Episode 5 Timing！Let‘s move on！

——《你可能不知道的verilog提高班指南》By TA-马子睿

前言

大家好呀！经历了编码器和译码器两类组合逻辑的洗礼，你是否已经对理论课上那个永不停息的时钟充满了好奇了呢？不用着急，本辑将带你深入剖析verilog对时序的描述与操作，让你更轻松地玩好时序电路！

目录

"<="还是"="？我偷偷换用一下应该没人发现吧！

错误的！当然有人会发现！（这里不点名助教了）

在理论课上，我们给了这两个符号两个很正经的定义：

- **<=：非阻塞赋值**
- **=：阻塞赋值**

并且老师还告诉我们：**组合电路块要用"="，时序电路块要用"<="。**这当然是万年不变的真理，但很多同学都有疑问：为什么要这样做？

下面，我将通过一段最简单的代码，帮助大家理解并记忆这两个赋值：

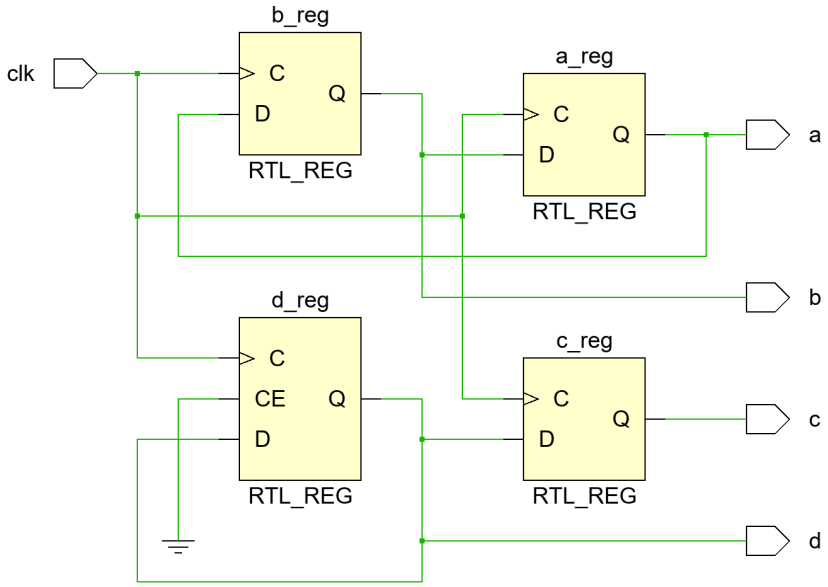
```
/* Timing */
reg a, b;
initial begin
    a = 1'b1;
    b = 1'b0;
end
always @(posedge clk) begin
    a <= b;
    b <= a;
end

/* Combination */
reg c, d;
initial begin
```

```
    c = 1'b1;
    d = 1'b0;
end
always @(posedge clk) begin
    c = d;
    d = c;
end
```

这两个电路块都试图在时钟上升沿到来时，交换两个寄存器的值。可是，只有使用了非阻塞赋值才可以实现不停交换，使用阻塞赋值的电路块最后两个寄存器都将变为0！

“阻塞赋值”顾名思义，你可以把它理解为，它真的会**从上到下一条一条执行**（即先阻塞住靠后的赋值语句）。我们打开这一个模块的电路可以看到：



如果不仔细看，确实很难看出ab和cd之间的不同。我们可以看到，**d寄存器的CE（clock enable）接口始终置零**，也就代表时钟一直无效。仔细思考这一段代码也可以看出，d的输入就是d的输出，d的值确实一直都没有发生变化，因此Vivado帮我们自动化简了电路。

另外，还记得我们在第二辑中讲过的**默认赋值**吗？

```
always @(*) begin
    b = 0;
    if(a[0])        b = din1;
    if(a[1])        b = din2;
end
```

你有没有发现，这类默认赋值只有在always@(*)，即组合电路中，才会出现呢？

我们可以把它理解为阻塞赋值的特性：每次赋值会将b变为0，之后如果满足下面几个条件，再将其赋值为对应的值。当然，这是一种软件的思维，**事实上对于默认赋值，Vivado会采用多选器进行实现。**

那么，为什么在时序电路中需要使用非阻塞赋值呢？这是由时序电路的性质决定的：**当且仅当时钟上升沿到来时，所有同一优先级的赋值必须同时进行，而不应进行等待**，这必须使用非阻塞赋值才能实现。而对于组合电路，**“赋值”事实上就是一根电线，赋值无时无刻不在进行**，这时，如果出现相关（即某一模块输出是另一模块输入），那么这时赋值会出现先后，使用阻塞赋值是很重要的。

看过这一段讲解，希望大家不仅能正确使用两种赋值**（即时序电路用非阻塞赋值，组合电路用阻塞赋值）**，更能理解这两种赋值的内在逻辑。有了这些理解，更能深刻地理解时序和组合的关系：

> 我们的思路是时序的，永远要跟着那一个个时钟上升沿走。上升沿的间距里，是组合电路工作的时间。

p.s. 这里说的所有赋值都是不带assign的，如果是assign后边的等号，那么大家直接把他理解为“连线”就好啦！

rstn？为什么是'n'？

在理论课上，大家知道有些触发器是带复位功能的，接口一般叫做reset或者rst。但在开发板上，我们使用的是rstn——即对rst信号进行取反。大家看到开发板右侧的CPU RESET按钮，这个按钮非常独特，**在被按下时它输出0，在不按下时它输出1！（一定要记住这一点，很多人被它坑过若干次）**，这也正应了它名字中的"n"（not）。

在一般的时序电路设计中，clk和rstn两个信号通常是一个时序模块的头两个信号：

```
module module_name(
    input clk,
    input rstn,
    ......
);
```

这里的rstn，一定对应了开发板上的CPU RESET按键，不要把它接给其他的模块（比如另一个模块的输出线直接接到rstn上），企图用其他模块来调控这个模块的rstn信号，一定不要这么做！这是非常违背设计规范性的行为！如果确实需要用其他模块的输出来对另一个模块进行复位，请额外给出接口（比如clear）来实现：

```
module module_name(
    input clk,
    input rstn,
    input clear,
    ......
);
```

对于所有时序电路，物理复位的优先级一般是最高的，如下所示：

```
always @(posedge clk) begin
    if(!rstn) dout <= 0;
    ......
end
```

最后，**请不要把rstn这个信号接入组合电路块**，虽然也可以实现对应功能，但那着实是没有意义的，还会白白增加电路的组合延时。

同步复位与异步复位：步调一致得胜利

大家在理论课上可能已经学习了同步复位与异步复位，那么它们在verilog里面是如何实现的呢？

同步复位

同步复位，顾名思义，就是**和时钟信号看齐的复位。**时钟上升沿不来，任你复位信号保持多久，都没有用！

换句话说，同步复位信号只有在时钟上升沿到来时才发挥作用！

再来回顾上面的例子：

```
always @(posedge clk) begin
    if(!rstn) dout <= 0;
    else ...
end
```

这是一个相当典型的同步复位模块。可以看到，这个模块只会响应时钟上升沿，**如果时钟上升沿没有到来，rstn信号置得再高也没有用！**同步复位是我们最常用的复位，因为它更符合时序，可以**保证不会在组合电路计算中途打断计算直接复位**，而且在实际应用过程中，由于时钟频率很高，也可以在人类的观感上，基本做到rstn来到时就可以复位。

异步复位

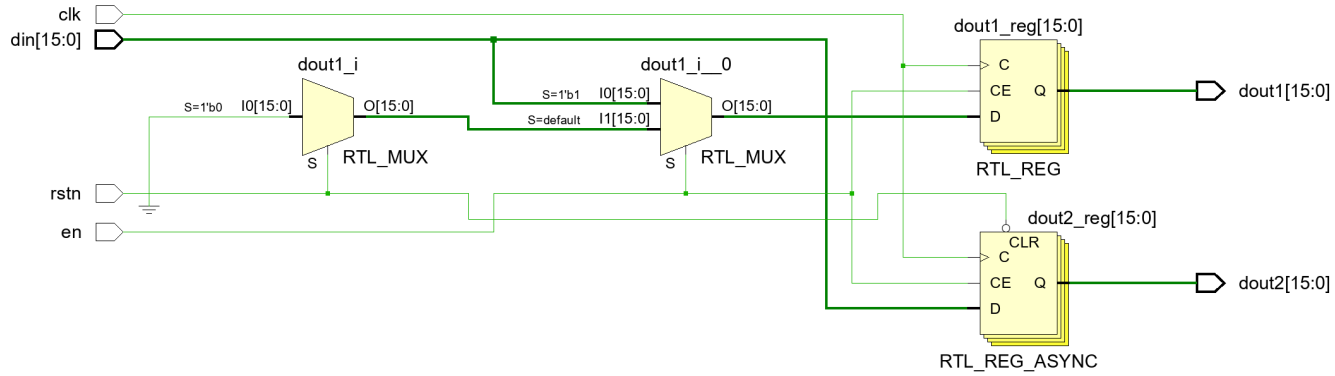
在异步复位中，只要按下rstn，就会立刻执行复位，因此这个模块需要响应的信号有两个：**clk上升沿和rstn下降沿**

```
always @(posedge clk or negedge rstn) begin
    if(!rstn) dout <= 0;
    else ...
end
```

然而，异步复位很容易在书写时出错，我们再来看一下下面的模块：

```
module test(
    input clk,
    input rstn,
    input en,
    input [15:0] din,
    output reg [15:0] dout1,
    output reg [15:0] dout2
    );
    always @(posedge clk or negedge rstn) begin
        if(en) dout1 <= din;
        else if(!rstn) dout1 <= 0;
    end
    always @(posedge clk or negedge rstn) begin
        if(!rstn) dout2 <= 0;
        else if(en) dout2 <= din;
    end
endmodule
```

我们看到，这个模块好似描述了两个异步复位的寄存器，但真的是这样吗？我们来看看电路图：



大家注意看右边的元件，REG好像不太一样呀！上面的REG是前一个always块描述的，它并不像下面的REG一样，有CLR（CLEAR的缩写）接口，这证明，它纯纯是一个**同步复位寄存器！**为什么会这样呢？

主要原因还是我们之前讲过的if和else if的绝对优先性的问题。在第一个模块里，rstn比en的优先级低，而这显然不符合异步复位的要求。因此，编译器认为这应当是一个同步复位。因此，**如果要描述异步复位，那么在第一个if条件判断中一定要是rstn，否则编译器不会认为这是一个异步复位！**

initial块在仿真中的使用

在时序电路里，我们要使用大量的寄存器，而这些寄存器往往需要在上电前赋予初值。如果你需要这么做的话，initial块将会是你不二的选择。事实上，initial被大家误会良久，大家之前普遍认为initial中的赋值在上板时是没有用的，但事实证明，如果reg型变量在电路中真的充当了存储器或寄存器，那么它上板后的初始赋值是有效的，而且initial赋初值的操作可以防止仿真时最开始出现X的情况。

不过，initial的赋值我们只建议大家在仿真文件中使用，可综合的电路如果需要initial，可以在寄存器中if(!rstn)中将这些信号置为initial中的值。上板那一刻，其实电路会自行rstn一次的！（这一点并没有

我们来看一段代码：

```
integer i;
initial begin
    for(i = 0; i < 64; i = i + 1)begin
       #10 rank[i] = 8'b00011011;
    end
end
```

大家看到了initial块的用法，**其实是非常简单的，对你需要的reg型变量直接赋值就行了**

……等等！

initial里面这是什么东西！for……循环？

没错，initial块实在是简单，大家只需要知道有这么个东西，需要用的时候直接用就行了！

那么下面，就让我们进入大家期待已久的：

for循环？你真的是for循环？

看到这个标题，大家应该有预感：verilog里面的for循环，和C语言里面的for循环完全不同。在verilog硬件描述里，是没有“循环”这个概念的。因此，我们有以下的箴言：

> **展开for循环，把每条“语句”都并行起来，就是你描述的电路**

一般情况下，我们会把for循环分为三种：always中的for循环、在外层的for循环。

initial中的for循环

这里的for循环是大家喜闻乐见的，因为它完全可以利用C语言的理解去使用for循环。原因是，initial中的所有东西都在上电之前完成，所以并不并行没那么重要。我们再来回顾一下上面的例子：

```
integer i;
initial begin
    for(i = 0; i < 64; i = i + 1)begin
        #10 rank[i] = 8'b00011011;
    end
end
```

这里面的rank是一个长度为8的寄存器组，我们通过循环赋值的方式对其中的值进行初始化。这里有几点需要注意：

- 循环变量i要声明为integer，不要声明为genvar
- 没有i++这种操作，体谅一下verilog编译器吧！

always中的for循环

在always中的for循环使用格式与initial中大体相似，但是实际的实现可谓大相径庭。

在always中，for循环就必须考虑到并行了。我们必须要保证，**for循环中的所有语句展开后，仍然构造了正确的电路。**来看一个例子：

```
integer j;
always @(posedge clk) begin
    if(clear_mem != 0) begin
        case(clear_mem)
        CLEAR_ALL: begin
            for(j = 0; j < TLBNUM; j = j + 1) begin
                tlb_e[j] <= 0;
            end
        end
        ...
        endcase
    end
end
```

看到这里大家可能有体会了：完全可以写TLBNUM行语句，对tlb_e[0]、对tlb_e[1]、……、对tlb_e[TLBNUM-1]、分别赋值为0，for循环只是提供给了我们一种简少代码行数的写法，如果足够勤奋，你一定可以替代for循环，因为**for循环终结判断语句一定是迭代变量和一个常量或宏定义来比较。**同时大家仍然要注意，**always里面使用的for循环的迭代变量也要是integer！**

在外层的for循环

外层，也就是不在always和initial中。for循环在这里的**基本性质与在always中无异**，只是，**迭代变量必须声明为genvar**！

一般我们会利用这个语法来进行assign或者例化，如下：

```
genvar i;
for(i = 0; i < TLBNUM; i = i + 1) begin
    assign found0[i] = all_e[i] &&
          (all_g[i] || (all_asid[10*i+9:10*i] == s0_asid)) &&
          ((all_ps[6*i+5:6*i] == 6'd12) ? (all_vpn2[19*i+18:19*i] == s0_vpn2) :
                                          (all_vpn2[19*i+18:19*i+9] == s0_vpn2[18:9]));
```

对于这种过长的逻辑语句和一个TLBNUM=32的设计，把这个语句重复32遍显然是过于糟糕了。for循环给我们提供了一种很好的解决方法。值得注意的是，这里看到有很多乘号，但他们一定不会被分析为乘法器，因为当我们把循环展开，这些乘号在编译阶段就会被自动计算好，等到上板，他们就是已经确定的电路了！

综上所述，无论是哪种for循环，其本质都是为了减少代码量。所以，for循环并不是一个必需的语法，大家可以试着去使用，多去观察生成的电路图，从而将其使用的更加得心应手。

结语

在本辑中，我们简单介绍了一些基础的时序逻辑电路描述的技巧。当然，加上时序后，电路会变得无比复杂，一次秘籍必然不能涵盖全面。后面我们会继续慢慢将时序电路的技巧和注意事项慢慢告诉大家哦！