**实验1**

**Verilog基础语法（上）**

# **实验介绍**

在本次实验中，我们将要学习Verilog语言的基本知识，并进行简单的Verilog编程工作。Verilog的全称为Verilog HDL，在不引起混淆的情况下，我们一般将其简称为Verilog。Verilog是目前在FPGA及数字IC设计中应用最为广泛的一种硬件描述语言。

硬件描述语言（Hardware Description Language, HDL）是一种使用形式化方法描述逻辑电路和系统的语言。具体来说，用户借鉴层次化的设计思想完成自顶向下的基本单元电路设计，并进行功能仿真验证，随后使用自动综合工具将模块组合转换到门级电路网表，接下来再用专用布线工具把网表转换为要实现的具体电路结构。

HDL语言借鉴和继承了一般高级语言的很多特点和语法结构，但二者存在如下的区别：

1. 硬件描述语言具有时序（或时钟）的概念，而一般的高级程序语言则没有。在硬件电路中，信号通过物理器件实现电平的转换，这个过程存在一定的延迟，经过不同路径后的信号时序也会变得不同。为了准确、客观地表达电路的情况，HDL语言必须引入时序的概念。
2. 硬件描述语言具有并行处理的功能，即同一时刻并行执行多条代码。这和一般高级设计语言（如 C 语言等）串行执行的特征存在本质上的不同。
3. 形式化表示电路的结构或行为。HDL语言源于高级程序设计语言，其描述方法与高级语言相同，但更注重实现硬件电路具体连接结构的描述。HDL语言与其说是在「写代码」，更像是在「画电路」。

目前主流的三大硬件描述语言为：Verilog、VHDL以及System Verilog。其中Verilog在电子信息集成化设计领域被广泛使用。在本次实验中，我们将对其语法特点及应用进行初步的介绍。



# 实验目标

1. 能够掌握基本的Verilog语法知识；
2. 能够阅读并编写简易的Verilog代码；
3. 能够将Verilog语言与硬件电路对应起来，理解硬件描述语言的内涵。

# 实验内容

1983年，Gateway Design Automation（GDA）公司的Phil Moorby创建了一门新的语言，主要用于公司内部的逻辑建模和仿真验证。这门语言被命名为 Verilog。随着公司模拟、仿真器产品的广泛使用，Verilog作为一种实用语言逐渐为众多硬件设计者接受。

1995 年，在Open Verilog International（OVI）组织的推动下，IEEE制定了Verilog的第一个国际标准 [IEEE Std 1364-1995](https://ieeexplore.ieee.org/document/803556)（Verilog 1.0），随后在2001年发布第二个国际标准 [IEEE Std 1364-2001](https://ieeexplore.ieee.org/document/954909)（Verilog 2.0）。迄今为止，Verilog已经成为电路设计中最流行的硬件描述语言。

现在，我们的故事，就要从Verilog最基础的语法知识开始讲起。

## 1 语法入门

### 1.1 语句与注释

Verilog是一种区分大小写的编程语言。与C语言类，Verilog的每一条语句可以写在一行内，也可以跨行编写，但都需要以分号结尾。例如，下面两段代码最终的效果完全一致。



代码中额外的空白符（例如换行符 \n、制表符 tab 以及空格 space）都没有实际意义，在编译阶段将被忽略。

与C语言类似，Verilog中也有两种注释方式：单行注释与多行注释。其中：

* 单行注释以双斜线 // 开始，表明从这里开始到本行结束均为注释语句。
* 多行注释以 /\* 开始，以 \*/ 结束，二者之间的多行内容均为注释语句。



### 1.2 数值系统

硬件描述语言的一切都是建立在硬件逻辑之上，因此Verilog具有一套独特的、基于电平逻辑的数值系统。Verilog通常使用下面四种基本数值表示电平逻辑：

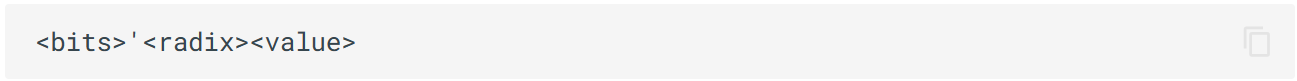
* 0：表示低电平或False；
* 1：表示高电平或True；
* x与X：表示电平未知。也就是说实际情况可能为高电平，也可能为低电平，甚至二者都不属于；
* z与Z：表示高阻态。这种情况常常源于信号没有驱动。

高阻，或者说“阻高”，是一种电路分析时的特殊情况。简单来说，它可以被看作是具有极高的输入（输出）电阻，在极限状态下就成为了断路。高阻态（High impedance state）、高输出态（High output state）和低输出态（Low output state）共同组成了三态逻辑。

这里要说明一下：真实电路中电平信号是连续变化的，我们定义VCC代表电路电压（也就是电压上限值），而数字电路则对0~VCC的范围进行了区间划分。例如0 ~ 0.3VCC为低电平（0），0.7VCC ~ VCC为高电平（1），0.3VCC ~ 0.7VCC 之间的则不属于工作电压，而高阻态往往被认为是理想的0电压。 因此，低输出态不等于没有输出；高阻则在行为上更接近于没有输出。

因此，如果两个输出端口被导线直接相连，且二者一个处于高电平状态，另一个处于低电平状态，此时相连的导线上电平就会出现混乱，即 1+0=?。但处于高阻态的端口不会影响另一个端口的输出结果，即 z+0=0, z+1=1。这一特性使得三态门被广泛应用与总线（Bus）结构之中。

除了电平逻辑，我们在编写Verilog程序时也经常用到整数。整数可以简单地使用十进制表示，例如25、-7等，也可以使用下面的基数格式进行表示：



其中：

* bits代表二进制位宽，是一个正整数。如果空缺不填则会由编译器根据后面的数值自动分配。
* radix代表进制，包括四种：十进制（d或D）、十六进制（h或H）、二进制（b或B）以及八进制（o或O）。
* value代表实际数值。有时会插入下划线 \_ 以保证更好的可读性

例如：从数值上来说，-4'hf = -4'd15 = -15；9'o210 = 9'b010\_001\_000 = 'b0\_1000\_1000 = 136。特别地，十六进制中的 a~f 字符是不区分大小写的，例如 4'ha=4'hA。

整数表示中，如果填写的二进制位宽小于实际位宽，则会根据填写的位宽对实际数据进行截断。在编译器中，这种操作极有可能会触发警告。例如：3'hfff 填写的二进制位宽为3，最终会被截断为 3'b111。

### 1.3 标识符与变量

标识符是编程的时候为变量赋予的「名字」。Verilog中的标识符可以是任意一组字母、数字以及美元符号 $ 和下划线 \_ 的组合。标识符是大小写敏感的，且第一个字符必须是字母或者下划线。例如：reg\_2、\_Add\_input 都是合法的标识符，而 5Reg\_in、\_ustc-123 都不是合法的标识符。

C语言的变量类型很多，如int、char、float等。而在Verilog中，变量主要有两种类型：wire型和reg型，其余类型可以理解为这两种数据类型的扩展。

* wire用于声明线网型数据。wire本质上对应着一根没有任何其他逻辑的导线，仅仅将输入自身的信号原封不动地传递到输出端。该类型数据用来表示以 assign语句内赋值的组合逻辑信号，其默认初始值是z（高阻态）。wire是 Verilog的默认数据类型。也就是说，对于没有显式声明类型的信号，Verilog一律将其默认为wire类型。
* reg用于声明在always语句内部进行赋值操作的信号。一般而言，reg型变量对应着一种存储单元，其默认初始值是x（未知状态）。为了避免可能的错误，凡是在always语句内部被赋值的信号，都应该被定义成reg类型。

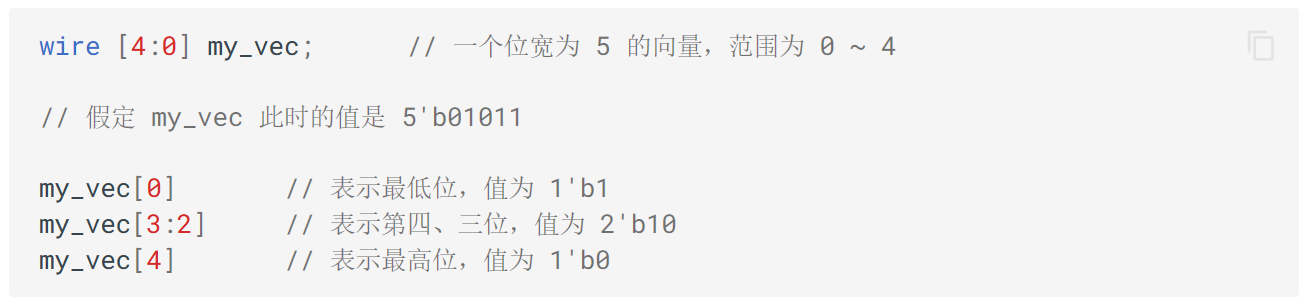
值得注意的是，reg作为Verilog的类型关键字实际上是具有误导性的。与wire类型代表线网不同，一个reg类型的变量不一定对应一个寄存器（Register）。reg关键字只是声明了一个在always语句中进行赋值的信号。如果always描述的是组合逻辑，那么reg就会综合成一根线，如果always描述的是时序逻辑，那么reg才会综合成一个寄存器。

在声明变量时，我们一般使用如下的格式：



其中width是我们声明的信号的位宽。例如 reg [3:0] ans; 声明了一个reg类型、位宽为4的变量ans。如果省略width不写，则默认变量的位宽为1（等价于 [0:0]）。无论是wire型变量还是reg型变量，Verilog统一将位宽为1的变量称作标量（Scalar），位宽大于1的变量称作向量（Vector）。

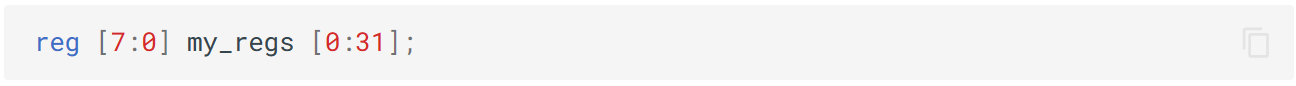
在表达式中，我们可任意选择向量中的一位或相邻几位，分别称为位选择（bit-select）和域选择（part-select），例如：



此外，Verilog中也是存在数组的概念的。你可以按照如下的格式初始化一个数组：



例如：



这段代码声明了一个数组，该数组由32个位宽为8的reg型变量组成。

### 1.4 运算符

表达式由操作符（运算符，Operators）和操作数构成，其作用是根据操作符的意义对操作数进行运算，最终得到相应的结果。Verilog 中的运算符按照功能可以分为以下几类：算数运算符、关系运算符、逻辑运算符、归约运算符、条件运算符、移位运算符、拼接运算符。我们将逐一进行介绍。

**算数运算符**

算术运算符又称为二进制运算符，共五种，均为双目运算符。为：

1. +（加）
2. -（减）
3. \*（乘）
4. /（除）
5. %（取模）

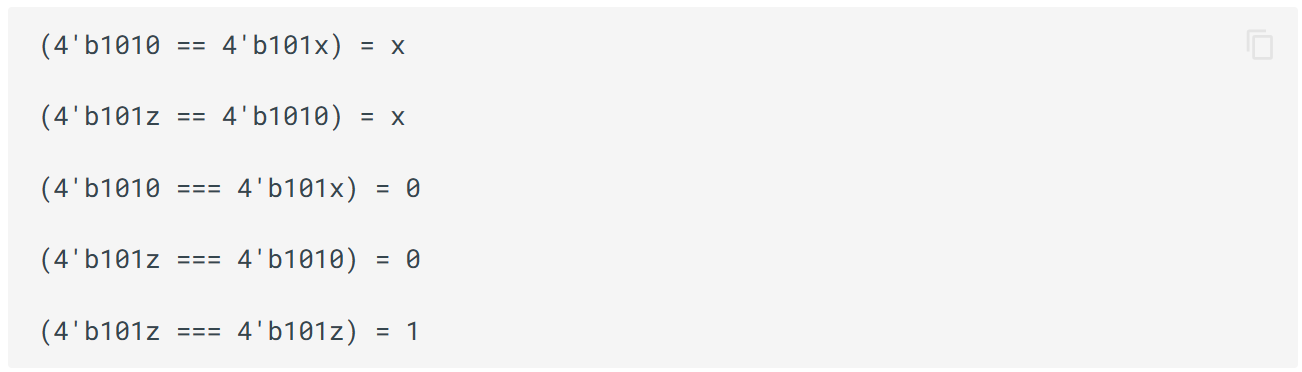
例如：a + 3'b101 表示两个数 a 和 3'b101 相加。除了表示加法和减法运算，+ 和 - 也可以作为单目操作符使用，用于表示操作数的正负性。此类操作符优先级最高。

**关系运算符**

共八种。为：

1. >（大于）
2. <（小于）
3. >=（大于等于）
4. <=（小于等于）
5. ==（等于）
6. !=（不等）
7. ===（全等）
8. !==（非全等）

关系运算符的运算结果固定为 0（False）或 1（True），是一个 1bit 的值。如果某一个操作数中有一位为 x 或 z，则前六种关系运算的结果为 x。全等比较则对为 x 或 z 的位也进行比较，只有两个操作数完全一致时，其结果才是 1，否则结果是 0。例如：



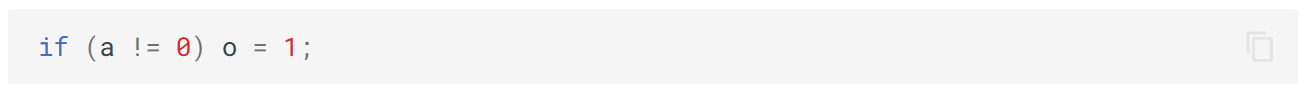
**逻辑运算符**

共三种。为：

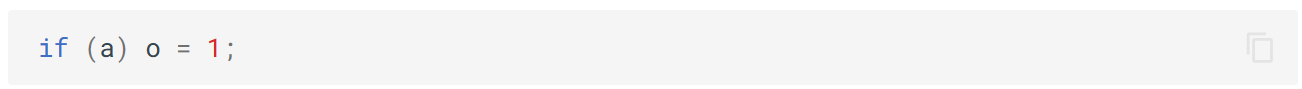
1. &&（逻辑与）
2. ||（逻辑或）
3. !（逻辑非）

逻辑运算符的计算结果也是一个 1bit 的值，其中 0 表示 False，1 表示 True，x 表示不确定。如果一个操作数不为 0，则在逻辑运算时它等价于逻辑 1；如果一个操作数等于 0，则它运算时等价于逻辑 0。如果它任意一位为 x 或 z，则它等价于 x。

一个很常见的例子是我们接下来会介绍的 if 语句：



它实际上等价于

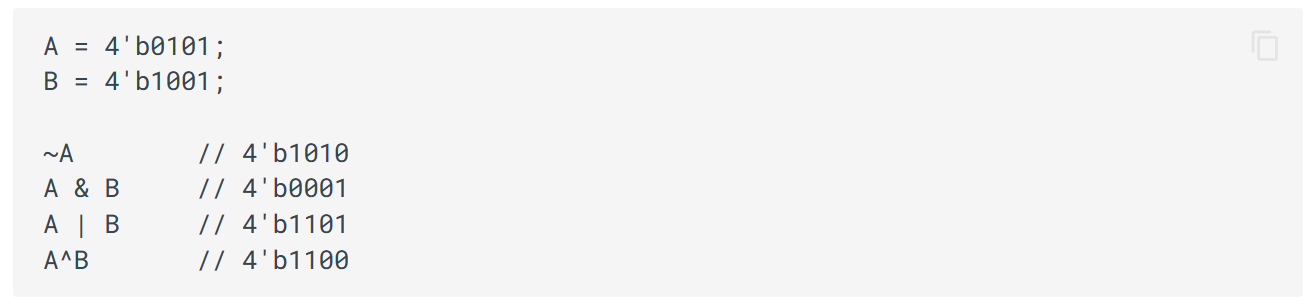


**按位运算符**

共五种。为：

1. ~（按位非）
2. &（按位与）
3. |（按位或）
4. ^（按位异或）
5. ~^ 或 ^~（按位同或）

按位运算符对 2 个操作数的每 bit 数据进行按位操作。如果 2 个操作数位宽不相等，则用 0 向左扩展补充较短的操作数。按位非是单目运算符，它对操作数的每 bit 数据进行取反操作。例如：

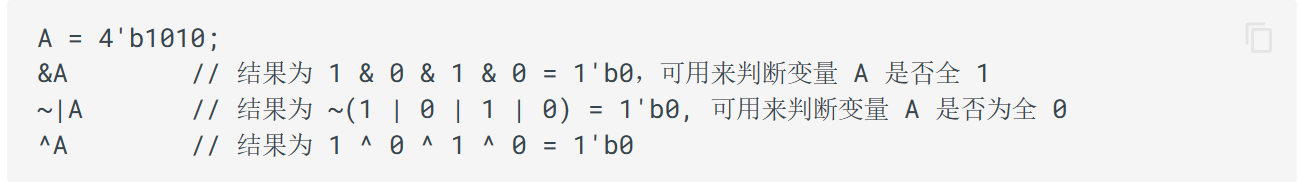


**归约运算符**

共六种。为：

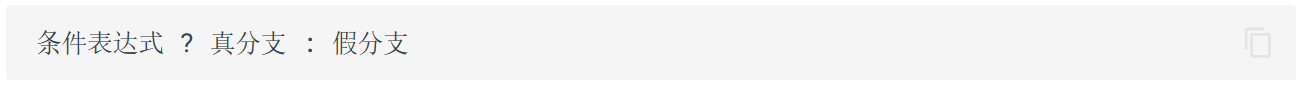
1. &（归约与）
2. ~&（归约非与）
3. |（归约或）
4. ~|（归约非或）
5. ^（归约异或）
6. ~^（归约同或）

归约运算符与按位运算符的符号相同，但规约运算符都是单目运算符。它对多位操作数逐位进行操作，最终产生一个 1bit 结果。例如：



**条件运算符**

该类操作符只有一个，是一个三目操作符，其一般形式为



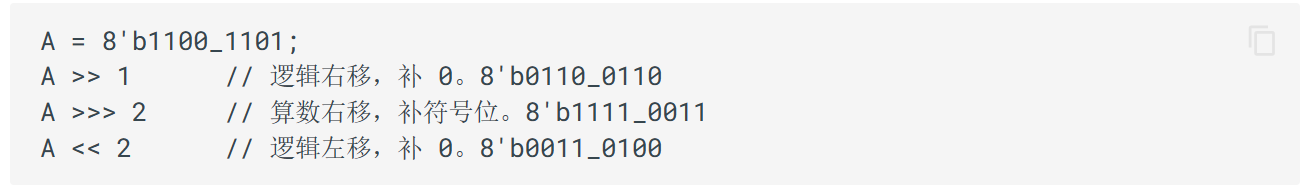
计算时，如果条件表达式为真（逻辑值为 1），则运算结果为真分支的结果；如果条件表达式为假（逻辑值为 0），则计算结果为假分支的结果。条件运算符也可以嵌套使用，以进行更为复杂的逻辑选择。

**移位运算符**

共四种。为：

1. <<（逻辑左移）
2. >>（逻辑右移）
3. <<<（算数左移）
4. >>>（算数右移）

在算术左移和逻辑左移时，右边低位会补 0。在逻辑右移时，左边高位会补 0；而算术右移时，左边高位会补充符号位，以保证数据缩小后值的正确性。例如：

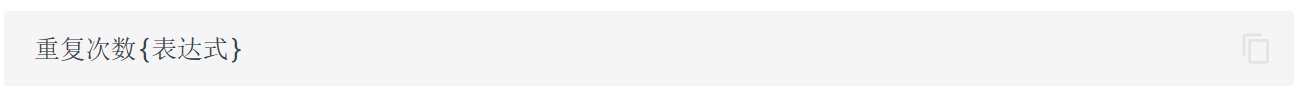


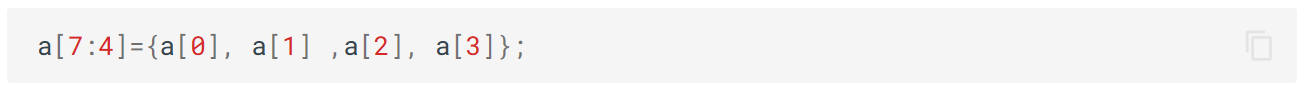
**拼接运算符**

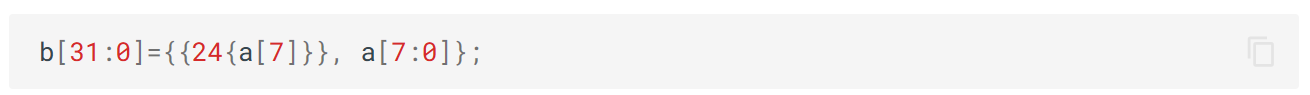
拼接运算符用大括号表示，一般形式为

拼接运算符用于将多个操作数拼接成新的操作数。其中的表达式既可以是常数也可以是变量，但是位宽必须是确定且不可变的。

与拼接操作经常一起使用的是重复操作，其一般格式是：

例如：

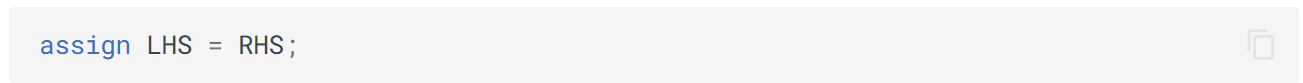
表示将 a 信号的低 4 位颠倒并赋值给高 4 位，

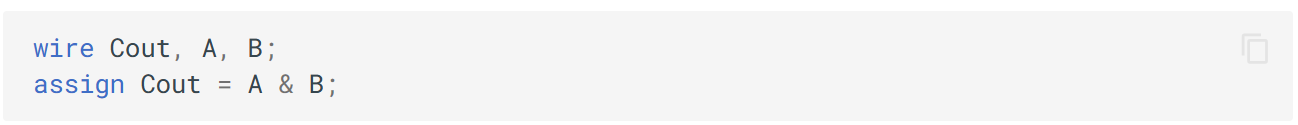
表示将 a 信号的低 8 位先符号扩展到 32 位后再赋值给 b 信号。

## 2 Verilog 语句

### 2.1 连续赋值：assign

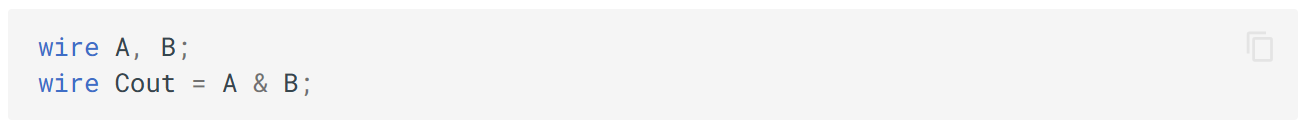
连续赋值语句用于对 wire 型变量进行赋值。其通用格式如下：

其中 assign 为 Verilog 的关键字，LHS（left hand side）指赋值操作的左侧（左值），RHS（right hand side）指赋值操作的右侧（右值）。下面是一个简单的示例：

这里有一些语法细节：

* LHS 必须是一个 wire 型变量，而不能是 reg 类型的变量；
* RHS 的类型没有要求；
* 只要 RHS 表达式的操作数有事件发生（值的变化）时，RHS 就会立刻重新计算，同时赋值给 LHS。这体现了 Verilog 语言的硬件特征：assign 语句实际上是构建了一段门电路，会长期存在于数字系统之中。

Verilog 还提供了另一种对 wire 型赋值的简单方法，即在 wire 型变量声明的时候同时对其赋值。例如，下面的赋值方式和上面赋值例子的效果是一致的。

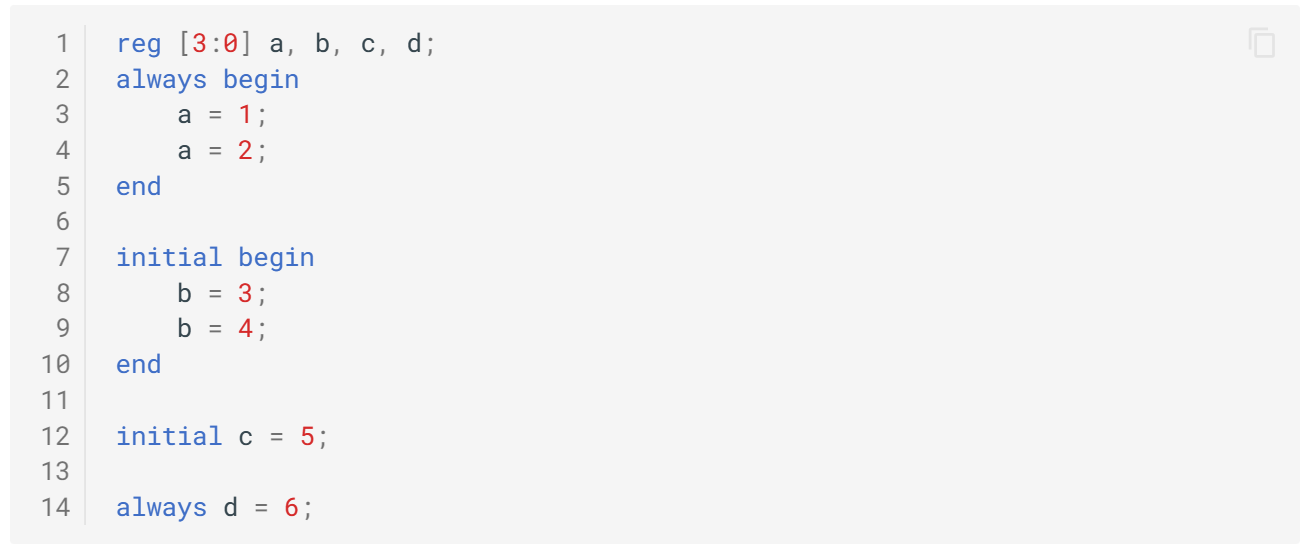


### 2.2 过程赋值：always

过程赋值语句用于对 reg 型变量进行赋值，由 2 种关键字引导，分别是 initial 与 always。这两种语句不可嵌套使用，彼此间并行执行（执行的顺序与其在模块中的前后顺序没有关系）。如果 initial 或 always 语句内包含多个语句，则需要搭配关键字 begin 和 end 组成一个块语句。

每个 initial 语句或 always 语句都会产生一个独立的控制流，执行时间都是从 0 时刻开始。二者的区别在于 initial 仅在 0 时刻开始执行一次内部的语句，而 always 语句块从 0 时刻开始执行，当执行完最后一条语句后，便再次执行语句块中的第一条语句，如此循环反复。

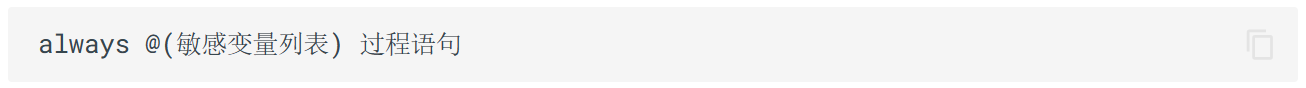
以下面的 Verilog 代码为例：



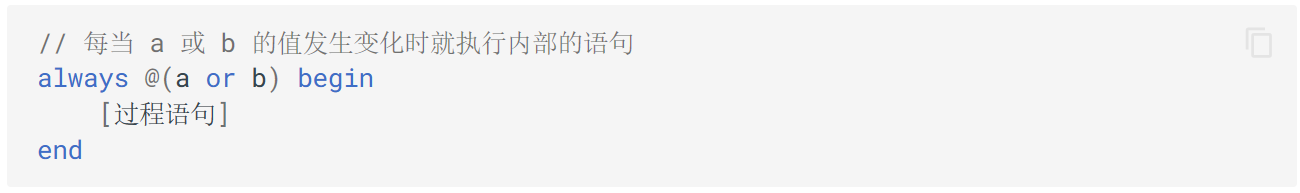
变量 a、b、c、d 会被同时赋值。

需要注意的是，上面的代码实际上是存在时序错误的。以第 14 行的 always d = 6; 为例，按照 always 的语句逻辑，电路将无限循环执行 d = 6 的赋值过程，且不存在任何延迟。这将导致严重的时序问题，并报出错误提示 A runtime infinite loop will occur.。

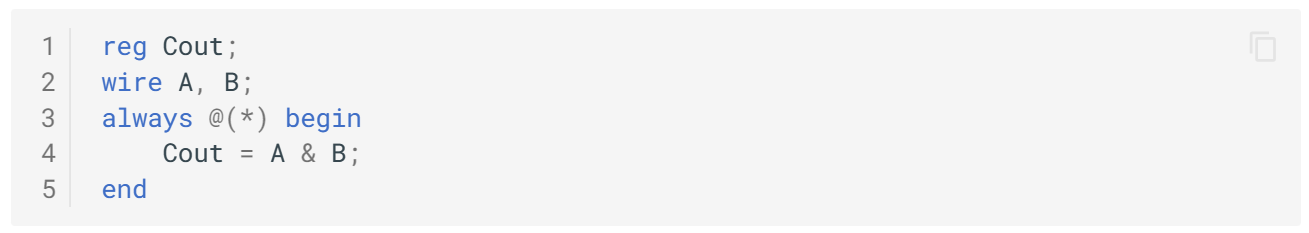
实践证明：没有任何条件限制的 always 并不是那么好用。因此，always 引入了敏感变量的概念。我们常常以下面的格式使用 always 语句：



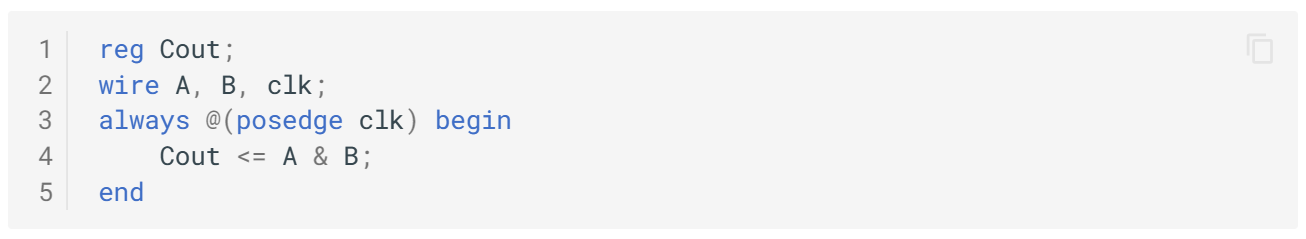
敏感变量就是触发 always 块内部语句的条件。加入敏感变量后，always 语句仅在列表中的变量发生变化时才执行内部的过程语句。



有的时候，敏感列表过多，一个一个加入太麻烦，且容易遗漏。为了解决这个问题，Verilog 2001 标准允许使用符号 \* 在敏感列表中表示缺省，编译器会根据 always 块内部的内容自动识别敏感变量。例如，先前的例子可以写为：



除了直接使用信号作为敏感变量，Verilog 还支持通过使用 posedge 和 negedge 关键字将电平变化作为敏感变量。其中 posedge 对应上升沿，negedge 对应下降沿。例如：下面的代码仅在 clk 从低电平（0）变为高电平（1）时触发。



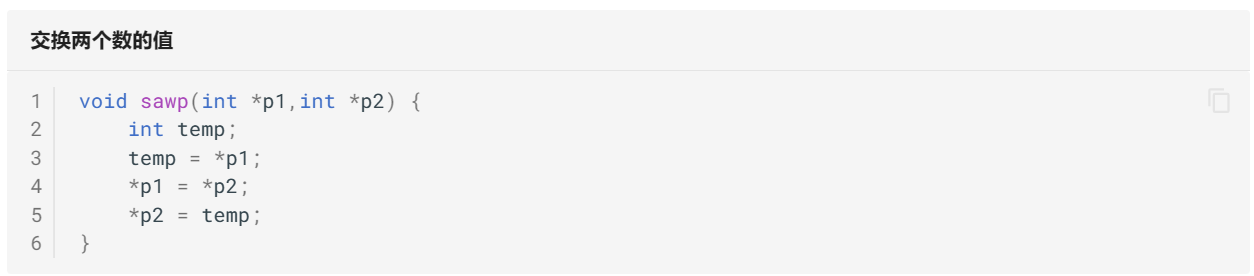
### ****2.3 阻塞赋值与非阻塞赋值****

注意到上面两段代码中，我们使用了两种运算符进行赋值操作：= 和 <=。它们分别对应着阻塞赋值和非阻塞赋值。

* 阻塞赋值是顺序执行的，即下一条语句执行前，当前语句一定会执行完毕。这与 C 语言的赋值思想是一致的。阻塞赋值语句使用等号 = 作为赋值符。
* 非阻塞赋值属于并行执行语句，即下一条语句的执行和当前语句的执行是同时进行的，它不会阻塞位于同一个语句块中后面语句的执行。非阻塞赋值语句使用小于等于号 <= 作为赋值符。

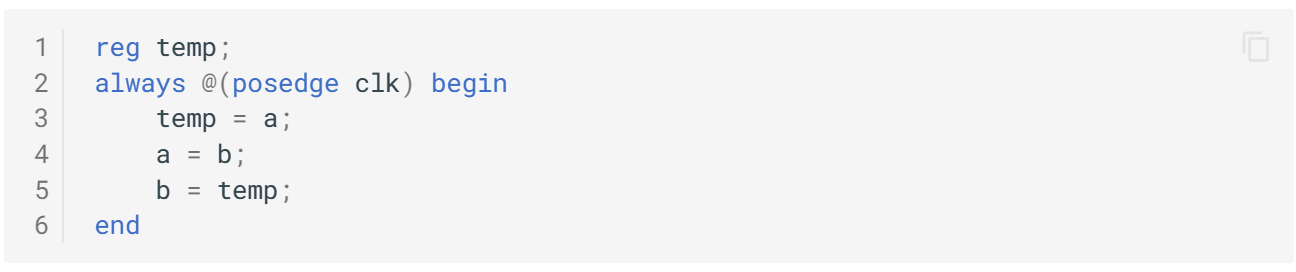
一般而言，在设计电路时，always 时序逻辑块中多用非阻塞赋值，always 组合逻辑块中多用阻塞赋值；在仿真电路时，initial 块中一般多用阻塞赋值。在实际的 Verilog 代码设计时，不要在一个过程结构中混合使用阻塞赋值与非阻塞赋值。两种赋值方式混用时，时序不容易控制，很容易得到意外的结果。

我们以一个例子加以分析。大家在学习 C 语言的时候，一定编写过这样一个函数：

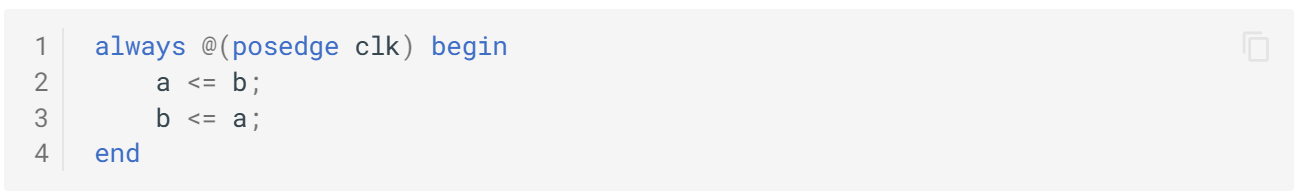


该函数实现了对两个 int 型数据的交换。那么，我们如何用 Verilog 实现这一功能呢？

假定现在有两个 reg 型变量 a 和 b，以及一个以一定周期进行电平翻转的信号 clk。我们希望在 clk 的上升沿交换 a 和 b 中的数值。由于 Verilog 中没有类似 C 语言中函数的概念（其实有，但我们目前没有教过），凭借着编程的直觉，我们不难写出下面的代码：



很幸运，这个代码是正确的。阻塞赋值保证了这三条语句是从上到下顺次执行，因此实现了值的交换。但实际上，我们并不需要使用中间变量 temp。下面的代码依然可以实现变量交换的功能。

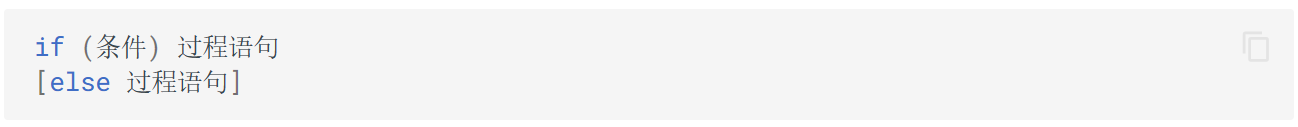


这段代码可能让你感到费解，但只要记住非阻塞赋值对应着同时执行，因此这两条语句不会有先后的差异，也就实现了值的交换。

### ****2.4 条件语句：if 与 case****

为了增加功能的多样性，Verilog 引入了条件分支语句。

if-else 语句用于实现带有优先级的条件分支，一般出现在 always 语句中，而不能直接在模块内部单独出现。其用法为：

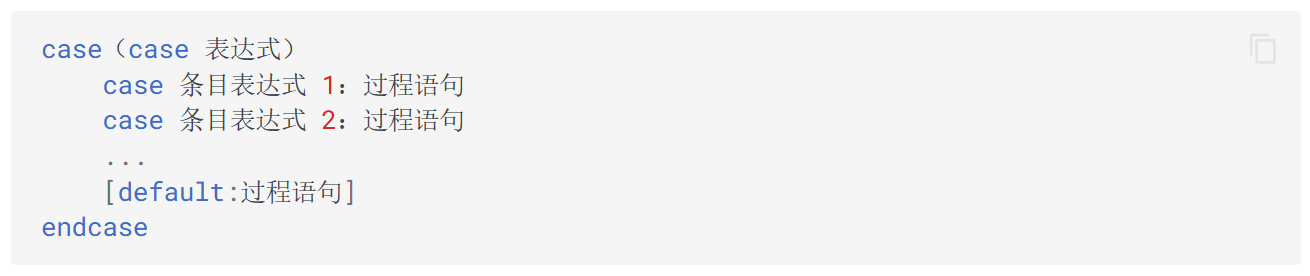


这里的 else 分支可以省略，但可能带来语法歧义以及其他的一些问题。因此，我们建议大家不要省略 else。

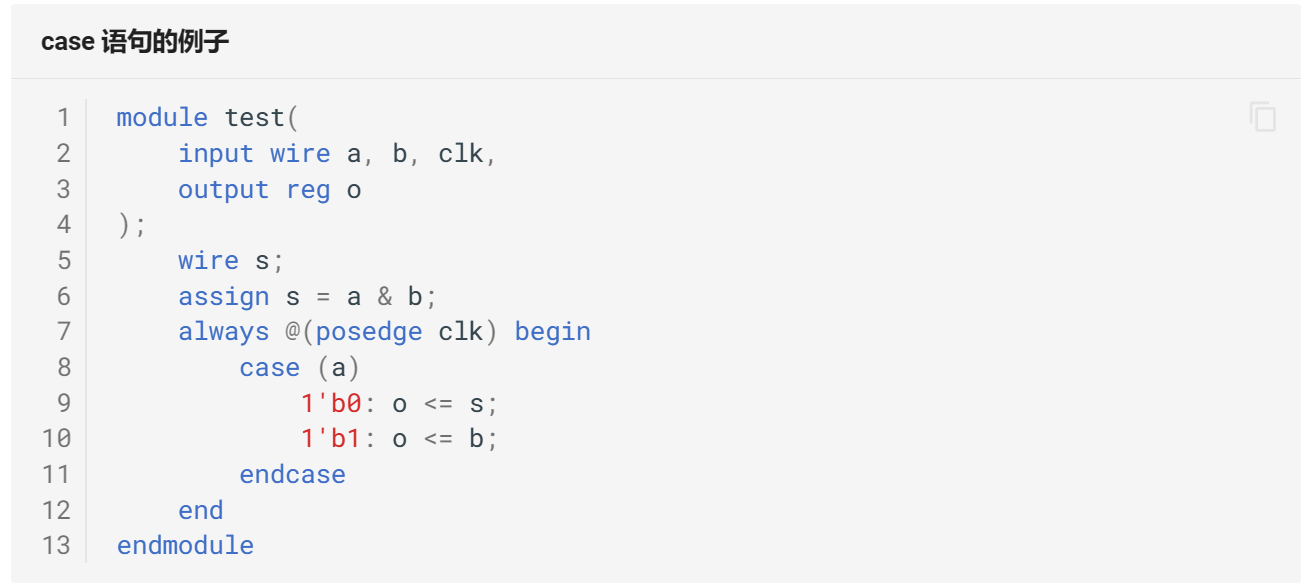


当时钟 clk 的上升沿到来时，我们首先判断 a != 0 是否成立。如果成立，则将 b 的值赋值给 o；否则将 s 的值赋值给 o。

除了 if 语句，Verilog 还提供了 case 语句用于具有相同优先级的条件分支。case 和 endcase 两个关键字必须成对出现。与 if-else 语句一样，case 语句出现在 always 的中，而不能在模块内部单独出现。其用法如下：



default 语句是可选的。在一个 case 语句中不能有多个 default 语句。过程语句可以是一条语句，也可以是多条。如果是多条语句，则需要用 begin 与 end 关键字进行说明。同样地，我们建议大家不要省略 default。



## ****3 基本单元：模块****

接下来，让我们来学习 Verilog 的基本单元：模块。

模块是具有输入和输出端口的逻辑块。它可以代表一个物理器件，也可以代表一个复杂的逻辑系统，例如基础逻辑门器件（三态门，与或非门等）或通用的逻辑单元（寄存器、计数器等）。

一个数字电路系统一般由一个或多个模块构成，每个模块实现某一部分的逻辑功能，而模块之间又需要按一定方式连接在一起实现所需求的系统功能。因此，数字电路设计也是使用硬件描述语言对数字电路/系统的基本模块以及模块之间的互连关系进行描述的过程。

### ****3.1 模块结构****

所有的模块以关键字 module 开始，以关键字 endmodule 结束。从 module 开始到第一个分号之间的部分是模块声明，它包括了模块名称与输入输出端口列表。模块内部由可选的若干部分组成，分别是内部变量声明，数据流赋值语句（assign），过程赋值语句（always）以及底层模块例化。这些部分出现顺序、出现位置都是任意的。变量声明的位置没有严格的要求，但必须保证在使用之前进行声明。

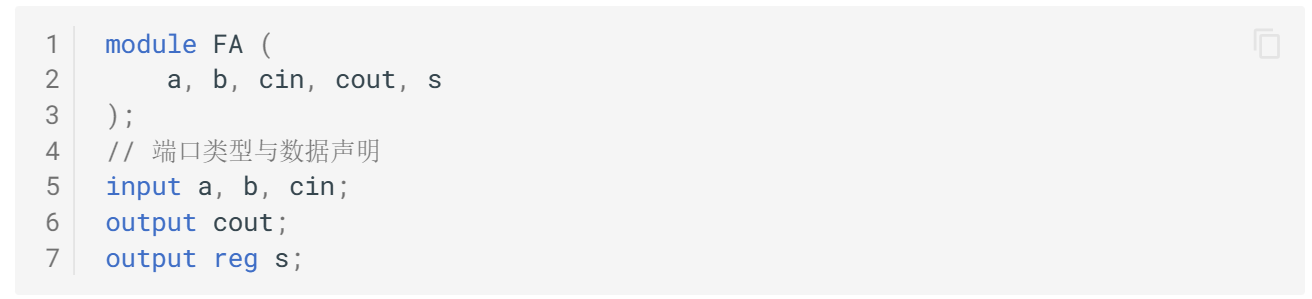
端口是模块与外界交互的接口。对于外部环境来说，模块内部的信号与逻辑都是不可见的，对模块的调用只能通过端口进行。端口列表是用于指定端口性质的集合，它包含了一系列端口信号变量。根据端口的方向，端口类型有 3 种： 输入端口（input）、输出端口（output）和双向端口（inout）。

你可以形象地将端口理解为函数调用的参数接口。端口的存在允许我们将模块视作一个黑盒，只需要正确连接端口，而无需关心模块内部的细节。下面是几种常见的端口声明方式。

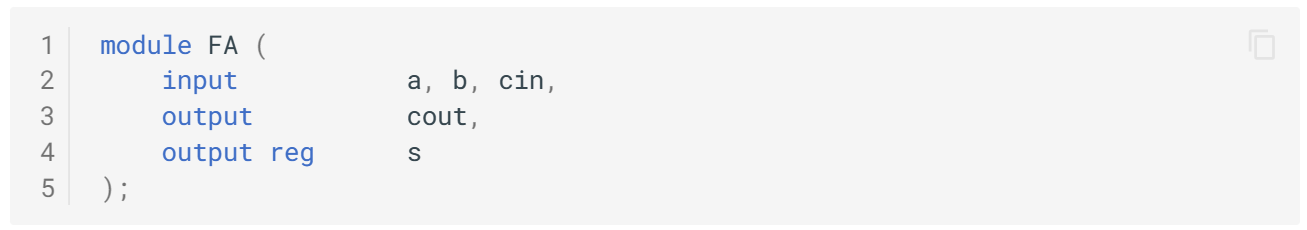


input、inout 类型的端口不能声明为 reg 数据类型，因为 reg 类型常用于保存数值，而输入端口只反映与其相连的外部信号的变化，不应保存这些信号的值。output 类型的端口则可以声明为 wire 或 reg 数据类型。

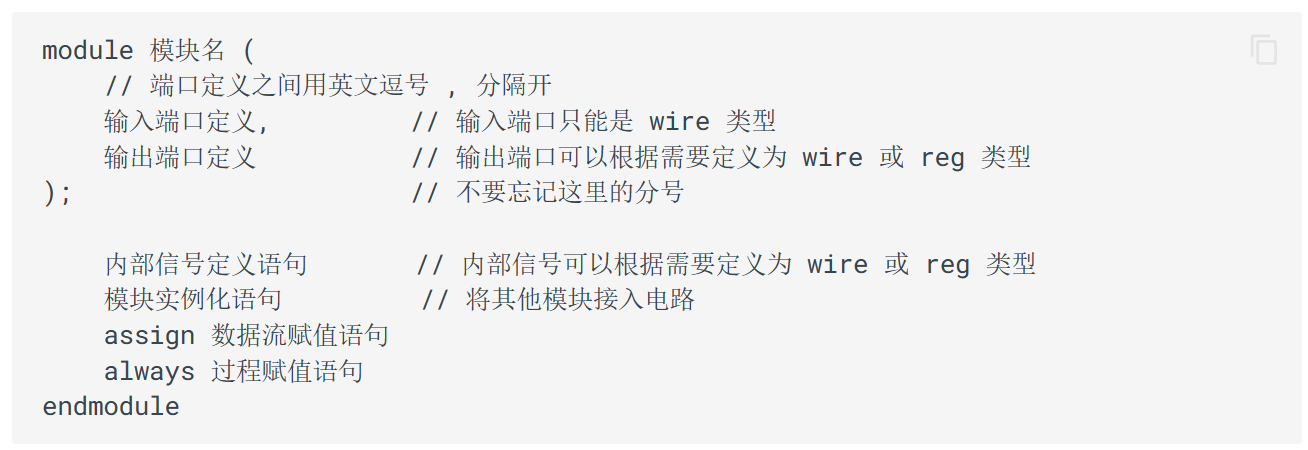
我们先前提到过，在 Verilog 中，wire 型为默认数据类型，因此当端口为 wire 型时，不用再次声明端口类型为 wire；但是当端口为 reg 型时，对应的 reg 声明不可省略。此外，端口类型和数据类型可以同时指定。基于此，上面的例子可以简化为：



实际编程时，更简洁且更常用的方法是在模块声明时就陈列出端口及其类型。



综上，Verilog 模块的基本结构可以总结为

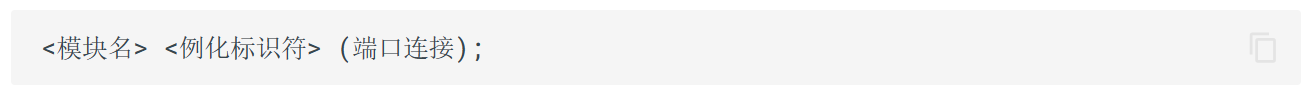


我们简单概括如下。

* 第一：每个模块都是由关键字 module 开头，由 endmodule 结束。
* 第二：每个模块都应该有一个唯一的模块名，模块名不能使用 Verilog 语法的关键字。
* 第三：模块名后面的括号内是对输入输出信号的定义，除后面实验中要讲到的仿真文件外，任何能实际工作的模块都应该有输入和输出信号。
* 第四：模块主体部分只能出现四类语句（仿真文件中会用到的 initial 语句等暂不考虑）：内部信号定义、模块实例化、assign 语句、always 语句，每类语句的数量与顺序不受限制，但要遵循变量先定义后使用的原则。

### ****3.2 模块例化****

模块例化是指在一个模块中引用另一个模块，对其端口进行相关连接的过程。例化的基本形式为：

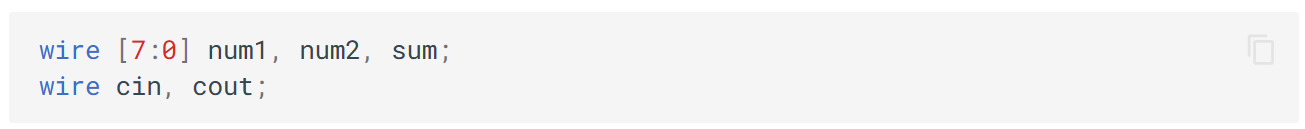


你可以简单地认为：模块例化就是在电路中放入了一个具有特定功能的集成电路。模块声明可以对标函数的声明，模块例化可以对标函数的调用。

假定我们已经写好了一个模块，其模块定义部分如下：

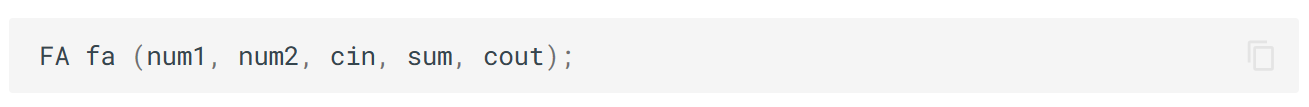


在顶层模块中，我们定义如下的变量



此时有两种可行的方式进行模块例化。

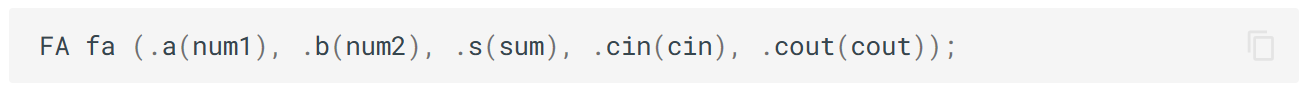
**1. 基于位置的端口关联**



这种方法将需要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，因此二者的位置要严格保持一致。虽然代码从书写上可能会占用相对较少的空间，但代码可读性低，也不易于调试。

在大型的设计中可能会有很多个端口，端口信号的数目和顺序也会有所改动，此时再利用顺序端口连接进行模块例化，显然是不方便的。

**2. 基于名字的端口关联**



这种方法将需要例化的模块端口与外部信号按照其名字进行连接，端口顺序可以与引用 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。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，甚至直接删除。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。

为了便于调试、保持良好的可读性，我们希望大家在例化时统一使用名字关联。大家可以按照下面的示例进行模块例化，每一行对应一个端口。

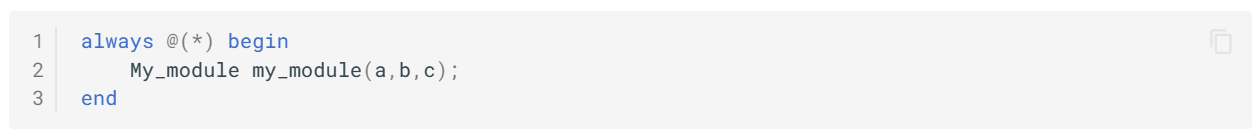


在名字例化方式中，我们需要额外介绍端口的连接规则。

* **输入端口：**模块例化时，input 端口可以连接 wire 或 reg 型变量；模块声明时，input 端口必须是 wire 型变量。
* **输出端口：**模块例化时，output 端口必须连接 wire 型变量；模块声明时，output 端口可以是 wire 或 reg 型变量。
* **输入输出端口：**模块例化和模块声明时，inout 端口都必须连接 wire 型变量。

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，即端口例化处保持空白。当 output 端口悬空时，我们甚至可以在例化时将其省略。input 端口悬空时，模块内部输入的逻辑功能表现为高阻状态（逻辑值为 z）。因此，例化时一般不能将悬空的 input 端口删除，否则编译会报错。我们建议大家对于 input 端口不要做悬空处理，在没有其他外部连接时使用常量进行连接。

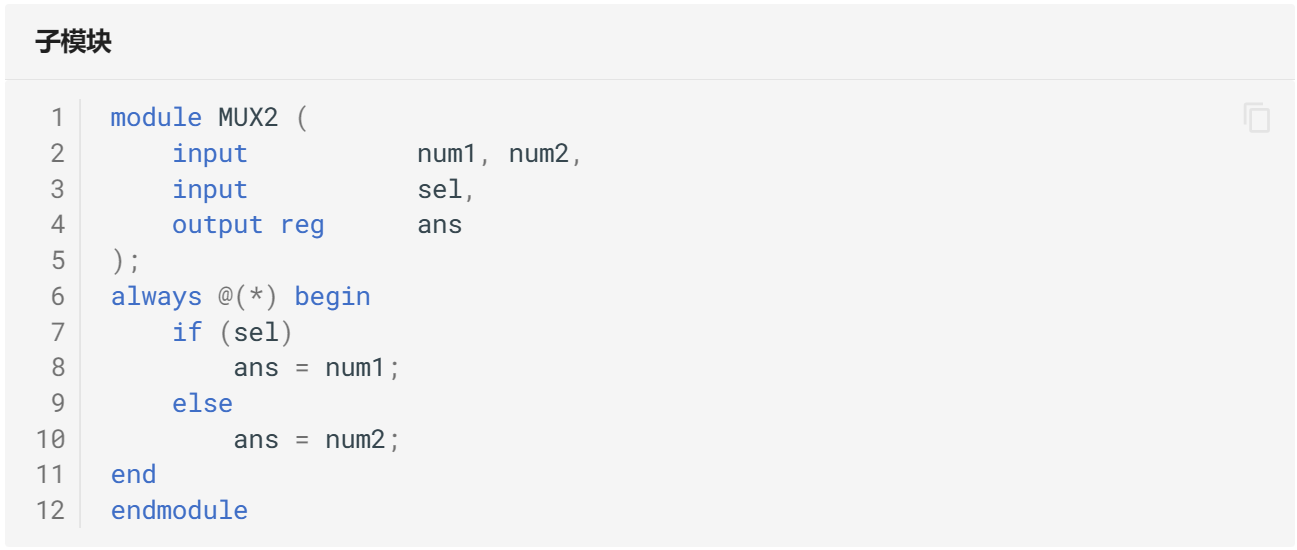
值得一提的是，多同学在初学 Verilog 的时候会写出下面的代码：



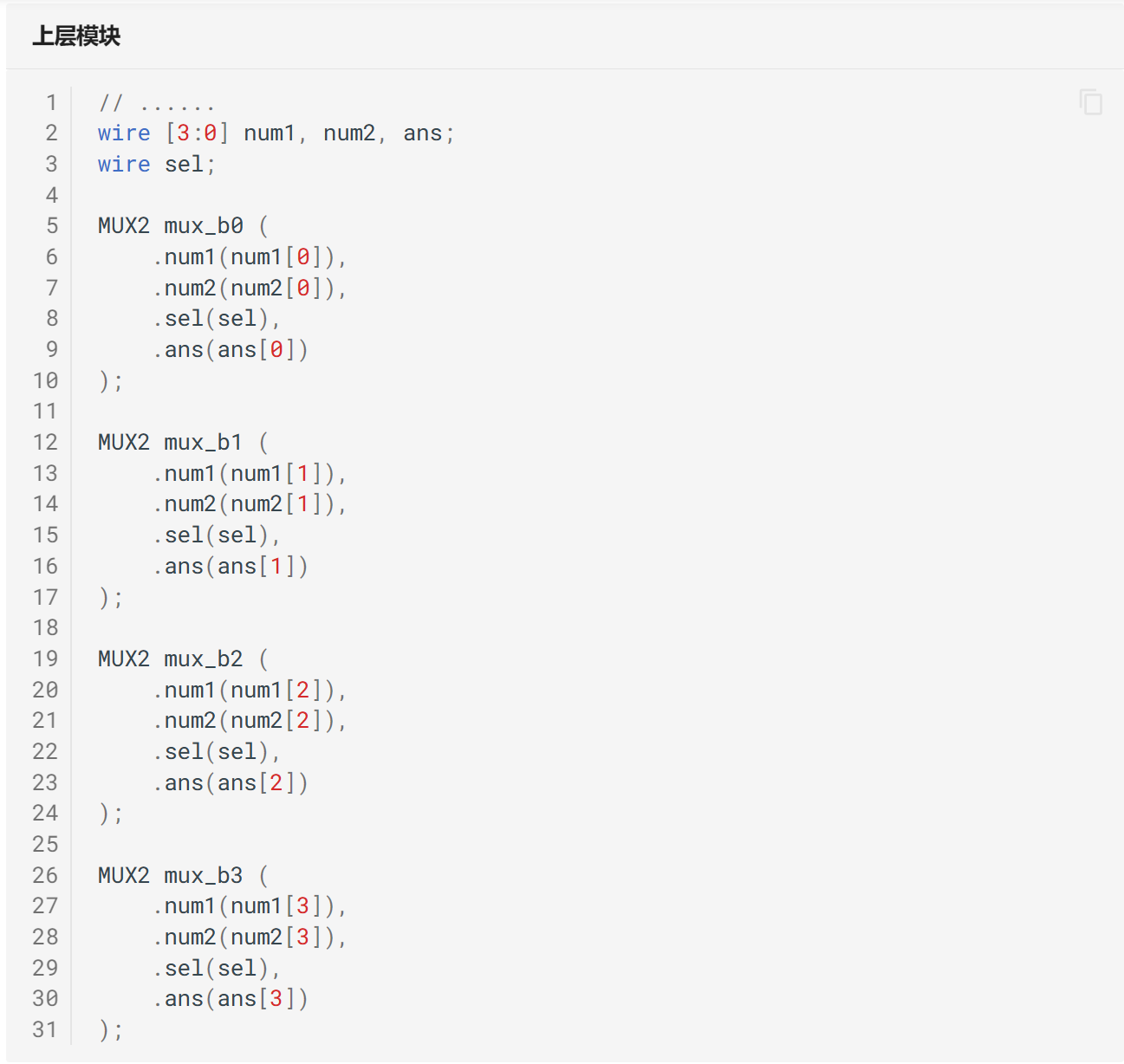
这是由于没有弄清楚 always 语句的意义而造成的。always 作为过程赋值语句，其与模块例化的关系是平等的，都是模块实现中的组成部分。我们不能将模块例化放入过程赋值语句中，因为这样做没有任何的道理。

### ****3.3 参数传递****

模块例化功能大大提升了 Verilog 的代码复用能力。假定我们有如下所示的模块代码：



不难看出，该模块接收三个 1bit 位宽数据的输入，输出一个 1bit 位宽的数据。但如果现在顶层模块的输入 num1、num2 是两个 4bits 数据，我们应该怎么办呢？自然，使用四个选择器分别选择每一位是一个可行的方案。



这种方法固然可行，但在数据位宽较大时便会十分繁琐。一种新的思路是：在编写子模块时并不预先指定位宽，而是在例化的时候根据需要确定位宽。此时我们可以使用 Verilog 的带参数例化功能：模块声明时使用 parameter 关键字指定参数，例化时将新的参数值写入模块例化语句，以此来改写子模块的参数值。

上面的代码可以更改为：



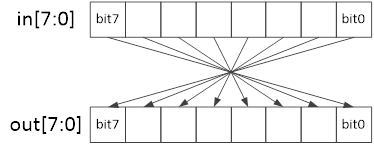
此时，子模块中的变量 num1、num2 和 ans 都是位宽为 WIDTH 的信号变量。参数 WIDTH 的默认值为 8。



在顶层模块中，我们指定子模块的参数值为 4，便可以正常输入 4bits 位宽的数据。

# 思考与练习

1. 逻辑非 ! 和按位非 ~ 在什么情况下等价？在什么情况下不等价？逻辑或 || 和按位或 | 呢？逻辑与 && 和按位与 & 呢？
2. 关系运算符 <=（小于等于）和非阻塞赋值的符号相同。二者会出现冲突吗？
3. 能否给出一个 if-else 语句中，因为省略else而导致歧义的例子？
4. 请编写 Verilog 代码，将 8bit 的输入信号按位翻转，并输出到输出端口。该过程如下图所示：

[](https://soc.ustc.edu.cn/Digital/lab1/figs/tasks/q1.png)