Verilog 开发经验总结

- 以硬件为基础的原则

  Verilog 是硬件描述语言，所谓描述就是是在在描绘已经设计好的电路。尤其是在刚开始学习 HDL 时，还没有能直接把语言对应到具体电路的能力，更不能上手直接写 Verilog 代码。所以最优方案是先设计好硬件电路，再按照电路编写 Verilog。

- 模块化思想

  这个思想不仅仅是 Verilog 的开发了，就算是 C 语言甚至现在的超高级语言，也一直在强调封装的概念。Verilog 开发遵循的是自顶向下的模块化设计，思路基本是从最终功能不断细分，直到 Verilog 可以很直接地描述最基础的硬件单元，例如加法器，移位寄存器等等。模块划分一定要尽量细，功能单一，且一定要留出使能、复位等接口以便于系统搭建。

  说明以下，所谓 Verilog 可直接描述指的是按照规范描述出来的电路，开发工具能够很清晰地理解所要描述的功能，而不会发生误解等现象。一个只使用编译器能理解的代码开发的电路，错误率会大大降低。相反，如果功能划分不够清晰，使得一个模块的功能过于庞杂，不仅描述困难，编译器也可能产生很大的误解。把握编译器的理解方式是有助于做 Verilog 的开发的，但这就需要一个长远的积累了，我在最后会给出一些例子。

- 时序电路与逻辑电路完全分开

  Verilog 中除了数据流模型和门级模型以外，最常用的是 always 即行为级模型描述电路。一个 always 块可以理解为一个电路，或者实物上的一个芯片。所以不要在一个 always 里杂糅时序电路和逻辑电路，那样很可能导致编译器综合出一些奇葩的结果。时序 always 模块的敏感变量有且最多两个，一个是时钟边沿，一个是复位边沿，没有再多的敏感变量了。组合逻辑 always 中敏感变量列表必须包含该模块所有涉及到的变量，或者直接用(*)代替，个人推荐后者。

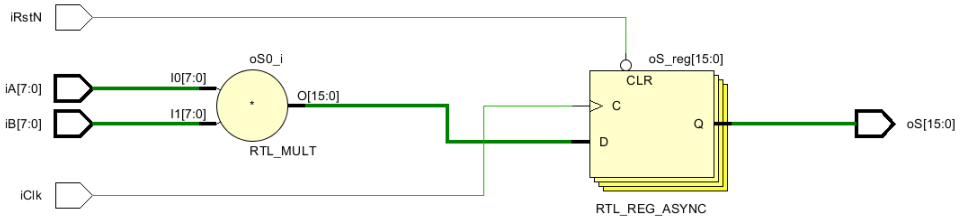
- 时序 always 块编写规范

  在时序 always 中，统一使用非阻塞赋值<=，因为时序 always 中在赋值号左侧的变量是真实的触发器，非阻塞赋值在触发时是同时赋值的，这很符合触发器在上升沿到来后同时将 D 输出到 Q 的实际情况。

  简单分析一下以下语句：

```
23  module timetest(
24      input iClk,
25      input iRstN,
26      input [7:0] iA,
27      input [7:0] iB,
28      output reg [15:0] oS
29      );
30
31      always @(posedge iClk or negedge iRstN) begin
32          if (~iRstN) begin
33              // reset
34              oS <= 16'd0;
35          end
36          else begin
37              oS <= iA * iB;
38          end
39      end
40  endmodule
```

这是一个非常简单的乘法器，我们看看编译器如何理解这个 always 模块的：



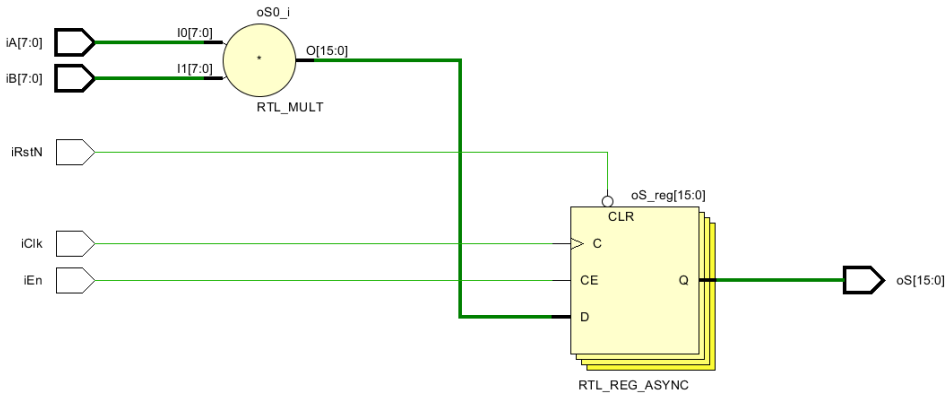
可见 always 时序模块左侧变量为触发器(oS)，右侧的表达式会用逻辑电路实现后接到除法器的 D 端同时可以看到在有两个触发信号中优先级最高的信号会自动设置为复位信号。

我们再给这个模块加上使能，代码如下：

```
module timetest(
    input iClk,
    input iRstN,
    input iEn,
    input [7:0] iA,
    input [7:0] iB,
    output reg [15:0] oS
    );

    always @(posedge iClk or negedge iRstN) begin
        if (~iRstN) begin
            // reset
            oS <= 16'd0;
        end
        else if(iEn)begin
            oS <= iA * iB;
        end
    end
endmodule
```

综合生成的电路如下：



可见在电路中配对复位的 else if 中的变量会被综合成使能信号。所以我们正常描述这类电路时也就要按照这一类的规范来进行。

时序 always 块是整个同步电路设计的基础，这只是一个很简单的只有一组触发器的电路，尚不涉及时序逻辑分析。在面对需要流水线作业的情况下，就必须熟悉同步时序逻辑的特点并分清楚各寄存器之间的关系，这些就属于开发者的事了，在这里不多做讨论。

- 组合逻辑电路描述

组合逻辑通常使用 assign 数据流模型和组合 always 块来描述。Assign 语句通常只用于描述很简单的逻辑，其中跟多的是描述选择器，如下：

assign out = en?in:1'b0;

assign 配合三目运算是最常用的，当然也可以描述简单的或之类的运算和加法、乘法等等。

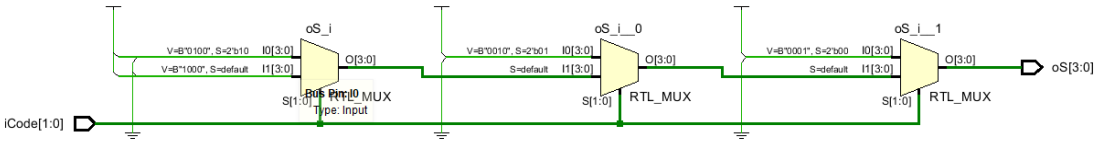
对于较为复杂的组合逻辑，通常使用 always 来描述，因为在 always 中可以更简单地使用 c 语言中的 if，case 语句来描述，但是他们之间的区别很大。

例如描述一个 2-4 译码器，如果用 if 来实现，代码如下：

```
module iftest(
    input [1:0] iCode,
    output reg [3:0]oS
    );

    always @(*) begin
        if (iCode == 2'b00) begin
            oS = 4'b0001;
        end
        else if (iCode == 2'b01) begin
            oS = 4'b0010;
        end
        else if (iCode == 2'b10) begin
            oS = 4'b0100;
        end
        else begin
            oS = 4'b1000;
        end
    end
endmodule
```

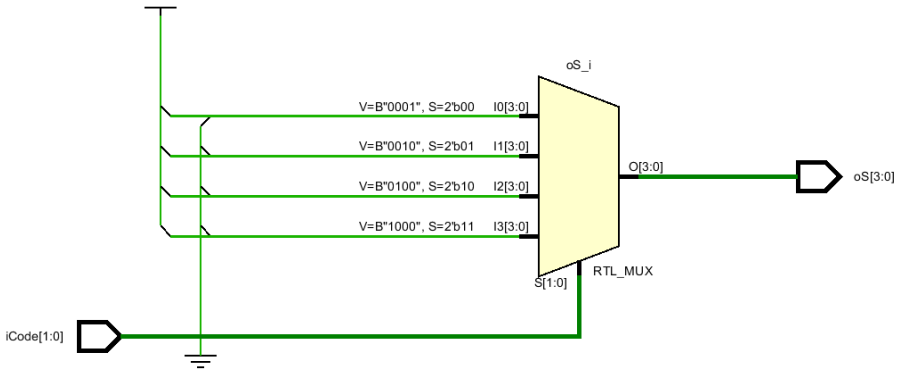
电路如下：



可见，if 是带有优先级的，所以必然会使用多个选择器来实现上述电路，若采用 case 语句来描述，代码如下：

```
    always @(*) begin
        case(iCode)
        2'b00:oS = 4'b0001;
        2'b01:oS = 4'b0010;
        2'b10:oS = 4'b0100;
        2'b11:oS = 4'b1000;
        endcase
    end
```

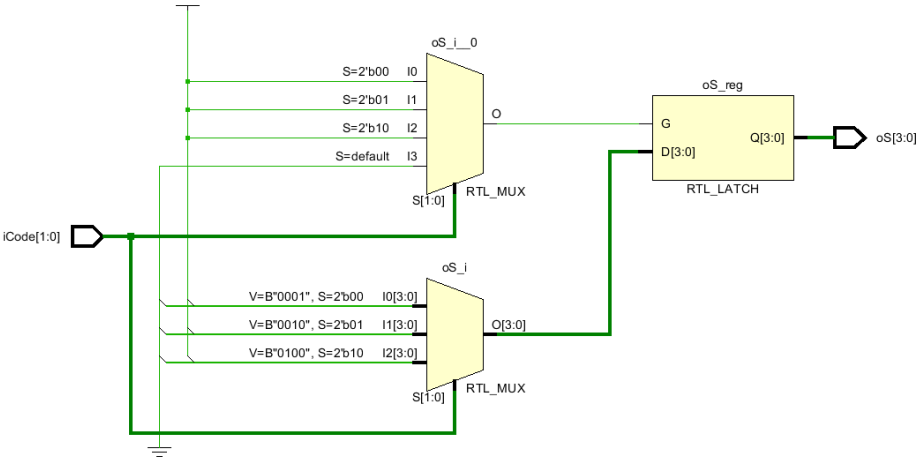
综合出来的电路如下：



可见是没有优先级的，是一个完整的数据选择器。如果译码功能更为复杂，对于 case 语句编译器会采用 ROM 来实现，及把输出值存进输入地址对应的存储空间中。

我们可以发现，虽然我们把 oS 定义为了 reg 型变量，但分析出来的电路中它任然是一个线网型的。因为在 always 中被赋值的变量必须是 reg 型，所以才申明为 reg，但它究竟是什么是需要靠实际电路来决定的。

需要格外强调的一点是，组合逻辑与时序逻辑不同，它不应该具备存储功能，换言之对于任何输入，都必须给出一个输出值。如果我们把 case 中最后一条注释掉，会发现分析出来的电路变成了这样：



可以看到电路复杂很多，更是多了 RTL_LATCH 这个块，及锁存器。原因很简单，因为当输入为 2'b11 时，根据我们的代码它没有给出 oS 应该是什么，那么它只能保持之前的值，及有了存储功能。但又没有时钟，那它只能成为锁存器。这样的设计是 FPGA 中需严格避免的，因为它会占用大量资源而且时序上不稳定。所以编写组合逻辑的 Verilog 代码时，不管是使用 if 还是 case，都必须保证该变量无论输入为何都必须给出明确的输出。另外，在组合逻辑中推荐使用阻塞赋值=，因为组合逻辑块其实跟类似与 C 语言，需要利用=的先后特性。

- 代码规范

代码规范是一个老生常谈的问题，但凡是真正做过项目的人都很厌恶不良的代码风格。一个优秀的代码风格不仅能提高代码的可读性，也能大大降低出错的可能性并提高排查错误的效率。

首先是模块的定义，不要采用在一行上堆满各种端口的形式，可参考一下 Xilinx 官方的写法，我将代码附上：

```
`timescale 1 ns / 1 ps

	module AXI_GPIO_pro_v1_0 #
	(
		// Users to add parameters here

		// User parameters ends
		// Do not modify the parameters beyond this line


		// Parameters of Axi Slave Bus Interface AXI_Lite
		parameter integer C_AXI_Lite_DATA_WIDTH	= 32,
		parameter integer C_AXI_Lite_ADDR_WIDTH	= 6
	)
	(
		// Users to add ports here
		inout [31:0]GPIO_1,
		inout [31:0]GPIO_2,
		inout [31:0]GPIO_3,
		inout [31:0]GPIO_4,
		inout [31:0]GPIO_5,
		inout [31:0]GPIO_6,
		inout [31:0]GPIO_7,
		inout [31:0]GPIO_8,
		// User ports ends
		// Do not modify the ports beyond this line


		// Ports of Axi Slave Bus Interface AXI_Lite
		input wire	axi_lite_aclk,
		input wire	axi_lite_aresetn,
		input wire [C_AXI_Lite_ADDR_WIDTH-1 : 0] axi_lite_awaddr,
		input wire [2 : 0] axi_lite_awprot,
		input wire	axi_lite_awvalid,
		output wire	axi_lite_awready,
		input wire [C_AXI_Lite_DATA_WIDTH-1 : 0] axi_lite_wdata,
		input wire [(C_AXI_Lite_DATA_WIDTH/8)-1 : 0] axi_lite_wstrb,
		input wire	axi_lite_wvalid,
		output wire	axi_lite_wready,
		output wire [1 : 0] axi_lite_bresp,
		output wire	axi_lite_bvalid,
		input wire	axi_lite_bready,
		input wire [C_AXI_Lite_ADDR_WIDTH-1 : 0] axi_lite_araddr,
		input wire [2 : 0] axi_lite_arprot,
```

```
        input wire    axi_lite_arvalid,
        output wire    axi_lite_arready,
        output wire [C_AXI_Lite_DATA_WIDTH-1 : 0] axi_lite_rdata,
        output wire [1 : 0] axi_lite_rresp,
        output wire    axi_lite_rvalid,
        input wire    axi_lite_rready
    );
// Instantiation of Axi Bus Interface AXI_Lite
    AXI_GPIO_pro_v1_0_AXI_Lite # (
        .C_S_AXI_DATA_WIDTH(C_AXI_Lite_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_AXI_Lite_ADDR_WIDTH)
    ) AXI_GPIO_pro_v1_0_AXI_Lite_inst (
        .S_AXI_ACLK(axi_lite_aclk),
        .S_AXI_ARESETN(axi_lite_aresetn),
        .S_AXI_AWADDR(axi_lite_awaddr),
        .S_AXI_AWPROT(axi_lite_awprot),
        .S_AXI_AWVALID(axi_lite_awvalid),
        .S_AXI_AWREADY(axi_lite_awready),
        .S_AXI_WDATA(axi_lite_wdata),
        .S_AXI_WSTRB(axi_lite_wstrb),
        .S_AXI_WVALID(axi_lite_wvalid),
        .S_AXI_WREADY(axi_lite_wready),
        .S_AXI_BRESP(axi_lite_bresp),
        .S_AXI_BVALID(axi_lite_bvalid),
        .S_AXI_BREADY(axi_lite_bready),
        .S_AXI_ARADDR(axi_lite_araddr),
        .S_AXI_ARPROT(axi_lite_arprot),
        .S_AXI_ARVALID(axi_lite_arvalid),
        .S_AXI_ARREADY(axi_lite_arready),
        .S_AXI_RDATA(axi_lite_rdata),
        .S_AXI_RRESP(axi_lite_rresp),
        .S_AXI_RVALID(axi_lite_rvalid),
        .S_AXI_RREADY(axi_lite_rready),
        .GPIO_1(GPIO_1),
        .GPIO_2(GPIO_2),
        .GPIO_3(GPIO_3),
        .GPIO_4(GPIO_4),
        .GPIO_5(GPIO_5),
        .GPIO_6(GPIO_6),
        .GPIO_7(GPIO_7),
        .GPIO_8(GPIO_8),
    );

    endmodule
```

上述代码中，包含了定义和例化的两种规范，给大家参考。同时，还有一下这些简单要求：

1. 就算只有一句话，也必须使用 begin 和 end
2. 每个 begin 和 end 下的内容统一向右一个 tab（四个空格）
3. Case 语句下的各情况对齐
4. If 中不省略条件，不写 if(a)，写成 if(a == 1'b1)
5. 在设计中不使用 initial 语句来初始化，初始化统一放到 rst 下
6. 有限状态机采用二段式或者三段式的标准描述方法
7. 不依赖于运算符优先级，统一用括号来说明运算顺序
8. 变量名要表达实际含义，但不要使用汉语拼音