数字电路

实验总结

现阶段 verilog 以及 vivado 掌握内容

助教

2021-11-1

目录

Verilog 语法简介

Verilog 与 module

Verilog HDL 是一门硬件描述语言，用于描述电路的一个部件。通过 verilog 语言描述得到一个模块（module），类似于一个模型，模型是没有办法直接用的，需要将模型实例化为某个电路中的一个具体的模块，才能够达到电路的信号输入到该模块，然后从该模块输出的目的。

Verilog 具体语法

参考书籍：

【1】 Verilog - A Guide to Digital Design and Synthesis

【2】 Verilog 数字系统设计教程-夏宇闻

Verilog 与 C 的异同

Verilog 与 C 语言的语法相似，例如 verilog 采用 begin end 包含一定范围的代码块（C 语言采用{}），verilog 采用模块实例化的形式调用模块，C 语言采用函数调用的形式让函数真正发挥作用。

Verilog 与 C 语言的不同，使用 verilog 编写 module 时需要理解不同变量的并行性，例如：

```
always @(clk)begin
    tmp1 = tmp2;
end

always @(clk)begin
    tmp1 = tmp3;
end
```

以上两个 always 块内的语句都是对 tmp1 进行赋值，然而需要注意的是 verilog 语言描述的是电路部件的实际结构，这两个 always 块在电路中会被翻译为：tmp1 这个单元分别连接到 tmp2 和 tmp3 两个信号；当 clk 信号发生变化时，always 块中的语句开始执行，此时在物理部件上，tmp2 和 tmp3 的信号都会传递到 tmp1，导致 tmp1 的值不确定。

相比于 C 语言的顺序执行，verilog 的并行性提醒我们采用 verilog 编写 module 时需要考虑对相同变量赋值时是否会发生冲突。

Verilog 的设计

我们学到的包括行为级和结构级描述都是可以使用的，但是对于复杂的逻辑电路来讲，行为级描述是更为常用的描述方法，使用行为级描述更能够简化我们的设计难度。

Verilog 语法注意点

下面介绍一些 verilog 在编写时的一些习惯。注意这只是一些推荐习惯，不代表其他的就一定错。

module 的端口类型

module 的端口类型默认为 wire 类型，在实际使用中，我们可能会遇到如下情况

```
module tmp(a,b,c);
    input a;
    input b;
    output c;

    always@(posedge a)begin
        c = b;
end
endmodule
```

代码中信号 c 需要在 always 中使用，此时会报错，因为 module 声明时默认信号类型都是 wire 类型，而 always 块中等号左边的信号必须是 reg 类型，最简单的解决办法就是将 output c 声明为 reg 类型：

```
module tmp(a,b,c);
    input a;
    input b;
    output reg c;

    always@(posedge a)begin
        c = b;
end
endmodule
```

这样就不会报错了，且省略了定义一个新信号的过程。

assign 的含义

assign 经常被用来给 wire 类型信号赋值。但是在 verilog 语法中 assign 实际上是一种连续赋值的含义。例如：

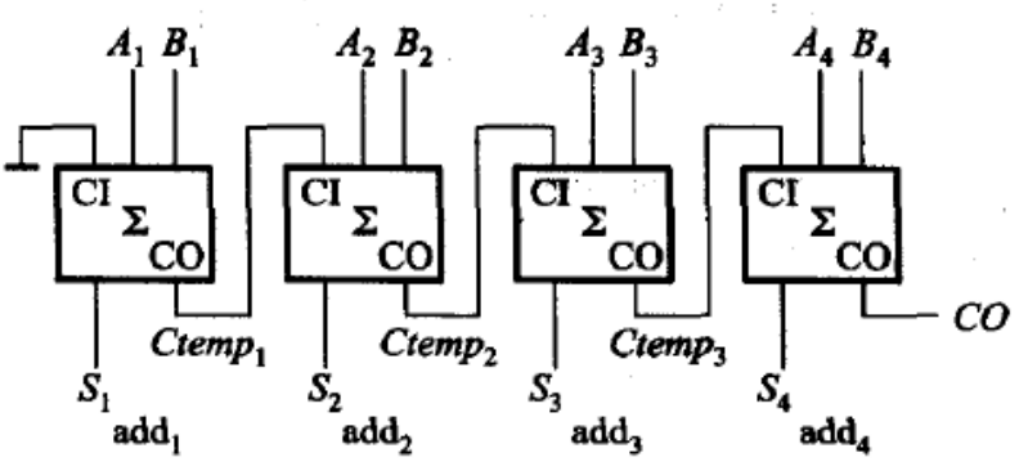
```
assign out = input_data;
```

我们可以理解为将 input_data 这根线和 out 这根线连接在一起，那么无论 input_data 这个 wire 信号如何变化，out 的值都和 input_data 的值保持一致，因此 assign 通常使用在组合逻辑电路中，即当目的电路是组合电路时，我们使用 assign 语句来让信号始终与等号右边的计算结果保持一致。

注意 assign 不可以写到 always 块中！

模块实例化

好多同学不理解模块实例化的含义，我们采用 verilog 编写一个 module，这个 module 只是一个部件，类似于一个简单的芯片，有 input，有 output，但是当你单独一个 module（芯片）时，你没有办法让它工作。而让它工作就需要从外部给它传递信号，信号通过 input 端口进入到 module 内部，此时 module 才算正常工作。那么一个模块实例化的例子：




```
module add_16(A,B,Cin,Out,Cout);
input [15:0] A;
input [15:0] B;
input Cin;
output [15:0] Out;
output Cout;

wire [2:0] tmp;

add_4 add_4_0(.A(A[3:0]), .B(B[3:0]), .Cin(0), .Out(Out[3:0]), .Cout(tmp[0]));
add_4 add_4_1(.A(A[7:4]), .B(B[7:4]), .Cin(tmp[0]), .Out(Out[7:4]), .Cout(tmp[1]));
add_4 add_4_2(.A(A[11:8]), .B(B[11:8]), .Cin(tmp[1]), .Out(Out[11:8]), .Cout(tmp[2]));
add_4 add_4_3(.A(A[15:12]), .B(B[15:12]), .Cin(tmp[2]), .Out(Out[15:12]), .Cout(Cout));
endmodule
```

图中 16 位加法器实例化了 4 个串行进位 4 位加法器，也就是说当我们设计一个 16 位加法器时，我们只需要若干个 4 位加法器就可以，而 4 位加法器我们用 verilog 实现以后，只需要模块调用就可以完成设计了。在实际应用中，一个大型的芯片通常包括多个模块，而其中的多个模块是有不同企业或个人设计的具有 IP（知识产权）的模块，因此当你使用别人的模块设计自己的芯片时，只需要了解对应模块的输入和输出参数即可，而不需要设计芯片的每个模块的每个部分，减少了芯片设计的时间。

模块实例化的语法

了解过模块实例化的同学应该知道模块实例化有两种方式，一种是端口按照顺序对应的方式，我们并不推荐这种方式，因此只介绍另一种方式：

```verilog
module add_4(A,B,Cin,Out,Cout);
input [3:0] A;
input [3:0] B;
input Cin;
output [3:0] Out;
output Cout;

assign {Cout, Out} = A + B + Cin;
endmodule
```

```verilog
module add_16(A,B,Cin,Out,Cout);
input [15:0] A;
input [15:0] B;
input Cin;
output [15:0] Out;
output Cout;

wire [2:0] tmp;

add_4 add_4_0(.A(A[3:0]), .B(B[3:0]), .Cin(0), .Out(Out[3:0]), .Cout(tmp[0]));
add_4 add_4_1(.A(A[7:4]), .B(B[7:4]), .Cin(tmp[0]), .Out(Out[7:4]), .Cout(tmp[1]));
add_4 add_4_2(.A(A[11:8]), .B(B[11:8]), .Cin(tmp[1]), .Out(Out[11:8]), .Cout(tmp[2]));
add_4 add_4_3(.A(A[15:12]), .B(B[15:12]), .Cin(tmp[2]), .Out(Out[15:12]), .Cout(Cout));
endmodule
```

模块调用的语法为
module_name module_instance_name(.module port(current module port), …
.module port(current module port),);

module_name 指的是被实例化的 module 名字，例如 add_4，module_instance_name 相当于是你在当前的模块中给你被调用的模块命名一个临时的名字，例如 add_16 中实例化了 4 个加法器，那么这四个加法器需要一个名字，即 add_4_0,add_4_1,add_4_2,add_4_3，这样每个实例化的模块有了标签，相当于你真的在一个 16bit 加法器中放置了 4 个 4bit 加法器元件。

这就是模块调用的含义和使用方法。

{}和多位取的使用

在 verilog 中，变量的定义默认是 32 位的，我们在使用中经常会遇到一个变量其中几位连接一个模块或者一个变量，另外几位连接到一个模块或者一个变量，verilog 中的多位取简化了我们的设计。例：

```verilog
wire [16:0] X;
wire [3:0] Y1;
wire [3:0] Y2;
wire [4:0] Y3;
wire [8:0] Y4;

assign Y1 = X[3:0];
assign Y2 = X[10:7];
assign Y3 = X[14:10];
assign Y4 = {Y[3],Y[2]};
```

图中 16 位的信号 X 将低 4 位赋给 Y1，中间 4 位赋给 Y2，Y4 则是由 Y2 和 Y3 的位合并创建的，这就是{}的语法使用。

{}可以将{}里面的内容按照变量的顺序依次并排使用，从左到右的顺序是高位到低位，因此 Y4 的低位对应 Y2 的位数，高位对应 Y3 的位数。{}可以用在赋值语句的左右两边。

{}除了可以合并变量之外还可以合并简单的信号值，如：

```
assign Y3 = {5{1'b1}};
```

式子中的 5 表示里面的{1‘b1}复制多少次，而外面的{}表示将这些复制的信号合并为一个值，即 Y3=5‘b11111.

always 的使用

always 作为 verilog 中常用的语法需要熟练掌握，always 后面通常需要加上@()，括号里面用来加入触发 always 块执行的敏感信号列表，然后 always 的语句块需要被 begin end 包住。例如：

```
wire a;
wire b;
reg c;
always@(a or b or c) begin
    c = a+b;
end
```

@()里面写了 always 块里面可能变化的所有敏感变量，当这些变量的值发生变化时，always 块里的语句将会被执行，这点和 assign 里面的连续赋值是不一样的。有时候为了方便会写为@(*)，表示 always 块里所有信号都是敏感信号。

后续我们会接触时序逻辑电路，always 块的触发条件会进一步扩展。

If else 的用法

Verilog 里面 if else 是一个常用的行为级的描述方法，只需要写出 if else 的逻辑关系式就可以实现逻辑功能，底层实现交给仿真工具来进行。例如：

```
always@(*)begin
    if(a>b)begin
        out = a;
    end
    else if (a==b)begin
        out = a+b;
    end
    else begin
        out = b;
    end
end
```

通常 ifelse 语句需要写在 always 块内，并且和 C 语言类似，if else 语句也需要 begin end 包住，来指明 begin end 中的所有语句在满足 if else 条件时执行。

Case 的用法

Case 语句作为另一种条件语句在条件单纯依赖某个信号时常被使用，例如：

```
always@(A)
begin
    case(A)
        4'b0001: Y = 0;
        4'b0010: Y = 1;
        4'b0100: Y = 2;
        4'b1000: Y = 3;
        default: Y = 2'bX;
    endcase
end
```

上图中通过比较 A 的值来确定执行那一条语句，注意例如在译码器中，默认输入只有一个 1，但是当输入出现多个 1 时，正常情况下没有正确输出时需要在 case 里面加入 default，即表示 default 上面的 case 情况都不满足时，执行 default 对应的语句。

为什么使用 vivado

Vivado 是一个比较成熟的 FPGA 集成开发环境，我们只是使用了它的一小部分功能，后续大家还会使用到它的其他功能，虽然它有一定的上手难度，但是当大家熟练以后就会发现 vivado 强大的功能。

实验机房主要方便安装 vivado，并且本地环境安装 vivado 也比较方便。

现阶段必须掌握的 verilog 语法和 vivado 的使用

根据实验一和实验二的内容，我们希望大家能够：

- Vivado
  - 能够自己创建工程文件，并且添加相应的设计文件（your module）。
  - 能够自己添加测试激励文件（testbench）。
  - 能够流畅的使用 vivado 的 simulation 功能，点击 run simulation 后跑出仿真波形。
  - 能够熟练的打开波形，并结合键盘和鼠标调出便于观察的波形。
  - 能够熟练使用波形图中的搜索工具和进制转换工具，改变波形图中输出的表现形式。
  - 能够从波形图中找到有问题的波形，并且结合自己的代码分析错误结果出现的原因。
  - 能够在多个 testbench 中设置 set as top 选择目标仿真文件。
- Verilog
  - 能够熟练的定义 module 的端口信号。
  - 能够熟练的采用 assign 和 always 语法实现相应的逻辑功能。
  - 能够在给出简单的组合逻辑电路（真值表、逻辑电路图等）情况下写出对应的 verilog

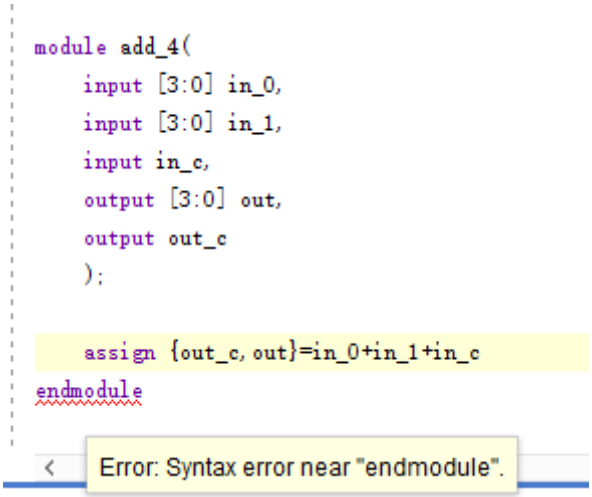
代码。

- 能够分析相应的逻辑功能，选择合适的 wire 和 reg 信号，并结合 if else 或者 case 语句写出简洁的 verilog 代码。
- 能够自行编写 testbench，生成对应端口的激励信号，即 initial 语句的使用。
- 能够明白模块调用的原理，掌握模块在调用不同模块时的端口连接，正确连接对应的端口，参考 16bit 比较器的实现。
- 能够完成现阶段的实验任务，包括 4bit 比较器（if else 实现），4bit 全加器（assign 语法），16bit 比较器（模块调用），附加题译码器（case 实现），要求能够自己独立完成。

使用 vivado 出现 Error 时怎么办

Stage 1. Syntax Error

句法错误，一般是最先遇到的问题。这种错误可以由 IDE 检测出来——即图中红色波浪线的部分。

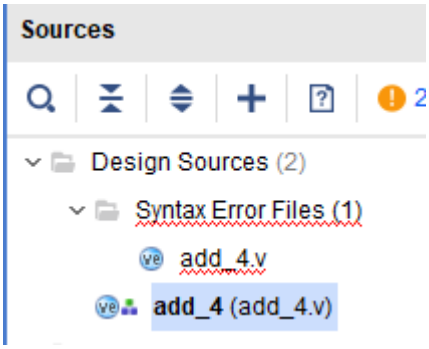


A. 解决方法

将鼠标移至波浪线上停留，会出现如上图所示的错误提示信息，根据错误提示信息解决问题。

B. 如何快速察觉与定位?

1. **文件编辑时&保存后**：注意编辑器右部的滑动栏。最上边的红色区块给出了所有错误情况概括，而下方的红色扁平则可以帮助快速定位错误位置与错误原因。
2. **文件保存后**：注意 source 栏中的情况，有如下图者，则说明有句法错误的文件。

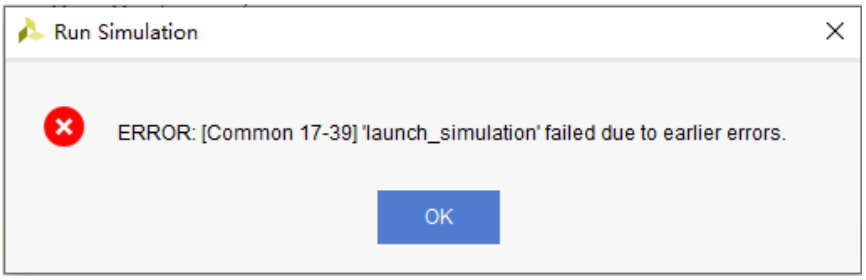


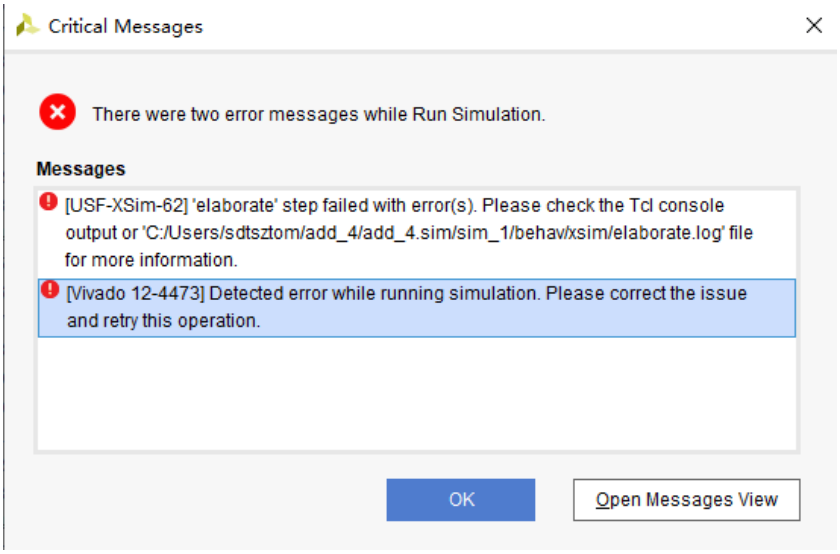
注意点

1. 多处错误时，可以优先解决最前面的错误。因为有些后面的问题是由前面的错误引发的。
2. 很多时候，错误提示是："error near ..."，即错误可能在此之前，也可能在此之后，而不仅仅局限于该行，应该上下多看看
3. 注意一些隐蔽的错误，比如用了中文字符的括号；分号与冒号的调换使用……

Stage2 Compilation Error

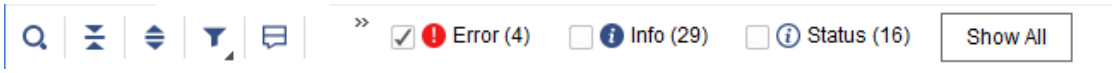
当你 run simulation 之后，倘若无法进行仿真时的报错，如下图所示。此时应该点击两个确定，关闭这些错误提示窗口。





A. 解决方法

1. 打开底部的“Message”窗口，**展开**和查看所有错误信息并根据提示进行 debug。各个信息的图标含义和过滤方式可见上方 Menu

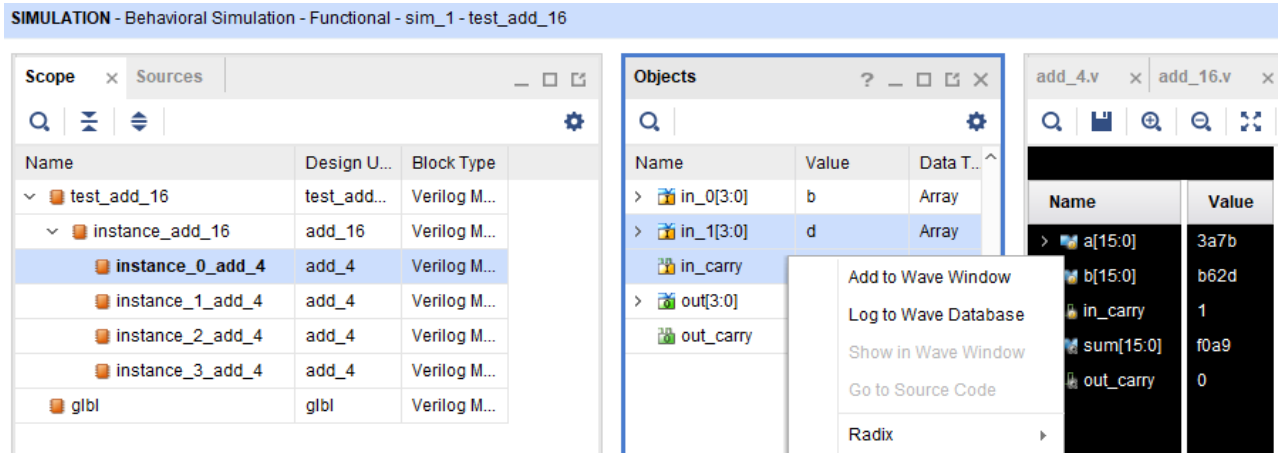


2. 根据 Message 或者 TCL console 窗口的提示，打开对应的 log 文件查看，如

Stage3 Simulation Error

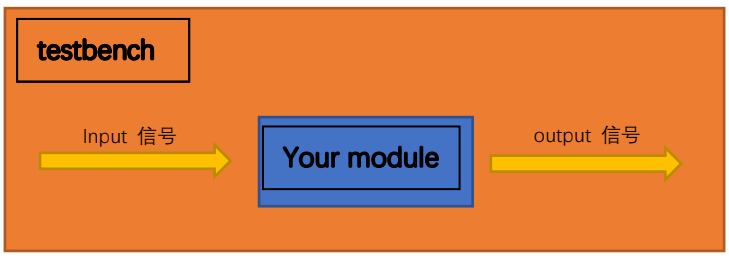
监视模块内部的信号

默认的波形界面只会给出 testbench 中的信号。倘若想要查看内部模块中信号信息，可打开 Scope 界面，选中对应的模块，再在对应 Object 界面加入想要监视的信号。不过刚加入波形界面，可能没有记录信息，此时就需要 restart 然后再重新 run 一遍。



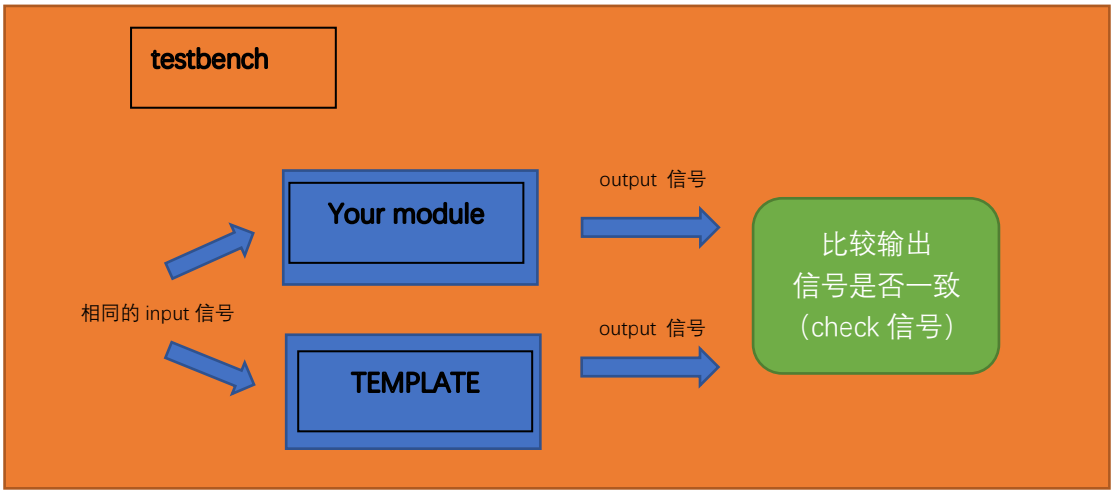
testbench 和自己 module 的关系

testbench 同样是一个模块，但是 testbench 不需要输入和输出端口，testbench 的作用是生成激励信号，输入到需要测试的模块中，同时将被测试模块的输出接口连接到自己设定的内部信号中，通过波形图观察输入被测模块的信号和从被测模块输出的信号，判断被测模块的功能是否符合预期。如图：



Testbench 通过模块调用的方式获取被测试模块输入和输出，根据波形图的结果判断模块功能，目前仅使用到仿真阶段。

为什么使用模板（TEMPLATE）验收



TEMPLATE 对应的 testbench 一共实例化了两个模块，一个是同学们写的模块，另一个是 TEMPLATE 模块，两个模块输入相同的输入，等待两个模块的输出，然后比较两个模块的输出结果，用 check 信号来看二者的输出结果是否一致，当完全一致时说明两个模块的功能一样，同学们的设计满足实验要求。

TEMPLATE 帮助同学们和助教快速发现 module 的错误，及时确定输出逻辑错误的地方，同学们也可以利用 TEMPLATE 寻找自己 module 错误的地方，自行 debug。