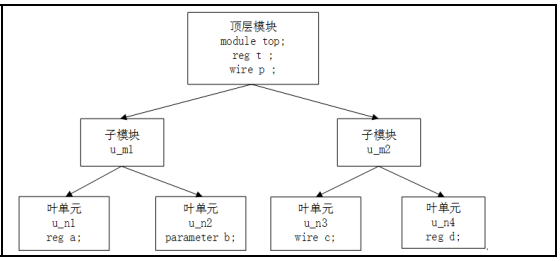
虽然代码从书写上可能会占用相对较少的空间，但代码可读性降低，也不易于调试。有时候在大型的设计中可能会有很多个端口，端口信号的顺序时不时的可能也会有所改动，此时再利用顺序端口连接进行模块例化，显然是不方便的。所以平时，建议采用命名端口方式对模块进行例化。

连接端口的信号类型可以是：1）标识符，2）位选择，3）部分选择，4）上述类型的合并，5）用于输入端口的表达式。端口连接规则如下：

输入端口	模块例化时，从模块外部来讲， input 端口可以连接 wire 或 reg 型变量。这与模块声明是不同的，从模块内部来讲，input 端口必须是 wire 型变量。
输出端口	模块例化时，从模块外部来讲，output 端口必须连接 wire 型变量。这与模块声明是不同的，从模块内部来讲，output 端口可以是 wire 或 reg 型变量。
输入输出端口	模块例化时，从模块外部来讲，inout 端口必须连接 wire 型变量。这与模块声明是相同的。
悬空端口	模块例化时，如果某些信号不需要与外部信号进行连接交互，我们可以将其悬空，即端口例化处保留空白即可，上述例子中有提及。 output 端口正常悬空时，我们甚至可以在例化时将其删除。 input 端口正常悬空时，悬空信号的逻辑功能表现为高阻状态（逻辑值为 z）。但是，例化时一般不能将悬空的 input 端口删除，否则编译会报错。
位宽匹配	当例化端口与连续信号位宽不匹配时，端口会通过无符号数的右对齐或截断方式进行匹配。

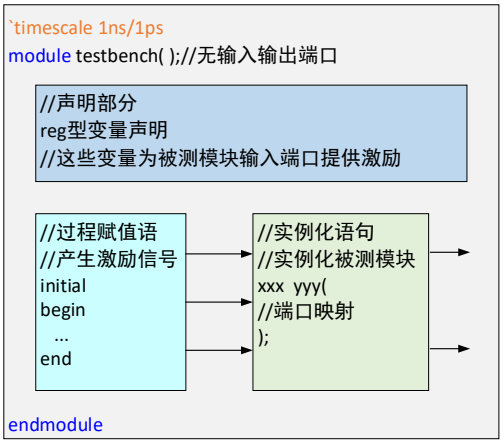
模块例化时，信号名字可以与端口名字不同，也可以一样，但如论如何他们的意义是不一样的，分别代表的是 2 个模块内的信号。每一个例化模块的名字，每个模块的信号变量等，都使用一个特定的标识符进行定义。在整个层次设计中，每个标识符都具有唯一的位置与名字。

Verilog 中，通过使用一连串的 . 符号对各个模块的标识符进行层次分隔连接，就可以在任何地方通过指定完整的层次名对整个设计中的标识符进行访问。层次访问多见于仿真中。例如，有以下层次设计，则叶单元、子模块和顶层模块间的信号就可以相互访问。

//u_n1 模块中访问 u_n3 模块信号:
a = top.u_m2.u_n3.c ;
//u_n1 模块中访问 top 模块信号
if (top.p == 'b0) a = 1'b1 ;
//top 模块中访问 u_n4 模块信号
assign p = top.u_m2.u_n4.d ;
	

Verilog 仿真

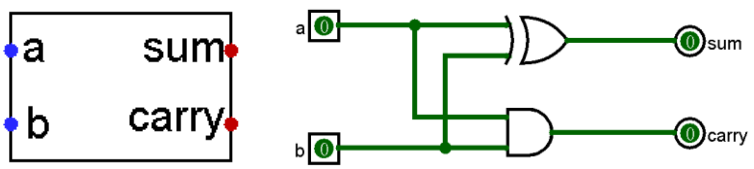
Verilog 代码编写完成之后一般不会直接烧写到 FPGA 上，或是拿去制作专用集成电路（ASIC），因为开发人员编写的代码中不可避免的会存在一些设计缺陷，电路越复杂，存在缺陷的可能性就越大，缺陷数量也越多，因此，我们需要对电路进行仿真，即通过电脑模拟的方式来验证 Verilog 代码所对应的电路是否符合设计要求。



仿真可分为功能仿真和时序仿真两种。功能仿真又称为行为仿真或前仿真，主要目的是确定一个设计是否实现了预定的功能（或者说设计意图），是证明设计功能正确性的过程。时序仿真使用布局布线后器件给出的模块和连线的延时信息， 在最坏的情况下对电路的行为作出实际地估价。时序仿真使用的仿真器和功能仿真使用的仿真器是相同的，所需的流程和激励也是相同的；惟一的差别是为时序仿真加载到仿真器的设计包括基于实际布局布线设计的最坏情况的布局布线延时， 并且在仿真结果波形图中，时序仿真后的信号加载了时延， 而功能仿真没有，这里我们主要介绍功能仿真。

组合逻辑仿真实例

下面通过对半加器电路仿真的例子来说明



其 Verilog 代码为：

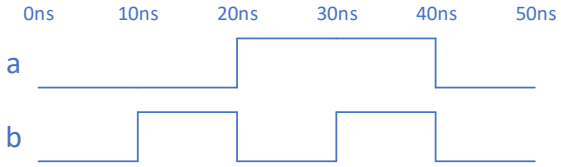
```
module half_adder(
input   a,
input   b,
output  sum,
output  carry);
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```

该电路的仿真文件为：

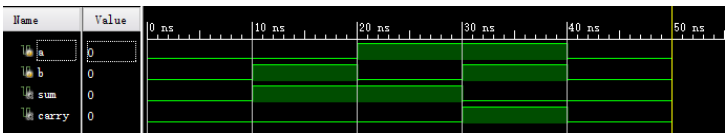
```
`timescale 1ns / 1ps
module tb_half_adder();
reg     a,b;
wire    sum,carry;
half_adder ha(
.a      (a),
.b      (b),
.sum    (sum),
.carry  (carry)
);
initial
begin
    #0  a = 0; b = 0;
    #10 a = 0; b = 1;
    #10 a = 1; b = 0;
    #10 a = 1; b = 1;
    #10 a = 0; b = 0;
    #10 $finish;
end
endmodule
```

在上面的例子中，“`timescale 1ns/1ps”用来定义仿真时间单位和精度，此处时间单位为 1ns，时间精度为 1ps，“#10”表示延时 10 个时间单位，即 10ns。initial 过程语句与 begin/end 顺序块配合使用，顺序块内的语句会顺序执行。

在上面的例子中，0ns（即仿真开始）时将变量 a 设置为 0,然后将变量 b 设置为 0。虽然对 a、b 的赋值是先后执行的两条语句，但是由于中间没有延时语句，因此是在同一时刻发生的。10ns 时 a=0,b=1，20ns（这里指仿真开始后 20ns，也就是前一条语句的 10ns 后）时 a=1,b=0，后面依次类推。仿真时一般通过波形图来检查电路功能，对于 1bit 位宽的信号来说，用高电平表示 1，低电平表示 0，本例中的 a、b 信号波形图变化如下：



在实际电路仿真过程中，会通过专门的工具进行代码分析，并产生波形图文件，代码中的所有信号都可以通过波形图查看工具进行检查，对于上面的例子来说，除了 a、b 信号之外，还可以查看半加器的输出信号 sum、carry，下图是通过仿真工具产生的波形图，从波形图可以看出，该半加器从功能上来看是正确的。



时序逻辑仿真实例

略

前面提到过，Verilog 从语法上可以分为可综合和不可综合两大类，只有使用可综合语法编写的 Verilog 代码才能够转换成真实的物理电路。编写仿真文件时可以综合和不可综合语法都可以使用。下面系统的对 Verilog 仿真中会用到的一些不可综合语法进行介绍。

关键字

always	defparam	for	instance	notif0	realtime	strong1	use
and	design	force	integer	notif1	reg	supply0	vectored
assign	disable	forever	join	or	release	supply1	wait
automatic	edge	fork	large	output	repeat	table	wand
begin	else	function	liblist	parameter	rnmos	task	weak0
buf	end	generate	library	pmos	rpmos	time	weak1
bufif0	endcase	genvar	localparam	posedge	rtran	tran	while
bufif1	endconfig	highz0	macromodule	primitive	rtranif0	tranif0	wire
case	endfunction	highz1	medium	pull0	rtranif1	tranif1	wor
casex	endgenerate	if	module	pull1	scalared	tri	xnor
casez	endmodule	ifnone	nand	pulldown	showcancelled	tri0	xor
cell	endprimitive	incdir	negedge	pullup	signed	tri1	
cmos	endspecify	include	nmos	pulsestyle_onevent	small	triand	
config	endtable	initial	nor	pulsestyle_ondetect	specify	trior	
deassign	endtask	inout	noshowcancelled	rcmos	specparam	trireg	
default	event	input	not	real	strong0	unsigned	

使用 Verilog 进行仿真时，使用最多的关键字是 initial，用来生成激励信号的时序变化，经常用到的关键字还有：for、forever、repeat、while、wait 等，配合 initial 和 begin/end 顺序块，可以实现循环或时序控制，此外 task、function 等过程语句相关的关键字也经常用到。

过程语句

在 Verilog 中，可以利用任务（关键字为 task）或函数（关键字为 function），将重复性的行为级设计进行提取，并在多个地方调用，来避免重复代码的多次编写，使代码更加的简洁、易懂。

函数是可综合的，但在仿真中使用的更多，函数只能在模块中定义，位置任意，并在模块的任何地方引用，作用范围也局限于此模块。函数主要有以下几个特点：

- 不含有任何延迟、时序或时序控制逻辑
- 至少有一个输入变量
- 只有一个返回值，且没有输出
- 不含有非阻塞赋值语句
- 函数可以调用其他函数，但是不能调用任务

Verilog 函数声明格式如下：

```
function [range-1:0]        function_id ;
input_declaration ;
  other_declaration ;
```

```
procedural_statement ;
endfunction
```

函数在声明时，会隐式的声明一个宽度为 range、 名字为 function_id 的寄存器变量，函数的返回值通过这个变量进行传递。当该寄存器变量没有指定位宽时，默认位宽为 1。
函数通过指明函数名与输入变量进行调用。函数结束时，返回值被传递到调用处。
函数调用格式如下：

```
function_id(input1, input2, ···);
```

函数在声明时，也可以在函数名后面加一个括号，将 input 声明包起来。
例如：

```
function [N-1:0]        data_rvs（
input        [N-1:0] data_in
      ……
      ）  ;
```

常数函数是指在仿真开始之前，在编译期间就计算出结果为常数的函数。常数函数不允许访问全局变量或者调用系统函数，但是可以调用另一个常数函数。
这种函数能够用来引用复杂的值，因此可用来代替常量。
例如下面一个常量函数，可以来计算模块中地址总线的宽度：

```
parameter      MEM_DEPTH = 256 ;
reg   [logb2(MEM_DEPTH)-1: 0] addr ; //可得 addr 的宽度为 8bit
      function integer        logb2;
      input integer        depth ;
            //256 为 9bit，我们最终数据应该是 8，所以需 depth=2 时提前停止循环
      for(logb2=0; depth>1; logb2=logb2+1) begin
            depth = depth >> 1 ;
      end
endfunction
```

在 Verilog 中，一般函数的局部变量是静态的，即函数的每次调用，函数的局部变量都会使用同一个存储空间。若某个函数在两个不同的地方同时并发的调用，那么两个函数调用行为同时对同一块地址进行操作，会导致不确定的函数结果。
Verilog 用关键字 automatic 来对函数进行说明，此类函数在调用时是可以自动分配新的内存空间的，也可以理解为是可递归的。因此，automatic 函数中声明的局部变量不能通过层次命名进行访问，但是 automatic 函数本身可以通过层次名进行调用。
下面用 automatic 函数，实现阶乘计算：

```
wire [31:0]              results3 = factorial(4);
function automatic    integer              factorial ;
      input integer        data ;
      integer                i ;
      begin
            factorial = (data>=2)? data * factorial(data-1) : 1 ;
      end
endfunction // factorial
```

和函数一样，任务（task）可以用来描述共同的代码段，并在模块内任意位置被调用，让代码更加的直观易读。函数一般用于组合逻辑的各种转换和计算，而任务更像一个过程，不仅能完成函数的功能，还可以包含时序控制逻辑。下面对任务与函数的区别进行概括：

比较点	函数	任务
输入	函数至少有一个输入，端口声明不能包含 inout 型	任务可以没有或者有多个输入，且端口声明可以为 inout 型
输出	函数没有输出	任务可以没有或者有多个输出
返回值	函数至少有一个返回值	任务没有返回值
仿真时刻	函数总在零时刻就开始执行	任务可以在非零时刻执行
时序逻辑	函数不能包含任何时序控制逻辑	任务不能出现 always 语句，但可以包含其他时序控制，如延时语句
调用	函数只能调用函数，不能调用任务	任务可以调用函数和任务
书写规范	函数不能单独作为一条语句出现，只能放在赋值语言的右端	任务可以作为一条单独的语句出现语句块中

任务在模块中任意位置定义，并在模块内任意位置引用，作用范围也局限于此模块。

模块内子程序出现下面任意一个条件时，则必须使用任务而不能使用函数。

- 子程序中包含时序控制逻辑，例如延迟，事件控制等
- 没有输入变量
- 没有输出或输出端的数量大于 1

Verilog 任务声明格式如下：

```
task          task_id ;
      port_declaration ;
      procedural_statement ;
endtask
```

任务中使用关键字 input、output 和 inout 对端口进行声明。input 、inout 型端口将变量从任务外部传递到内部，output、inout 型端口将任务执行完毕时的结果传回到外部。

进行任务的逻辑设计时，可以把 input 声明的端口变量看做 wire 型，把 output 声明的端口变量看做 reg 型。但是不需要用 reg 对 output 端口再次说明。

对 output 信号赋值时也不要用关键字 assign。为避免时序错乱，建议 output 信号采用阻塞赋值。

例如，一个带延时的异或功能 task 描述如下：

```
task xor_oper_iner;
      input [N-1:0]     numa;
      input [N-1:0]     numb;
      output [N-1:0]    numco ;
      //output reg [N-1:0]   numco ; //无需再注明 reg 类型，虽然注明也可能没错
      #3    numco = numa ^ numb ;
      //assign #3 numco = numa ^ numb ; //不用 assign，因为输出默认是 reg
endtask
```

任务可单独作为一条语句出现在 initial 或 always 块中，调用格式如下：

```
task_id(input1, input2, …,outpu1, output2, …);
```

任务调用时，端口必须按顺序对应。

输入端连接的模块内信号可以是 wire 型，也可以是 reg 型。输出端连接的模块内信号要求一定是 reg 型，这点需要注意。

对上述异或功能的 task 进行一个调用，完成对异或结果的缓存。

因为任务可以看做是过程性赋值，所以任务的 output 端信号返回时间是在任务中所有语句执行完毕之后。

任务内部变量也只有在任务中可见，如果想具体观察任务中对变量的操作过程，需要将观察的变量声明在模块之内、任务之外。
和函数一样，Verilog 中任务调用时的局部变量都是静态的。可以用关键字 automatic 来对任务进行声明，那么任务调用时各存储空间就可以动态分配，每个调用的任务都各自独立的对自己独有的地址空间进行操作，而不影响多个相同任务调用时的并发执行。
如果一段任务代码被 2 处及以上调用，一定要用关键字 automatic 声明。没有使用 automatic 声明任务时，任务被 2 次调用，可能出现信号间干扰。

时序控制

时延控制：基于时延的时序控制出现在表达式中，它指定了语句从开始执行到执行完毕之间的时间间隔。
时延可以是数字、标识符或者表达式。根据在表达式中的位置差异，时延控制又可以分为常规时延与内嵌时延。遇到常规延时时，该语句需要等待一定时间，然后将计算结果赋值给目标信号。格式为：#delay procedural_statement，例如：

```
reg   value_test ;
reg   value_general ;
#10   value_general      = value_test ;
```

该时延方式的另一种写法是直接将“#”独立成一个时延执行语句，例如：

```
#10 ;
value_ single              = value_test ;
```

遇到内嵌延时时，该语句先将计算结果保存，然后等待一定的时间后赋值给目标信号。
内嵌时延控制加在赋值号之后。例如：

```
reg   value_test ;
reg   value_embed ;
value_embed            = #10 value_test ;
```

需要说明的是，这 2 种时延控制方式的效果是有所不同的。
当延时语句的赋值符号右端是常量时，2 种时延控制都能达到相同的延时赋值效果。
当延时语句的赋值符号右端是变量时，2 种时延控制可能会产生不同的延时赋值效果。
在 Verilog 中，事件是指某一个 reg 或 wire 型变量发生了值的变化。
事件控制用符号 @ 表示。
语句执行的条件是信号的值发生特定的变化。
关键字 posedge 指信号发生边沿正向跳变，negedge 指信号发生负向边沿跳变，未指明跳变方向时，则 2 种情况的边沿变化都会触发相关事件。例如：

```
//信号 clk 只要发生变化，就执行 q<=d，双边沿 D 触发器模型
always @(clk) q <= d ;
//在信号 clk 上升沿时刻，执行 q<=d，正边沿 D 触发器模型
always @(posedge clk) q <= d ;
//在信号 clk 下降沿时刻，执行 q<=d，负边沿 D 触发器模型
always @(negedge clk) q <= d ;
//立刻计算 d 的值，并在 clk 上升沿时刻赋值给 q，不推荐这种写法
q = @(posedge clk) d ;
```

当多个信号或事件中任意一个发生变化都能够触发语句的执行时，Verilog 中使用"或"表达式来描述这种情况，用关键字 or 连接多个事件或信号。这些事件或信号组成的列表称为"

敏感列表"。当然，or 也可以用逗号 ，来代替。例如：

```
//带有低有效复位端的 D 触发器模型
always @(posedge clk or negedge rstn)       begin
//always @(posedge clk , negedge rstn)       begin
//也可以使用逗号陈列多个事件触发
      if（! rstn）begin
            q <= 1'b ;
      end
      else begin
            q <= d ;
      end
end
```

当组合逻辑输入变量很多时，那么编写敏感列表会很繁琐。此时，更为简洁的写法是 @* 或 @(*)，表示对语句块中的所有输入变量的变化都是敏感的。例如：

```
always @(*) begin
//always @(a, b, c, d, e, f, g, h, i, j, k, l, m) begin
//两种写法等价
      assign s = a? b+c : d ? e+f : g ? h+i : j ? k+l : m ;
end
```

前面所讨论的事件控制都是需要等待信号值的变化或事件的触发，使用 @+敏感列表 的方式来表示的。

Verilog 中还支持使用电平作为敏感信号来控制时序，即后面语句的执行需要等待某个条件为真。Verilog 中使用关键字 wait 来表示这种电平敏感情况。例如：

```
initial begin
      wait (start_enable) ;          //等待 start 信号
      forever begin
            //start 信号使能后，在 clk_samp 上升沿，对数据进行整合
            @(posedge clk_samp)   ;
            data_buf = {data_if[0], data_if[1]} ;
      end
end
```

循环控制

forever、repeat、while、for

Verilog HDL 有四种循环语句，用于控制语句执行的次数。这四种循环语句分别为：

- `forever`：连续执行语句。
- `repeat`：连续执行语句 n 次。
- `while`：执行语句直到表达式变为假。从表达式为假开始，该语句将不再执行。
- `for`：条件循环语句。

forever 循环语句用于持续不断的执行语句块，通常来产生周期性的波形，用于仿真激励信号。forever 语句通常在 initial 过程语句中。

```
      forever   begin
            语句块;
```

```
    end
```

repeat 循环语句执行指定循环数，如果循环计数表达式的值不确定，即表达式的值为 X 或 Z 时，循环次数为 0。

```
    repeat(循环次数表达式)  begin
        语句块;
    end
```

循环次数表达式用于指定循环次数，可以是一个常量、变量或者数值表达式。如果是变量或者数值表达式，其数值只在第一次循环时得到计算，从而确定循环次数。

语句块是重复执行的循环体。 在可综合设计中， 循环次数表达式必须在编译过程中保持确定不变。

while 循环语句执行语句直到循环条件表达式变为假。从表达式为假开始，该语句将不再执行。

```
    while(循环条件表达式)  begin
        语句块;
    end
```

while 语句在执行时，首先判断循环条件表达式是否为真，若为真，则执行或湖面的语句块，然后再次判断循环条件表达式是否为真，如此不断，直到循环条件表达式为假，则语句块将不再执行。

for 循环语句用于条件循环，通过三个过程控制其关联语句的执行，具体过程如下：

- 初始化循环变量，该变量控制循环数数；
- 判断循环结束条件，如果为真，则执行其关联的语句，然后进入第三个过程；如果为假，则退出 for 循环语句；
- 按照循环变量步进值修改循环控制变量值值，然后重复第二个过程。

```
    for(循环变量赋初始值;循环结束条件;循环变量步进值)  begin
        语句块;
    end
```

for 循环语句是比较常用的一条循环语句，在仿真时经常用来产生一些周期性的激励信号。for 循环语句是可综合的，但是在实体硬件描述时很少使用，这主要是因为 for 循环会被综合为所有变量情况的并行结构，每个变量独立占用寄存器资源，不能有效地复用硬件逻辑资源。简言之，for 循环语句循环几次，就是需要将相同的电路复制几次，因此循环次数越多，占用面积越大。

显示任务

Verilog 中主要用以下 4 种系统任务来显示（打印）调试信息：$display, $write, $strobe, $monitor。

$display 使用方法和 C 语言中的 printf 函数非常类似，可以直接打印字符串，也可以在字符串中指定变量的格式对相关变量进行打印。例如：

```
$display("This is a test.");    //直接打印字符串
$display("This is a test number: %b.", num); //打印变量 num 为二进制格式
```

```
This is a test.
This is a test number: 0001.
```

如果没有指定变量的显示格式，变量值会根据在字符串的位置显示出来，相当于参与了字符串连接。例如：

```
$display("This is a test number: ", num, "!!!");
```

```
This is a test number:  1!!!
```

如果没有指定格式，$display 默认显示是十进制。$displayb, $displayo, $displayh 显示格式分别为二进制、八进制、十六进制。同理也有 $writeb, $writeo, $writeh, $strobeb 等。

下表是常用的格式说明。

%h 或 %H	十六进制格式输出	%c 或 %C	ASCII 码格式输出
%d 或 %D	十进制格式输出	%e 或 %E	指数格式输出
%o 或 %O	八进制格式输出	%f 或 %F	浮点数 (real 型) 格式输出
%b 或 %B	二进制格式输出	%t 或 %T	当前时间格式输出
%s 或 %S	字符串格式输出	%m 或 %M	当前层次访问路径输出

还可以使用转义字符显示特殊字符，例如：

\n	换行符	%%	百分号"%"
\t	制表符（Tab 键）	\0	八进制代表的字符
\\	反斜杠"\"符	\0x	十六进制代表的字符
\"	双引号		

$wirte 使用方法与 $display 完全一样，只是前者会在每次显示信息完毕后不会自动换行，后者会自动换行。当输出后不需要换行时，可以使用显示任务 $write。

```
$write("This is a test");
$write("number: %b", num);
$write("!!!\n");
```

```
This is a testnumber: 0001!!!
```

$strobe 为选通显示任务。$strobe 使用方法与 $display 一致，但打印信息的时间和 $display 有所差异。

当许多语句与 $display 任务在同一时间内执行时，这些语句和 $display 的执行顺序是不确定的，一般按照程序的顺序结构执行。

$strobe 则是在其他语句执行完毕之后，才执行显示任务。例如：

```
    reg [3:0]   a ;
    initial begin
        a = 1 ;
        #1 ;
        a <= a + 1 ;
        //第一次显示
        $display("$display excuting result: %d.", a);
        $strobe("$strobe excuting result: %d.", a);
        #1 ;
        $display();
        //第二次显示
        $display("$display excuting result: %d.", a);
        $strobe("$strobe excuting result: %d.", a);
    end
```

执行结果如下所示。
执行第一次显示任务时，非阻塞赋值与 $display 同时执行，$display 显示赋值之前的变量值，而 $strobe 显示赋值之后的变量值。这正体现了 $strobe 的选通显示特性。

```
$display excuting result:  1.
$strobe excuting result:  2.

$display excuting result:  2.
$strobe excuting result:  2.
```

再看一个例子：

```
    integer   i ;
    initial begin
        for (i=0; i<4; i=i+1) begin
            $display("Run times of $display: %d.", i);
            $strobe("Run times of $strobe: %d.", i);
        end
    end
```

执行结果如下：
$display 按照程序结构，执行显示操作 4 次。而此循环语句是在 0 时刻执行的，所以 $strobe 显示的变量值是循环结束时变量的结果，即 i=4 退出循环后 $strobe 才会执行。这就体现了显示任务 $strobe 的时刻显示特性。

```
Run times of $display:            0.
Run times of $display:            1.
Run times of $display:            2.
Run times of $display:            3.
Run times of $strobe:            4.
```

$monitor 为监测任务，用于变量的持续监测。只要变量发生了变化，$monitor 就会打印显示出对应的信息。例如:

```
    reg [3:0]      cnt ;
    initial begin
        cnt = 3 ;
        forever begin
            # 5 ;
            if (cnt<7) cnt = cnt + 1 ;
        end
    end
    initial begin
        $monitor("Counter change to value %d at the time %t.", cnt, $time);
    end
```

```
Counter change to value  3 at the time                0.
Counter change to value  4 at the time             5000.
Counter change to value  5 at the time            10000.
Counter change to value  6 at the time            15000.
Counter change to value  7 at the time            20000.
```

文件操作

Verilog 提供了很多可以对文件进行操作的系统任务。经常使用的系统任务主要包括:

- 文件开、闭：**$fopen, $fclose, $ferror**

- 文件写入：$fdisplay，$fwrite，$fstrobe，$fmonitor
- 字符串写入：$sformat，$swrite
- 文件读取：$fgetc，$fgets，$fscanf，$fread
- 文件定位：$fseek，$ftell，$feof，$frewind
- 存储器加载：$readmemh，$readmemb

使用文件操作任务（尤其注意 $sforamt, $gets, $sscanf 等）对文件进行操作时，需要根据文件性质和变量内容确定使用哪一种系统任务，并保证参数及读写变量类型与文件内容的一致性，不要将字符串类型和多进制类型相混淆。

用于打开文件的系统任务 $fopen 格式如下：

```
fd = $fopen("<name_of_file>", "mode")
```

和 C 语言类似，打开方式的选项 "mode" 意义如下：

r	只读打开一个文本文件，只允许读数据。
w	只写打开一个文本文件，只允许写数据。如果文件存在，则原文件内容会被删除。如果文件不存在，则创建新文件。
a	追加打开一个文本文件，并在文件末尾写数据。如果文件如果文件不存在，则创建新文件。
rb	只读打开一个二进制文件，只允许读数据。
wb	只写打开或建立一个二进制文件，只允许写数据。
ab	追加打开一个二进制文件，并在文件末尾写数据。
r+	读写打开一个文本文件，允许读和写
w+	读写打开或建立一个文本文件，允许读写。如果文件存在，则原文件内容会被删除。如果文件不存在，则创建新文件。
a+	读写打开一个文本文件，允许读和写。如果文件不存在，则创建新文件。读取文件会从文件起始地址的开始，写入只能是追加模式。
rb+	读写打开一个二进制文本文件，功能与 "r+" 类似。
wb+	读写打开或建立一个二进制文本文件，功能与 "w+" 类似。
ab+	读写打开一个二进制文本文件，功能与 "a+" 类似。

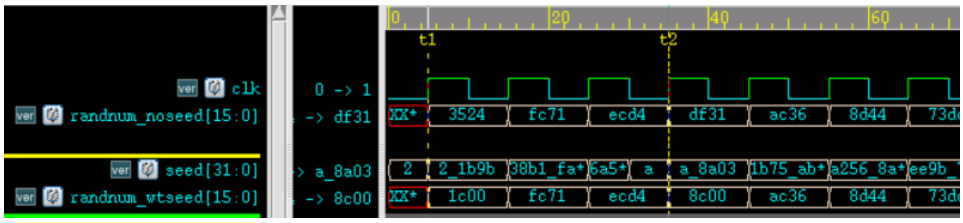
随机数任务函数

Verilog 中使用系统任务 $random(seed) 产生随机数，seed 为随机数种子。
seed 值不同，产生的随机数也不同。如果 seed 相同，产生的随机数也是一样的。
可以为 seed 赋初值，也可以忽略 seed 选项，seed 默认初始值为 0。
不使用 seed 选项和指定 seed 并对其修改来调用 $random 的代码如下所示：

```
    //seed var
    integer seed ;
    initial begin
        seed = 2 ;
        #30 ;
        seed = 10 ;
    end
    //no seed
    reg [15:0]     randnum_noseed ;
    always@(posedge clk) begin
```

```
    randnum_noseed <= $random(); //不指定随机种子
end
//with seed
reg [15:0]     randnum_wtseed ;
always@(posedge clk) begin
    randnum_wtseed <= $random(seed); //指定随机种子
end
```

仿真波形图如下。



无论是否赋初值，每产生一次随机数后，seed 值改变，随机数也随之改变。

每改变一次 seed 值，当前输出的随机值会改变；但是下一个状态时，随机数的走向又恢复成系统内部产生的随机序列。

例如仿真图中 t1 和 t2 时刻随机种子不同，产生是随机数也不同。但是其他时钟周期，产生的随机数都是相同的。

建议调用系统任务 $random 时，不指定 seed 选项，或指定 seed 选项时使用变量传递参数。

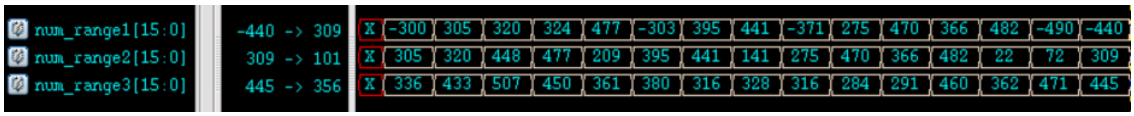
不建议调用 $random 时，将常数项写到 seed 参数处。此时 seed 值被固定，可能只会产生一个随机数。例如以下写法是不建议的：

```
randnum_wtseed <= $random(2); //不建议将常数项指定给 seed
```

可以使用取余的方法，将随机数限定在一定的数据范围内。例如：

```
    //with a range
    parameter       MAX_NUM = 512;
    parameter       MIN_NUM = 256;
    reg [15:0]     num_range1, num_range2, num_range3 ;
    always@(posedge clk) begin
        //产生的随机数范围为 -511 ~ 511, ±（MAX_NUM-1）
        num_range1 <= $random() % MAX_NUM;
        //产生的随机数范围为 0 ~ 511, （0 ~ MAX_NUM-1）
        num_range2 <= {$random()} % MAX_NUM;
        //产生的随机数范围为 MIN_NUM ~ MAX_NUM，包含边界
        num_range3 <= MIN_NUM + {$random()} % (MAX_NUM-MIN_NUM+1);
    end
```

随机数按照有符号、十进制格式显示，前几个数据结果如下：



Verilog 提供了许多按一定概率分布产生数据的系统任务，简单描述如下：

系统任务	调用格式	任务描述
均匀分布	$dist_uniform(seed, start, end);	start、end 为数据的起始、结尾

正态分布	$dist_normal (seed, mean, std_dev);	mean 为期望值，std_dev 为标准差
泊松分布	$dist_poisson(seed, mean);	mean 为期望（等于标准差）
指数分布	$dist_exponential(seed , mean);	mean 为单位时间内事件发生的次数
卡方分布	$dist_chi_square(seed, free_deg);	free_deg 为自由度
t 分布	$dist_t(seed, free_deg);	free_deg 为自由度
埃尔朗分布	$dist_erlang(seed, k_stage, mean);	k_stage 为阶数，mean 为期望

其它任务函数

$finish 与 $stop 都有让仿真停止的功能，但他们还是有区别的。
$finish 是结束本次仿真。$stop 是暂停当前的仿真。仿真暂停后通过 Verilog 仿真工具或命令行还可以使仿真继续进行，而结束仿真后仿真无论如何也不能再进行。$stop 类似于 C 语言的断点调试功能。

系统任务	调用格式	任务描述
退出仿真	$finish(type) ;	结束仿真，参数 type 可选择退出仿真时是否打印信息 type=0: 直接退出不打印 type=1: 打印仿真时间和该语句所在的位置行信息 type=2: 打印仿真时间、位置、存储器和 CPU 时间的使用情况
暂停仿真	$stop(type) ;	暂停仿真，用法格式与 $finish 相同

下面用仿真说明 type 类型对应的打印信息。

```
initial begin
    forever begin
        #100;
        if ($time >= 10000)   $finish(0) ;
        //if ($time >= 10000)   $finish(1) ;
        //if ($time >= 10000)   $finish(2) ;
    end
end
```

$finish(0)，仿真退出时不打印任何信息。
$finish(1)，仿真退出时打印仿真时间和 $finish 所在的行信息，如下所示。
模块 test 时间精度为 ns，但是仿真器时间精度为 ps，所以打印的时间信息相差 1000 倍。

```
$finish called from file "../tb/test.v", line 14.
$finish at simulation time             10000000
```

$finish(2)，仿真退出时不仅打印仿真时间和行信息，还打印 PC 机使用的时间以及及存储器的使用情况，如下所示。

```
$finish called from file "../tb/test.v", line 22.
$finish at simulation time             10000000
CPU Time:      0.340 seconds;       Data structure size:   0.0Mb
```

当显示任务（如 $display、$monitor 等）和文件写任务（如 $display 等）使用格式 "%t" 进行数据输出时，$timeformat 可以指定时间单位信息的输出格式。

系统任务	调用格式及说明
打印时间 单位和精度	$printtimescale(hierarchy) ;

	该系统任务会按照如下格式打印 timescale 信息 TimeScale of (hierarchy) is 1 (unit) / 1 (precision) hierarchy 为模块访问层次，可省略，此时打印当前模块的 timescale 信息
设置时间 单位和精度	$timeformat(unit_num, precision_num, suffix_string, min_field_width)
	unit_num，设置时间单位，默认以 `timescale 为准 precision_num，设置时间单位中小数的有效位数，默认为 0 suffix_string，设置时间后缀信息，例如 "ns" 等，默认为空 min_field_width，设置时间信息所占的字符节数，默认为 20

$timeformat 中 unit_num 是使用有符号数来指定时间单位的，其对应关系如下表所示：

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

利用如下代码对时间刻度的 2 个系统任务进行简单的仿真。

```
//change timescale
initial begin
    # 10 ;
    //ps 精度，小数为 5 位有效数字，单位后缀显示"my-ps"，占 15 个字符大小的长度
    $timeformat(-12, 5, " my-ps", 15) ;
end
initial begin
    # 5 ;
    $printtimescale() ;
    $display("Time before resetup: %t", $time);
    # 10 ;
    $printtimescale() ;
    $display("Time after resetup: %t", $time);
end
```

仿真 log 如下所示。

由图可以对比 $timeformat 设置前后时间的显示格式。

此外，此过程中 timescale 始终是没有变的，$timeformat 只是改变了时间的显示格式。

```
TimeScale of test is 1 ns / 1 ps
Time before resetup:                5000
TimeScale of test is 1 ns / 1 ps
Time after resetup: 15000.00000 my-ps
```

系统任务调用	说明
$time	返回一个 64bit 整数型时间值
$stime	返回一个 32bit 整数型时间值
$realtime	返回一个实数型时间值，可以是浮点数

仿真代码如下：

```
    initial begin
        #10;
        $display("$time output1: %t", $time);
        $display("$stime output1: %t", $stime);
        $display("$realtime output1: %t", $realtime);
        #3.2;
        $display("$time output2: %t", $time);
        $display("$stime output2: %t", $stime);
        $display("$realtime output2: %t", $realtime);
        #5.6;
        $display("$time output2: %t", $time);
        $display("$stime output2: %t", $stime);
        $display("$realtime output2: %t", $realtime);
    end
```

仿真 log 如下所示。

由于仿真时间短，$time 与 $stime 是没有区别的。

但是 $realtime 会按照当前的时间精度对仿真时间进行准确读取，而 $time 和 $stime 会根据时间精度对当前时间进行四舍五入的读取。

```
$time output1:                  10000
$stime output1:                  10000
$realtime output1:                  10000
$time output2:                  13000
$stime output2:                  13000
$realtime output2:                  13200
$time output2:                  19000
$stime output2:                  19000
$realtime output2:                  18800
```

Verilog 还提供了交互任务 $test$plusargs 和 $value$plusargs ，仿真时可通过命令行传参的方式进行参数的传递，为仿真调试提供了极大的便利。

系统任务	调用格式	任务描述
字符串传参	$test$plusargs(str) ;	仿真时通过命令行传递的字符串数据如果和 str 一致，则返回值为 1，否则为 0。
数值传参	$value$plusargs(str, var) ;	仿真时通过命令行传递的字符串数据如果和 str 一致，则返回值为 1，否则为 0。 需要在 str 内部指定传递给 module 内 var 变量的类型，格式可参考显示任务 $display

使用 $test$plusargs(str) 时，只需在仿真命令行中加入"+str "即可。

使用 $value$plusargs(str，var) 时，需要在 str 内部指定传递参数时数值的类型。而在命令行传递参数时，数值不需要添加任何有关进制的说明，只保留相关进制的数值即可。命令行传递参数的格式需要参照 $value$plusargs 时 str 声明的格式。

下面用仿真说明：

```
    initial begin
```

```
        if ($test$plusargs("DISPLAY_CTRL")) begin
            $display("Display simulation information!!!");
        end
    end
    reg   [1:0]     display_sel ;
    initial begin
        if ($value$plusargs("INFO_SEL=%b", display_sel)) begin
            $display("Parameter transfer succeeds!!!");
        end
        else begin
            display_sel = 2'b0 ;
        end
    end
    initial begin
        #1 ;
        if (display_sel == 2'b01)
            $display("You have selected Runoob!!!");
        else if (display_sel == 2'b10)
            $display("You have selected Verilog!!!");
        else if (display_sel == 2'b11)
            $display("You have selected Me!!!");
        else
            $display("What do you really what???");
    end
```

在仿真命令行中增加： +DISPLAY_CTRL +INFO_SEL=01

则仿真 log 如下，由此可知，字符串参数和二进制参数均传递正确。

```
Display simulation information!!!
Parameter transfer succeeds!!!
You have selected Runoob!!!
```

时序控制

延时控制、电平敏感事件控制、边沿触发敏感事件控制

延迟时序控制使用关键字 # 和延迟时间来定义，延迟时间可以是数字、变量或者表达式。延迟时序控制指定了从最初遇到该语句到该语句实际执行之间的延迟时间。延迟表达式可以是电路状态的动态函数，也可以是一个数字。延迟时序控制可以分为常规延迟和内嵌延迟两种类型。

常规延迟在赋值语句的左边，系统执行到这一行代码时，系统先进行延迟，延迟完成后，再计算表达式，并将结果赋值给左边的变量。

内嵌延迟在赋值语句的右边，系统执行到这一行代码时，系统先立即计算表达式，再进行延迟，最后把表达式的结果赋值给左边的变量。

parameter latency = 8; initial begin	#5 x = 3 使用了常规延迟：延迟 5 个时间单位后，给 x 赋值为 3

x = 1;
y = 2;
#5 x = 3;
#latency y = 4;
z = #10 (x+y);
end
	#latency y = 4 使用变量进行常规延迟：延迟 8 个时间单位后，给 y 赋值为 4
	z = #10 (x+y) 使用内嵌延迟：先计算(x+y)的值，然后延迟 10 个时间单位后，给 z 赋值

常规延迟是先延迟再计算表达式，这时表达式的自变量可能已经发生了变化；而内嵌延迟在延迟前就已经进行了计算，表达式的自变量在延迟过程中发生的变化，对已经计算的表达式结果没有影响，延迟只是让这个结果等待一段时间，然后再赋值给左边的变量。

事件时序控制使用关键字 @ ，变量值的更改或已声明事件的发生，可以用作触发语句执行的事件，一旦指定的事件发生，则代码被触发执行。通常，事件是时钟信号的上升沿（posedge）或下降沿（negedge）。

@r rega = regb;
always @(posedge clock) rega = regb;
forever @(negedge clock) rega = regb;
	@r rega = regb 当变量 r 改变时，rega 被赋值为 regb
	always @(posedge clock) rega = regb 当 clock 上升沿时，rega 被赋值为 regb
	forever @(negedge clock) rega = regb 当 clock 下降沿时，rega 被赋值为 regb

多事件时序控制（event or operators）使用关键字 or 或逗号 "," 。OR 事件时序控制可以使用任何数量的变量或事件作为触发条件，当任意一个事件的发生，则代码被触发执行。如果需要监视的变量很多，则可以使用 @* 或 @(*) ，表示触发条件是代码中的所有输入变量。此外，触发条件可以是前面所提到过的常规事件。

always @(a, b, c, d, e) always @(posedge clk, negedge rstn) always @(a or b, c, d or e)	always @(a, b, c, d, e) 当变量 a 或 b 或 c 或 d 或 e 改变时触发 always @(posedge clk, negedge rstn) 当 clk 上升沿或者 rstn 改变时触发 always @(a or b, c, d or e) 当变量 a 或 b 或 c 或 d 或 e 改变时触发

testbench 仿真举例

前面的章节中，已经写过很多的 testbench。其实它们的结构也都大致相同。
下面，我们举一个数据拼接的简单例子，对 testbench 再做一个具体的分析。
一个 2bit 数据拼接成 8bit 数据的功能模块描述如下:

```
module   data_consolidation
    (
        input               clk ,
        input               rstn ,
        input [1:0]       din ,              //data in
        input               din_en ,
```

```
        output [7:0]       dout ,
        output             dout_en          //data out
    );

// data shift and counter
  reg [7:0]                data_r ;
  reg [1:0]                state_cnt ;
  always @(posedge clk or negedge rstn) begin
      if (!rstn) begin
          state_cnt      <= 'b0 ;
          data_r         <= 'b0 ;
      end
      else if (din_en) begin
          state_cnt      <= state_cnt + 1'b1 ;      //数据计数
          data_r         <= {data_r[5:0], din} ;   //数据拼接
      end
      else begin
          state_cnt <= 'b0 ;
      end
  end
  assign dout            = data_r ;

  // data output en
  reg                    dout_en_r ;
  always @(posedge clk or negedge rstn) begin
      if (!rstn) begin
          dout_en_r      <= 'b0 ;
      end
      //计数为 3 且第 4 个数据输入时，同步输出数据输出使能信号
      else if (state_cnt == 2'd3 & din_en) begin
          dout_en_r      <= 1'b1 ;
      end
      else begin
          dout_en_r      <= 1'b0 ;
      end
  end
  //这里不直接声明 dout_en 为 reg 变量，而是用相关寄存器对其进行 assign 赋值
  assign dout_en         = dout_en_r;

endmodule
```

对应的 testbench 描述如下，增加了文件读写的语句:

```
`timescale 1ns/1ps

  //============== (1) ==================
```

```
    //signals declaration
module test ;
    reg                 clk;
    reg                 rstn ;
    reg [1:0]       din ;
    reg                 din_en ;
    wire [7:0]      dout ;
    wire                dout_en ;

    //============== (2) ==================
    //clock generating
    real                CYCLE_200MHz = 5 ; //
    always begin
        clk = 0 ; #(CYCLE_200MHz/2) ;
        clk = 1 ; #(CYCLE_200MHz/2) ;
    end

    //============== (3) ==================
    //reset generating
    initial begin
        rstn          = 1'b0 ;
        #8 rstn           = 1'b1 ;
    end

    //============== (4) ==================
    //motivation
    int                 fd_rd ;
    reg [7:0]       data_in_temp ;    //for self check
    reg [15:0]      read_temp ;         //8bit ascii data, 8bit \n
    initial begin
        din_en        = 1'b0 ;              //(4.1)
        din             = 'b0 ;
        open_file("../tb/data_in.dat", "r", fd_rd); //(4.2)
        wait (rstn) ;       //(4.3)
        # CYCLE_200MHz ;

        //read data from file
        while (! $feof(fd_rd) ) begin    //(4.4)
            @(negedge clk) ;
            $fread(read_temp, fd_rd);
            din       = read_temp[9:8] ;
            data_in_temp = {data_in_temp[5:0], din} ;
            din_en = 1'b1 ;
        end
```

```
        //stop data
        @(posedge clk) ;    //(4.5)
        #2 din_en = 1'b0 ;
end

//open task
task open_file;
        input string          file_dir_name ;
        input string          rw ;
        output int             fd ;

        fd = $fopen(file_dir_name, rw);
        if (! fd) begin
                $display("--- iii --- Failed to open file: %s", file_dir_name);
        end
        else begin
                $display("--- iii --- %s has been opened successfully.", file_dir_name);
        end
endtask

//============== (5) ==================
//module instantiation
data_consolidation       u_data_process
(
   .clk                  (clk),
   .rstn                 (rstn),
   .din                   (din),
   .din_en               (din_en),
   .dout                  (dout),
   .dout_en              (dout_en)
 );

//============== (6) ==================
//auto check
reg    [7:0]              err_cnt ;
int                        fd_wr ;

initial begin
        err_cnt     = 'b0 ;
        open_file("../tb/data_out.dat", "w", fd_wr);
        forever begin
                @(negedge clk) ;
                if (dout_en) begin
```

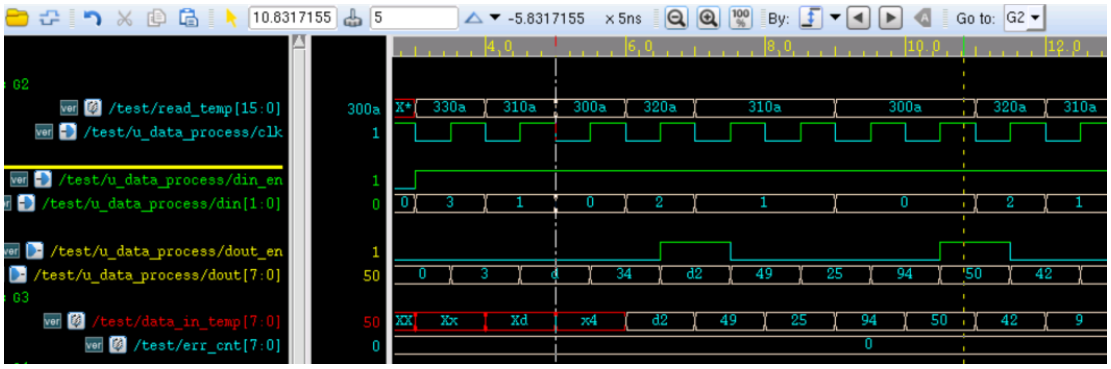
```
                $fdisplay(fd_wr, "%h", dout);
            end
        end
    end

    always @(posedge clk) begin
        #1 ;
        if (dout_en) begin
            if (data_in_temp != dout) begin
                err_cnt = err_cnt + 1'b1 ;
            end
        end
    end

    //============== (7) ==================
    //simulation finish
    always begin
        #100;
        if ($time >= 10000)   begin
            if (!err_cnt) begin
                $display("------------------------------------");
                $display("Data process is OK!!!");
                $display("------------------------------------");
            end
            else begin
                $display("------------------------------------");
                $display("Error occurs in data process!!!");
                $display("------------------------------------");
            end
            #1 ;
            $finish ;
        end
    end

endmodule // test
```

仿真结果如下。由图可知，数据整合功能的设计符合要求:



testbench 具体分析

1）信号声明

testbench 模块声明时，一般不需要声明端口。因为激励信号一般都在 testbench 模块内部，没有外部信号。

声明的变量应该能全部对应被测试模块的端口。当然，变量不一定要与被测试模块端口名字一样。但是被测试模块输入端对应的变量应该声明为 reg 型，如 clk，rstn 等，输出端对应的变量应该声明为 wire 型，如 dout，dout_en。

2）时钟生成

生成时钟的方式有很多种，例如以下两种生成方式也可以借鉴。

```
initial clk = 0 ;
always #(CYCLE_200MHz/2) clk = ~clk;
```

```
initial begin
    clk = 0 ;
    forever begin
        #(CYCLE_200MHz/2) clk = ~clk;
    end
end
```

需要注意的是，利用取反方法产生时钟时，一定要给 clk 寄存器赋初值。

利用参数的方法去指定时间延迟时，如果延时参数为浮点数，该参数不要声明为 parameter 类型。例如实例中变量 CYCLE_200MHz 的值为 2.5。如果其变量类型为 parameter，最后生成的时钟周期很可能就是 4ns。当然，timescale 的精度也需要提高，单位和精度不能一样，否则小数部分的时间延迟赋值也将不起作用。

3）复位生成

复位逻辑比较简单，一般赋初值为 0，再经过一段小延迟后，复位为 1 即可。

这里大多数的仿真都是用的低有效复位。

4）激励部分

激励部分该产生怎样的输入信号，是根据被测模块的需要来设计的。

本次实例中:

(4.1) 对被测模块的输入信号进行一个初始化，防止不确定值 X 的出现。激励数据的产生，我们需要从数据文件内读入。

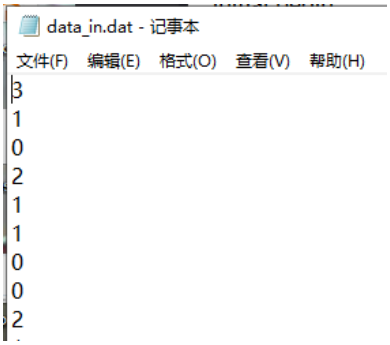
(4.2) 处利用一个 task 去打开一个文件，只要指定文件存在，就可以得到一个不为 0 的句柄信号 fp_rd。fp_rd 指定了文件数据的起始地址。

(4.3) 的操作是为了等待复位后，系统有一个安全稳定的可测试状态。

(4.4) 开始循环读数据、给激励。在时钟下降沿送出数据，是为了被测试模块能更好的在上升沿采样数据。

利用系统任务 $fread ，通过句柄信号 fd_rd 将读取的 16bit 数据变量送入到 read_temp 缓存。

输入数据文件前几个数据截图如下。因为 $fread 只能读取 2 进制文件，所以输入文件的第一行对应的 ASCII 码应该是 330a，所以我们想要得到文件里的数据 3，应该取变量 read_temp 的第 9 到第 8bit 位的数据。



信号 data_in_temp 是对输入数据信号的一个紧随的整合，后面校验模块会以此为参考，来判断仿真是否正常，模块设计是否正确。

(4.5) 选择在时钟上升沿延迟 2 个周期后停止输入数据，是为了被测试模块能够正常的采样到最后一个数据使能信号，并对数据进行正常的整合。

当数据量相对较少时，可以利用 Verilog 中的系统任务 $readmemh 来按行直接读取 16 进制数据。保持文件 data_in.dat 内数据和格式不变，则该激励部分可以描述为：

```
    reg [1:0]       data_mem [39:0] ;
    reg [7:0]       data_in_temp ;   //for self check
    integer          k1 ;
    initial begin
        din_en       = 1'b0 ;
        din          = 'b0 ;
        $readmemh("../tb/data_in.dat", data_mem);
        wait (rstn) ;
        # CYCLE_200MHz ;

        //read data from file
        for(k1=0; k1<40; k1=k1+1)   begin
            @(negedge clk) ;
            din       = data_mem[k1] ;
            data_in_temp = {data_in_temp[5:0], din} ;
            din_en = 1'b1 ;
        end

        //stop data
        @(posedge clk) ;
        #2 din_en = 1'b0 ;
     end
```

5）模块例化

这里利用 testbench 开始声明的信号变量，对被测试模块进行例化连接。

6）自校验

如果设计比较简单，完全可以通过输入、输出信号的波形来确定设计是否正确，此部分完全可以删除。如果数据很多，有时候拿肉眼观察并不能对设计的正确性进行一个有效判定。此时加入一个自校验模块，会大大增加仿真的效率。

实例中，我们会在数据输出使能 dout_en 有效时，对输出数据 dout 与参考数据 read_temp（激励部分产生）做一个对比，并将对比结果置于信号 err_cnt 中。最后就可以通过观察 err_cnt 信号是否为 0 来直观的对设计的正确性进行判断。

当然如实例中所示，我们也可以将数据写入到对应文件中，利用其他方式做对比。

7）结束仿真

如果我们不加入结束仿真部分，仿真就会无限制的运行下去，波形太长有时候并不方便分析。Verilog 中提供了系统任务 $finish 来停止仿真。

停止仿真之前，可以将自校验的结果，通过系统任务 $display 在终端进行显示。