Episode 4 How to make Correct Digit Circuit?

——《你可能不知道的verilog提高班指南》By TA-马子睿

前言

大家好！在第一次实验中，各位同学面对未知的verilog和vivado，火力全开各显神通，但有时面对纷繁复杂的电路，总是愁眉紧锁、一筹莫展。在本辑中，我将结合我的自身经验，告诉大家一些硬件Debug的技巧和流程，希望大家能借助这些技巧，将自己的数字电路设计的正确、简洁、美观。

目录

构建一个“看得懂”的设计文件

大家学习C语言的时候，程序设计的助教一定也和大家强调了代码的可读性、规范性。一份可读性高的代码是非常容易Debug的，而对于verilog这种动辄几百行的代码更是这样。下面我来介绍几个让代码变得优美的小方法：

使用行为级描述

verilog给大家提供了很多描述电路的方法，其中大家喜爱的主要是门级电路描述和行为级电路描述。事实上，**行为级电路描述更为高级，它相较于门级电路更容易看出错误，更容易让自己理解**。

verilog中的行为级描述一般是由always块和case、if、elseif、else语句构成的。这些行为级描述更像是大家熟悉的C语言，但又在某些方面区别于C语言。

- **case...endcase语句**

  case...endcase语句描述了一个由多选器构成的电路，我们用一个实例来了解一下：

```
always @(*) begin
    case(way_sel)
    4'b0001: replace_tag = tag_0;
    4'b0010: replace_tag = tag_1;
    4'b0100: replace_tag = tag_2;
    4'b1000: replace_tag = tag_3;
    default: replace_tag = 0;
    endcase
end
```

这个电路的解读是：**一个多选器有4个输入：tag_0，tag_1，tag_2，tag_3，而way_sel信号作为了多选器的选择信号，replace_tag作为了多选器的输出。**这种描述的好处是，如果选择信号十分复杂，即不是自然编码或者是独热编码，而是如格雷码或自定义编码的信号，那么我们就不用关心这个多选器的选择底层逻辑是如何实现的了。比如：

```
always@(posedge clk) begin
    case(wrt_type)
    4'b0001: begin
        case(addr_rbuf[1:0])
        2'b00: ...
        2'b01: ...
        2'b10: ...
        2'b11: ...
        endcase
    end
    4'b0011: begin
        case(addr_rbuf[1])
            1'b0: ...
            1'b1: ...
        endcase
    end
    4'b1111: ...
    default: ...
    endcase
end
```

这个例子不仅说明了，在wrt_type信号是自定义信号格式时case语句描述的便捷性，更说明了case语句是可以嵌套的。

那么当选择信号位宽过大时，怎么让case语句变得更简洁呢？显然，十六进制给了我们很好的解决方案。我们用十六进制重写第一个例子：

```
always @(*) begin
    case(way_sel)
    4'h1: replace_tag = tag_0;
    4'h2: replace_tag = tag_1;
    4'h4: replace_tag = tag_2;
    4'h8: replace_tag = tag_3;
    default: replace_tag = 0;
    endcase
end
```

看，这样是不是简洁多了呢？**注意，十六进制虽然是h，但前面的数字依然是二进制位宽！**

使用case语句经常会出现下面的问题：

- **没写default的情况**：如果**在组合电路中**列出的选择信号的情况**没有涵盖选择信号位宽下所有可能的数字**（比如2位宽就是0-3，4位宽就是0-15），那么就会出现**逻辑不完备**，Vivado会分析出**锁存器导致电路错误**。如果你无法理解什么叫“选择信号位宽下所有可能的数字”，那就只要看到case就default一下好啦！
- **没写endcase**：这个一定会报错，一定要记得写！

- **if，else if，else语句**

  if选择语句事实上也描述了选择器，但if和case语句是有以下区别的：

  - case语句不会出现冲突：每一种情况是完整的（在不使用x和z的情况下）。case的每种情况是选择信号全位宽的一种可能的取值，任何两个case间不会有交集，所以**case语句描述的所有情况没有优先级区别**。
  - if，else if，else语句是有先后顺序的：Vivado会使用一串的多选器来实现一个if...else if...else语句块，if中的情况优先级最高，每个else if的情况顺序检查，这就体现了每一个判断情况的优先级：

    ```
    if(exception_temp != 0) nxt = IDLE;
    else if(uncache) begin
        if(op == READ) nxt = REPLACE;
        else if(is_atom_rbuf && op == WRITE && !llbit_rbuf) nxt = EXTRA_READY;
        else nxt = MISS;
    end
    else if(is_atom_rbuf && op == WRITE && !llbit_rbuf) begin
        nxt = EXTRA_READY;
    end
    else if(cache_hit) begin
        if(valid) nxt = LOOKUP;
        else nxt = IDLE;
    end
    ```

    这一段代码中，exception_temp的优先级最高，满足其不为0，则按照软件语言的思维，“下面的比较不会进行"。但在生成的电路中，exception_temp是这个if...else描述的语句块的最后一个选择器的选择信号，最后一个选择器的两个输入是IDLE和uncache信号所控制的选择器的输出，这也就体现了exception_temp无视其他信号，直接决定了nxt的值。如果exception_temp == 0，uncache信号决定的多选器输出才会有意义。
  - 在使用单个if...else if...else语句块的时候，大家完全可以按照C语言的思路来构建电路。但是，对于并列if的情况，请大家参见Episode 2 中的讲解，理解两个if并列的情况。
  - if也需要逻辑完备性，即每个if必须匹配一个else，否则如果if旗下的else if没有把所有可能情况列举完全，则会出现锁存器导致电路错误。

一个文件只写一个module

很多时候，对于比较简单的设计，大家喜欢把不同的module写到同一个设计文件中。但是，这样往往会导致代码文件过长，从而在找错误时容易晕头转向。非常推荐大家"专模块专用"，这样可以非常方便地根据要求对之前写过的模块进行改动，同时可以避免引入冗余的模块。

当然，有些模块是唇齿相依的，属于必定互相依存的，那么可以把它们写到同一个文件中。

RTL分析，一个让bug胆寒的功能

RTL分析，也就是Elaborated Design，是一种高效、简洁的纠错方式。**在写完设计文件后，第一件事就是RTL一下**，看看电路图以及下方Message栏中Elaborated Design文件夹下报出的warnings、critical warnings和errors。下面我对几个常见的问题进行简要分析：

warning：位宽不匹配

> [Synth 8-689] width (1) of port connection 'dout' does not match port width (4) of module 'mux2_1'

位宽不匹配是非常致命的错误（虽然它只是个warning），因为它会导致仿真中出现大量的z的x。例如上面这条错误，我们就理解为，例化mux2_1，本应是一个4位宽的接口dout却接上了一个位宽为1的线路。

warning：空载

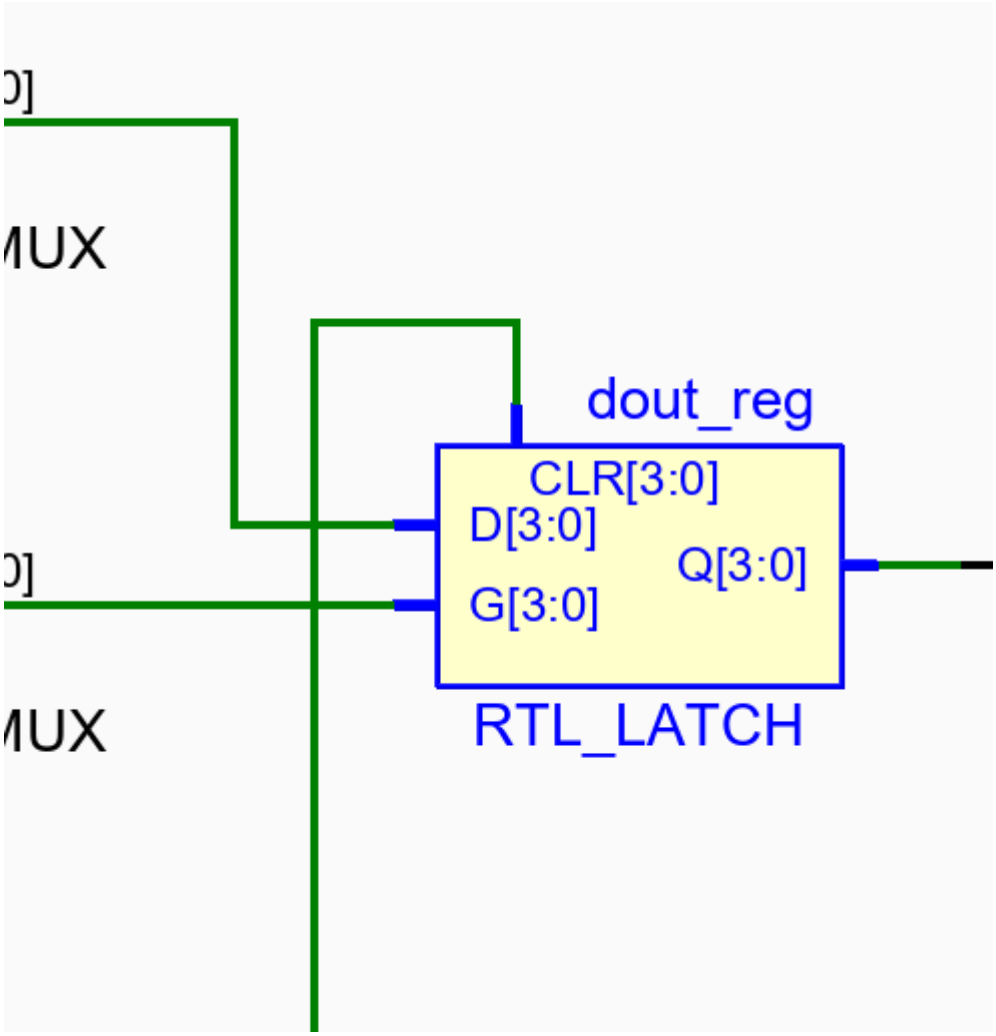
> [Synth 8-7071] port 'din2' of module 'mux2_1' is unconnected for instance 'decoder_input_mux'

空载是指在例化时忘记写某个模块的某个接口，这往往是因为接口太多导致的粗心错误。这里已经写的非常清楚：mux2_1模块的din2接口没有接上。

warning：推断出锁存器

> inferring latch for variable 'psum_reg'

这种错误非常隐蔽，**常常不出现在Message中！**大家要做的是点开每个模块的电路图，看看是不是存在一个叫做LATCH的元件，如图：



学习了计算系统概论大家知道，门控D锁存器只要控制信号不变，内部存储的数据就不会改变。但这并不符合组合逻辑的定义。

为什么会出现锁存器呢？这一定是**逻辑不完备**导致的。如果**case没有default，if没有else**，那么只要你陈列的情况不够完整，锁存器一定会出现。这个锁存器的意义是：**当输入不属于你列举的情况集合时，电路输出应当不改变状态。**于是Vivado使用一个锁存器来存储电路情况。但这已经不符合组合逻辑的定义了，因此**出现锁存器，电路设计一定是出现问题的。我们必须保证电路中没有RTL_LATCH，才能进行后续的测试。**

以上是RTL部分可以报出的常见错误，当然，大家如果仔细看电路，可以看出多驱动、组合环之类的错误，但他们不易被发现，且RTL分析无法检测出（这些错误我们将在下文进行分析）。因此，**不管哪一步出现了错误，我们都应该结合错误信息来看RTL电路图**，因为它是最直观、最简单的纠错方法。

仿真测试，妙不可言

在解决了RTl分析的所有错误之后，进行仿真是非常重要的步骤。这一部分我们将来介绍一下Vivado的仿真

仿真文件的意义在于：给某个模块特定的输入，观察它的输出是否如我们期望的那样。所以，我们需要设置这个模块的输入，来获取输出信息

仿真文件的组成

1. **例化需要测试的模块：**

   这里需要注意：不管是在仿真文件里测试，还是在设计文件中例化，**被例化模块的input接口可以使用reg或wire型去连接，但output接口，不管是output还是output reg的接口，都只可以用wire型去连接。**原因很容易理解：作为一个输出接口，我们应当使用一根线去接入，而不能用一个寄存器来接，因为寄存器是可以存储值的。

   在测试文件中，我们一般使用reg型变量来连接输入接口（因为reg型变量可以非常方便地进行赋值），用wire型变量来连接输出接口（这是verilog的语法要求）

2. **设计测试的流程：**

   我们一般使用一个initial块来进行一次测试。

   - 在initial块的前两行，一般是要对所有的输入变量来赋予初值（否则他们在仿真波形里会出现X，影响我们的判断）
   - 之后，我们需要借助仿真语句来设计测试流程，一条仿真语句组成是：

     > #n 执行语句

     其中n是非负数，是指这一条语句和上一条语句执行的间隔时间。例如我们有如下的测试：

```
initial begin
    input_signal = 4'b0; //这里对input_signal作了一次初始化
    //第一个时间单位内，input_signal为4'b0
    #1 input_signal = 4'b1;//第二个时间单位内，input_signal为4'b1
    #1 input_signal = 4'b10;//第三个时间单位内，input_signal为4'b10
    #1 input_signal = 4'b11;//第四个时间单位内，input_signal为4'b11
    #1 input_signal = 4'b100;
    #1 input_signal = 4'b101;
    #1 input_signal = 4'b110;
    #1 input_Signal = 4'b111;

    input_signal = 4'1111;//这里执行完上面所有操作后，对input_signal又作了一次初始化
    #1 input_Signal = 4'b1110;
    #1 input_Signal = 4'b1101;
    #1 input_Signal = 4'b1100;
    #1 input_Signal = 4'b1011;
    #1 input_Signal = 4'b1010;
    #1 input_Signal = 4'b1001;
    #1 input_Signal = 4'b1000;
end
```

这里我们可以看出，在前八个时间单位内，input_signal每隔一个时间单位就自增1，在后面的时间里，input_signal每隔一个时间单位就自减1，这样就达到了**指定输入观察输出**的效果。

当然，如果你想在某个时间单位内对多个信号进行改变，需要使用begin...end来聚合起来：

```
#1 begin
    input_signal1 = 1'b1;
    input_signal2 = 1'b0;
end
```

不过，如果一定要每个时钟周期都指定信号，对于有规律变化的信号来说，实在是太麻烦了。因此，verilog提供了repeat和forever语句，用以简化仿真。不过一定要注意：**repeat和forever语句仅限在仿真文件中使用！设计文件中不可以使用！**

- repeat语句：

  repeat语句的格式是：

  ```
  repeat(times) begin
      #m begin
          执行语句1
          执行语句2
          ...
      end
  end
  ```

  repeat后括号内的数字是重复的次数，#号后的数字是每次循环间隔的时间，例如：

  ```
  repeat(10) begin
      #1 begin
          din = din << 1;
      end
  end
  ```

  每隔1个时间单位，din左移1位，循环10次

- forever语句：

  forever语句的格式是：

  ```
  forever begin
      #m begin
          执行语句1
          执行语句2
          ...
      end
  end
  ```

  forever一般用于比较规律的长时间变化的信号，比如创建一个模拟时钟信号：

  ```
  initial begin
      clk = 0;
      forever #1 clk = ~clk;
  end
  ```

- 我们总说“时间单位”，那这个时间单位究竟是什么呢？这就要看到每一个设计文件最顶层的一行了：

```
`timescale 1ns / 1ps
```

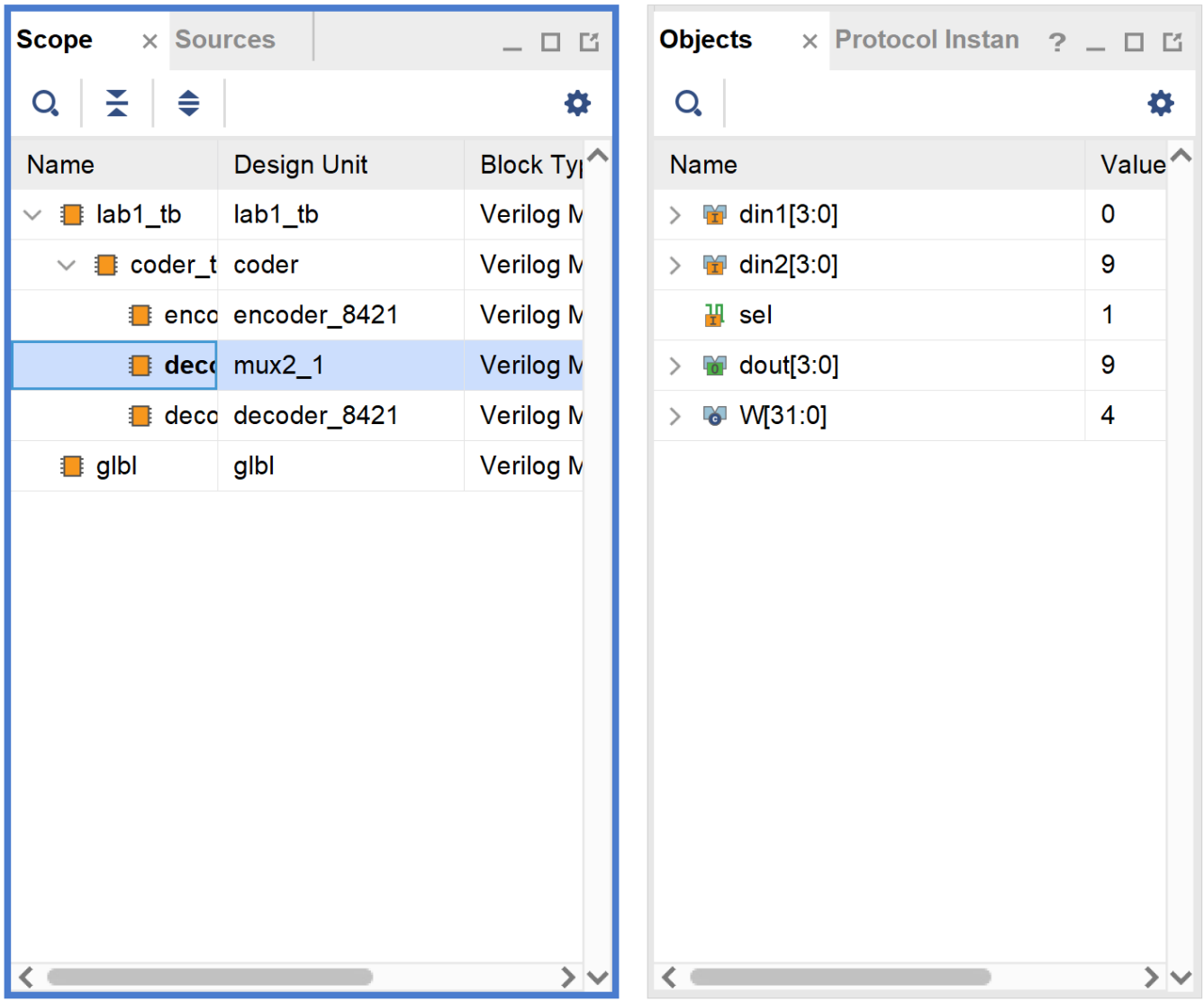
1ns表示每个"#"后面的数字的单位，1ps表示数据精度。如果你写了#3.6452832946384932，虽然它的单位是ns，但这个精度要求太大，仿真只能精确到1ps，在这里，这个时间只能精确到3.645ns。

以上就是仿真文件的基本语法。掌握了这些，你就可以应对基本上所有的仿真需求了。

仿真信号的查看方法

如果仿真信号出现了错误，那么我们很有可能会将其锁定在几个模块中。那么对于顶层模块的仿真，如何看子模块的信号呢?

打开仿真界面，我们看到中间偏左的位置有：



scope中，你可以展开每个模块，并选择你觉得会出现问题的模块。在右侧的objects中，你只需要将你觉得出问题的模块的某个信号拖入右侧的仿真波形图，就可以看到这个信号的变化方式啦！

揪出综合电路中的错误

经过了RTL分析和仿真，你以为电路就没有问题了？非也！在综合时，可能会出现更多的问题。在这里我们介绍两个主流的综合电路错误，供大家进行参考。

值得注意的是，这些问题即便存在，综合依旧能够完成，所以大家需要仔细查看警告信息。

Critical Warnings：多驱动（multi-driven）

> [Synth 8-6859] multi-driven net on pin d_OBUF[1] with 1st driver pin 'd_OBUF[1]_inst_i_1/O'

一根数据线，可以连接多个输入接口，但只能连接一个输出接口。如果连接了超过一个输出接口，那么这根线上的数据是不确定的，即两个模块共同驱动了这根数据线，因此称为多驱动。在这条报错信息里，d这根wire型变量被作为了多个模块的输出，因此是错误的。

解决多驱动没有很好的办法，可以看电路图来找出错误，也可以看代码找到这个变量被当做了哪些模块的输出。

Critical Warnings：组合环（combination loop）

这种错误非常易于理解，就是一个组合逻辑电路块的输出被间接或直接引入到自己的输入，比如：

```
assign dp = dp & 1;
```

但是，聪明的Vivado总会想尽办法自己解决这样的错误，这虽然是一个看起来必错的语句，但Vivado却并没有报错。因此，这就需要大家在电路出现问题的时候，通过查看电路图，来找到一个可能的组合环。

一般地，组合环的报警信息会在组合环较长的时候报出。大家如果有幸见到了这样的报警，那么一定要检查长组合逻辑通路是否存在输出作为输入的情况。

结语

在本辑中，我们讲解了对电路进行Debug的几种方法。我们可以看到，电路的错误一般是由两种错误构成：

- “我怎么这么傻”类型错误：可以通过RTL分析和综合电路来解决
- “哦，原来是这样啊”类型错误：可以通过仿真来解决

希望大家能够通过本辑，总结更多的电路编写经验，在之后的实验中能够自如、顺利地完成各项设计！