有人认为我验证做得很牛，也有人认为我的验证早就丢下了；有人认为我发现了各个项目的不少问题，也有人认为我在 CMM 库的几百个问题单大部分属纯净水。

好吧，无论怎样，我还是把我在验证中如何发现和定位 Bug 的思路稍微描述总结一下，纯属灌水。以前华仔曾经叫我写过一次，我随手写了一点点，这次还是详细一点吧，主要分几点：视角、技巧、思路、经验。

这里主要还是共享给验证的同志们，但对设计的同志其实我觉得是没有什么差别的。

目的：

发现 Bug，发现所有的 Bug，或者证明没有 Bug，是验证存在的唯一目的。无论任何验证语言、任何验证环境、任何验证方法学、任何 Feature List，都是为了达成这一目的而使用的方法，或者所手段。偏离了这一目的任何工作和努力，都是屎、大便、Shit。

绝对不要被任何华丽的技巧、方法、经验所迷惑，无论验证环境有多么美丽，无论验证语言有多么的 High Level，都不要迷惑。不要为了追求完美、高效的环境而沉迷其中，陷阱往往就在美丽的后面。有时候，最简单的，才是最直接的，任何武术，直拳最有效。

以 SV 为例，SV 有高层次的语法和结构，能够更大限度发挥激励的控制和 Random 测试的效率。但是对于发现 Bug 的目的而言，它只对其中的 20%目标达成有突出贡献，而剩余的 80%，其作用和普通的 Verilog 并无二致。当然，我不是指要放弃 SV，因为其有效贡献的 20%工作，是普通 Verilog 很难或者无法完成的工作。OK，所以顺便涉及另一个问题，设计人员需要学习 SV 吗？有多少设计人员能够在检视或简单 UT 中发现 80%的 Bug，而需要 SV 去完成最后 20%？不要看见别人用 SV，就屁颠屁颠地跟潮流，想清楚 SV 能为达成最终的目的带来什么贡献才是关键。设计人员和验证人员相互沟通，真正的障碍是验证方法学，而不是验证语言。

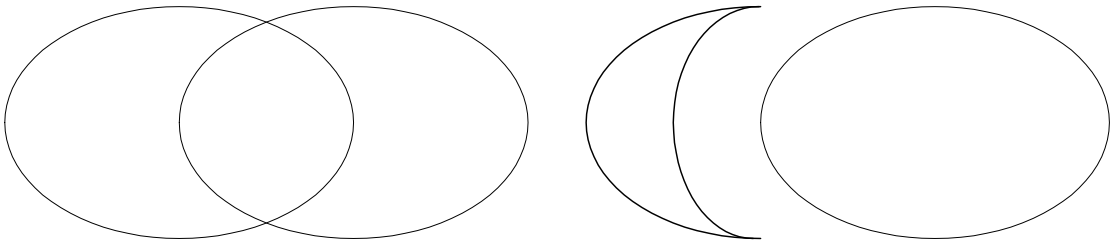
以 TC 为例，对于一个验证人员，跑通全部 TC，意味什么？代码覆盖率 100%，意味什么？验证差不多完成？在我看来，相当于验证工作大致完成了 90%，而有一句老话怎么说的？行百里路，半九十。也就是所，实际上剩下 10%，才是最艰辛的工作。也许某条 TC 什么也没干，然后因为什么也没干而 Pass 了，或者没有实现验证者的意图，所以也 Pass 了。

只有，而且也只有，有充足信心证明全部 Bug 被发现、或者没有 Bug。但这个充足的信心怎样说明？后面我再详细说明。

视角：

有多大的视角，就能发现多少的 Bug。引用 CCTV 的一句台词，心有多大，舞台就有多大。

我比较不喜欢看到的，就是一个验证人员跑来告诉设计人员，说某某 TC Fail 了，波形在 XXX，请分析。我不能认定这位验证人员的工作是否合格，只能表达强烈的情绪，特别是最后发现 Fail 的原因是验证环境问题的时候。这种验证人员，对设计人员、项目经理，都是巨大的风险。因为设计和验证，是一定需要有交集的，并且耦合越大，风险越小，只能提 Feature、写 TC 的验证人员，就像初三的新月一样，反而需要别人去耦合，如果设计人员视野不足，野心不够，就存在空隙了。



一个验证人员，如果能够发现设计中的 Critical Path 并告诉 PR，一定不会得到批评，反而会在实现工作中得到更多的发言权，和更多的发展。一个验证人员，如果仅仅只能跑写 TC、跑 TC，那么多年得不到晋升恐怕也怨不得别人。

OK，回到原点。验证人员必须要懂得代码，懂得分析逻辑，甚至能够通过代码分析出可能的疑点，更好的，能够理解整个系统的运作，理解前端后端的实现，找出设计人员视角的盲区，才能更好的发现 Bug，解决 Bug。

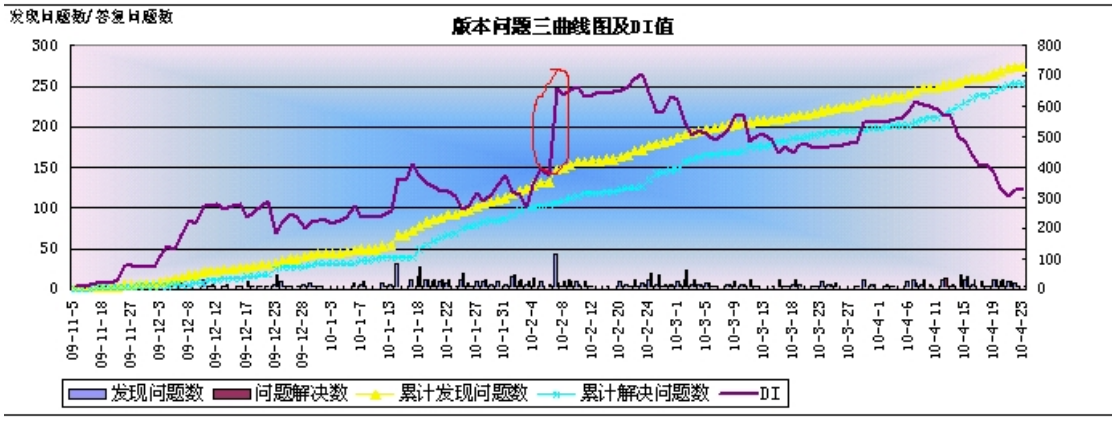
当然，某些同志会认为，验证人员，发现实现的问题耽误了主业，而且实现的问题，实现人员更容易发现。OK，这里同样存在一个视角的问题，你的视角和实现人员的视角是不一样的，也许觉得很容易发现的问题，恰好别人不容易发现呢？反过来说，实现人员或设计人员还可以觉得代码 Bug 对于验证人员是很容易发现的呢。此外还有一个时间成本的问题，任何问题，遗留的时间约长，代价越大。

所以我说一句，验证人员，一定要放开视角，努力去看你所能够看到的，然后，你能够看得更多。然后再补充不务正业的说明，验证人员的目的是发现 Bug，这是唯一的目的，不仅仅是一个 TC 所能发现的 Bug，而是整个芯片可能存在于任何环节、任何位置的 Bug。只有芯片的成功，才是真正的成功，而一个 Bug，就可以毁掉一个芯片，而覆巢之下，安有完卵？

当然，验证人员会问，整个芯片太大了，扩展视角，不是不努力，而是看不到啊。OK，我再说一句，对于验证人员，最简单，最真切的视角就在脚下。TC，每一条 TC，每一个 TC 的波形，都代表了芯片中的全部或部分，真实运作的场景，有血有肉。如果把波形当作 TC Pass 的附属物，那么，恭喜，验证人员，你拿了芝麻丢了西瓜。波形真的可以告诉你很多、很多。

我甚至可以公布我做验证的时间分布（不包括最初搭建环境的时间），20%时间写 TC，10%时间调环境，50%时间看波形，确认 TC 达到我想要的意图（TC Log 中的 Pass？噢，对不起，这种狗屎信息我向来忽略），剩余 20%时间？对，剩余 20%的时间，是我固定的，从当前表面上正确运行的波形中，对照代码，寻找其他可能发现的时间。

不要跟我说现在系统太复杂，看波形效率太低。OK，Hi1380 的系统复杂不？整个波形我也从头到尾看过啊。而且，就在我看波形的第一天，就是从一个已经 Pass 的，好像是 GIC 的系统验证波形中，拿到了超过 20 个问题单（加上代码检索的 30 个问题单，创造了下图中陡升的曲线，不过可惜了，没能突破 300）。



淘金的执念：

缺陷就在哪里，静静地躺在哪里。没错，一定在，而且马上就能看到!! 执念，这是一种执念!! 作为验证人员，一定要有这种强烈的，不可动摇的执念或者说饥渴感，而且是和设计人员强烈对抗的执念。

实际上，目前看到的所有芯片，都已经证明，投片后，依旧有缺陷遗留在其中，没有被发现。所以，这种执念，无比正确。只有疯子，才能发现隐藏得最深的金子。

我开始做设计之后，这种执念消失了很多，总是希望系统确实在完美地运行，失败，很是失败。不过对他人设计的模块，以及不是我负责的项目，这种执念还是非常强烈，呵呵，这也是我在 1380 和 P600 中疯狂创造问题单的原动力。

这跟淘金的人，可能是差不多的。金子就在这里，一切的希望都在这里，再挖一锄头，就找到了。只有疯子，才能成功。

淘金的技巧：

指定找一块地，疯狂地朝下挖？No，No，疯子都会 B4 你，淘金也是有技巧的。

很多方法，其实说白了，很简单的。

表层的土是最容易挖的，那么，别人没有挖过的地方，最有可能在表层找到金子。为什么别人没有挖？很简单，盲区。两种盲区：

1） 明明每天都能看到，却没有人想到去挖一挖的地方。以 1380 为例，天天都有人跑 ARM，就硬是没有人去分析一下 ARM，如果最开始，就能多看看 ARM 的 ACP 代码，宝藏啊，宝藏啊，ACP 虽然没有错，可是上游会冲下来多少金子积累在这里啊。

2） 别人不屑于去挖的地方。IO_CFG 简单吧？TEST_MUX 简单吧？TOP 层的 IO 互联简单吧？不屑啊，多少验证人员对此不屑一顾啊，那眼神就是在说，真 TMD 没技术含量。是啊，再 TMD 没有技术含量，也是金子啊。

对于表层土的挖掘，不要太固执于一点，广撒网，多捕鱼。如果十锄头都没有挖到金子，马上换地方，对于别人刚挖过，还没深挖的地方，也来上几锄头，说不定就可以让前一个家伙悔到死。

表层土都差不多了，就需要找关键部分深挖了。如何找关键部分，是非常讲究的事情，兼顾风水、心理、外交、直觉等多方面知识，很难给出综合性的分析。下面几点可作为 Hint：

1） 如前面所说的，只知道跑 TC，跑错后让设计人员定位的验证人员负责的区域；

2） 对实现没什么概念的设计人员设计的模块；
3） 责任人变来变去的地方；
4） DFT 相关的地方（验证人员的 DFT 知识严重缺乏）；
5） 规格老是变来变去的地方及其可能影响的地方；
6） 第一次做代码集成人员连接的顶层位置；
7） 浮浮躁躁、毛毛糙糙的新员工负责的地方；
8） 时钟域（几乎当前所有的验证人员都不关心时钟正确性，只要能跑 TC）；
9） 所有人都认为没有问题的地方；
10） 验证人员宣称放弃的地方；
11） 技术难度比较高的地方；
12） 你以前项目发生过问题的地方（相同或类似的问题很大几率存在）；
13） 整个系统中相关性非常高的一连串区域；
14） 协议和时钟转换的区域；
15） 其他隐藏在内心深处的秘密。

需要注意的是，在挖掘这些 Hint 点的时候，并不一定能保证挖到金子，而且即使有金子，你也并不一定能够挖到，人品，人品很重要。

OK，关键部分都挖得差不多了，剩余的金子基本上就埋藏得比较深了，这个时候发现的金子都将比较可观。再不济，也能够成为荣誉奖、星星奖之类，要搞得恰当了，直接拿 A 也不是梦想。当然，如果没能发现金子，一无所获的可能性也很大。

收益和风险是成正比的，淘金人在这个阶段一定要能够沉下心来，冷静思考。楼上提了这么多 Hint，那个地方还比较薄弱？整个项目统观下来，还有哪里有薄弱的？你在思考，项目经理也在思考，验证经理也在思考，SE 也在思考。如何超越项目经理、SE、验证经理的思考发现金子？我非常、非常难以回答。提供我已有的两个经验是：

1） 反向思考。最后阶段，大部分人员的思路都已经固化了，像一条绳子一样，不断的朝一个方向缠绕、缠绕。反向的思考往往能突破这个限制。当然，反向思考这个东西，很多时候就是忽悠，难以做到。我的一个经验是可以多听取一下局外人的一些意见，例如软件人员的意见。当然，这其中大部分的意见都是无关痛痒的瞎扯，但偶尔、偶尔会出现一些能够引发进一步思索的缺口。
2） 和谐。这里没有任何问题，芯片运作一切正常，没有任何差错。但是你拿着架构图看、或者拿着时钟结构图看、或者打开最复杂的 ST 波形看，心中却总是有一种说不清道不明的感觉，没错，虽然一切正常，但是某个地方，却有那么一点点不和谐，就像合唱团中插入了一个走调的家伙一样。可能是非常微妙的一个路径，可能是波形上非常诡异的一个脉冲。对了，就是这个地方，追下去，即使工作正常，这里也可能存在和设计意图不符的东西存在。

书签

开门红：

根据规格分解 FeatureList，根据 FeatureList 对应 TC，然后再一条一条仿真 TC 反过来

映射 FeatureList 和规格。没错，这是最通常的做法，可惜我不这样做。

世间有 80：20 原则，验证也是，80%的问题都可以通过 20%的测试和时间去发现和解决，而剩余 20%的问题需要 80%的测试和时间去解决。

所以，按照我的思路，会有几个最初级的 TC，可以用来测试最基本的通路能否冒烟，这几条 TC，可以划归到 TC List 中，也可以不划归。然后，一定有一条开门的 TC，这是一条复杂的 Directed TC，一条可以覆盖 70%的 Feature 的 TC。

这条 TC 并不负责任何 Corner、异常覆盖，不做任何特殊的思考，一切都是直接对 Feature 的连续描述（也可以是若干条 TC 的直接串联），因此即使有些许问题，修改的难度也比较低。

这条 TC 能够帮助设计人员定位超过 70%的问题，如果设计人员足够聪明，这个 TC 可以解决 90%的问题。

这条 TC 的寿命可能将超过一个月，这一个月足够设计人员在其中沉沉浮浮，使得代码达到 95%的交付情况。而验证人员在这一个月中，有足够的时间完善 Corner 的 TC、Random 的 TC 和环境，然后集中精力完成剩下 10%问题的解决。

检视：

代码检视是最容易发现问题的步骤，从写第一行代码开始，到最后一个 Tag 结束，都是如此。

代码检视不仅仅是设计人员的事，也是验证人员的事。我知道很多人都不认同这样的观点，正如我不明白为什么有些扫一遍代码就能发现的问题，有些验证人员还那么兴致勃勃、废寝忘食地编写 TC，然后再辛辛苦苦跑 TC 来发现一样。

正因为我做过设计人员，所以我感受非常深刻，设计人员绝对都是极度乐观、自信的，特别是代码刚刚完成那一霎那，瞬间的快感，Oh，凤姐啊，芙蓉啊，让设计人员全身都在颤抖。破绽啊，这里有太多的破绽了。

所以对于新交付的代码，按照我的经验，建议验证人员先检视（尤其是设计人员是两年以内设计经验的），不过，这个检视绝对不要是傻看代码，要跑一条 TC，最简单的，就一个读写就可以了，保存所有信号的波形，然后打开 Verilog 代码，对照着波形检视。

1） 所有信号全部抓出来看一遍，红色的（X）、黄色的（Z），简单确认一下，然后 Alt+Tab，切换到 CMM 页面即可（百试百灵，至今为止从未失手，nLint 不是万能的）。

2） 模块间的握手信号，全部抓出来看一下，是脉冲信号还是电平信号（98.765%的设计人员，都不会在信号名上注明是脉冲或电平），脉冲信号是必须立刻采样的，电平信号是需要鉴沿的，如果握来握去，如果还有异步，基本上，检视出问题的概率非常高。

3） 在一个 always 中，对多个信号赋值的；在一个 always 中，elsif 数量超过 6 个的；在一个 always 中，if 的条件组合中包括超过 5 个信号的；都是高产田。

4） 协议理解上和自己理解相异的，例如对于我，AXI 相关设计未按照我《AXI 总线设计的二十一条忠告》的。

代码，是设计人员思路的直接映射，而设计人员的思路，有时候真的是一根筋。通过检视，或者加上设计人员的讲解，可以直接了解一下到设计人员的思路、逻辑思维模式，非常有助于去构造一些检测其思路正确或不正确的点，验证人员的思维其实很简单，抬竹杠就好。

请读者回忆一下刚刚经历过的项目，在 100%网表之后，是否有好几个 ECO，都是从检

视中出现的（代码或脚本或 TC）。而每一个这样的事件，都是那么神奇的偶然。可能是某位新员工周末偶然在学习老员工的代码时发现异常；可能是某人在项目经理逼迫下第三次检视某个模块时，惊讶地发现了一个低级错误；可能是某个 IP 交付团队某天突然想起说有一个连接的错误忘了改正，但幸好在 ECO 前发现。反正，总是奇迹一般，让项目经理觉得自己是世界上最幸运的项目经理。

明白了吗？

我经历的多个项目，每次都有这样的奇迹，其比例占 ECO 的约 30%～50%，这不是偶然，是一个说不清，道不明的必然。也许，只是 100%后，同志们有更多的时间投入检视而已。

怎么能不检视？

检视，在任何时候进行，都不算早，也不算晚。

OK，这里就扯到另一个话题，某些同志经常反馈，检视工作非常不受待见。不做，没人管，做了，看不到绩效，即使检视出问题，也会有人跳出来说，“这么简单的问题啊”，特没成就。OK，我认为这是管理问题，典型的。无论对于海思的投资团队，还是对于项目经理本身，用最小的投资，或者说最小的人力、最短的时间，努力去发现项目中的问题的活动，难度不是最应该鼓励的吗？OK，如果你真有这样的感觉，我的建议是，将检视问题提问题单，再不济也是一个严重，发现阶段为 ST，不用觉得害臊，害臊的应当是管理者。

此外，到最后阶段的检视，对于验证人员，可能需要更加地扩大视角，围绕代码为核心，TC、波形、脚本都需要涉及。对于这里，我还是再强调一遍吧。验证人员拥有设计人员所不同的视角，所以一定能够发现潜藏在其中的问题，对于单个项目而言，验证人员会认为是一个偶然，而从多个项目而言，我的经验，这是一个必然。

检视的经验：

OK，这里我可以再补充一下我个人检视代码的经验，我的步骤如下：

1） hdl_stat 统计整个模块代码行数，及各个子模块代码行数分配；
2） 打开代码顶层，快速浏览整个代码，获取这几个信息：代码风格、设计人员的思维成熟度、代码结构、重用度、逻辑类代码和集成类的分布、关键 C Path 和关键 D Path 的位置；
3） 按照现有的经验，其实已经大致能够推断出该模块整体的缺陷数量和缺陷分布了；
4） 先简后难，先扫除直接就能够看出的 Bug，这种 Bug 分布比较散，没什么特别的依据，但很多问题，真的很简单，就在设计人员鼻子底下（不要超过 2 小时）；
5） 用 Verilog 构造最简单激励给模块，保留波形供对照，如有疑问，更改激励再仿（不要超过 3 小时）；
6） 然后，因为我有设计经验，我就会思考如果自己是设计人员，我会怎样划分模块、描述关键控制逻辑，如果和设计者不符的地方，着重分析；对识别出来的关键控制逻辑，例如异步握手、堆栈、链表、数据拼接，静下心来，慢慢看，慢慢看。对于疑点，构造简单激励，出波形对比。

OK，我拿我曾经的一个模块 Security Engine 代码作为实例，如果我进行检视如何进行。

1） 代码行数约 17000，行数最多的集中在几个整体控制模块：sec_ctrl（2235）、sec_slave（2121）、sec_channel（1669）、sec_master（1193），除了这几个大模块，几个算法都分散为非常多小模块，相互调用搭成 aes_core、kasumi_core 等算法模块被顶层调用。

2） 第一轮检视，快速浏览。各个算法模块的输入输出非常干净，都是通过 run、done 进行握手，其内部都是轮运输，通过 round 控制，其中 aes_core 还是纯复用以前项目的模块，而顶层几个 ctrl 模块，则相互交互非常复杂，特别是代码数量最多的模块，数据交互特别复杂的 sec_ctrl 和 set_master，居然没有状态机，看起来设计人员是希望通过自己的逻辑思维，直接描述其控制；而 sec_slave，纯寄存器描述，而且是大量复制代码搭建，技术含量低；结构上，sec_channel 是一级控制，sec_ctrl 和 set_master 是二级控制，各个算法 Core 是三级控制。代码风格上，设计人员部分遵守代码规范，但很多地方自以为是，为了自己方便写了不少擦边球的代码。

3） 分析，aes_core 理论上缺陷将很少，而其他几个算法 Core，如果 round 控制上没有发现错误，那么错误通过 RM 比对验证，效率更高；然后，sec_channel，可以着重关注状态机和状态机对应的控制信号是否正确；最后，sec_ctrl 和 set_master 的交互，一定是关键，特别是代码量大的部分。

4） 第二轮检视，对算法 Core 的各个 round 控制信号检视，是否符合 run、done 控制；对 sec_slave 的寄存器读写控制检视，是否有笔误和拷贝的错误；检视整个代码集成和互联；简单查看其它代码中 if 和 else 比较复杂的地方，记录可能的疑点。

5） 构造一条 TC，正常而言，是通过 sec_channel 调配 sec_ctrl 完成一次算法运算（用最简单的 Verilog 搭建 TB，超过 2 小时是否非常失败的事情）。根据波形，将 sec_channel、sec_ctrl 和 set_master 主要逻辑，全部过一遍。

6） 关键逻辑，重点关注，关键疑点，修改 TC，重点覆盖。

再谈检视：

首先引用一个对检视的不同观点：

review 真的最有效吗 or 导致更多的 BUG?

review:中文叫评审。本人见过这个做法的最早出处是朱兰的质量手册。在很长一段时间被软件行业认为是最有效的保证代码质量的手段。这段时间的质量高压之下,我们再次见到了红红火火的各种代码 vreview,自检,互检,飞检,X 检。这让我想起了考试,考试完了都要自己检查几遍再交卷。(当然是在能够把题目做完的情况下),偶尔我们也会在考场上互检(不过这个可能属于作弊)。不过从以上最简单的例子可以看出,互检应该比自检效果好很多,不然也不会有很多学生冒着风险去互检了。

但是:上周在和敏捷顾问一起参加一个项目组的回顾会议的时候,发生了这样一个状况,大家在讨论下轮迭代需要改进的时候,都提出来要加强 LLT 测试用例的 review,顾问一直追问我们为什么要这么做？是不是上轮迭代的结果出了什么问题？我们这样做能够带来哪些改善？我们的 UT 一直做得很弱,顾问非常奇怪为什么我们不多花些时间做 UT？而要花时间去做 LLT 用例的 review？当问到从迭代结果上除了什么问题而导致我们想加强 LLT 用例的检视的时候,大家都找不出直接的证据,只是说:如果不评审,风险会很大。(但是上轮迭代的最大问题大家已经搞清楚了——是:低层 BSP 没有人力投入,导致其中 4 个相关的 Story 无法全部完成。

我也很差异顾问提出来这个问题,他继续讲:review 有可能是一种浪费。my lady gaga! 我自己从来没有听说过 review 可能是一种浪费。顾问为什么能够提出这样的疑问？

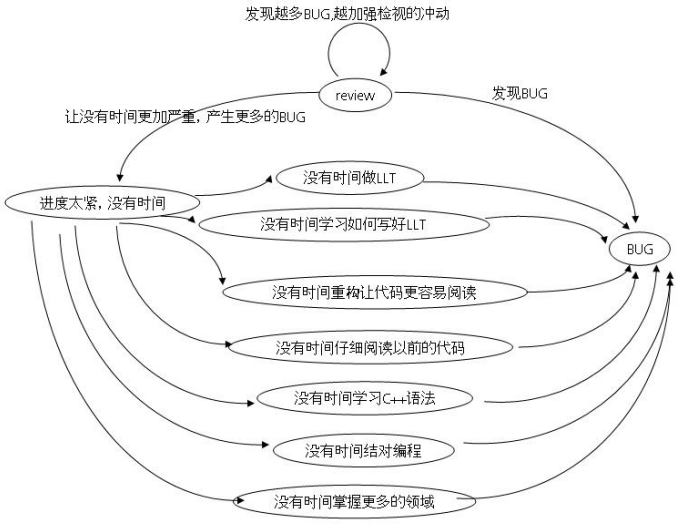
其实这段时间经常会收到不少项目组发的邮件,宣称自己团队组织封闭检视又发现了几

百个问题,似乎是这样做还能得到一些表扬。当时,我内心深处觉得有那么些不对劲,团队 1 个月的编码工作,怎么能在短短半天的时间里面就可以检视出这么多问题？这也许说明了检视有效,但是是不是更加说明我们前面的工作并没有做好呢？(再次 Oh,my lady gaga,我自己怎么都对 review 产生了怀疑。)

后来忽然在温伯格(«软件程序开发心理学»«顾问工作的秘密»等书的作者)的一本书籍目录中看到了这样一句话,"任何质量措施,益早不益重"。但是非常遗憾,我没有看到全文,只能根据这句话来进行推测和分析(在信息不完整的情况下进行分析是敏捷教练应该具备的能力之一——someone said)。

好吧,我们来看看 REVIEW 的效果吧。优点:review 能够发现问题。缺点:review 无法象测试一样可以重复性地保证缺陷没有被引入。

另外:我们可以按照下图方式画出 review 和 BUG 的系统控制图:如果我们在如下场景下加强 REVIEW,将会进入一个非常有意思的循环,加强 REVIEW->发现问题->占用了时间,更没有时间做测试等等->生产更多的问题->review 会发现更多的问题->大家认为 vreview 很有效果 -> 大家会用更多的时间 REVIEW-> 于是产生更多的 BUG^^^^^^^^^^



神啊，GAGA 啊，感谢这些不同的观点吧，正是有了不同的观点，才让我们能够更加深入讨论的机会。

在我写完前一期的经验总结后，接收到了很多关于检视的不同的观点，很耐人寻味的是，这些反面的观点，基本上全部来自于从 Intel、BroadCom 等公司背景的高端同事（当然也包括了我引用的这位外籍专家），在这些外来的，充满魅力、经验丰富的男人们看来，强调检视是属于海思（从引用的专家的指向，整个华为也是如此）的专利，而被人直接用肉眼发现代码中的缺陷，简直就是人生中的奇耻大辱，真正保证设计正确的，只能是有激励的仿真。

其实我非常赞成楼上最好的一个描述，真的，非常赞成。检视，只是检视，只是能偶尔地，发现一些错误而已，检视什么也保证不了，检视既不能保证 Bug 被全部发现，也不能保证这些偶尔发现的 Bug 能被修正或不被重犯，更重要的是，检视根本无法保证设计最终能够正确运行，最终能够保证设计正确运行的，只有仿真。

但是，为什么我们常常从检视活动中受益呢？难道真的如我引用的文字所述，我们也陷入了强调检视，而弱化测试，然后检视出更多 Bug，而更加强化检视、弱化测试的循环？

下来后，我进行了反复思考和交流，得出了我的一些观点，以供参考。OK，我并不是想驳倒前面的观点或证明什么，不同观点之间的激荡间，我相信读者自有能力分辨和获取自己的观点。

首先，那些经验丰富的男人们，也是会进行代码串讲的，他们不把这个算成检视（我们算检视），而算设计流程的一部分。

OK，这一点点误解只是一个欲盖弥彰的借口，呵呵，连我自己都害臊了。

我真正的观点之一，是我们的设计人员，还不够成熟。在国外，十年甚至二十年经验的编程人员比比皆是，所以，这些经验丰富的男人们，周围也是经验丰富的男人，而在国内，十年或者二十年经验从业的，只要还没累死，基本上不是领导就是架构师了，写代码？还是给新员工锻炼的机会吧！所以，我们的代码，往往都是经验小于 3 年的人员编写的。我妄自揣测，在海思内部，编写逻辑代码（非 C、JAVA）超过 10 万行的，屈指可数，超过 20 万行的，很可能已经绝迹。在这种 2～3 年经验设计人员编写的代码中，认真检视一下，发现问题，其实并不是件很难的事。嗯，我并不是在嘲弄我们的设计人员，因为，一个人的成长，一定是需要经历磨练的，正如《成长的烦恼》所演绎的，没有人能够一天就长大。我自信我不是很愚钝的人，但我也是整整写了 3 年的代码之后，才真正醒悟：一份代码，在写完之后，一定要再经过一次或多次整理和打磨，才能算完成的；一份代码，一定要把其有效代码行，精简、锤炼到最少、最短、最有效，才能算完成的。前一个月，海思有一位叫聂世玮的兄台讲过一次 Verilog 编码，也是这么个道理，只是不知道有多少听者真正明白了，不过不明白也没关系，再经历几年磨砺，如果还在写代码，就明白了。所以，我们的代码，还需要检视。

我真正的观点之二，检视，并不能替代仿真，甚至不能让整个仿真的动作有任何减少，但检视确实能够缩短整个仿真的收敛时间。因为，我们设计人员不成熟的同时，验证人员同样不成熟。我看过太多的验证人员，随机向量跑出来一个 Bug，然后和设计人员抱着显示器追波形，太阳升起，然后落下，再升起，再落下。如果定位并解决一个 Bug 平均需要半天（真不算慢，虽然那些经验丰富的男人们认为这只是分分钟的事），那么一个设计如果有 40 个 Bugs（不是夸张，比这数字高的多得要命），光定位就花掉了 20 天，还不算期间项目组例会、设计组例会、验证组例会、攻关组例会等等的时间。真是累啊，不是兄弟们不努力啊，而是敌人太凶残啊。所以，我们要检视，只要通过一整天的时间检视，能够发现其中 1/4 的 Bug，就赚大发了啊。当然，检视发现的问题，一定是需要仿真再覆盖确认的，仿真的工作，提取 Feature，写 TC，一个都少不了，但定位和回归，可以大幅缩短。

我真正的观点之三，检视，能够偶然发现某些仿真遗漏的重大问题，对，是偶然。但是，我也曾经说过，一次是偶然，两次是偶然，三次就是必然。验证空间是无限大的，而如何使用有效的仿真覆盖无限的空间，我们依旧经验不足。所以，这里还是有检视的空间。

基于观点一、二、三，所以我认为，检视，能够很大几率缩短项目收敛时间，并发现一些隐蔽的问题，颇具投资价值。

检视很重要，但千万不要迷恋检视。检视的成功，是典型的不可复制型。所以，检视，可以成为流程中的一个步骤，但绝不是流程中用以保证结果正确的方法，换句话说，我们可以定义一个设计需要检视多少次，但绝对不应当定义这个设计的 Bug，有多少比例应当在检视中发现。因为，影响检视效果的因素太多了，天时、地利、人和，缺一不可。某位检视人员前天和老婆吵架了，或者会议室空调太冷，都可能使得检视效果发生天翻地覆的变化，所以，不可依靠检视。如果检视效果不佳，一切还是只有靠仿真。

这里，再共享一个我检视的经验。我把检视分为私下单 P 的检视和会议室中的群 P 的检视，单 P 的检视，一定要结合波形进行，后面专门讲，这里讲群 P，群 P 的大部分时间，我都是精神不振的，因为一群大男人聚在一个小屋子里，实在没啥好鸡动的。等待，耐心等待，一旦讲解代码的设计人员语速放慢了，或者出现迟疑了，发出了例如“嗯”、“这个”之类的拖延时间的助词，马上提问，要求讲清楚，不清晰的地方反复提问，要追根究底，飞流直下三千尺，语不惊人死不休。能确认的，现场就可提单，有疑问的，下来对照波形 Check。

最后，Remember，我们的目标只有一个，发现 Bug，检视只是实现这个目标的一个手段而已。刀在手，是屠夫还是侠客，存乎一心。

重要的是人

项目是人做的，设计是人做的，验证也是人做的。IT 行业，是人的行业，是创造的行业，工作是以有效性，而不是以记件式来衡量。工作就必然是受到个人影响的，具体个体特征创造活动。

所以，重要的是人。

我的观点一，让合适的人做合适的事情，以及合适的搭配和互助。OK，我承认这是 Absolutely 的空话，所以我只能贡献一下我自己的经验。我个人比较讨厌从上到下强制的任务分配，虽然有时候这是紧急或迫不得已的，并且服从团队的利益，也是一个合格工程师的必备品质，但我还是希望，在大多数时间，能够让任务被自发的分配。我在曾经负责过的团队中通常都会这样去做（当然我的团队都非常小，最大 8 个人），在组建团队后的磨合和学习阶段，反复强调项目目标，鼓励所有人都去熟悉和了解整个任务的全部环节，并以项目经理的角度去思考。这样，需要承担的任务及将自发被团队成员所分解和理解，然后，在对整个项目的理解过程中，人员的特质将被识别，某些成员会退缩，会自发开始进行自己最为熟悉和有把握的工作，这与自发展开架构或 ST 的工作的，或主动承担难度较高或较繁琐的工作的成员，同样值得鼓励，而依旧疑虑并无法熟悉系统的成员，如果乐于被分配到模块级或简单繁琐工作，也是应当鼓励的。我的实际经验上并不会出现任务难以分配或争抢的局面，因为在相互讨论和学习的过程中，团队成员之间也会有相互之间的了解和欣赏，所以实际分配任务时，成员之间会默认某些工作更加适合于某人承担，并且我常常还会主动混淆任务的分割，使得某些任务的边界部分被多个成员承担，至少感觉上，这样有利于任务在出现空隙的时候，其相关的同志能够更加顺畅和自然地提供帮助。当然整个过程并不是完全随意的，项目的目标强调非常重要。此外，某些成员，例如眼高手低，或者非常内向的，能够被识别出来，并在项目过程中及早关注，也是非常必要的。除了任务，还要尽量撮合合适的人员进行验证和设计的搭配。也许是我个人的错觉，长期坐在一起的同事，或者居住在一个区域共同做班车的搭档，往往搭配起来能够更快完成任务的收敛。我觉得顺畅的沟通和理解，在其中起了很大的作用，在管理中，沟通可以建立，理解难以达成。所以，我会倾向于让工作和生活上的伙伴来搭档设计和验证，并且，这也是在任务分配中常常自发出现的。

从我观点一来看，我倾向建立完全自发型的人员组成，强调团队的目标，并使团队内的任务相互交叉，鼓励生活和工作的伙伴搭配完成任务。OK，说到这里，我承认我其实非常的肤浅，虽然我一直有这样的意愿，并一直努力这样做，但真正的理解和把握是相当薄弱的，没法给出更多的总结。

我的观点二，针对验证人员，建议不要仅仅按照验证的流程一步一步执行，而要针对同

伴量体裁衣。不同的设计人员，有不同的特征的。粗心的、内向的、喜欢忽悠的、身兼多职的，识别出来并不困难。对于粗心的设计人员，可以集中力量在前期检视，迅速发现其大量Bug，通过交付规范的制定，推动其返工，帮助其重新细致思考；对于内向的，则需要四面出击，多多帮助其和周边、上下游沟通，特别避免因为信息不畅而出现的规格遗漏、需求理解错误、前后数据接口的握手等地方，所以 Feature 的整理和澄清需要很多时间；对于喜欢忽悠的（自信心膨胀的设计人员往往都是如此），就需要沉下心来，反向推敲其思路，着重在 Corner 和强反压的随机测试中；对于身兼多职的，其交付、沟通等多个环节都可能会出现遗漏，所以验证人员应当更进一步，主动承担某些设计人员应有的工作，不要说 nLint，帮助设计人员理解协议、分析代码、修改代码，也不是不可以的。也许这些事情，并不在流程中被标识，但伙伴，就一定要能够相互扶持、共同进退。一对搭档，共同的成功才是真正的成功。

我的观点三，项目经理（或验证经理），建议在看测试报告的同时，也要看人。即使都是符合交付流程的测试报告，不同的人交付，其背后的含义都是不一样的。新员工往往注重形式而不清楚其内涵，老员工往往以完成为目的而不注重形式，有的同志会忽视条件覆盖率而只关注 TC 和代码行的覆盖，有的同志思考问题过于理想，以至于主动避开一些 Corner Case，还有的同志重随机而轻直接，或重直接而轻随机。这些都是需要进行识别和分析的。流程解决不了这些问题，即使包括评审，亲身经历。如何进行识别呢？我的一个建议是做对比，将多份报告同时打开，多份验证环境同样打开，相互对比，分析其中的差异。这一点，在流程中，是没有的，而且，基本上，验证人员本身也都很少主动进行这种对比的。

算了，写不下去了，以上内容均为废话。很多内容，只是我个人的一种 Feeling。我只是觉得啊，流程，只是指导人正确的做事情，却往往没有办法保证人能够将事情完全做正确，能把事情做正确的，最后还是人。所以啊，很多内容，实在没有办法整理清晰并总结成文字，嗯，回头看整篇文字，也很是没有条理，唉，原谅我吧。

测试数据会撒谎

测试数据会撒谎，没错，测试数据会撒谎。

当黑暗蒙住双眼，迷途的羔羊啊，如何才能追随上帝的步伐。

当信息不能最直接显示，而需要通过其他现象推导或表现，并且人数超过 3 个的时候，则测试数据会撒谎。

这种情形，主要出现在 FPGA 测试和样片测试中。当问题出现在这两种测试环节中时，现象往往都非常表面，虚得很，就像浮在水面上，而真正的 Bug，往往潜伏在深邃的水底，而水面上，还经常有浮萍啊、乌龟啊，或者紫金矿业的污染物啊之类的阻挡，要像 EDA 验证一样，啪嗒一下直接找到问题（EDA 我们往往都没法啪嗒），其难度，不亚于逮住看贴不回帖的潜水艇。

所以，我们往往会收集各种测试数据和现象，进行分析，甚至投入巨大人力，成立攻关组。

这些收集的数据和现象，以我已有的经历看，其中 10%～20%是谎言。

这里，并不是有测试或攻关的兄弟有意在欺骗大家，嗯，绝对不是，兄弟们可都是老实淫啊。

记得有这么一个笑话，一个 GG 问一个 PPMM

“你是处女？”

“不告诉你，你是？”

“我天枰！”

“……”

就是酱紫，明白了吧，当本身不是清晰源头的信息，在汇总和整理时，一定会存在理解上的偏差。这种偏差的引入，我所见，有如下几种可能：

1、 其他相关问题引入，表现好像和本问题相关；
2、 误操作，或板级接触不良、示波器探头抖动之类偶发故障；
3、 疲劳的测试人员眼睛一花，看错了行、读错了数字、找错了寄存器；
4、 测试人员本身有主观分析，就认定是 XX 的 YY 肯定有错，因此数据或意见上具有某些倾向性；
5、 某些无厘头的信息；
6、 佛曰，不可说。

所以，对待测试数据，特别是 FPGA 和样片测试数据，要有自己的想法。

曾经，有一个项目（不是 P600），为定位一个样片测试还是 FPGA 测试的问题，投入数人、数周，方法思考定位，搞得大家的衣服都变胖了。最后却发现，一个最初的测试现象是假象，而以此引申出的后续所有分析和论证的现象，是错误的。具体是哪一个项目，到底是哪一个问题？我现在都已经完全想不起来了，但那种强烈的挫败感和失意感，在我的脑海中，久久挥之不去，唉，问君能有几多愁，恰似一群太监上青楼。

所以，要有自己的想法。样片测试和 FPGA 测试，对于所有测试现象和数据，要有自己的想法。

1、 多个信息指向某个结论的时候，某个结论才是比较可靠的，比较的；
2、 单个信息指向的结论，要思考，思考如果其信息不准确，可能的影响，随时准备退路，如果有可能，亲手复现该现象确认（如果亲手确认的信息还是错的，唉，爷们我认了）；
3、 如果某个现象的现场被破坏，不能复现，麻烦了，一定要慎重排查版本、环境，争取重现，否则其现象及其推导的结论不可控的可能性很大。（这里也提醒某些测试人员，保存现场及其重要，活要见人死要见尸，无凭无据如何能替月行道）
4、 如果某个现象是偶发的，不能稳定复现，不要紧张，以我所见，任何偶发的现象，通过这样或那样的条件设置，一定，一定是可以变成固定重现的；
5、 相互矛盾的现象，要么是没有看到本质（大多数情况），要么测试有问题，切勿南辕北辙各自行事；
6、 建议，对各种现象进行分析的时候，心中要有权重，那些可尽信，那些可斟酌，要梳理出自己心中的条理；
7、 对于整个攻关团队，保证一定程度冲突非常有必要，如果有两种不同的意见和分析，不可强行统一。这里有两种可能，也许两条路都是正确的（很多），只要中间的一环一打破，豁然开朗，也可能其中一条路是死路，那另一条很大可能就正确了。再说了，与其在一棵歪脖树上吊死，不如在两棵歪脖树上吊死来的帅气。

最后，切记，最终真相只有一个，上帝会掷骰子，但不会在芯片内部掷。IC 之神，关羽、张飞，都是实在人，任何缺陷，任何问题都有其唯一的原因的，一便秘就怪地球没引力是没有道理的。认真分析、思考，要有自己的想法。

波形为王

波形不撒谎，这是我做验证的格言。

波形是真理，可以击破一切虚假、迷乱的谎言。

波形，是一个逻辑正确运行的最直观的体现，是逻辑在每一个时钟沿，触发每一个信号的跳变或不跳变，进而产生美丽的，如波浪般运转的脉动。

中医看病，讲究的是，望、闻、听、切，验证看波形 Check 缺陷，正如中医诊断的切脉诊病，除非医术达到精深广博，否则仅靠望、闻、听断病，是不够的。

规格是人写的，详细设计是人写的，激励是人写的，RM 是人写的，自动比对及 Log 打印也是人写的。在做过的几个项目中，我无数次得看到这些内容在编写和理解的过程中出现误差，而导致最终的结果和实际不符，期间诞生了多少的磨难。。。。。。正是经历了太多的磨难，我最终成为了一个验证原旨主义者，任何可以提升验证效率的自动化工具，我都会首先进行排斥，即使使用也充满疑虑。跟我共事过的兄弟应该比较了解了，呵呵，这和转载我文字的接入的傅晓兄弟几乎是南辕北辙的两极。因为我始终认为，自动化虽然提升了效率，但是自动化屏蔽了非常多信息，如果验证的员工对验证的根本性质理解不足的时候（特别是新员工），这种屏蔽非常有害，并且常常容易将错误隐藏起来，而这种隐藏，比效率低下更加致命。所以我始终认为，某些效率提升所带来的收益，是小于问题被遗留到最后才发现所造成的损害的，虽然，这不符合公司效率提升的导向。例如 SV，我认为，在没有理解验证理念和方法之前，连《Writing Testbench》都没有看过两遍（注意，是两遍），还没明白验证不仅仅是跑 TC 之前，就开始玩 SV 的 Class，有害无益（所以我是 DT003 做 SV 最坚决的反对者，很多新员工，甚至连 OOP 是啥都不知道）。想想我们自己是怎样前进的吧，从 Verilog 到 Vera、SystemC、SV，一步步走来，经历了多少磨难。我看到有太多的，这两年新进的员工，表面上什么都知道、什么都会做，但在需要沉下心来分析和理解的问题上，却屡屡碰壁。所以，直至今天，我推崇，绝大部分工作，采用最原始的纯手工方式打造，包括设计和验证。我写的 RTL 代码，都是手工一行一行敲入的，我所验证的逻辑，都是一个一个波形 Check 的。我没有觉得自己效率非常差，反而乐在其中。呵呵，其中的乐趣，是那些只看 Log，忽视波形的人可能无法体会的了。正如“向前一小步，文明一大步”这种长短之间的哲理，女人永远无法理解一样。

呵呵，当然，我上面这一观点并不适用于任何人，应该会有很多兄弟拍砖的。

波形，反映的是逻辑运行的规律，是最切近于最终芯片工作状态的表现。波形中的每一个时钟脉冲，每一个毛刺，每一个 Delay，都将最终完完全全映射到最终的芯片中（特别是后仿）。所以，看波形，能够最真切地发现芯片实际运行中可能的缺陷和风险。作为一个经历过三个项目的项目经理，我特别建议项目经理（其实包括所有有责任感的工程师）能够多看看波形。每一颗芯片，项目经理在其中经历了多少的艰辛和风险，实在不足为外人道。所以，每一颗芯片，对于项目经理而言，都是一个珍贵的孩子，能够在它出生之前，触摸它的每一次心跳，感受它的血管的流动和神经的传导，是多么的难以言语的经验。所以，最近的三次项目，我都承担了后仿的相当一部分工作，我会精心地 Check 每一个时钟和复位，在最典型的激励下，仔细的检查各个模块之间的数据流交互，还有每一个 IO 的延迟和时序，因为我知道，在波形中我看到的，就是最终芯片运转的情形，所以我一定会尽我最大的心力去保护它，毫不留情地去除掉任何一个可能会影响其健康成长的因素，让它能够健康出生、健康成长。

为人父者，无它，其责必重，其谋必远，其心必仁，而其言行必慎！总此四德，天下为父者不可不遵也！

怎样追波形

曾经，有同事仿真挂死，抱着显示器看波形，看了两天，没有结果，给我，我看了 30 分钟，找到了原因；曾经，在同事已经仿真 Pass，最简单的中断测试波形中，我找到了超过 20 个 Bug（和中断测试无关）。所以有同事问我怎么做到的，所以引出了我写这个连载。在这个连载的最后一节，我最后分享一下，我通过波形发现问题，及问题的原因的一些经验。

一回生，二回熟。

很多新晋的验证人员抱怨，这么多信号，这么复杂的连接关系，千头万绪，眼睛都看得长挑针，还是看不出东西。OK，我说，这是没办法的事情，看波形，追波形，是一个经验积累的过程，任谁都逃不掉。爷爷都是从孙子走过来的。越是看，越是明白，越是不看，越是不懂。看得多了，自然就知道应该抓那些信号，如何分类，如何追溯了。所以我奉劝某些希望通过全自动的 Log 和信息推导结果，或者每次一有问题就找设计人员看波形的验证人员，回头是岸。波形，是逻辑运行的最真实的表现，逃不掉的。为什么我看波形快？无他，唯手熟尔。

先看 X 和 Z。

任何一个波形，无论是验证的前期、中期、后期，到手之后，先刷屏，找 X 和 Z，确认，某些 Z 和 X 是可以存在的，例如某些 IP 模型，或者未初始化的寄存器和 RAM，但芯片开始正常后，Z 和 X，都不应当存在。OK，我承认这个经验非常简单，某些高层领导可能认为这简直就是幼稚。可惜，可惜的是，我至今为止看的，所有项目的波形，都能够在这上面找到 Bug，甚至我可以预计下一个项目，我继续看波形，还是能够找到。以我自己设计的 L2 Cache 为例，一个多年验证经验的老员工负责验证的，至今已经在多个项目中量产，还是在最近检查波形的时候发现有一个文件 wire 声明时把信号名写错了，悬空了（因为该信号是 input，隐含了 wire 声明，所以不影响功能）。OK，X 和 Z，一定会存在，第一时间找到它，可以节省非常多的验证定位时间，否则追波形半天发现是 X，真是浪费青春。X 和 Z，所对应的 Bug，可能有如下几种：

1、 IP（包括 Memory、PLL、Serdes 等等）例化时，某些信号悬空未接。也许某些模型允许 Power 信号悬空，或者某些信号是悬空给 DFT 处理（当下给 DFT 处理的信号是接零），但大多数 IP，输入信号是不可悬空的；
2、 信号位宽不匹配、信号多驱动、声明的信号名称写错、TB 级互联错误或 TB 中遗漏的 Force（额外小心隐藏的 Force），不要相信 nLint，特别是在芯片顶层或
3、 后仿真时序不满足时的 X 态传递；
4、 功能错误，某些模拟 IP 未能正确操作；
5、 功能错误，导致管脚冲突；
6、 功能错误，未能合理使用无复位端的寄存器和未初始化的 Memory。

再看时钟。

很多验证人员不看时钟。经过我反复的证明，这是一个非常正确的结论（经常在项目验证后期协助定位时钟不对齐导致的环境问题）。只要 TC 能够打印 Pass，很多验证人员不关心时钟是否有问题（甚至很多新验证人员，根本不明白Δ Delay 的概念）。大多数情况下，时钟不会有问题？No，大多数情况下，系统验证，时钟都有问题。记得以前在一个项目中推动 CRG 设计定义了一个规范，时钟分频寄存器，延迟 0.2ns，其他寄存器，延迟 0.3ns，

分频的原时钟，在输出前延迟 0.2ns 对齐，不知道看到本文的，数字平台部的验证人员，有多少能明白其中的用意?不解释，不明白的请自行蹲墙角反省。再想起一个以前海思的设计规范，要求寄存器赋值，不可加延迟，也是让人在风中凌乱啊。下面这个逻辑，寄存器赋值没有延迟，仿真能正常工作否？答案是可以！如果上帝比较仁慈，或者验证人员对仿真器的 always 执行顺序无限了解。

```
always@(posedge clkx2)   clk <= ~clk;
always@(posedge clkx2)   b<= a;
always@(posedge clk) if(clken) c <= b;
```

所以，在看完 X 和 Z 后，要将所有时钟拉到波形中 Check，看是否所有同步时钟（包括 1：N 倍频）的时钟沿是否严格对齐，CLKEN 时钟能够正确将倍频时钟上升沿罩住，关键地方寄存器赋值是否有 Delay（如果时钟间不存在Δ Delay，寄存器赋值可以没有 Delay）。别小看这个工作，Pxxx 验证组，记得哥当时被抓去协助定位了多少时钟问题吗？赔我青春损失费啊。。。。。。。

OK，我们进入正题，怎样看波形。没错，楼上都是废话，谁再让我看波形，结果是上面两种情况，我就要发飙了。

如何在一个看似无限复杂的挂死波形中定位根因？

如何在后仿波形中发现可能的问题？

如何在表面上没有问题的波形中发现问题？

面对波形，首先要端正心态，不要认为看波形是浪费时间，也不要因为一时无法发现其中的问题而焦虑烦躁，更不要盲目乐观，认为已经没有任何问题。要执着、坚定，充满勇气。

先不要看波形，对，先不要看，这是我很重要的经验之一。要先思考，我的做法是对照架构图，虚拟一个芯片运转的场景，即在脑海中想像一下当前这个激励下，波形应当是怎样运作的，激励怎样进入系统，然后怎样完成协议解析和转换，怎样到达了总线，然后出现 DDR 的吞吐，然后 CPU 取指、取数，完成处理。OK，也就是说，先要在心中预留一个完美的 Scenario，设想一下白雪公主和王子是怎样在城堡中幸福生活在一起的。

心中有了虚构的波形后，再使用 Verdi 打开波形。抓关键信号，分组，标识不同颜色，这些奇技淫巧应该不用多说，很多兄弟都比我这个验证原旨主义者来得厉害。只是需要说明的是，抓多少信号，怎样分组，很需要斟酌。其实原则只有一个，让尽可能精炼的信号在一屏内显示，这和代码的精简是一个道理。信号除了按功能分类，还要按信息量分权重。所以还是要说，很多验证人员，用着花哨的手指技法，一屏一屏的信号抓，刷刷几屏下来，跟瀑布一样，很是壮观，往往 Group 的数量比我总共抓的波形数量还要多。操，看个 AXI 总线，把 arlock 信号和 arvalid 信号一起抓出来，除了催眠看波形的人之外，有其他意思吗？以 AMBA 总线为例，APB 先看 PADDR、PSEL、PENABLE，AHB 先看 HADDR、HTRANS、HREADY，AXI 先看各个通道 ADDR、VALID、READY，如果这些信号不能说明问题，再逐步增加辅助信号观察，尽量保证在一屏中显示所有有效信号，如果信号太多，宁可删除部分。

然后，将展开的波形和脑海中已有的场景进行对照，看数据流是否按照脑海中预期的构想而流动。一般来说，实际波形和预想都不太能够对上，最开始的大多数情况下，波形是正确的，而脑海中的预想存在不足，这主要是因为我自己对架构的细节理解还不充分，对某些特殊逻辑处理方式不熟悉，或者某些逻辑相互连接后新增的耦合关系不了解等等原因导致。而这个时候，在我看来，也正是一个最好的时机，来进一步熟悉和理解芯片真实运转流程，弥补自己对系统结构、互联设计中各个细节的理解不足。在对细节的进一步理解和澄清的过

程中，我会逐步修正心中虚构的波形，使其逐渐接近真实的运作。当然，在修正过程中，会出现某些确实表现异常的波形，一些明显出乎设计预期的时序出现。OK，这是一个岔路口，先记录*.rc 波形现场，然后对该出乎意料的时序进行进一步深入追溯。正如前面所述，我常常在已经 Pass 的波形（或后仿波形）中发现 Bug，通常都是从这样的岔路口开始的。也许这个岔路口最终证明逻辑没有问题，还是细节理解不足导致，但也许就是一个芯片难以发现的致命缺陷。

看到这里，也许有同志会问，挂死的波形呢？怎么还没追挂死的点呢？Yes，就是还没开始，无论是挂死的波形，Fail 的波形，还是 Pass 的波形，在其实际波形和我大脑中虚构的波形没有完全吻合之前，我是不会开始定位问题的。这简直是浪费时间？嗯，在最开始定位的阶段确实如此，但在验证进入中后期之后，正常的波形和我虚构的波形已经调频到一个波段了，或者说，我已经确认白雪公主和王子开始幸福生活了。和波形到手，从前到后一扫，寻找那些地方异常，例如，白雪公主生下来的小王子永远长不高，那明显就是有问题嘛。然后，半小时发现问题，并不困难。当然，还是要说，我的经验，是不适合那些平时就只看 Log，忽视波形的人的。

最后，我还是共享一些我定位复杂挂死波形的根因的经验。起点，定位挂死的起点在那里？或者说，从那里开始追？很多同志认为寻找一个合适的起点很重要，或者说最好直接就找到引发挂死的点。我说，No!! 找到直接引发挂死的点，或者和该点直接相关的起点，是很困难的，我不做这种类似分析双色球中奖率的事情，任何挂死的波形，第一时间能够获得的信息，都是表象。我会将任意一个被阻塞的操作作为起点，开始我溯溪的旅途（请确信，无论那个溪流，最终都会汇到唯一的源头）。如有可能，用笔在沿途做下记号，保证在偏离方向的时候能够返回。溯溪的路途有艰辛、险阻，我看过太多的兄弟在中途放弃，或另寻他路，而有时候，他离最终的源头，仅一步之遥，所以，最后一个建议，请保持一颗坚定和勇敢的心。有一首我喜欢的歌，范玮琪的《最初的梦想》，艰难的时候，可以听一下。

最最后，再补充一点后仿定位的思路，供参考。

1、 还是先关注时钟，每一个模块，clk 和 clken 相位是否正确（PR 有时会对时钟取反，需要注意）；
2、 关注有门控功能的时钟和复位，确认是否存在毛刺；
3、 把 Top 层所有管脚和 oen、I、C 信号抓出来。除掉红色黄色，观察有特殊变化的 oen、i、c 信号,尤其是和管脚值不符的,还有毛刺；
4、 观察所有异步接口的时序，基本上都会发现毛刺。例如 EBI、Efuse，确认毛刺是否影响功能；
5、 观察顶层或 Subsys 独立处理的信号，特别针对还不够成熟的集成人员的特殊设计，例如 testmode、rst_out、系统控制器特殊的配置和检测信号；
6、 关注跨团队的模块，例如功能可控的 Memory BIST；
7、 以上所有这些,MAX 和 MIN 会有不同，要仿真 Typical 时序，并且确认以上的内容；
8、 后仿的波形，需要结合 Log 中的 Warning 一起 Check。

The END