**实验6**

**测试与仿真**

# **实验介绍**

在 Verilog 代码设计完成后，我们还需要进行一项重要的步骤：仿真。TestBench 是用于验证和测试设计模块的仿真环境，它可以生成输入信号并模拟设计模块的输出，进而测试设计电路的功能、性能与设计的预期是否相符。

一般来说，仿真阶段花费的时间会比设计花费的时间更多，因为需要根据各种可能的应用场景设计各式各样的样例，对应的代码编写也更加复杂。

# 实验内容

## 1 Testbench

### 1.1 概述

Testbench 的中文释义是试验台、测试架、试样、试验工作台，是硬件设计中用于测试设计模块的环境。那么，为什么在硬件设计领域会需要 Testbench 呢？首先，我们需要区分测试和 Debug 的概念。Debug 是程序开发阶段中消除逻辑错误的过程，它侧重于让程序能够正常运行；测试是编程完成后，测试程序的正确性的过程，它侧重于模拟尽可能多的输入情况，保证不同情况下程序都可以输出正确的结果。

例如：现在我们想用 C 语言编写一个冒泡排序程序。你可能会写出下面的代码：



写代码并保证其能正确排序的过程就是 Debug，而测试过程则对应着如下的思考：

* len 的值为 -1 时能否正常运行；
* 传入的 arr 大小和 len 不匹配时能否正常运行；
* 传入的 arr 是空指针时能否正常运行；
* ......

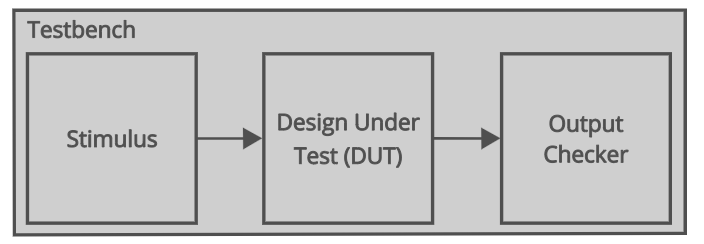
然而，在软件开发的过程中，我们很少涉及到测试的过程。一方面，我们的需求并不需要考虑这些奇奇怪怪的输入情况；另一方面，便捷高效的调试过程可以让我们立即发现并修复程序中可能的漏洞，把漏洞留给用户去上报再进行修复也是一种可接受的选择。

但是在硬件设计领域，一切都不同了。「你只能编写出自己能测试的模块」，一个自己设计但无力进行测试的复杂模块，往往也很难按照自己的预期进行工作。此外，硬件电路的工作状态是很难被我们获知的，因为芯片上没有 printf 函数，而为每一个元件连接一个显示器也不是什么好的选择。尽管一些芯片有内置的信息输出单元，但一方面其成本高昂，另一方面传输效率也十分低下。大多数情况下，摆在你面前的只有一个小小的、内部状态未知、工作不正常的芯片。

所以，我们选择在设计完成后引入 Testbench 进行测试，尽管编写 Testbench 的代价往往大于（甚至远大于）编写对应的待测试模块。如何编写高效的 Testbench 也成为了硬件开发中的重要一环。

### 1.2 基本结构

Testbench 由不可综合的 Verilog 代码组成，这些代码用于生成待测模块的输入，并验证待测模块的输出是否正确（是否符合预期）。下图展示了一个 Testbench 的基本架构。

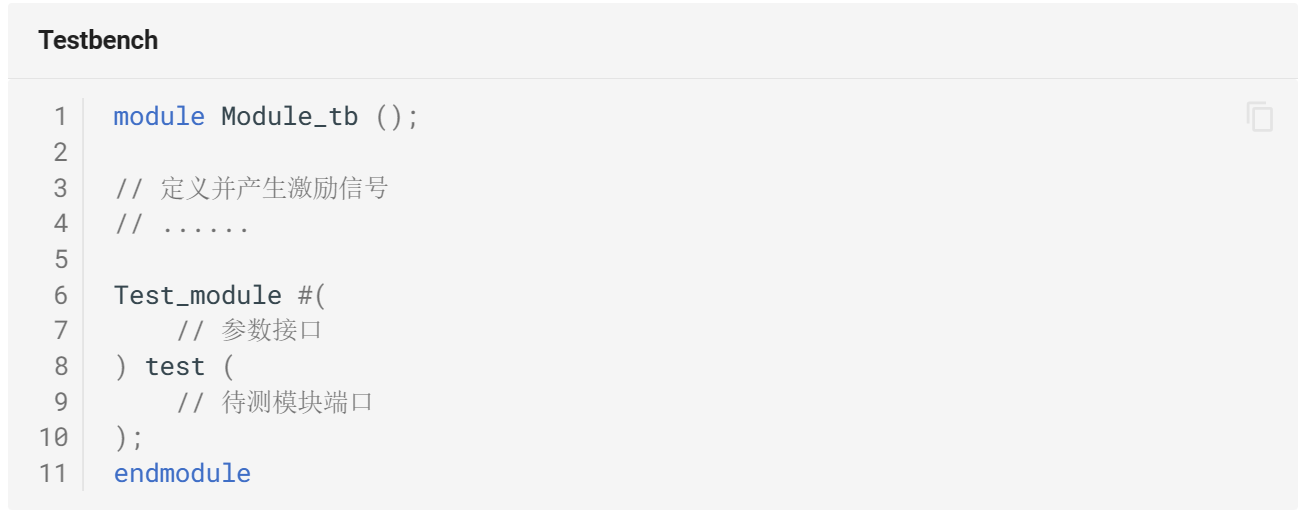
[](https://soc.ustc.edu.cn/Digital/lab2/figs/testbench/tb_structure.png)

其中：

* 激励（Stimulus Block）是专门为待测模块生成的输入。我们需要尽可能产生全面的测试场景，包括合法的和不合法的。
* 输出校验（Output Checker）用于检查被测模块的输出是否符合预期。
* 被测模块（Design Under Test, DUT。也称 Unit Under Test, UUT）是我们编写的 Verilog 模块，Testbench 的主要目的就是对其进行验证，以确保在特定输入下其输出均与预期一致。

编写 Testbench 的第一步是创建一个 Verilog 模块作为测试的顶层模块。与正常设计时的 Verilog module 不同，用于测试的模块应当没有输入和输出，这是因为 Testbench 模块应当是完全独立的，不受外部信号的干扰。

接下来，我们需要例化待测模块，将信号连接到待测模块以允许激励代码运行。这些信号包括时钟信号和复位信号，以及传入 Testbench 的测试数据。下面的代码片段展示了一个 Testbench 的基本框架。



## 2 Verilog测试语法

### 2.1 时序控制

与我们正常的设计代码不同，Testbench 中的代码并不需要被综合成实际电路，为此可以使用一些不可综合的语句，例如时序控制语句。综合是将 HDL 转换成较低层次电路结构的过程，包括查找表 LUT、触发器、RAM 等。有一些 Verilog 语法结构无法与这些电路结构对应，因此就产生了不可综合语句。例如，除了延迟语句外，循环次数不确定的循环语句也是不可综合的。

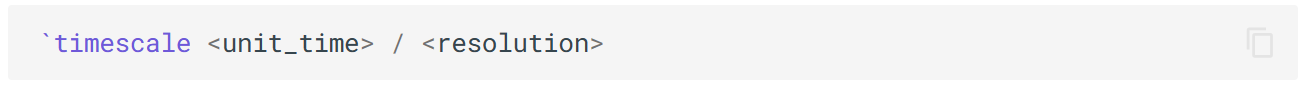
Verilog 中允许我们模拟两种不同的延时：惯性延时和传输延时。惯性延迟是逻辑门或电路由于其物理特性而可能经历的延迟，而传输延迟是电路中信号的『飞行』时间。

Verilog 使用 # 字符加上时间单位来模拟延时。例如 #10; 表示延迟 10 个时间单位后再执行之后的语句，对应着传输延迟。惯性延迟将延时语句写在与赋值相同的代码行中，这代表信号在延迟时间之后开始变化。例如：

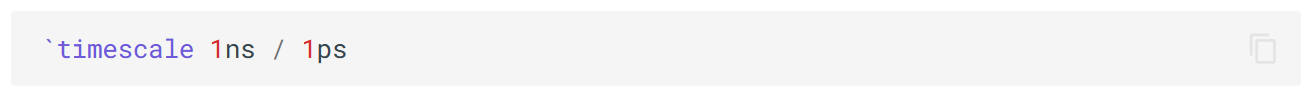


上面的代码中，A 或 B 任意一个变量发生变化，都会让 Z 在 10 个时间单位的延迟后得到新的值。如果在这 10 个时间单位内，A 或 B 中的任意一个又发生了变化，那么最终 Z 的新值会取 A 或 B 当前的新值。

为了明确在仿真期间所使用的时间单位，我们需要使用 `timescale 指令。其格式为：



其中， <unit\_time> 指定时间的单位，<resolution> 指定时间的精度，例如我们常常使用的



代表仿真的一个时间单位是 1ns，最小时间精度为 1ps。如果使用了 #1.1111; 指令，则最终的延迟为 1.111ns（四舍五入）。仿真时间单位和时间精度的数字只能是 1、10、100，不能为其它的数字。此外，时间精度应当不超过时间单位的大小。

除了直接控制延迟，Verilog 还支持基于事件的时间控制，这就是 @ 符号的作用。Verilog 有以下三种常用的事件控制方式：

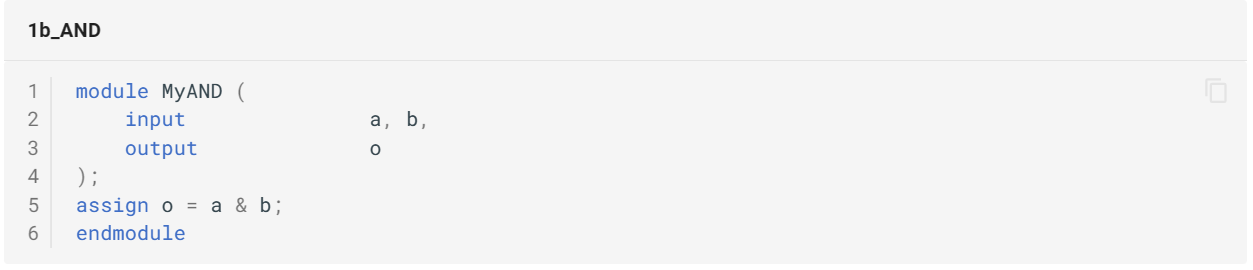
1. @ + 信号名，表示当信号发生逻辑变化时执行后面的内容。例如 @ (in) out = in;
2. @ + posedge + 信号名，表示当信号从低电平变化到高电平时执行后面的内容。例如 @ (podedge in) out = in;
3. @ + negedge + 信号名，表示当信号从高电平变化到低电平时执行后面的内容。例如 @ (negedge in) out = in;

这也印证了 always 语句的敏感变量语法。

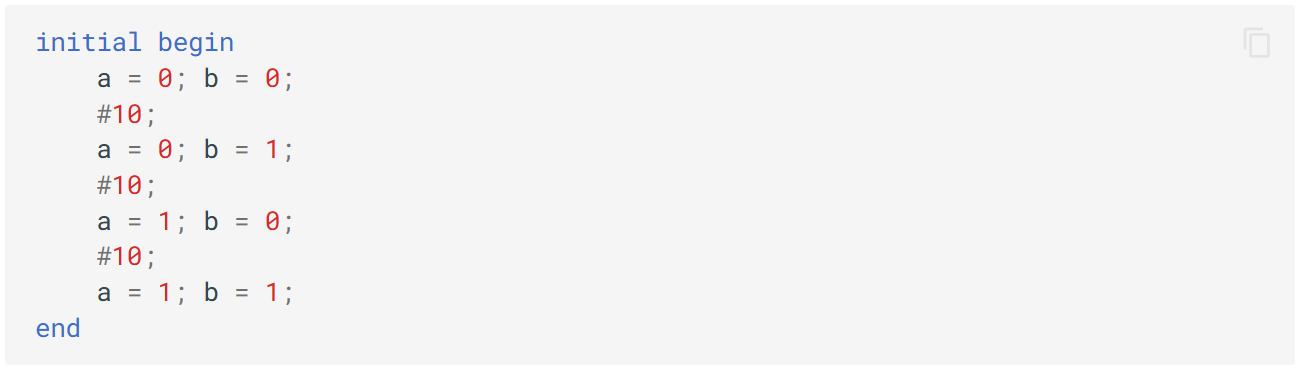
### 2.2 initial与always

在先前的实验中我们介绍过，initial 块中编写的任何代码都会在开始时执行，但仅执行一次，而 always 块则会循环执行内部的代码。与 always 块不同，在 initial 块中编写的 Verilog 代码几乎都是不可综合的，因此基本上只被用于仿真与初始化信号。

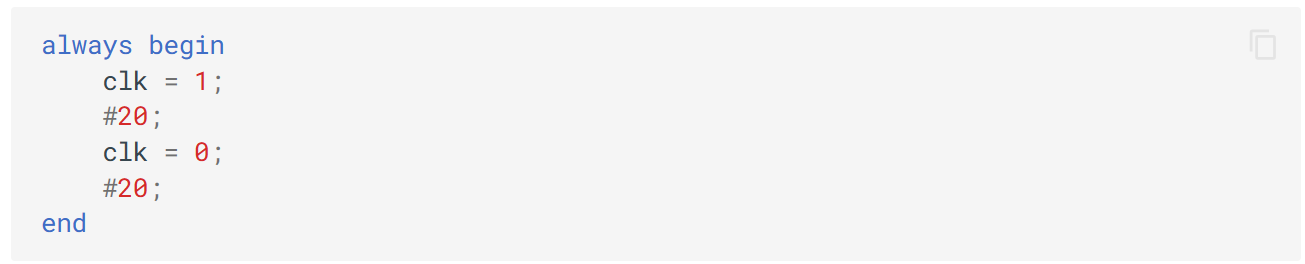
为了更好地理解 initial 块与 always 块在 Testbench 中的使用，我们来看一个例子。



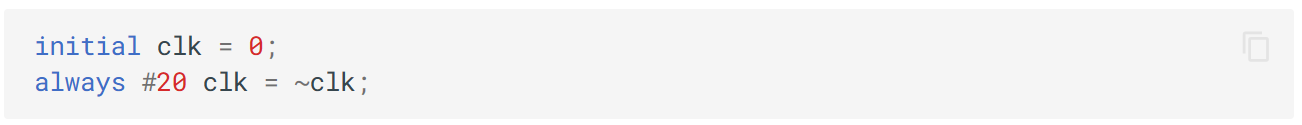
位宽为 1 的两输入与门一共只有 4 种可能的输入，因此我们可以直接枚举所有可能的情况。此外，我们还需要使用延时运算符在不同的输入之间增加一段延迟，便于我们观察到结果的变化。下面的 Verilog 代码展示了使用 initial 块编写 Testbench 的方法。



如果想要模拟一个时钟信号，则可以使用 always 语句。例如：



这段代码也可以写为

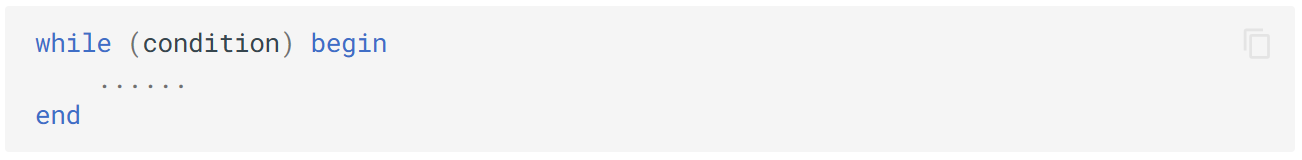


### 2.3 循环

除了先前介绍的赋值语句和分支语句，Verilog 中也有循环语句。它们分别是 while、for、repeat 和 forever 循环。循环语句只能在 always 或 initial 块中使用，其内部可以包含延迟表达式。这四种循环的介绍如下。

#### 2.3.1 While

while 循环的基本格式为：



while 循环的中止条件是 condition 为假。如果一开始 condition 已经为假，那么循环内的语句将一次也不会执行。一个简单的例子如下：



这段代码让 counter 从 0 开始，每 10 个时间单位增加 1。

#### 2.3.2 For

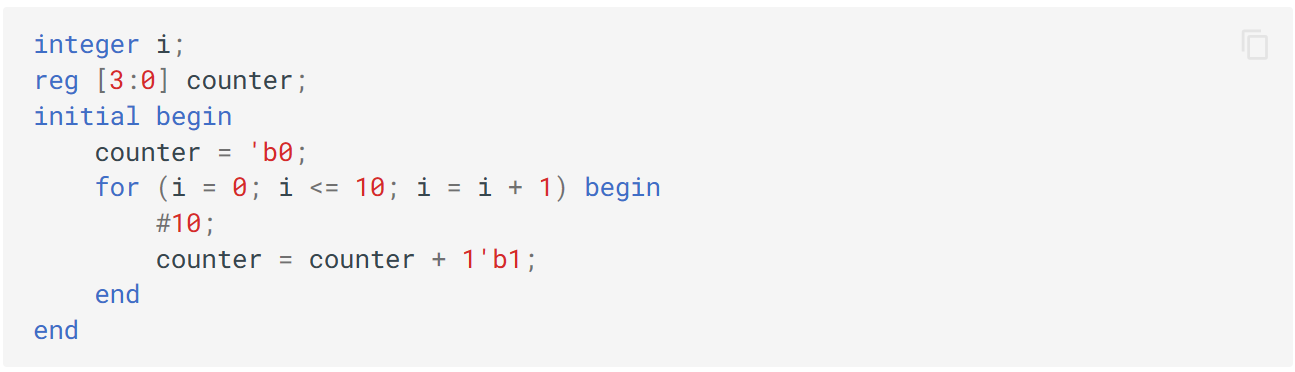
for 循环的基本格式为：



其中，initial\_assignment 为初始条件。condition 为循环条件，为假时立即跳出循环。step\_assignment 为改变控制变量的过程赋值语句，通常为增加或减少循环变量的值。

一般来说，因为初始条件和自加操作等过程都已经包含在 for 循环的头部，所以 for 循环写法比 while 循环更为紧凑，但也不是所有的情况下都能使用 for 循环来代替 while 循环。

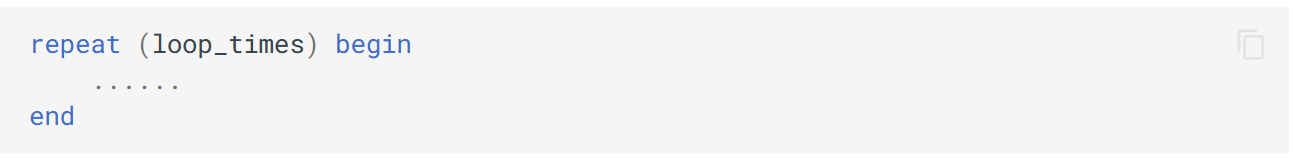
下面是一个 for 循环的例子，实现了与之前 while 循环例子一样的效果。



这里我们定义了一个 integer 类型的变量。integer 类型实际上是有符号的 reg 类型，一般用于描述循环变量或计算。通常来说，integer 类型的变量是 32 位的。值得注意的是，在 Verilog 语言里，i = i + 1 不能像 C 语言那样写成 i++ 的形式，i = i - 1 也不能写成 i-- 的形式。

#### 2.3.3 Repeat

repeat 循环的基本格式为：



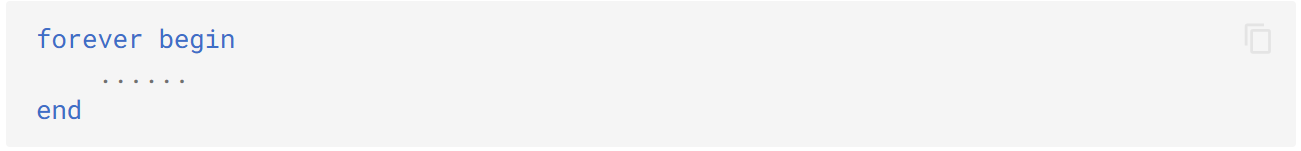
repeat 语句的功能是执行固定次数的循环，它不能像 while 循环那样用一个逻辑表达式来确定循环是否继续执行。repeat 循环的次数必须是一个常量、变量或信号。如果循环次数是变量信号，那么循环次数是开始执行 repeat 循环时变量信号的值。即便执行期间循环次数代表的变量信号值发生了变化，repeat 循环的执行次数也不会改变。

下面是一个 repeat 循环的例子。



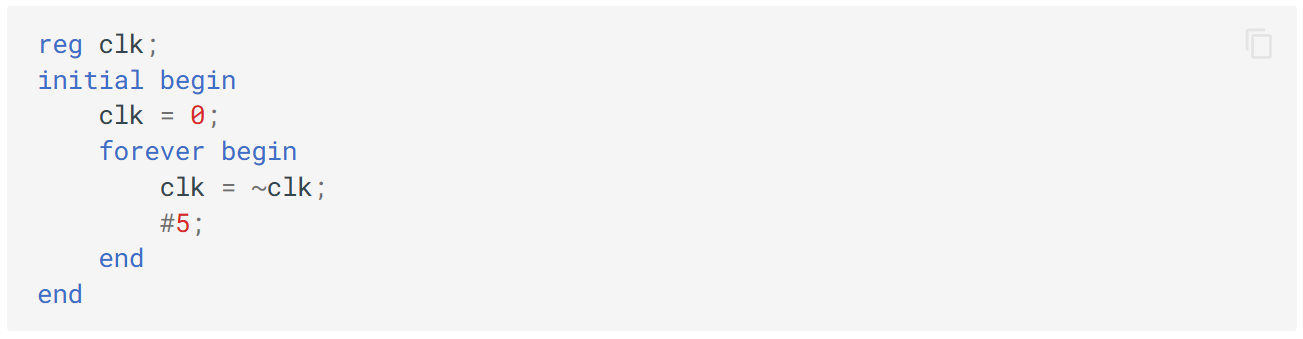
#### 2.3.4 Forever

forever 循环的基本结构为：

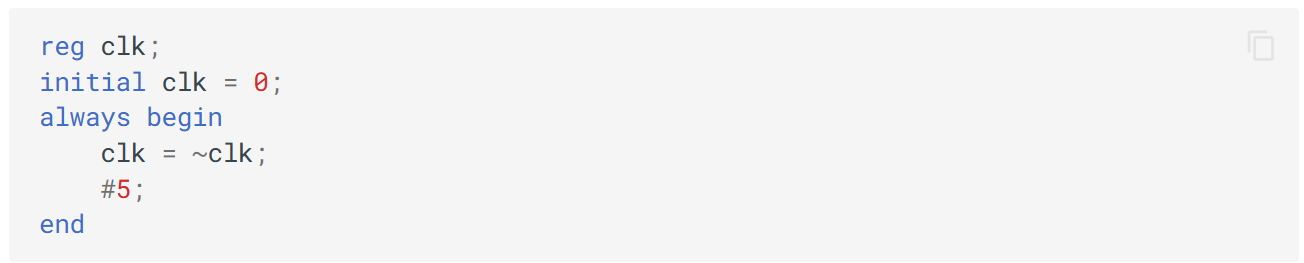


forever 语句表示永久循环，不包含任何条件表达式，一旦执行便永久执行下去。使用系统函数 $finish 可退出 forever 循环。

通常，forever 循环是和时序控制配合使用的。例如下面是使用 forever 语句产生一个时钟信号的 Verilog 代码：



这段代码等价于



### 2.4 系统任务

在 Verilog 中编写 Testbench 时，有一些内置的任务和函数可以为我们提供帮助。它们总是以美元符号 $ 开头，被统称为系统任务或系统函数。其中，下面三个是最常用的系统函数：$display、$monitor 和 $time。

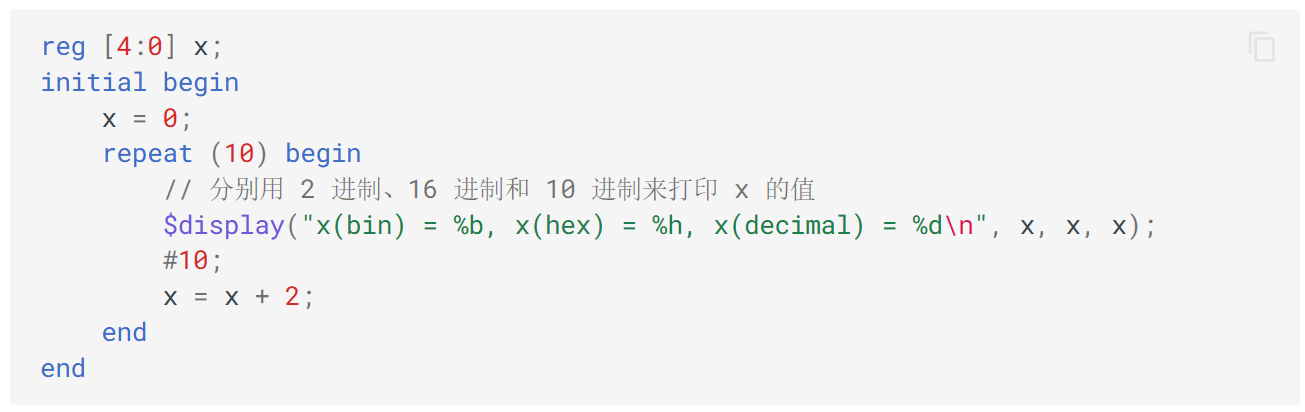
#### 2.4.1 $display

$display 允许我们在控制台上输出一条消息。该函数的使用方式与 C 语言中的 printf 函数非常类似，这意味着我们可以轻松地在 Testbench 中创建文本语句，并使用它们来显示有关仿真状态的信息。

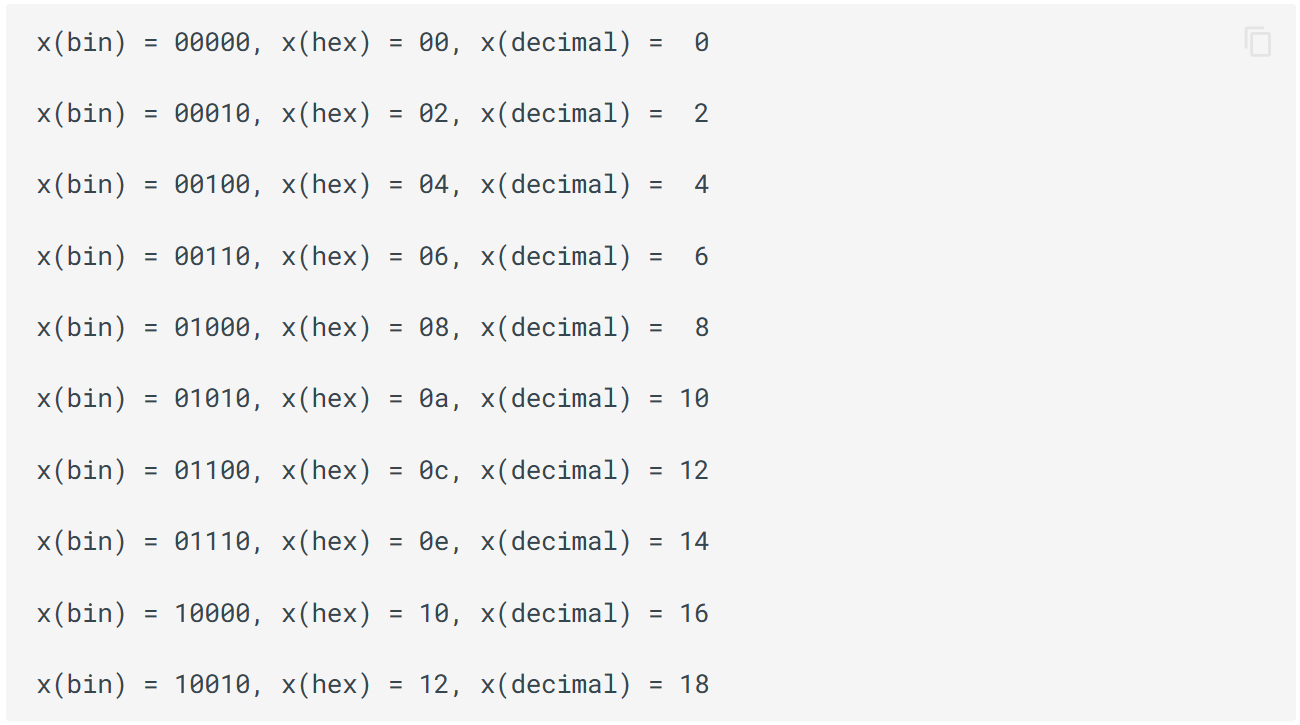
此外，我们还可以在字符串中使用特殊字符 % 来规范化显示信号数值。我们需要使用一个格式字母来决定以何种格式显示变量数值，在格式代码前面加上一个数字来确定要显示的位数。最常用的格式是 b（二进制）、d（十进制）和 h（十六进制）。

[](https://soc.ustc.edu.cn/Digital/lab2/figs/testbench/display.png)

下面是使用 $display 函数的一个例子：



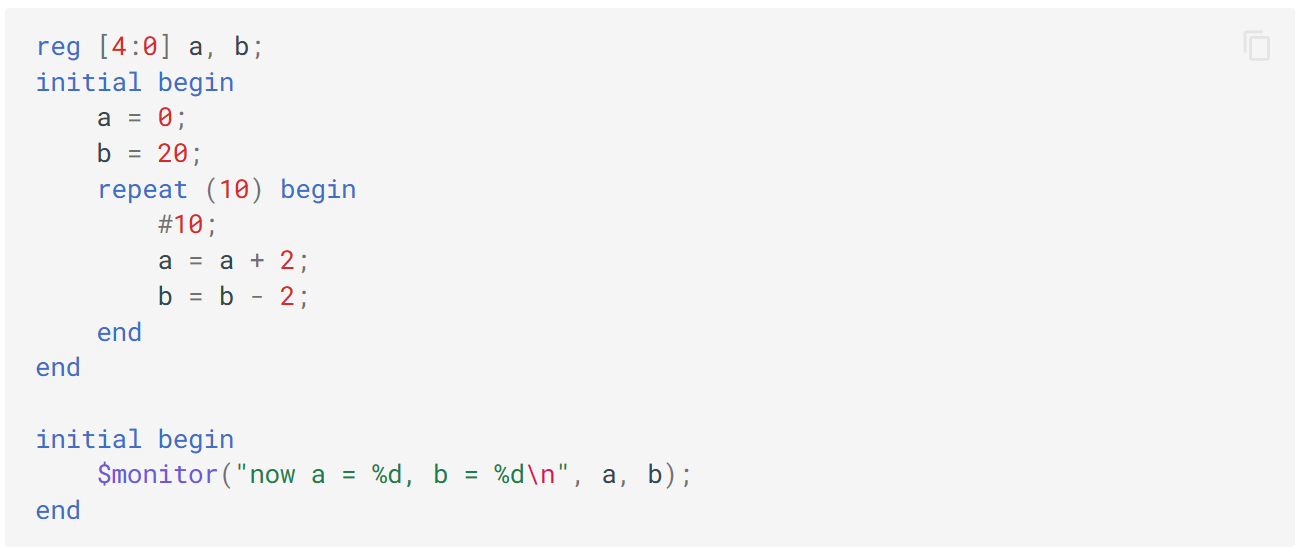
这段代码的输出结果为



#### 2.4.2 $monitor

$monitor 函数与 $display 函数非常相似，但它一般被用来监视 Testbench 中的特定信号。这些信号中的任何一个改变状态，都会在终端打印一条消息。

下面是使用 $monitor 函数的一个例子：



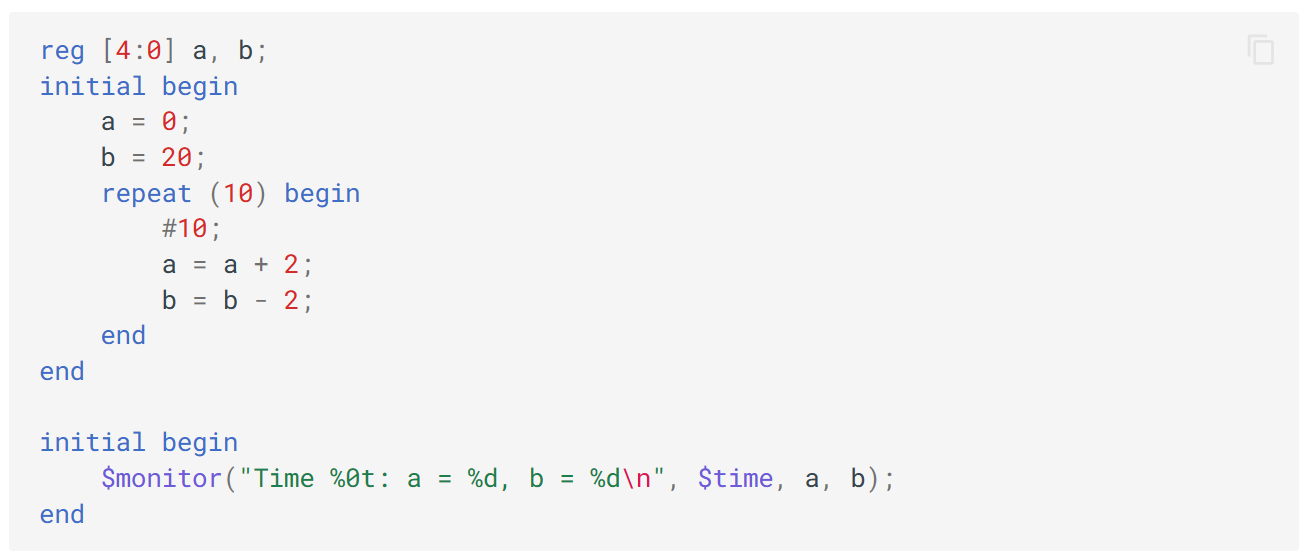
这段代码的输出结果为



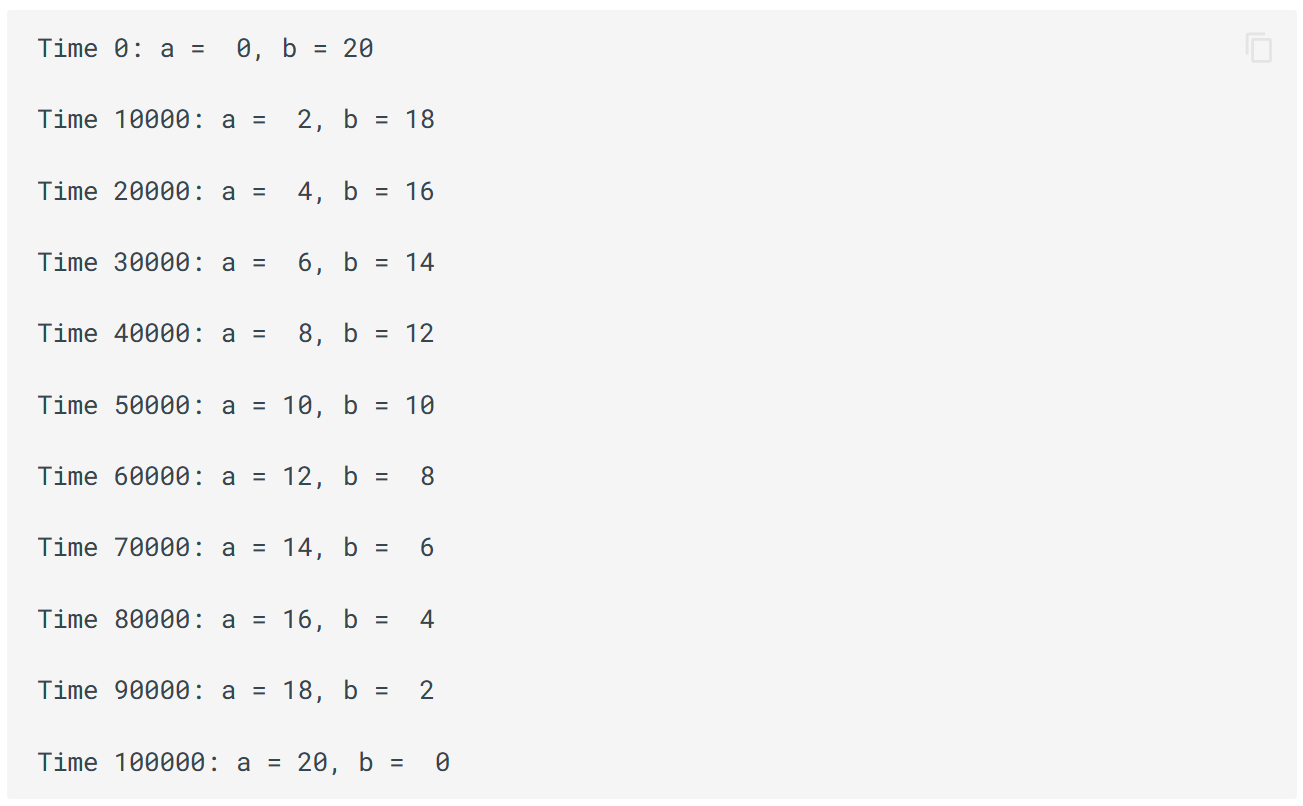
#### 2.4.3 $time

最后一个常用的系统任务是 $time，它可以用来获取当前的仿真时间。在 Testbench 中，我们通常将 $time 与 $display 或 $monitor 一起使用，以便在打印的消息中显示具体仿真时间。

下面是使用 $monitor 函数和 $time 的一个例子：

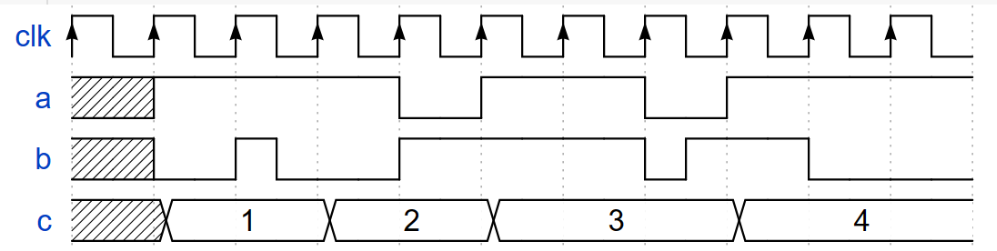


其中时间参数的设定为 `timescale 1ns / 1ps。这段代码的输出结果为：



# 思考与练习

1. 请编写仿真文件，生成下图所示的波形。时钟周期设定为 10ns，变量 c 的位宽自行指定（图中存在偏差，实际上变量 c 应在时钟上升沿变化）。图中的阴影部分代表电路状态不确定，你可以自行指定其状态。

[](https://soc.ustc.edu.cn/Digital/lab2/figs/tasks/q3.png)