

# Episode 2 Advaced Verilog!

——《你可能不知道的verilog提高班指南》By TA-马子睿

## 前言

相信大家已经在数字电路的课程上简单接触过verilog这门语言了。你是否已经把思维从软件编程转换为硬件编程了呢？本辑将会带你理解verilog这门语言的原理，并介绍那些理论课上不会讲解的verilog小技巧。

## 目录

- [什么？"@\(\\*\)"？](#)
- [硬件并行？那两个if怎么办？](#)
- verilog小技巧
  - [善用默认值，BUG都消失](#)
  - [我三目运算符又回来了！](#)
  - [例化模块，我选指定接口法，它才是众望所归！](#)
- [结语](#)

## 什么？"@(\*)"？

大家在理论课上已经学习了always块语法，简单来说，我们可以把always描述的电路分成两类

- **always @(posedge clk)：时序电路**
- **always @(\*)：组合电路**

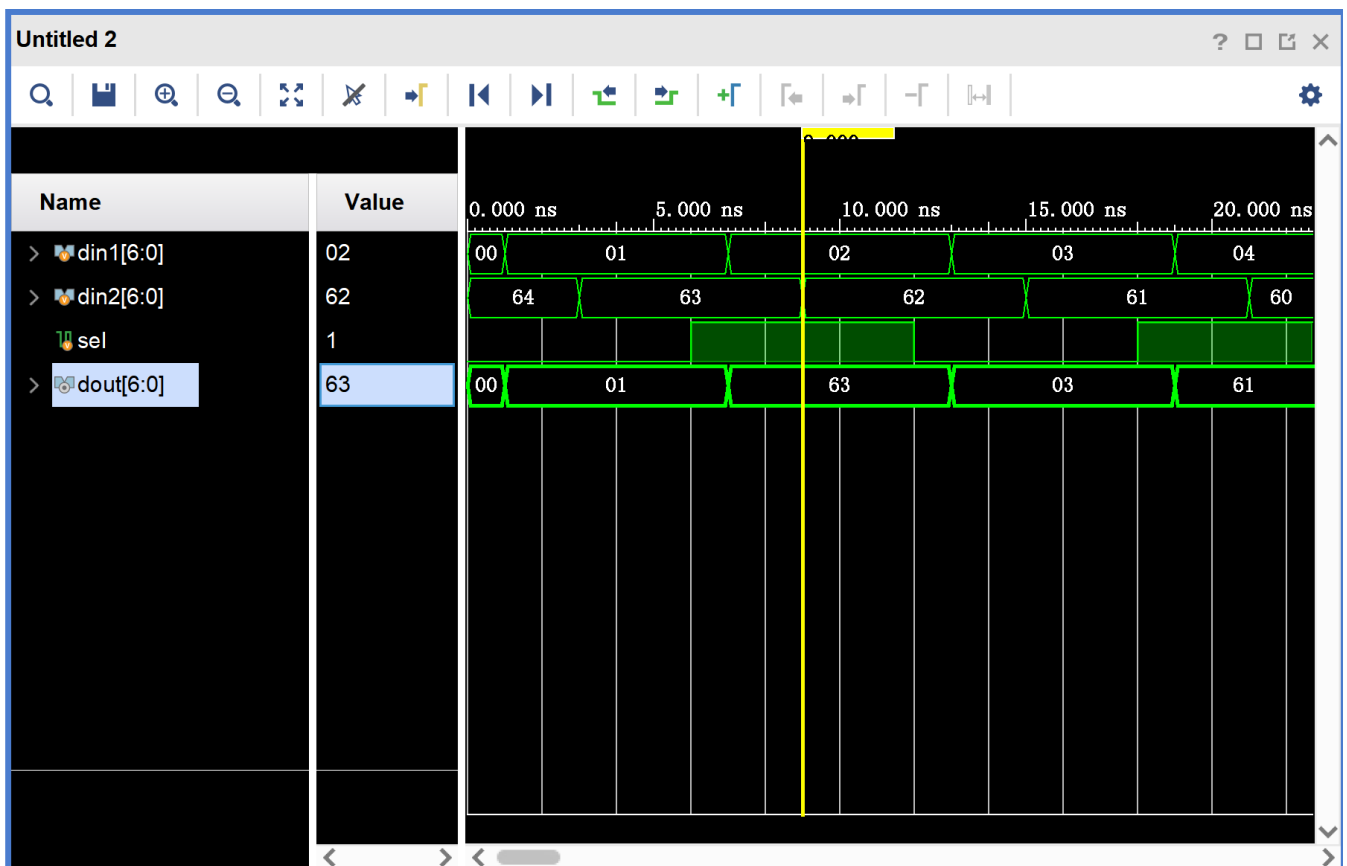
相信理论课老师在课上讲解的是：**@(\*)描述了一个组合电路**，这是最正确也是最合理的说法，但它肯定会引起大家的“云里雾里”。

可能的同学会有这样的理解：**@后括号中的信号名，表示这个always块对这些信号敏感，只要信号发生变化，对应的输出也就发生变化（“信号”就可以理解为“变量”）**。是的，这样的理解没有任何问题！那么@(\*)自然可以理解为，**这个always块对其中所有输入信号都敏感**。

我们来看一段代码：

```
module mux2_1(  
    input    [6:0] din1,  
    input    [6:0] din2,  
    input          sel,  
    output reg [6:0] dout  
);  
  
    always @(din1) begin  
        case(sel)  
            1'b0: dout = din1;  
            1'b1: dout = din2;  
        endcase  
    end  
endmodule
```

这还是那个二选一的多路选择器。看到这段代码，喜欢使用“敏感”的同学可能会说：这个代码写错了，din2改变的时候输出不会变的！是的！通过仿真我们确实可以看出，哪怕sel = 1，输出也依然会保持原样：



这里可以看到，在黄线处，din2由63变为62，但即便此时sel选择了din2，输出也没有变化，保持了63，这显然已经违背了一个多路选择器的设计初衷。因此，如果我们不使用@(\*)，我们就必须保证，把所有的输入敏感信号都写入@后的括号中，这显然太过麻烦（试想一个16选1的多路选择器），因此，verilog才提供给我们了一个优美的@(\*)，大大简化了我们的代码量。

可能有同学会发问：这样会不会导致这个组合电路块被一些其他信号干扰？答案是当然不会！这个(\*)只会对模块内部信号敏感，模块外部的信号对其没有任何影响。

所以，如果你想要描述一个组合电路，比如运算单元、多路选择器、译码器、编码器、状态转换机等，在目前阶段，如果你想描述组合电路，请无脑使用@(\*)，因为它真的很好用！

## 硬件并行？那两个if怎么办？

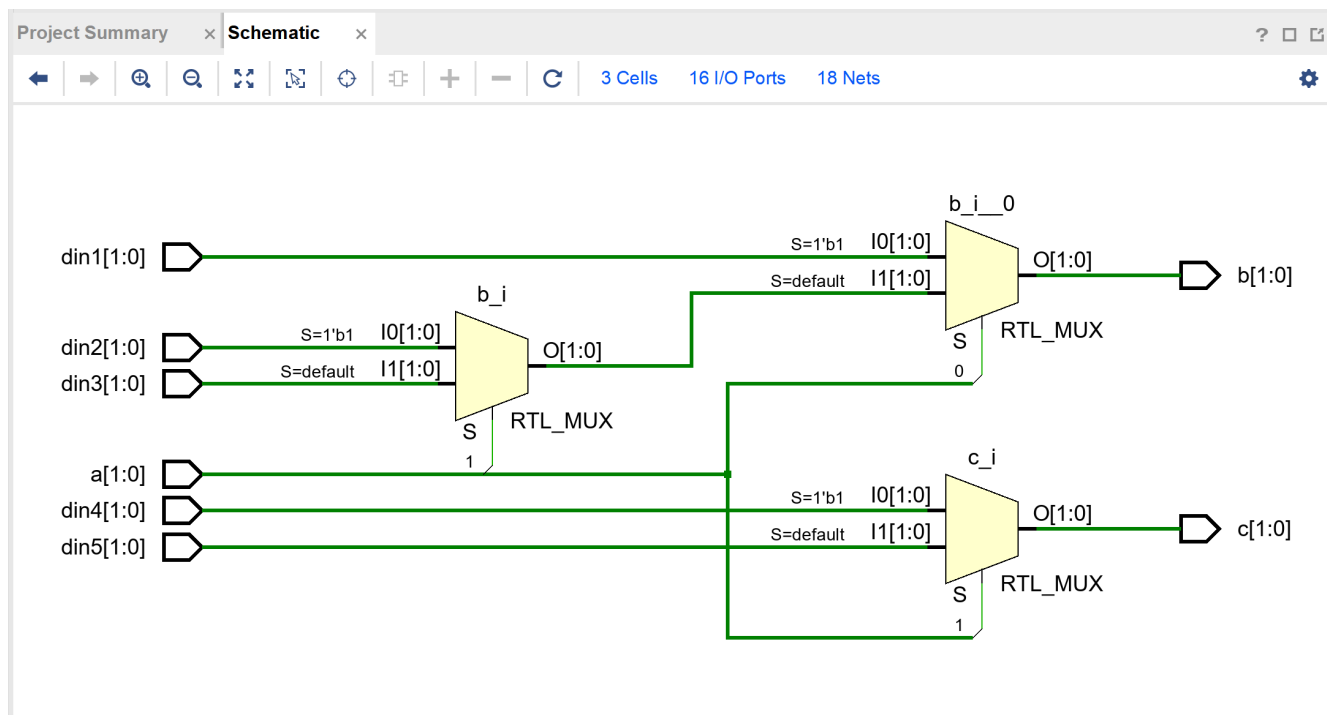
来看这样的一段代码

```
module test(  
    input      [1:0] a,  
    output reg  [1:0] b,  
    output reg  [1:0] c  
);  
always @(*) begin  
    // part 1  
    if(a[0])      b = 2'd1;  
    else if(a[1]) b = 2'd2;  
    else          b = 2'd0;  
    //part b  
    if(a[1])      c = 2'd2;  
    else          c = 2'd0;  
end
```

```
end
endmodule
```

如果把它看成C语言代码，你会毫不犹豫地说，他会一条一条顺序执行。但在verilog中，它并不是这样。

容易看出的是，part1和part2两部分是分离的，他们描述了两个不同的多选器，在电路中，他们不是串行的，而是并行的：



左侧和上方两个多选器描述了part1的电路，而下方的多选器描述了part2的电路。可以看出，上下两部分没有任何串行部分（即你的输出是我的输入），也就是**没有b和c之间的逻辑延迟**。

下面我们就来关注一下，if之间的两个重要问题：

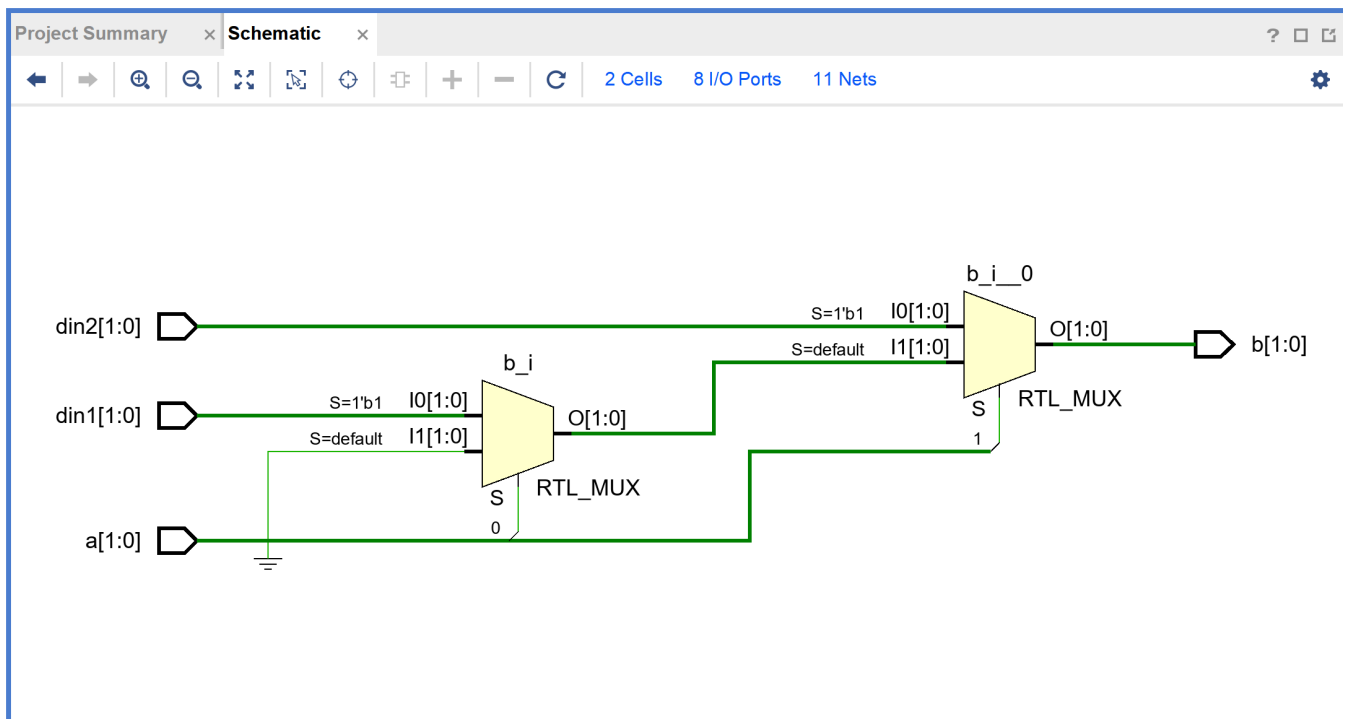
## 两个if之间真的都是并行的吗？

来看一段代码：

```
module test(
    input    [1:0] a,
    input    [1:0] din1,
    input    [1:0] din2,
    output reg [1:0] b
);
always @(*) begin
    b = 0;
    if(a[0])    b = din1;
    if(a[1])    b = din2;
end
endmodule
```

这段代码由一点令人难以理解：为什么在always下面紧接了一个b = 0呢？这是后面要分析的一些always高级用法，现在我们简单说明为：**如果下面所有的条件都不成立，那么b将被赋值成0**。这样也是为了避免在组合逻辑中出现锁存器。

我们来看它的电路图：



很显然，这两个多选器之间是串行的，观察信号，我们发现，**两个if都在对b赋值，而且赋值是不同的**，这种情况下，编译器会按照由上至下的顺序串行描述多选器。我们将其总结为：

- 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

## if和else if之间到底是什么关系？

看了刚才的分析，有同学可能自然地产生如下的想法：

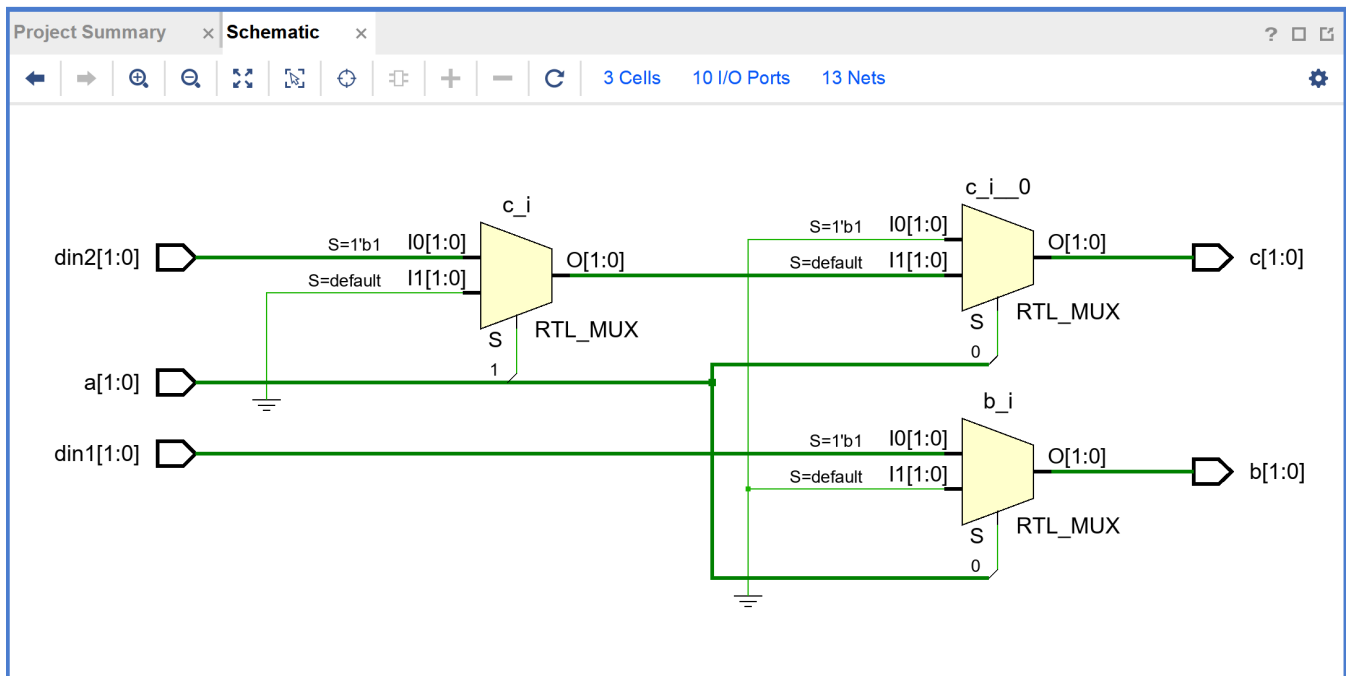
如果if和elseif的赋值对象也没有冲突，那么它们之间也是并行的

再来看一段代码：

```
module test(
    input    [1:0] a,
    input    [1:0] din1,
    input    [1:0] din2,
    output reg [1:0] b,
    output reg [1:0] c
);
always @(*) begin
    b = 0; c = 0;
    if(a[0])      b = din1;
    else if(a[1]) c = din2;
end
endmodule
```

如果按照上述思想，应该会出现两个选择器，一个选择b的输入，一个选择c的输入，它们是并行的。

可是，事实真的是这样吗？



可以看到，c的输出依然受到了两个选择门的影响。事实上，**赋值对象不同的if和else if之间的串行关系是绝对的**，只要if的条件成立，else if中的赋值就永远不会进行，哪怕它们没有冲突！

综上所述，两个if之间的关系，以及if和else if的关系是和软件描述语言略有不同的。我们可以将其简单总结为：

- **如果赋值对象没有冲突，那么两个if描述的多选器是并行的；**
- **赋值对象不同的if和else if的串行关系是绝对的。**

当然，实践才是检验真理的唯一标准。如果大家对自己写出的电路不自信，那么不妨画一下RTL电路图来看一下吧！当然，也非常欢迎大家对我的结论进行补充哦！

## verilog小技巧

### 善用默认值，BUG都消失

在always组合电路块中，为了避免锁存器的出现，我们不得不在每一个if分支中，对每一个输出信号进行赋值。但是，可能在很多情况下，会出现大量的if分支中都有相同的默认赋值的情况，让我们再回到这个例子：

```
module test(
    input    [1:0] a,
    input    [1:0] din1,
    input    [1:0] din2,
    output reg [1:0] b,
    output reg [1:0] c
);
always @(*) begin
    b = 0; c = 0;           // 默认赋值
    if(a[0])                b = din1;
    else if(a[1])            c = din2;
end
endmodule
```

注意这里的“默认赋值”，如果没有默认赋值，代码应写为：

```
module test(
```

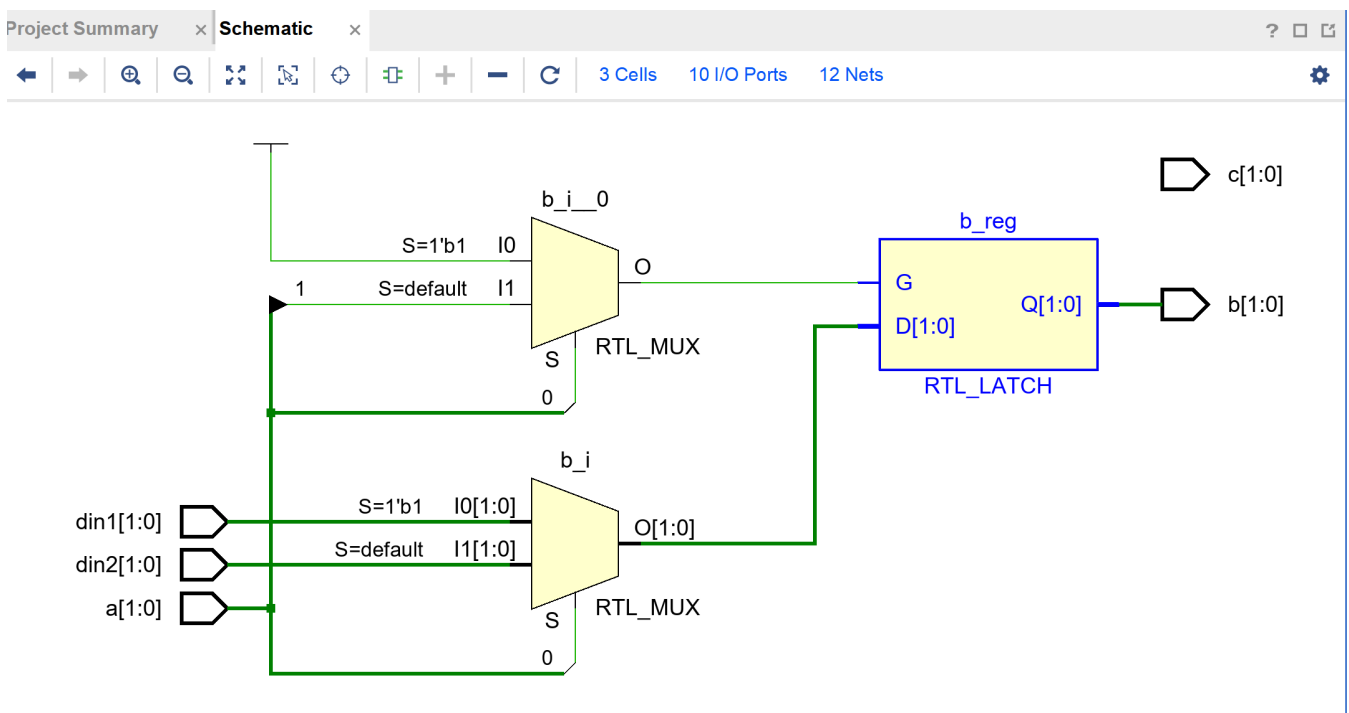
```

input      [1:0] a,
input      [1:0] din1,
input      [1:0] din2,
output reg [1:0] b,
output reg [1:0] c
);
always @(*) begin
    if(a[0]) begin
        b = din1;
        c = 0;
    end
    else if(a[1]) begin
        b = 0;
        c = din2;
    end
    else begin
        b = 0;
        c = 0;
    end
end
endmodule

```

这样一来，每一个if分支都需要对整个模块所有的输出信号都考虑周全，这在大型的工程中显然是不现实的。而且，在case和if分支中，**如果没有考虑到所有的情况，则必须使用default和else来保证逻辑完整性**，而特别是default，很容易被我们忘记，因此，使用默认赋值是十分必要的。

然而，即便使用了默认赋值，一些case代码块依然可能出现忘记对某个变量忘记赋值的情况。因此，**我们一定要养成看RTL电路图的习惯**，只要看到下图中的锁存器元件，那么一定要检查**逻辑完备性**（即在没有完全默认赋值的情况下，case和else if是不是真的涵盖了所有的可能）：



## 我三目运算符又回来了！

```

assign exception = 7'h48 ? tlb_exception : (exception_cache == 0 ? tlb_exception :
exception_cache);

```

看到这个代码，是不是DNA动起来了？像极了自己从前代码风格极差的C语言代码，但实际上：

- 为什么三目运算符依然被verilog保留了下来？这是因为，三目运算符可以非常简便地描述一个正经的二选一选择器，而这种选择器是一切选择器的根本。回想使用always块描述多路选择器的过程，无论是用if还是case，都是需要花费大量代码的。而一行assign和三目运算符，就可以描述出一个二选一选择器，是不是很方便呢？
- 在实际设计中，特别是在最顶层的模块中，我们偏好使用assign，这是因为**顶层模块主要是为了例化诸多模块**，这时如果加入一个always必然会显得不伦不类，但一句assign却是简洁、易懂的。
- 不过，三目运算符也有**缺点**：
  - **套用之后，逻辑变得很复杂**
  - **always块在综合实现电路时是有可能被优化线路的，但assign的线路是基本不会被优化的，因此，使用三目运算符来assign接口，基本确定了线路延迟不会减少**

最后附上三目运算符的语法：

```
dout = signal ? din1 : din2;
```

当signal不为0时，dout选择din1作为输入，否则dout选择din2作为输入

## 例化模块，我选指定接口法，它才是众望所归！

在verilog中，我们有两种**例化模块的方法**。一种是**顺序接口例化**，一种是**指定接口例化**。

下面我们先来探讨没有parameter的模块例化。顺序接口例化的含义是，按照模块中定义接口的顺序，顺序传递高层模块的接口。**这种方法实在不希望大家知道，也很不希望大家掌握，因为已经有无数的人被它坑过无数次。**所以，今天我们在这里只介绍**指定接口例化**的方法。

指定接口的格式如下：

```
模块名 为这个实例起的名字 (
    .模块接口名1      (变量名1),
    .模块接口名2      (变量名2),
    .模块接口名3      (变量名3)
);
```

这里给大家一个例子：

```
memory cache_mem(
    .clk          (clk),
    .r_addr       (addr),
    .w_addr       (addr_rbuf),
    .mem_din      (mem_din),
    .mem_we       (mem_we),
    .mem_en       (mem_en),
    .mem_dout     (mem_dout)    //注意这里不需要加逗号了！
); //注意这里要加分号
```

- memory是原模块的名字，在其他文件中，真的有个地方写着：module memory(...)
- cache\_mem是为这个例化出的实例起的名字，用来区分不同的实例
- **"."后面跟着的名字都是memory模块中input和output接口的名字**
- 括号中的名字是在当前文件中定义的wire或reg型变量，表示接入这个模块中的哪个接口。千万注意，**reg型变量不可以接入被例化模块的output接口，哪怕写的是output reg也不行，因为output reg只是为了方便定义在always块中被赋值的接口变量！**
- 最后一个例化接口不需要在后面跟着逗号，整个例化的括号后要加分号

下面我们再来看对带参数的模块进行例化的方法。有时我们需要定义一些可以变长度的模块，比如ALU、触发器、多选器，来提升代码的通配性。我们在定义模块的文件中一般写作：

```
module mux2_1(  
    parameter WIDTH = 32  
)(  
    input      [WIDTH-1:0] din1,  
    input      [WIDTH-1:0] din1,  
    input      sel,  
    output reg  [WIDTH-1:0] dout  
)  
;
```

那么例化时，如果我们需要一个16位宽的多选器，我们需要这样写：

```
mux2_1#(16) mux(  
    ...  
)  
;
```

当然，如果我们采用下面这种方法：

```
mux2_1 mux(  
    ...  
)  
;
```

那么生成的mux实例依然是32位的mux2\_1多路选择器。

## 结语

在本辑中，我针对了大家平时可能会疑惑的@(\*)、if硬件并行作了简单的解释，并介绍了几个verilog代码书写时的小技巧。希望能够帮助到大家！有关verilog的更多应用，我会在后面几辑中慢慢和大家讲解。如果本辑中有任何的谬误，欢迎大家指出哦！