整洁代码避免混乱

轻松设计整洁代码

代码整洁之道

干净的代码是程序员的核心技能

如何避免代码混乱

用好的代码实现 0 仿真 0 调试

（注：所有标题中可切入 FPGA 这个关键词）

在正式开始之前，我们先看看一位 FPGA 工程师的工作日常：

开始设计代码

开始写第一个 always 代码

发现要增加一个信号，因此写第二个 always,设计这个新增的信号

回到第一个 always 上，继续完善这个代码

开始写第三个 always 代码

感觉第一个 always 有情况没考虑到

一阵重新思考

回去修改第一个 always 的代码

写完后，得了，不检查代码了，仿真再说吧。

仿真过程：

每个时钟上升沿一个一个检查

发现这时某信号没有变高

检查代码，把 BUG 补上

继续检查波形，继续补 BUG

发现信号 A 和 B 时序对不齐

思考是打补丁呢还是打补丁呢

是改这个信号呢，还是改那个信号，还是加一个信号

一番折腾后，终于对齐了

修改测试文件，再测试

还是有 BUG，继续打补丁

该上板调试了

系统跑一会没问题，长时间跑就出 BUG

用调试工具各种分析各种定位

一番折腾后，终于找到 BUG

一个 corner 没想到/粗心大意漏了个条件/

早知道，要没这 BUG，我早就做完了

又出现 BUG 了，又要来折腾啦。

这个场景是不是觉得很熟悉？还有下面这些情形也许都遇到过：一个项目看上去很简单，精心设置了架构，结果越做发现冲突越多，直到整个逻辑完全混乱。本来一天可以的完成的事不知道怎么搞的一个星期还没有完成；本来只需要做一行更改，结果却涉及到 N 个模块；出现了一个非常小的 BUG 打了一个补丁，然后补丁越来越多，到最后

无法解决。诸如此类等等情况不一而足，究其原因，总离不开“混乱”两个字。这些混乱的根源是什么？又该如何解决呢？

一个好的 FPGA 项目的设计作品，不仅依赖于架构设计，优秀的代码也是必不可少的关键因素。而好的代码最基本的就是清晰整洁。整洁的代码运行稳定，也是后期维护和升级的基础。正如 C++语言发明者 Bjarne Stroustrup 说的那样：“代码逻辑应当直截了当，叫缺陷难以隐藏；尽量减少依赖关系，使之便于维护；依据某种分层战略完善错误处理代码；性能调至最优，避免其他人优化时不知所措从而出现混乱状态。整洁的代码只做好一件事。”

这段话说得实在太好了，整洁的代码只去做好一件事。事实上，有两点只要做到了，就可以大大提高自己代码的整洁度。第一、写简单的代码；第二、把复杂的代码简单化。下面我们通过一个小的实例来说明一下。我们先来看这样一组代码：

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n==1'b0)begin
3          shi_ge <= 0 ;
4      end
5      else if(((set_flag == 1'b1 && set_sel == 4)&& (key_vld == 1 && key_num ==
       4'b0010)) || shi_ge_add)begin
6          if(shi_shi ==2 && shi_ge == 3)begin
7              shi_ge <= 0 ;
8          end
9          else if((shi_shi == 0 || shi_shi ==1) && shi_ge == 9)begin
10             shi_ge <= 0 ;
11         end
12         else begin
13             shi_ge <= shi_ge + 1 ;
14         end
15     end
16 end
```

这个程序时一个数字时钟功能的其中一份关于小时个位的代码。

小时个位复位等于 0（第 3 行代码）；设置的语句（第 5 行代码），意思是当你选中小时的个位并且按键按下去，小时个位+1，或者说正常情况下一个小时+1。这里需要注意的是：首先小时的计数方式在 0：00——9：00，10：00——19：00，20：00——23：00 情况下+1；另外几个时间点清零。

我们来分析一下，在这份代码的设计中需要考虑到很多因素。第一、需要考虑按键；第二、按下去时与正常计数的关系；第三、需要数多少次清零，比如说 9 点、19 点、23 点清零；当很多因素混在一起去考虑，特别是格式没有被规范的时候，就容易出现混乱、遗漏点或是相互之间出现冲突，出错的可能性随之变大。

接下来我们来看另外一组代码的思路和操作。

首先，我们建立一个通用的计数器模板，命名为 jsq。每次遇到计数器，只需要输入 JSq，即可调入该模板。（注：关于模板的设置以后章节介绍）

```
1  always @(posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          cnt <= 0;
4      end
5      else if(add_cnt)begin
6          if(end_cnt)
7              cnt <= 0;
8          else
9              cnt <= cnt + 1;
10     end
11 end
12
13 assign add_cnt = ;
14 assign end_cnt = add_cnt && cnt== ;
```

接下来设置什么时候个位+1，分为两种情况：1、按键按下去；2、

自然计数+1；（第 13 行）

采用变量法设置 X-1；即先不用去管数多少下，反正数完就清零；（第 14 行）

最后我们设置数多少下。20:00 时数 4 下;其它时候数 10 下;（16~21 行）

```
1  always @(posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          cnt <= 0;
4      end
5      else if(add_cnt)begin
6          if(end_cnt)
7              cnt <= 0;
8          else
9              cnt <= cnt + 1;
10     end
11 end
12
13 assign add_cnt =((set_flag == 1'b1 && set_sel == 4)&& (key_vld == 1 && key_num ==
   4'b0010)) || shi_ge_add ;
14 assign end_cnt = add_cnt && cnt== x-1 ;
15
16 always  @(*)begin
17     if(shi_s == 2)
18         x = 4 ;
19     else
20         x = 10 ;
21 end
```

现在我们来回顾一下这段代码，从中不难发现，设计的总体思路有着严密的逻辑和步骤，并采取了便捷工具（模板）来规范了代码编写，减少了设计量。最重要的是设计者的意图清晰了然，控制语句直截了当，代码之间相互依赖性非常低，作者之外的开发者阅读和增补非常轻松。

这一节我们讲到了代码混乱的根源及解决这个问题的技巧，下一节我们要讲到的是简单代码规则的技巧。