**实验3**

**Verilog基础语法（下）**

# **实验介绍**

在上一次实验中，我们已经初步学习了 Verilog 硬件描述语言的基本语法。接下来，我们将站在更为系统的层面理解 Verilog 语言的特性。

# 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**

## 1 Verilog的三种描述层次

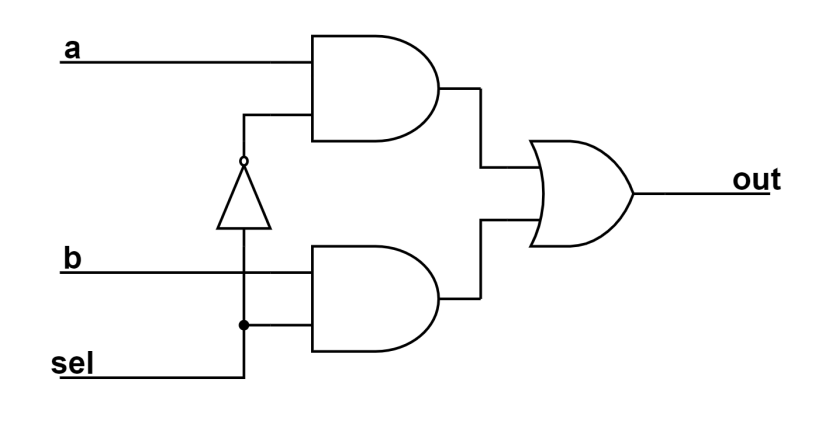
Verilog 可以使用三种不同的方式描述模块实现的逻辑功能。它们分别是：

* 结构化描述方式：调用其他已经定义过的低层次模块对整个电路的功能进行描述，或者直接调用 Verilog 内部预先定义的基本门级元件描述电路的结构进行描述。
* 数据流描述方式：使用连续赋值语句 assign 对电路的逻辑功能进行描述。该方式特别适合于对组合逻辑电路建模。
* 行为级描述方式：使用过程块语句结构 always 和比较抽象的高级程序语句对电路的逻辑功能进行描述。

我们用一个例子展示这三者的区别。

### ****1.1 结构化描述方式****

考虑下图所示的电路：



如果从结构化层面来描述电路，我们需要刻画与门、或门和非门，并将其正确连接。Verilog 常用的内置逻辑门包括：

* and（与门）
* nand（与非门）
* or（或门）
* nor（或非门）
* xor（异或门）
* xnor（同或门）

我们可以通过类似模块例化的方式使用这些逻辑门，进而实现一些简单的逻辑功能。

由于不知道输入信号的数目，门级单元无法采用基于名字的端口关联方式，只能基于位置进行端口关联。一般来说，门级单元的第一个端口是输出，后面其余的端口是输入。在例化调用的时候，我们也可以不指定门级单元实例的名字，从而为代码编写提供了方便（毕竟逻辑门的数目可能很多）。当输入端口超过 2 个时，只需要将输入信号在端口列表中继续排列即可，Verilog 可以自动识别输入和输出信号。

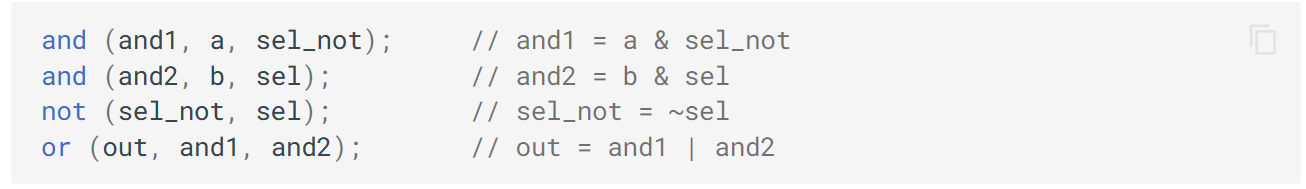
下面是使用门级单元结构化描述该电路的 Verilog 代码。



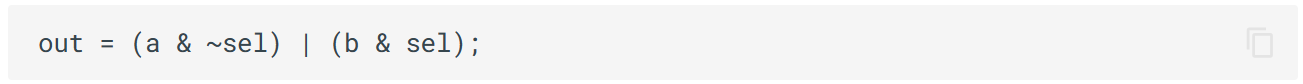
你可以将其与我们的电路图进行对应，理解每一个信号与实际连线的对应关系。

### ****1.2 数据流描述方式****

数据流描述方式需要我们得到逻辑表达式。我们可以将门电路转换为对应的逻辑表达式：



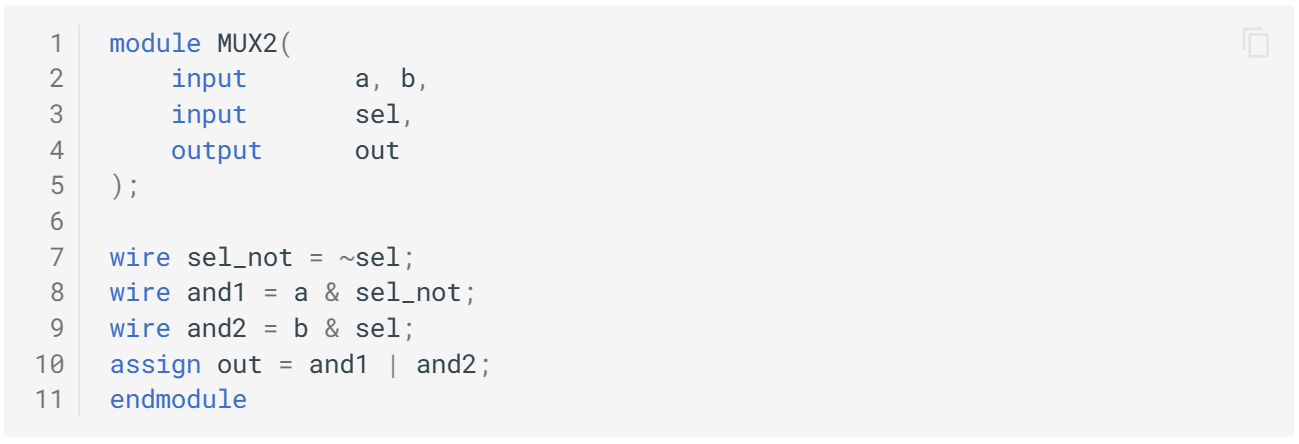
化简后，我们就得到了输出 out 关于输入 a、b 和 sel 的逻辑表达式：



由此可以得到基于 assign 语句的数据流描述。



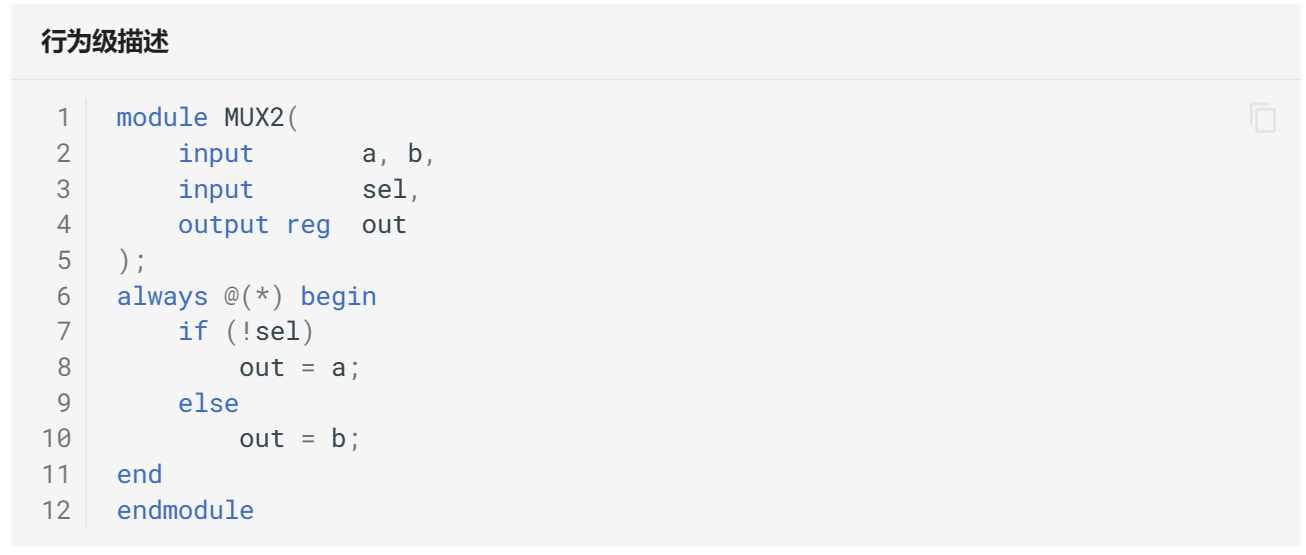
当然，你也可以写成下面的样子：



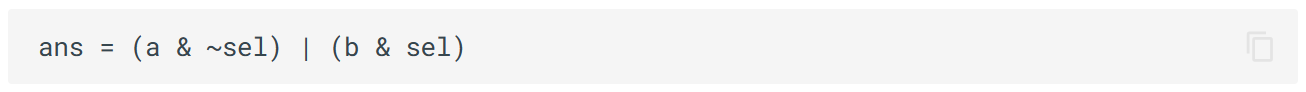
### ****1.3 行为级描述方式****

很多时候，我们难以得到模块的电路结构，或者得到的结构十分繁琐，这时我们就可以使用行为级描述，以类似于高级语言的抽象层次进行硬件结构开发。这一层面的描述过程更看重功能需求与算法实现，也是对于我们最为友好的描述方式。

下面是该电路的行为级描述代码。



我们可以简单分析一下。在 always 语句内部，当输入信号 sel 为 0 时，out 输出 a 的内容；当 sel 为 1 时，out 输出 b 的内容。其对应的逻辑表达式为：



这正是我们想要的结果。

在实际的硬件开发过程中，我们更多采用的是将三种描述方式结合起来，根据需要选择相应的描述方式，从而实现自己的预期设计。

## 2 两种特殊信号

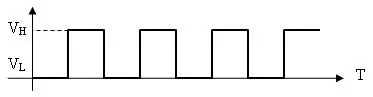
在数字电路设计中，有两种信号是十分重要的：时钟信号和复位信号。下面我们将对这两种信号进行介绍。

### ****2.1 时钟信号****

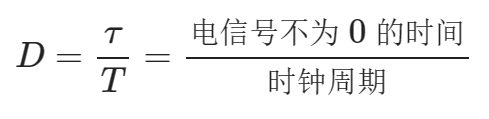
#### 2.1.1 基本概念

在之前的教程中，我们已经见过了一个特殊的信号 clk，这就是时钟信号（Clock）。时钟信号是数字电路中时序逻辑的基础，用于决定逻辑单元中的状态何时更新，是有着固定周期并与模块运行无关的信号量。

硬件电路中的时钟信号是由时钟发生器产生的。它只有两个电平，一个是低电平，另一个是高电平。高电平可以根据电路的要求而不同，例如理想情况下 TTL 标准的高电平是 5V，而低电平一般默认为 0V。时钟信号有固定的翻转频率（周期），以恒定的速度进行着高电平和低电平之间的转换。

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/clock.png)

除了周期 T、频率 f 之外，另一个重要的时钟属性是占空比（Duty Ratio）。对于周期恒定的时钟信号，占空比的定义为周期电信号中，有电信号输出的时间与整个信号周期之比。也就是：



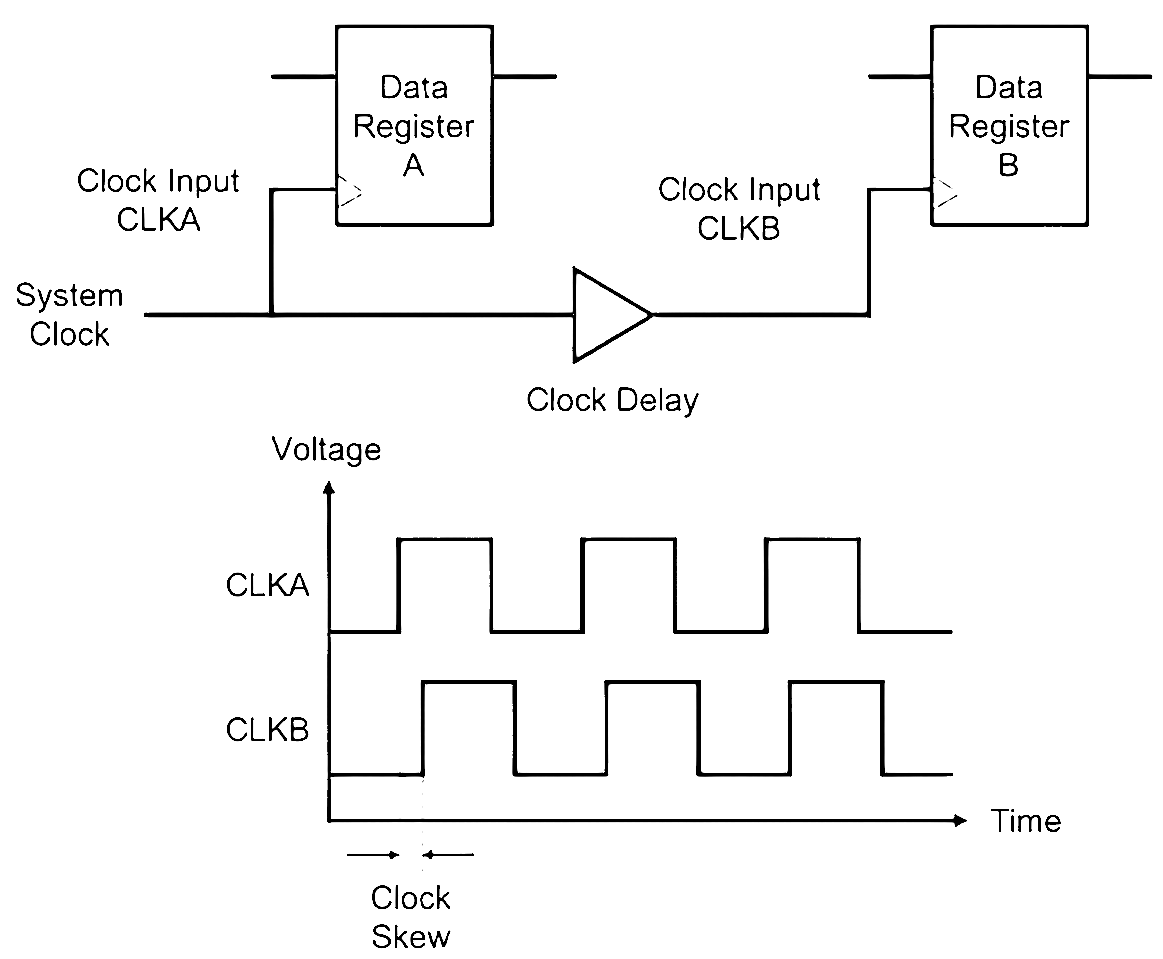
最常见的时钟信号占空比为 50% ，也就是说高电平和低电平的持续时间是一样的。

数字系统使用时钟信号的上升沿、下降沿或者双边沿作为同步驱动的参考，进而实现不同模块的同步运作，确保了整个系统的协调性与正确性。

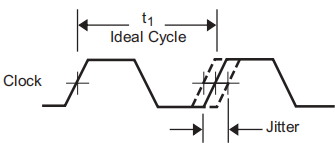
上面的讨论中，所有的时钟信号都是理想的：时钟的翻转是在瞬间完成的，模块之间的时钟沿都是对齐的，没有延迟，没有抖动。但在实际电路中，时钟在传输、翻转时都会有延迟。一个较好的数字设计也应该考虑这些不完美的时钟特性，否则会造成潜在的设计时序不满足的状况。

下面是一些常见的时钟特性：

* 时钟偏移（Skew）：由于线网的延迟，同一时钟的信号即使同时发出，也不能保证不同模块接收到的边沿是对齐的，即不同模块端口的时钟相位存在差异。这种差异称为时钟偏移。

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/clock_skew.png)

* 转换时间（Transition）：时钟从上升沿跳变到下降沿，或者从下降沿跳变到上升沿时，并不是"直上直下"、不需要时间的，而是以一种"斜坡式"的方式，需要一个过渡时间才能完成电平跳变。这个过渡时间称之为时钟的转换时间，
* 时钟抖动（Jitter）：相对于理想时钟沿，实际时钟中不随时间积累的、时而超前、时而滞后的偏移称为时钟抖动。时钟抖动可分为随机抖动和固定抖动。其中，随机抖动的来源为热噪声、半导体工艺等；固定抖动的来源为开关电源、电磁干扰或其他不合理的布局布线等。

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/clock_jitter.png)

#### 2.1.2 本地时钟

在上板调试与运行时，我们使用板载芯片的时钟作为时钟驱动。但在本地测试的时候，我们就需要自己编程生成符合预期的时钟信号。下面简单介绍一些基本的 Verilog 时钟编写方式。

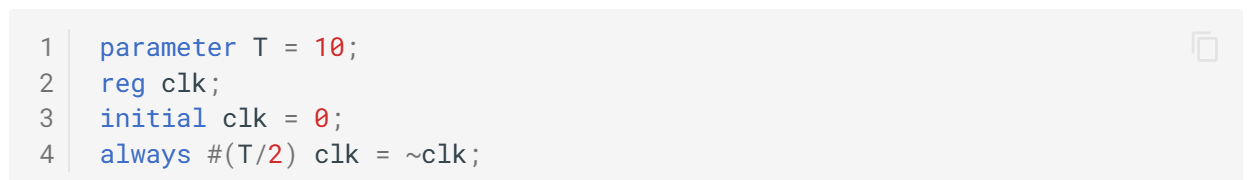
**基于 initial**

我们先前提到过，initial 仅在 0 时刻开始执行一次内部的语句。而时钟信号是一个长期翻转的信号，因此我们需要在 initial 内部添加一个死循环，用于生成周期性的信号。下面的代码使用 forever 关键字声明了一个周期为 T 的时钟。你可以简单地认为：forever 类似于 C 语言的 while(1)，而 # value 类似于 C 语言的 sleep(value)。

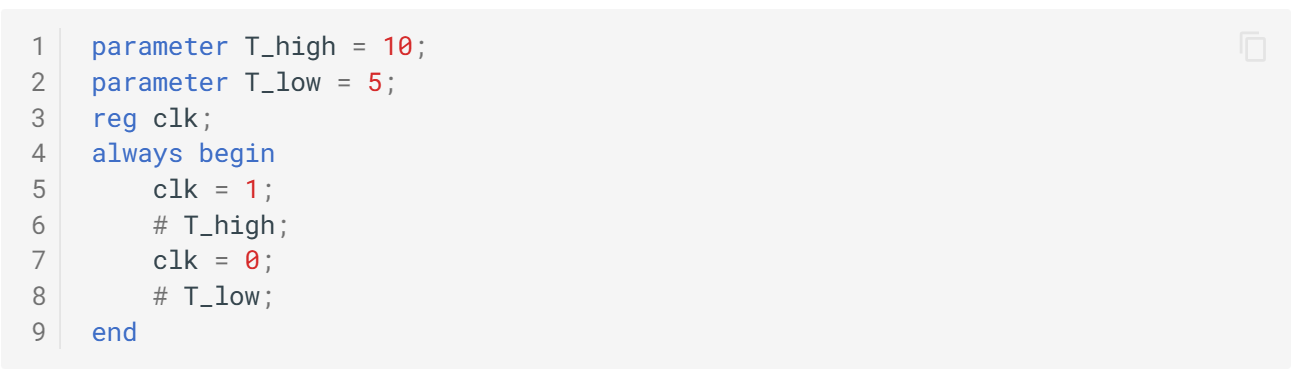


**基于 always**

与 initial 不同，always 语句将持续执行内部的语句，因此就不需要额外的死循环了。我们可以使用 initial 语句对 clk 变量初始化，再使用 always 语句实现永久的周期变化。



如果时钟的占空比并不是 50%，则可以参考如下代码编写：

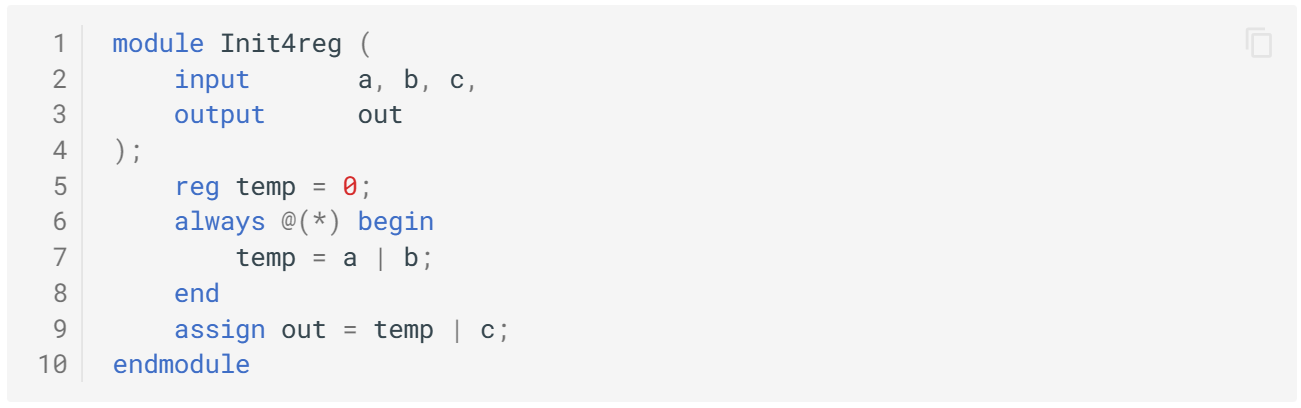


### ****2.2 复位信号****

我们知道，C 语言的指针（Pointer）需要进行初始化，否则访问时可能发生意想不到的错误。Verilog 的信号变量也是同理。wire 型变量的默认赋值是 Z，reg 型变量的默认赋值是 X。那么，我们应当如何为这些变量初始化呢？

首先，wire 类型的变量是不需要初始化的，因为其只要与电路相连，就一定有确定的输出（这是组合逻辑电路的特征）。我们只需要考虑 reg 类型的变量了。需要强调的一点是：我们不建议在 reg 类型的变量声明时对其进行初始化操作。尽管在本地测试的时候这样可以成功初始化，但在上板实际验证时，信号初始值依然是不确定的，从而成为潜在的错误（因为这种方式在语法上和逻辑上都没有问题）。

例如：下面这段代码能通过语法检查，本地测试也是正确的，但上板时依然可能运行异常：



那么，我们应该怎样进行初始化操作呢？这就需要引入复位信号了。

为确保实际环境下数字系统在上电后有一个明确、稳定的初始状态，且系统在运行紊乱时可以恢复到正常的初始状态，我们会在模块设计中添加复位模块。复位电路保证了系统工作的可控性，在一定程度上其重要性不亚于时钟信号。

从时序上来看，复位电路可分为同步复位（Synchronous Reset）和异步复位（Asynchronous Reset）两种。

#### 2.2.1 同步复位

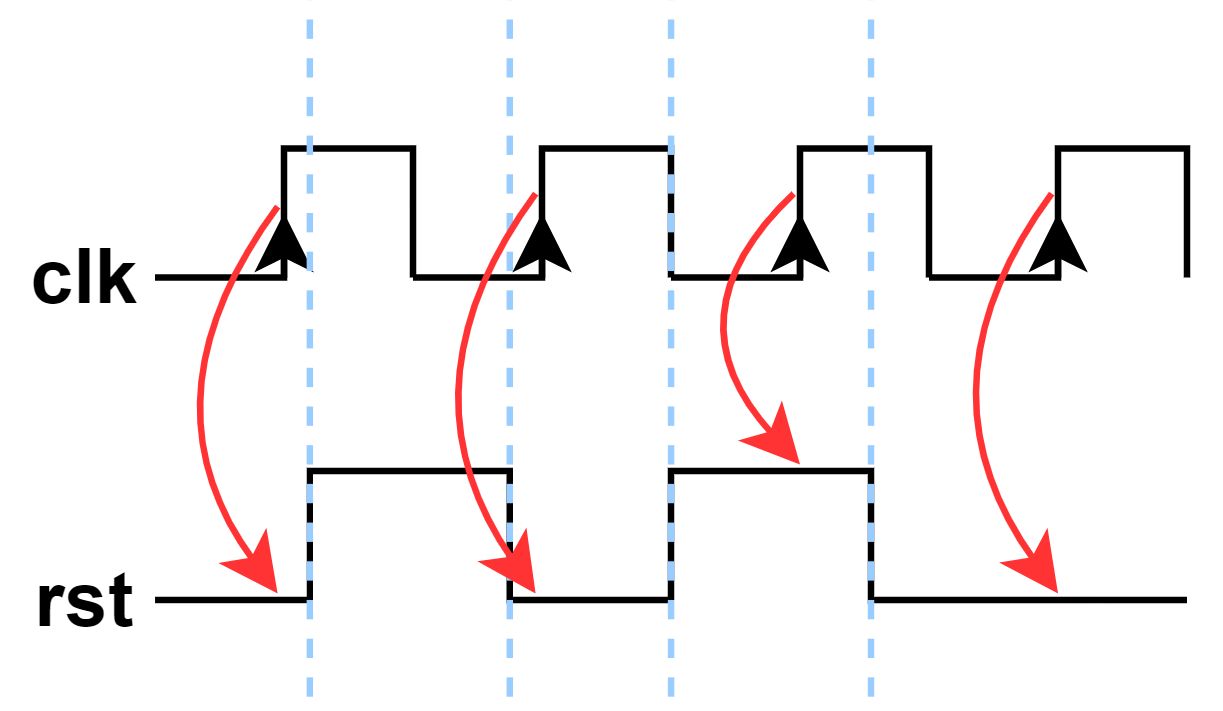
同步复位是指与时钟信号同步的复位信号。也就是说，复位信号仅在时钟的边沿到来时才有效。如果没有时钟边沿，无论复位信号如何变化，电路也不会进行复位操作。

一个同步复位的 Verilog 描述如下：



可以看到，上面这段代码仅在时钟信号 clk 的上升沿到来时才执行 always 内部的语句，即判断复位信号 rst 是否有效。其他时刻不管 rst 如何变化，内部信号 dout 都不会被复位。

同步复位保证了信号是与时钟同步变化的，有利于保证时序的稳定性，因此被广泛使用。但大多数时序逻辑单元并没有同步复位端，使用同步复位描述得到的电路往往会消耗更多的逻辑资源。此外，复位信号的宽度必须大于一个时钟周期，否则便有可能发生遗漏。

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/rst_miss.png)

如上图所示，复位信号 rst 是高电平有效。可以看到，前两个时钟上升沿到来时，rst 均为低电平，因此不会发生复位。在第三次时钟上升沿到来时，rst 为高电平，系统才能触发同步复位。

#### 2.2.2 异步复位

异步复位是指无论时钟到来与否，只要复位信号有效就会执行复位操作的信号。一个异步复位的 Verilog 描述如下：



上面这段代码中，always 语句在 clk 和 rst 信号的上升沿到来时均会执行。当复位信号 rst 由低电平变为高电平时，对应的上升沿便会触发 always 执行内部语句，进而执行复位操作。

目前大多数时序逻辑单元都提供了异步复位信号的接口（RST 或 CLR），因此异步复位描述不会占用额外的逻辑资源，设计上相对简单。但是，异步复位会导致复位信号与时钟信号之间没有明确的时序关系，并且复位信号容易受到外部因素的干扰，产生意想不到的复位操作。

#### 2.2.3 其他讨论

复位电路会为数字系统带来更多的硬件逻辑和资源消耗，增加系统设计的复杂度。所以，在一些初始状态不影响逻辑正确性的数字设计中（例如数据处理、高速流水线中的一些寄存器等）可以考虑去掉复位信号，以达到最佳性能。

如果某个模块确实需要复位操作，Xilinx 的建议是：使用同步复位。这是因为FPGA 芯片在通电后，会执行下面的一系列操作：

1. 触发一个复位事件，进入全局复位过程；
2. 复位过程中，控制逻辑检测所有器件的供电电压。如果在规定时间电压达到规定值并稳定下来，则进入后续的配置环节，否则就需要等待足够时间让电压达标；
3. 复位顺利完成后进入配置模式，随后按照用户意图将各个寄存器初始化为预期值（一般为 0）；
4. 初始化完成后进入用户模式，此时系统按照既定时序开始运转。

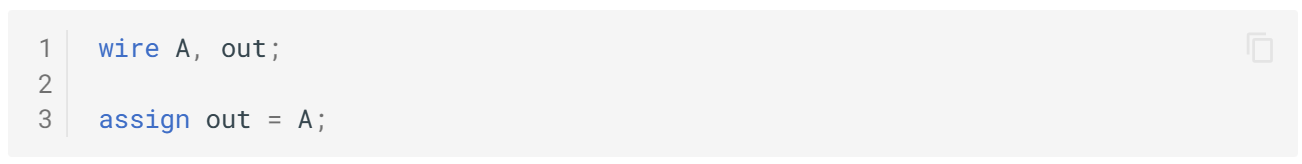
因此，每次工作开始前，FPGA 必定会进行复位、初始化等操作。所以理论上我们的变量是不需要复位的（默认赋值成 0）。但是，一方面我们需要保证寄存器中的值是我们期望的内容，另一方面也要避免可能出现的异常状态导致后续系统工作异常，因此数字系统中的复位设计是必不可少的。

## 3 Logisim与Verilog

最后，我们使用 Logisim 帮助大家更好地理解 Verilog 的设计与应用。我们反复强调了：Verilog 编程语言得到的是电路，而不是顺序执行的代码。因此，每一段代码都有着与之对应的电路结构。理解这种对应关系有助于我们更好地进行 Verilog 编程开发。

### ****3.1 wire型变量与逻辑门****

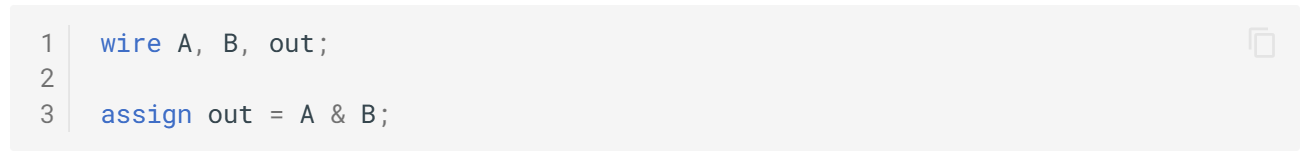
wire 型变量对应着普通的导线。例如：



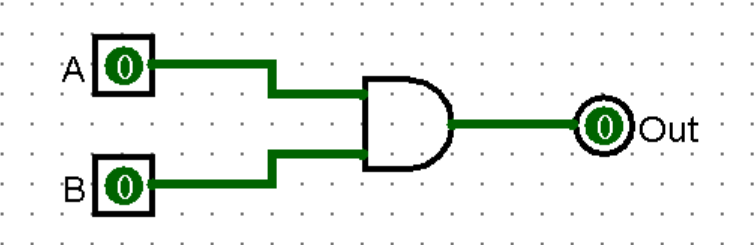
上面这段代码对应的电路结构如下：

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/wire_in_log.png)

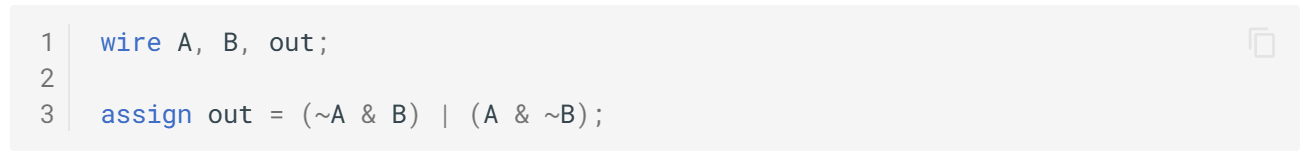
除了直接赋值，assign 里面也可以使用逻辑表达式。例如：



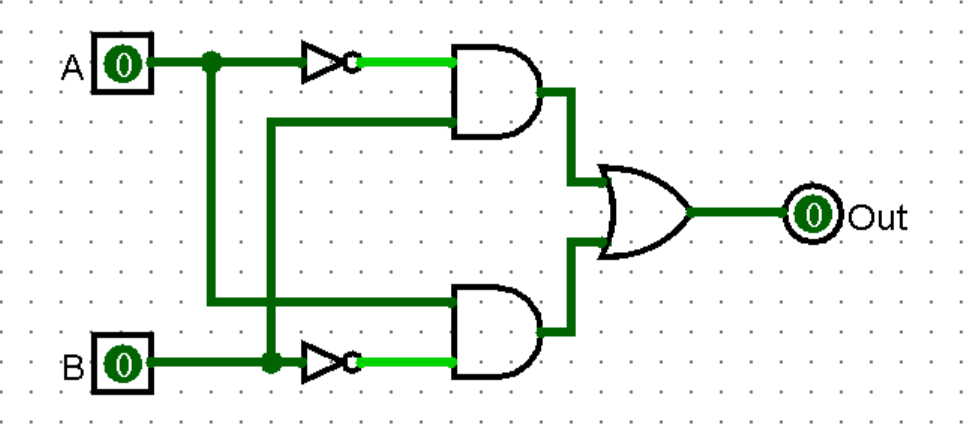
上面这段代码对应的电路结构如下：

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/and_in_log.png)

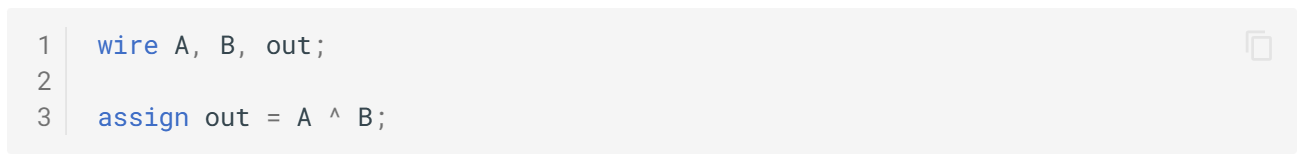
我们也可以使用一些更复杂的表达式。例如：



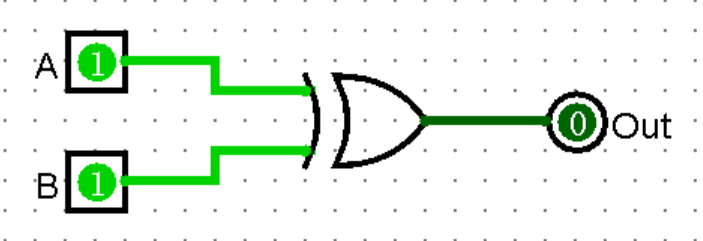
上面这段代码对应的电路结构如下：

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/xor_in_log_1.png)

当然，这实际上是异或门的逻辑表达式，因此也可以直接使用下面的 Verilog 代码描述：

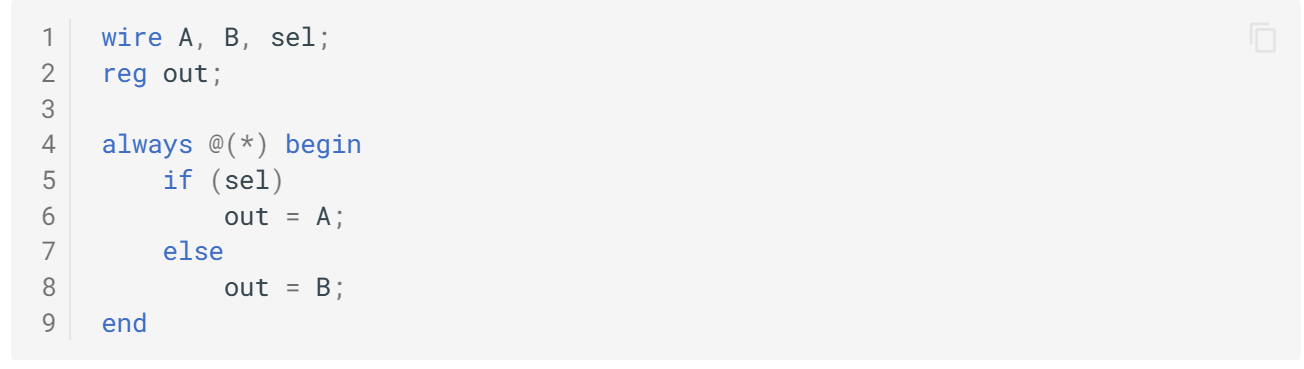


Logisim 中也有相应的异或门结构。

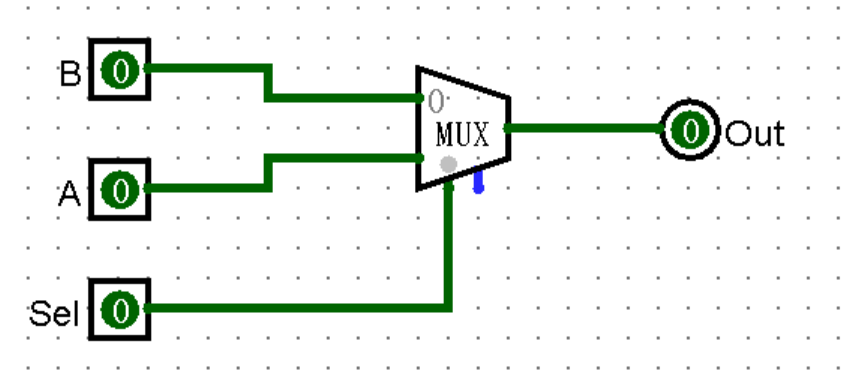
[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/xor_in_log_2.png)

### ****3.2 if语句****

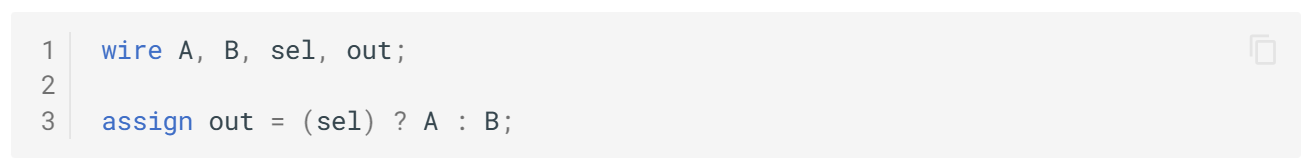
从 C 语言的逻辑上来看，if 实现了分支逻辑，即根据条件选择某一分支继续执行。在硬件电路中，实现选择功能的电路被称为选择器，其一般用于从多个输入中选择一个进行输出。



上面这段 Verilog 代码对应的电路图如下：

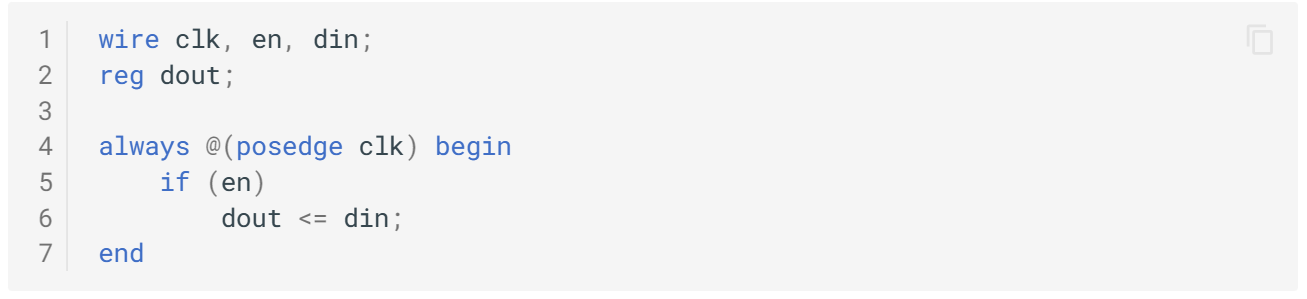
[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/mux_in_log.png)

选择器在图中以一个梯形结构作为示意。当然，我们也可以使用条件运算符实现相同的功能。即：



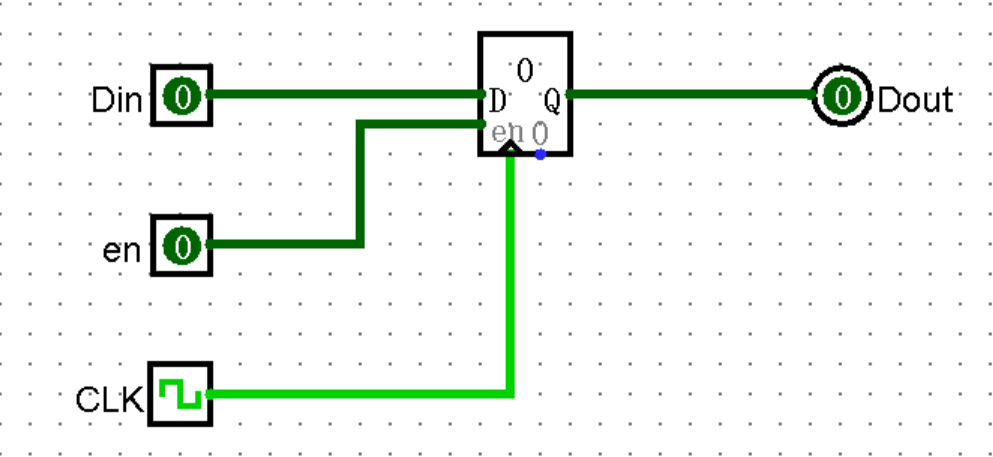
### ****3.3 寄存器****

以 always @(posedge clk) 为代表的语句对应的是寄存器结构，该结构可以保存电路中的信息，实现存储功能。



我们知道，在上面这段代码中，posedge 关键字决定了 always 语句仅在 clk 信号的上升沿（从 0 变为 1 的瞬间）执行。在上升沿到来时，首先判断 en 信号，如果为 1 则将 din 的值赋值给 dout。如果 en 信号为 0，代码中没有给出描述，表明为 0 时只需要维持原状即可，这也是寄存器「存储」功能的体现。

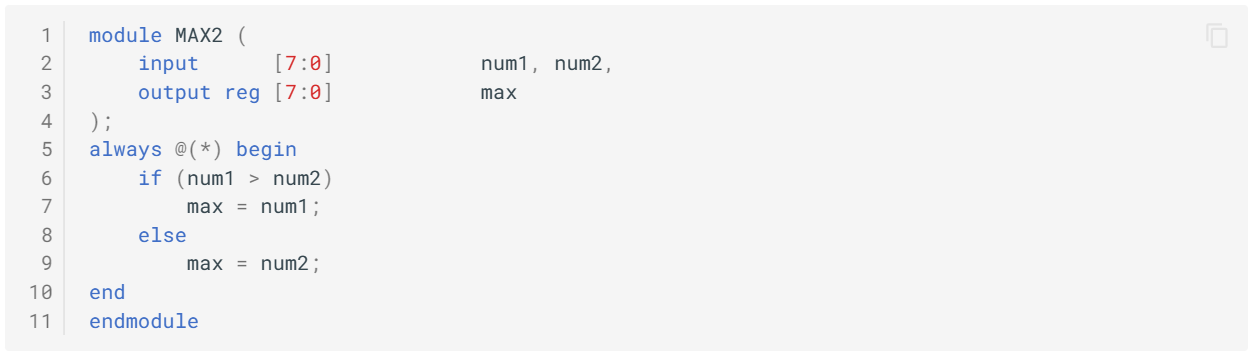
上面这段代码对应的电路结构如下：

[](https://soc.ustc.edu.cn/Digital/lab1/figs/verilog2/reg_in_log.png)

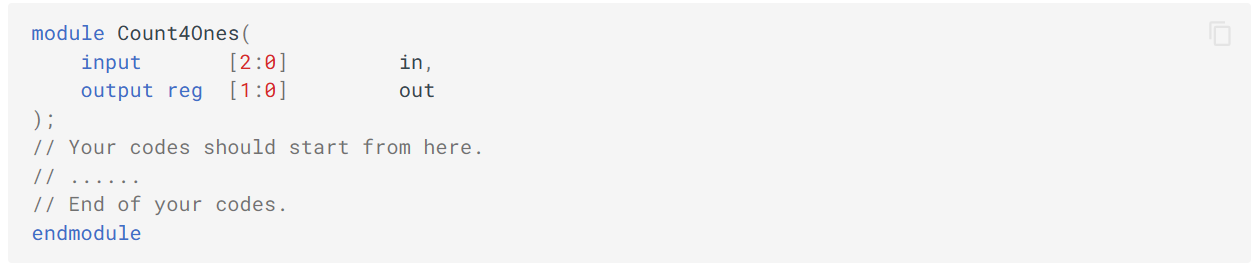
寄存器以一个矩形结构作为示意。我们将在后续的实验中介绍其详细结构。

# 思考与练习

1. 下面的 Verilog 代码片段实现了两个数取最大值的功能：

请据此完成下面的问题。

1. 请使用 assign 语句重新完成该模块的功能。
2. 现在我们需要获得三个数的最大值。请在下面的框架中通过例化 MAX2 模块实现该功能。
3. 给定一个位宽为 3 的信号 in，编写 Verilog 代码以输出其中为 1 的位的数目。例如：如果 in = 3'b011，则模块输出 2'd2；如果 in = 3'b111，则模块输出 2'd3。我们已经为你准备了可用的框架代码：



1. Verilog 支持对于数组的声明。请自行查阅有关资料，编写Verilog代码，声明一个由16个位宽为32的reg型变量组成的数组。