**实验4**

**Verilog进阶设计**

# **实验介绍**

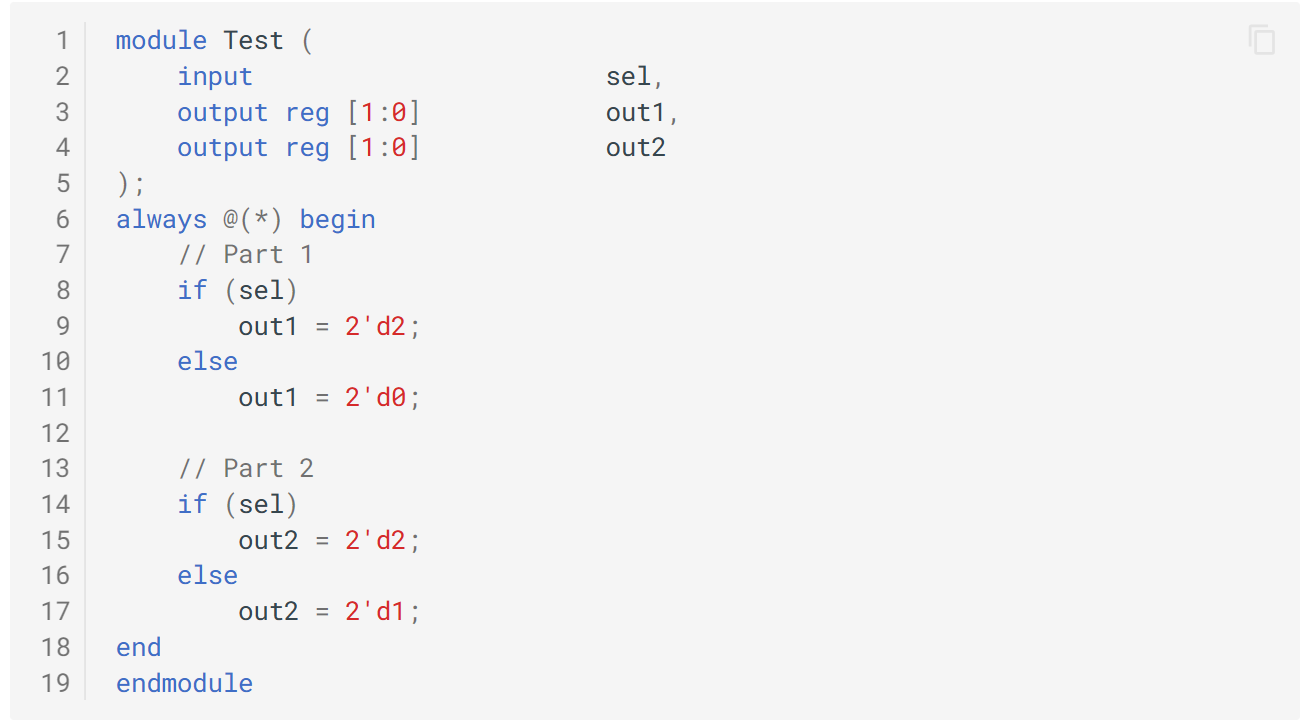
在了解 Verilog 的基础语法知识后，我们也将正式踏入 Verilog 硬件开发的阶段。下面，我们将学习更为深层次的 Verilog 设计思想，培养更为完善的系统观念。

# 实验内容

## 1 硬件层面的并行

我们知道，不同 always 块和 initial 块之间是并行执行的，而同一个模块里的 always 语句和 initial 语句执行顺序与其在模块中的位置无关。always 块和 initial 块内的过程赋值语句可以是阻塞赋值 =，也可以是非阻塞赋值 <=。其中阻塞赋值用于组合逻辑电路，为串行执行；非阻塞赋值用于时序逻辑电路，为并行执行。

然而，在同一个 always 块里的阻塞赋值语句也可以是并行执行的，只要其内部的信号不会产生冲突。我们来看下面这段代码：

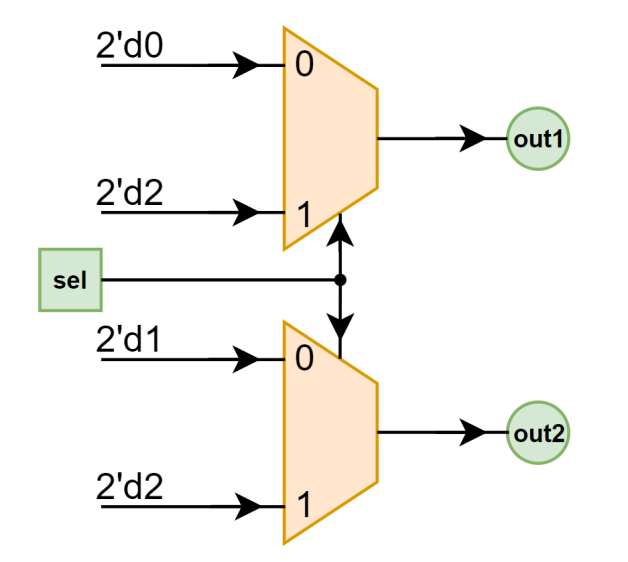


这段代码包含了一个 always @(\*) 语句块，表明当 sel、out1 或 out2 任何一个发生变化时，就执行内部的语句。不难发现，out1 和 out2 的逻辑是依赖于 sel 信号的，因此我们只需要关注 sel 信号的变化即可。

现在我们来思考下面的问题：当 sel 信号从 1'b0 变为 1'b1 的时候，out1 和 out2 谁先发生变化？按照 C 语言的理解，程序会首先执行 Part 1 中的 if 语句，得到 out1 的值为 2'd2，然后执行 Part 2 中的 if 语句，得到 out2 的值为 2'd2。因此 out1 应当比 out2 先发生变化。



但是在 Verilog 中，结果并不是这样。作为一门硬件描述语言，我们一直在强调：Verilog 中的每一条语句都对应着一种实际的硬件结构，例如 if 语句对应的是选择器。这段代码对应的硬件电路如下图所示：

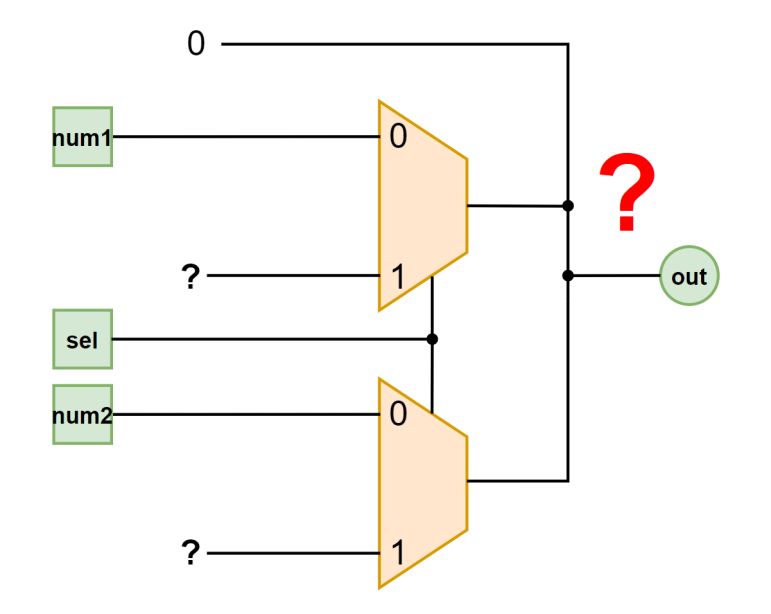
[](https://soc.ustc.edu.cn/Digital/lab2/figs/verilog/example_parallel_1.png)

不难看出，Part 1 和 Part 2 两部分是分离的，描述了两个不同的选择器。因此在硬件电路层面上，out1 和 out2 信号是同时生成的，二者之间不存在逻辑延迟。

那么，什么时候 Verilog 中的阻塞赋值会串行执行呢？答案是：在信号出现依赖与冲突时，自然就会串行执行了。例如下面这段 Verilog 代码：

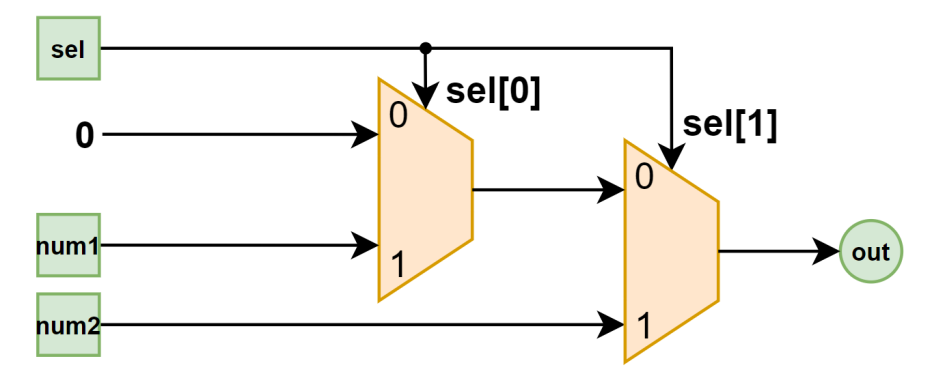


它对应了怎样的硬件结构呢？我们试着照葫芦画瓢画出它的结构：

[](https://soc.ustc.edu.cn/Digital/lab2/figs/verilog/example_parallel_2.png)

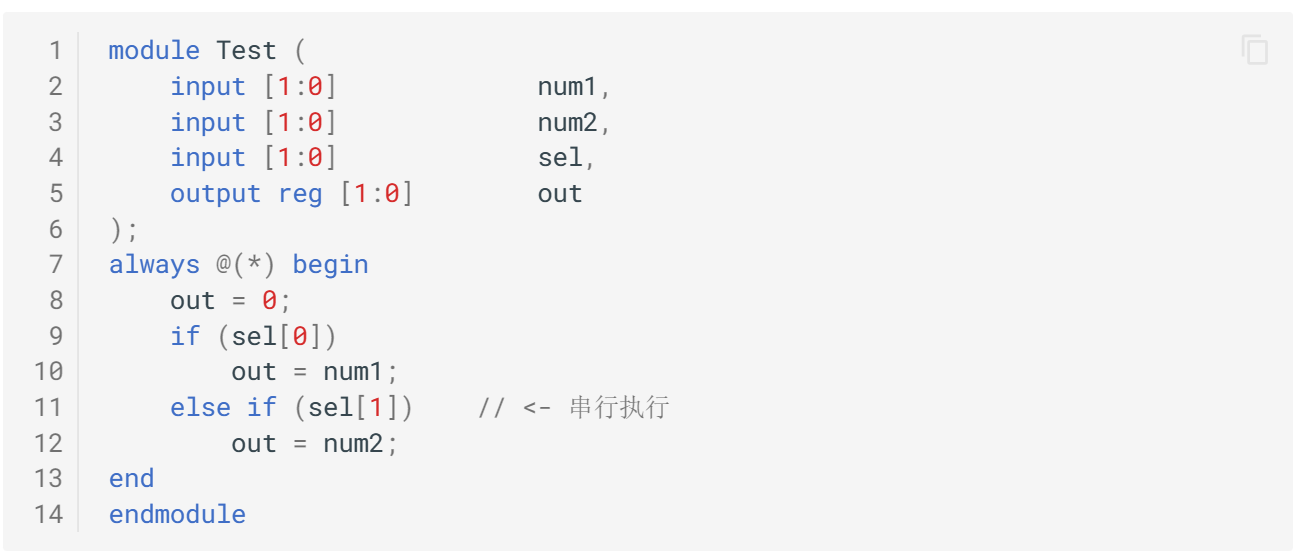
很明显，这里上下两部分的结果会产生冲突，因此这样的结构是错误的。在这里，always 中的语句是串行执行的，因为它们都对 out 变量进行了赋值操作。

我们来分析一下这一段 Verilog 代码的逻辑：当 always 语句执行时，首先执行第 8 行的赋值语句，out 的值为 0；接下来执行第 10 行的 if 语句：当 sel[0] 为 1 时将 out 赋值为 num1；然后执行第 14 行的 if 语句：当 sel[1] 为 1 时将 out 赋值为 num2。所以整段逻辑对应的硬件结构为：

[](https://soc.ustc.edu.cn/Digital/lab2/figs/verilog/example_parallel_3.png)

我们将上面的讨论结果总结如下：如果两个 if 的赋值对象没有冲突，那么两个 if 描述的多选器是并行的，否则是串行的。

当然，我们也可以使用 if-else-if 语句实现上面的结构，而 else-if 将显式指出多选器的串行执行顺序。



## 2 避免锁存器

锁存器（Latch）是最为基本的时序元件，我们将在后面的实验中对其进行详细介绍。简单来说，锁存器可以存储电路当前的状态保持不变，仅在外部控制信号的驱动下才能更改内部的信息。

在 Verilog 中，一个变量如果声明为寄存器类型（reg），它既可以被综合成组合逻辑的导线，也可能被综合成时序逻辑中的寄存器或锁存器。在使用 always @(\*) 语句时，我们会希望变量被综合成导线，但是有时候由于代码书写问题，它会被综合成我们不期望的锁存器结构，进而对电路带来危害。主要有：

* 电路的输出状态可能发生多次变化，增加了下一级电路的不确定性；
* 在大部分 FPGA 的设计里，锁存器结构会消耗更多的电路资源；
* 锁存器导致电路不能按照我们预期的方式工作，在调试时带来额外的问题。

因此，我们在代码书写时需要格外注意，应当避免出现锁存器。一个简单且好记的原则是：组合逻辑中不应出现记忆电路，即电路不能保存自身的状态。违反了这一原则的组合逻辑电路往往就会产生锁存器。

我们来看下面的这些例子。

### ****2.1 if-else逻辑缺陷****

在组合逻辑电路中，不完整的 if-else 结构会产生锁存器。例如下面这段 Verilog 代码：

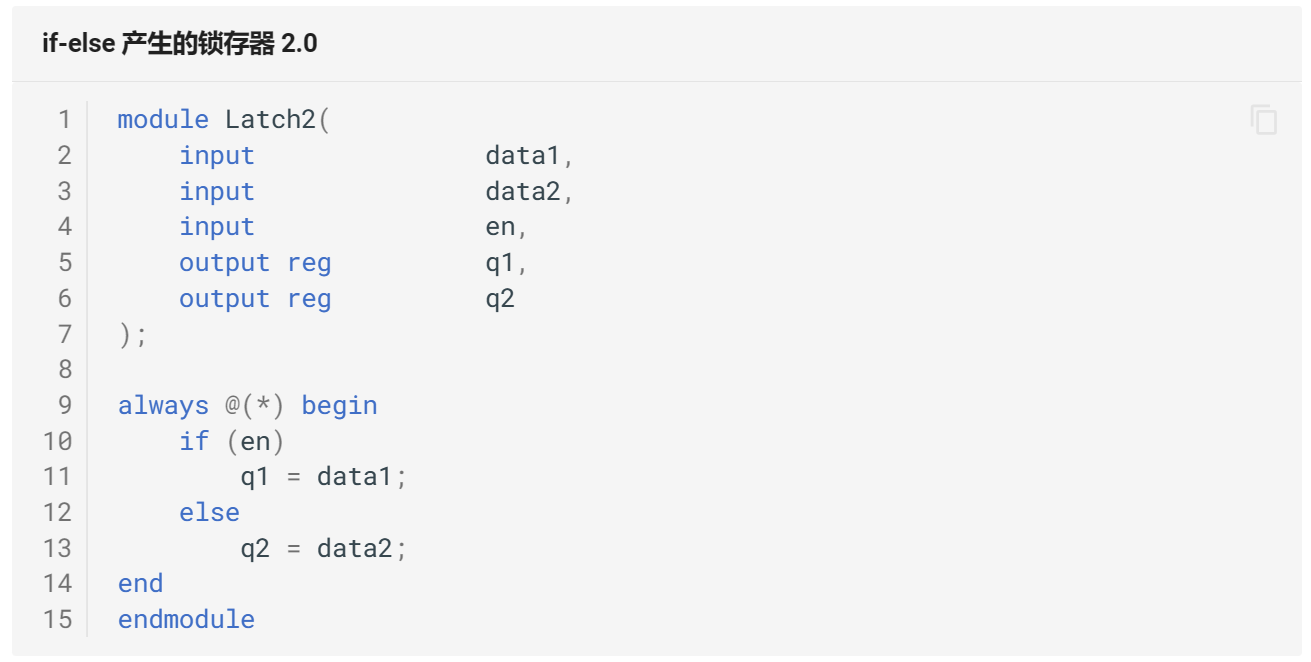


我们来分析一下。当 en 信号为 1 时，q 会被赋值为 data 的值；当 en 信号不变时，由于 always 语句里的 if 缺少对应的 else 分支，因此编译器默认 else 的分支下寄存器 q 的值保持不变。此时电路应当具有存储数据的功能，所以变量 q 会被综合成锁存器结构。

避免此类锁存器的方法主要有 2 种，一种是补全 if-else 结构，另一种是对信号赋初值。例如，上面的代码可以改为以下两种形式：

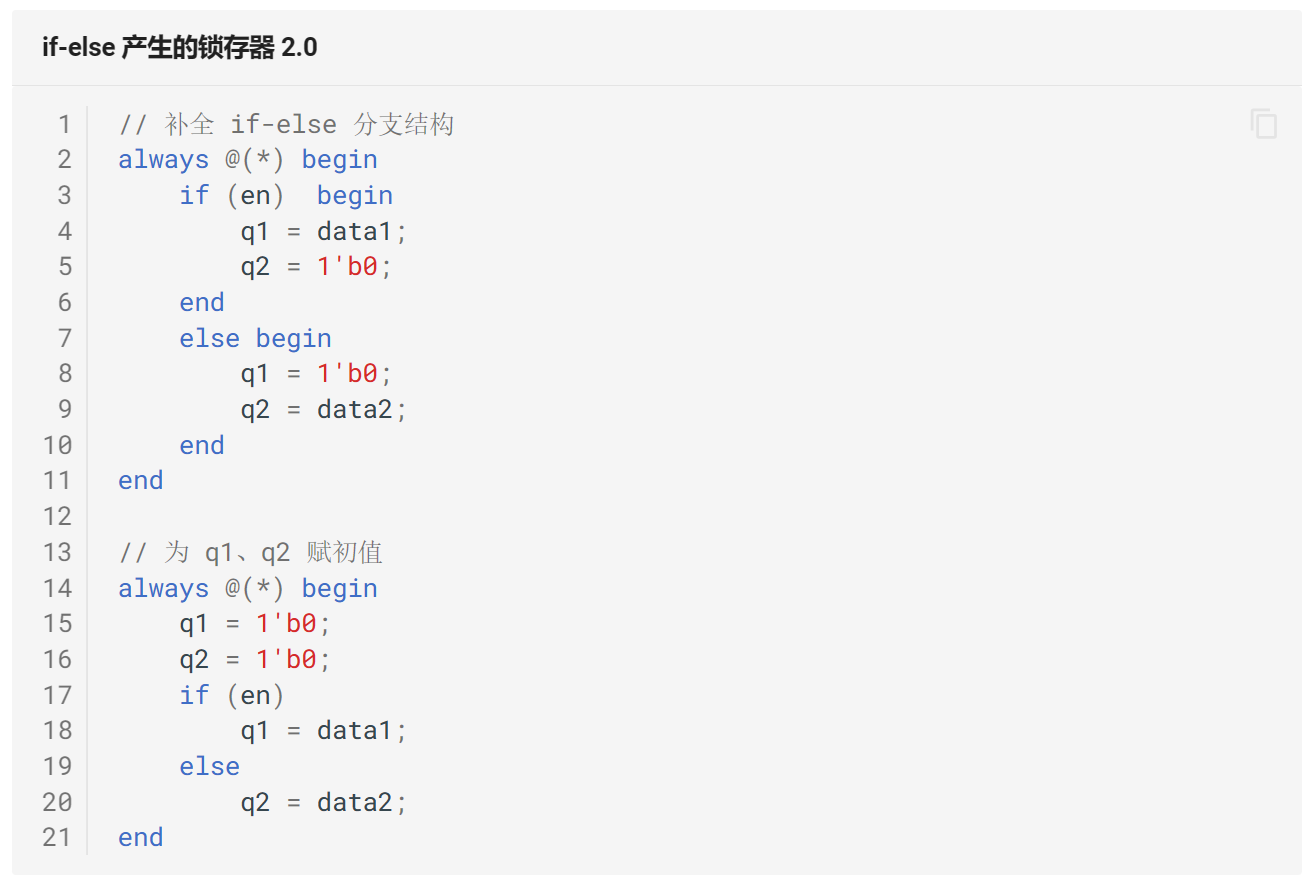


另外一种比较隐蔽的情况是：当条件语句中有很多条赋值语句时，每个分支条件下的逻辑不完整也是会产生锁存器的。例如：



这段代码看起来 if 和 else 都有了，但 if 部分里没有对 q2 赋值，else 部分里没有对 q1 赋值。从每个信号各自的的逻辑来看，这实际上也相当于是 if-else 结构不完整，相关信号缺少在其他条件下的赋值行为。

这种情况也可以通过补充完整赋值语句或赋初值来避免产生锁存器。例如：



### ****2.2 case逻辑缺陷****

case 语句产生锁存器的原理几乎和 if 语句一致。在组合逻辑中，当 case 选项列表不全且没有加 default 关键字，或有多个赋值语句不完整时，也会产生锁存器。例如：



这段代码会产生锁存器。因为在 sel 为 2'b10、2'b11 时，case 语句中并没有给出 q 的赋值结果，进而会被默认为保持原先的值不变。

同样地，消除此种锁存器的方法也是 2 种：将 case 选项列表补充完整，或对信号赋初值。补充完整 case 选项列表时，可以罗列所有的选项结果，也可以用 default 关键字来代替其他选项结果。

例如，上面的 always 语句有以下 3 种修改方式：



更特别地，当 if 和 case 组合起来时，我们往往就容易出现逻辑遗漏。例如下面这段 Verilog 代码：



这段代码中尽管有 endcase 语句，但依然会产生锁存器，因为 case 分支中的 if 逻辑不完整。一种比较好的策略是：在 always 语句块的一开始就进行默认赋值。这样可以避免潜在的逻辑不完整风险。



### ****2.3 自赋值与判断****

在组合逻辑中，如果一个信号的赋值源头有其信号本身，或者判断条件中有其信号本身的逻辑，也会产生锁存器。因为此时的信号也需要具有存储功能，能够获得先前时刻该信号的数值。此类问题在 if 语句、case 语句、问号表达式中都可能出现，例如：

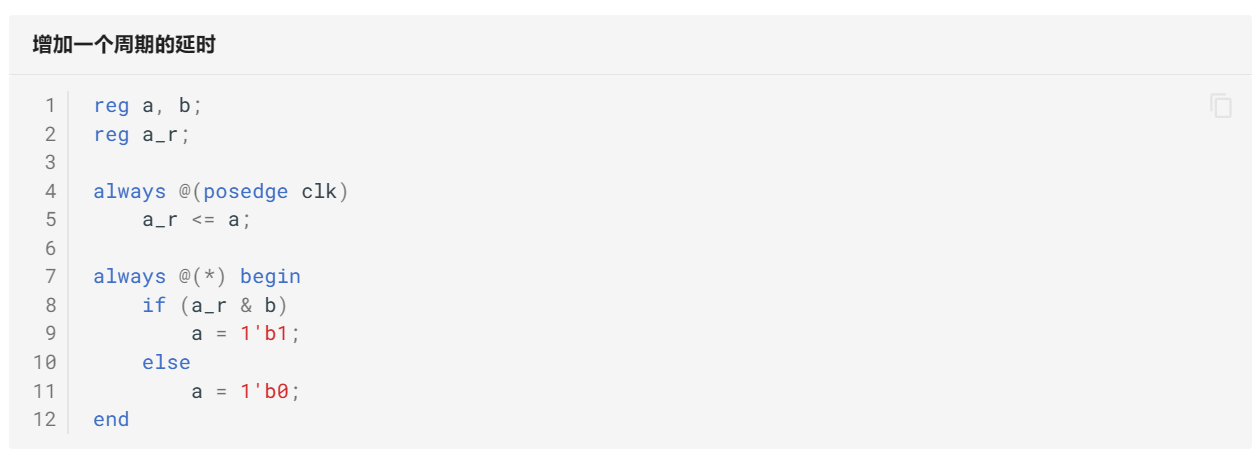


避免此类锁存器的方法只有一种，就是在组合逻辑中避免这种写法。时刻提醒自己：信号不要给信号自己赋值，且不要用赋值信号本身参与判断条件逻辑。

如果不要求下一时刻信号立刻输出，我们可以将信号进行一个时钟周期的延时后再接入组合逻辑。例如：



上面这段代码可以更改为：



这段代码不会生成锁存器，因为我们人为引入了 a\_r 作为寄存变量，用于存储变量 a 先前的值。

## 3 其他问题

### 3.1 多驱动

多驱动是指一个输出信号同时被多个输入信号驱动，这是一个非常严重的问题，因为这会导致电路的行为不确定。在编写代码时，以下几种情况会被认为存在多驱动情况：

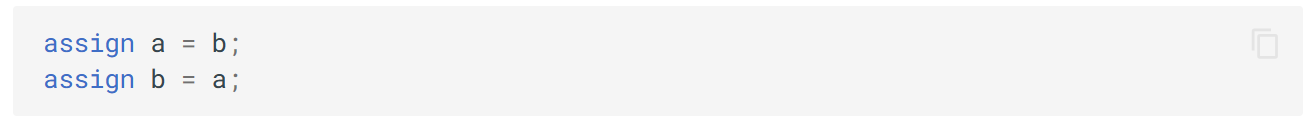
* 一个信号被多个 always 块赋值；
* 一个信号被多个 assign 语句赋值；
* 一个信号与多个模块的输出端口相连；
* 上述情况的混合。

例如，在下面这条报错信息里，d 这根 wire 型变量被作为了多个模块的输出，因此是错误的。

**[Synth 8-6859] multi-driven net on pin d\_OBUF[1] with 1st driver pin 'd\_OBUF[1]\_inst\_i\_1/O'**

### 3.2 组合环

组合环是指一个信号经过组合电路后又回到了自己，例如；



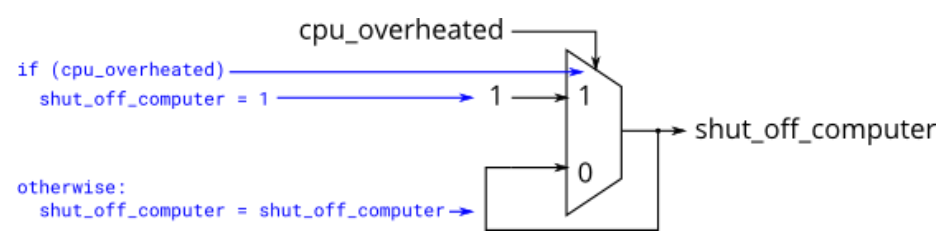
大多数程序的环路算法不是很好，组合环报错只能粗浅地具体到某个模块（甚至模块都定位不到，只报错「发现了组合环」），而由于模块之间的互联，这个环路可能会跨越多个模块，因此这个报错信息并不一定能够准确地定位到环路的位置。

想要解决组合环，首先应该对简单的电路部分进行检查，排除问题后再检查复杂的电路部分。一般情况下，组合环都是由于接线错误、输入接入输出等问题导致的，因此需要我们在编写代码时就注意这些问题。

# 思考与练习

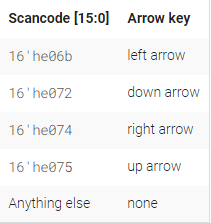
1. 有时候，语法正确的代码并不一定能产生功能正常的电路，一般来说都是因为不小心引入了锁存器造成的。例如下面的 Verilog 语句：



[](https://soc.ustc.edu.cn/Digital/lab2/figs/tasks/q1.png)

为消除锁存器，我们应当使组合逻辑过程块中的条件完备，即 if 语句后应加上 else 语句。试修改上述两段代码，以消除锁存器。

1. 假设您正在构建一个电路来处理游戏中 PS/2 键盘上的扫描码。现在给定接收到的扫描码的最后两个字节，您需要指示是否按下了键盘上的一个箭头键。这涉及到一个相当简单的映射，包含五种可能的情况：上、下、左、右，以及不属于任何一种。

[](https://soc.ustc.edu.cn/Digital/lab2/figs/tasks/q2.png)

请设计一个电路。该电路有一个 16bits 输入和四个 1bit 输出，该电路识别这四个扫描码并确认正确的输出。代码框架如下：

