

FPGA STORY 《工具篇 II》

# 仿真技巧 & Modelsim

芯驿电子科技（上海）有限公司  
*[Http://www.alinx.cn](http://www.alinx.cn)*

# 目录

前言.....	4
第一章 仿真的扫盲文.....	5
1.1 Modelsim 是电视机.....	5
1.2 仿真和调试.....	6
1.3 理想与物理.....	10
1.4 综合与验证.....	12
1.5 激励文本 ( Testbench ) .....	13
1.6 仿真流程 .....	17
1.7 建模的切糕.....	20
1.8 仿真的切糕.....	24
1.9 自动思想的简介 .....	30
1.10 不可仿真对象的简介.....	33
总结 : .....	34
第二章 Modelsim 就是电视机 .....	35
2.1 连接 Modelsim .....	35
2.2 自动编译, 半自动编译, 手动编译.....	37
2.3 自动编译的预先设置.....	41
2.4 常用界面.....	47
2.5 自动编译与半自动编译 .....	52
2.6 操作 Wave 界面.....	61
总结 : .....	75
第三章 理想就是美丽.....	76
3.1 理想时序.....	76
3.2 时间点事件还有即时事件的时序表现.....	86
exp01_simulation.vt .....	86
exp02_simulation.vt .....	89
exp03_simulation.vt .....	90
exp05_simulation.vt .....	95
exp06_simulation.vt .....	96
exp07_simulation.vt .....	98
3.3 指向时钟的 i.....	102
exp09_simulation.vt .....	108
3.4 指向过程的 i.....	111
exp09_simulation.vt .....	113
3.5 激励的好帮手是 i.....	117
3.6 协调的时序。 .....	122
exp10_simulation.vt .....	124
exp11_simulation.vt .....	127
总结 : .....	133

第四章 激励文本就是仿真环境.....	134
4.1 验证语言是什么？.....	134
4.2 验证语言与时序表现①.....	145
4.3 验证语言的时序表现②.....	154
4.4 激励文本的布局.....	165
selfadder_funcmod.v.....	167
selfadder_funcmod.vt.....	169
4.5 仿真模型与反馈输出①.....	175
hello_funcmod.v.....	179
hello_funcmod_simulation.vt.....	181
4.6 仿真模型与反馈输出②.....	186
hello_funcmod.v.....	187
hello_funcmod_simulation.vt.....	188
hello_funcmod_simulation.vt.....	192
总结.....	196
第五章 仿真就是人生.....	198
5.1 单向仿真与多向仿真.....	198
exp20_simulation.vt.....	200
exp21_simulation.vt.....	203
5.2 仿真的变数——时钟用量.....	209
exp22_simulation.vt.....	213
5.3 仿真的变数——信号数量与信号方向.....	215
5.4 理想变数与物理变数.....	221
5.5 仿真的必然——特定条件.....	228
exp23_simulation.vt.....	230
exp24_simulation.vt.....	232
exp25_simulation.vt.....	239
5.6 仿真迷宫①.....	244
5.7 仿真迷宫②.....	250
5.8 主动思想.....	257
总结：.....	266
第六章 结束就是开始.....	268
6.1 不可仿真对象.....	268
6.2 超烦模块与不等比例缩放.....	272
exp26_simulation.vt.....	272
exp27_simulation.vt.....	275
exp28_simulation.vt.....	277
6.3 不可仿真对象——超乱模块①.....	279
ram.v.....	280
exp30_simulation.vt.....	283
6.4 不可仿真对象——超乱模块②.....	289
exp31_simulation.vt.....	291
exp32_simulation.vt.....	296

6.5 主动设计① .....	301
<i>ram.v</i> .....	302
<i>spi_funcmod.v</i> .....	303
<i>spi_funcmod.vt</i> .....	305
<i>ctrlmod.v</i> .....	307
<i>exp33_simulation.vt</i> .....	309
6.6 主动设计② .....	313
<i>ram.v</i> .....	313
<i>spi_funcmod.v</i> .....	314
<i>spi_funcmod.vt</i> .....	317
<i>ctrlmod.v</i> .....	319
<i>exp34_simulation.vt</i> .....	321
6.7 仿真的决意 .....	326
总结： .....	329
后语 .....	330





## 前言

笔者一直以来都在纠结，自己是否要为仿真编辑相关的教程呢？一般而言，Modelsim 等价仿真已经成为大众的常识，但是学习仿真是否学习 Modelsim，笔者则是一直保持保留的态度。笔者认为，仿真是 Modelsim，但是 Modelsim 不是仿真，严格来讲 Modelsim 只是仿真所需的工具而已，又或者说 Modelsim 只是学习仿真的一部小插曲而已。除此之外，笔者也认为仿真可以是验证语言，但是验证语言却不是仿真，因为验证语言只是仿真的一小部分而已，事实上仿真也不一定需要验证语言。

常规告诉笔者，仿真一定要学习 Modelsim 还有验证语言，亦即 Modelsim 除了学习操作软件以外，我们还要熟悉 TCL 命令（Tool Command Language）。此外，学习验证语言除了掌握部分关键字以外，还要记忆熟悉大量的系统函数，还有预处理。年轻的笔者，因为年少无知就这样上当了，最后笔者因为承受不了那巨大的学习负担，结果自爆了。

经过惨痛的经历以后，笔者重新思考“仿真是什么？”，仿真难道是常规口中说过的东西吗？还是其它呢？苦思冥想后，笔者终于悟道“仿真既是虚拟建模”这一概念。虚拟建模还有实际建模除了概念（环境）的差别以外，两者其实是同样的东西。换句话说，一套用在实际建模的习惯，也能应用在仿真的身上。

按照这条线索继续思考，笔者发现仿真其实是复合体，其中包括建模，时序等各种基础知识。换言之，仿真不仅需要一定程度的基础，仿真不能按照常规去理解，不然脑袋会短路。期间，笔者发现愈多细节，那压抑不了的求知欲也就愈烧愈旺盛，就这样日夜颠倒研究一段时间以后，笔者终于遇见仿真的关键，亦即个体仿真与整体仿真之间的差异。

常规的参考书一般都是讨论个体仿真而已，然而它们不曾涉及整体仿真。一个过多模块其中的仿真对象好比一块大切糕，压倒性的仿真信息会让我们喘不过起来，为此笔者开始找寻解决方法。后来笔者又发现到，早期建模会严重影响仿真的表现，如果笔者不规则分化整体模块，仿真很容易会变得一团糟，而且模块也会失去连接性。

笔者愈是深入研究仿真，愈是发现以往不曾遇见的细节问题，然而这些细节问题也未曾出现在任何一本参考书的身上。渐渐地，笔者开始认识，那些所谓的权威还有常规，从根本上只是外表好看的纸老虎而已，细节的涉及程度完全不行。笔者非常后悔，为什么自己会浪费那么多时间在它们的身上。可恶的常规！快把笔者的青春还回来！所以说，常规什么的最讨厌了，最好统统都给我爆炸去吧！呜咕，过多怨气实在一言难尽，欲知详情，读者自己看书去吧 ...

akuei2 上

（15 日 08 月 2013 年）

# 第一章 仿真的扫盲文

## 1.1 Modelsim 是电视机

如果笔者提问 Modelsim 是何物？想必同学们都会认为“Modelsime 就是仿真”这种等价的关系。草草而言，该想法只是美丽的误会而已，笔者眼中 Modelsim 只是类似电视机的工具。我们知道电视机除了播放功能以外，它甚么也不是。换之，身为用户的我们，使用电视机就要学会开关和调节节目，而不是去研究构造和功能原理。

Modelsime 成功播放波形图以后自然可以功成身退。笔者一直以来都无法理解，为甚么会有那么多同学特别纠结 Modelsim 的五脏六腑，笔者不知要佩服他们的挑战精神才好，还是要讽刺他们皮痒才好呢？因为信心满满到的他们，最终都会被 Modelsim 搞到变成破烂回来，结果实在惨不忍睹。笔者也不是有意中伤他们，因为笔者也是经验者。

Modelsime 有各种各样的版本，除了官方的默认版本以外，还有第二方的自定义版本，如 Altera Modelsim SE 或者 Altera Modelsim PE。以上是 Altera 公司自定义的两个版本，SE 是 Start Edition 亦即入门版本，AE 是 Altera Edition，亦即是付费版本。

许多同学都认为 AE 一定比 SE 更强更加好用。逻辑上的确如此，不过 AE 与 SE 之间宛如 24 寸与 19 寸液晶显示器的大小差别而已。基本上 SE 已经足够应付一切仿真应用了，此外 SE 还有许多功能根本排不上用场。婆婆曾说过做人要节俭，东西够用就好，过多就是浪费，所以笔者告诫读者不要过度纠结 AE，如果读者硬要自寻烦恼的话，那么后果请自负。

笔者知道自己很烦，最后还是要再强调一下，目前同学们只要把 Modelsim 当成电视机就好，然而 Modelsim 除了学习开关之余，还有就是调节节目而已，至于详细的用法，往后我们会慢慢接触到。

## 1.2 仿真和调试

曾笔者还没讲述“仿真”之前，让我们在根本上先理解仿真和调试的区别。仿真这词的同义虽然接近调试可是仿真却不等于调试。调试用来观察结果，仿真则是观察过程。仿真有“功能仿真”和“行为仿真”两个专业分类，对此调试也有“下线调试”和“上线调试”两个专业分类，让笔者用表格来说明：

表格 1.2.1 仿真与调试的对应关系。

并行语言	顺序语言
仿真	调试
功能仿真	下线调试
行为仿真	上线调试

那么，什么是上线调试与下线调试呢？上线调试最为常见的方法就是将程序下载到开发板，然后再观察开发板的结果是否与预期一样，如发出“哔哔哔”声的程序，下去开发板后蜂鸣器发出“哔哔哔”的话，那么程序就合格。此外，上线调试还有较为细腻的方法，就是利用专用的上位程序或者集成环境同步测试开发板，详细情形笔者就不谈了，有玩过单片机的朋友一定略知一二。

下线调试就是隔着开发板在电脑上模拟程序的测试结果，让笔者用例子详细说明吧。

```
1. main()
2. {
3.     int varA = 1;
4.     printf( "%d", varA );
5. }
```

代码 1.2.1

代码 1.2.1 大意是指，第 3 行声明整型变量 varA 然后赋予初值 16，然后再第 4 行使用打印函数——printf 输出 varA 的储存结果。假设 varA 的储存结果 1 对应 LED 点亮，那 0 对应 LED 消灭。如代码 1.2.1 所示，它会告诉我们 LED 最终会点亮，因为 varA 被赋予初值 1。

所谓下线调试就是在没有开发板的情况下，利用模拟环境取得假想结果，假想结果再经由大脑进一步脑补（反映）实际的情况。如代码 1.2.1 所示，当我们在集成环境（IDE）按下 <F5> 或者 <F6> 的测试热键，varA 的输出结果“1”就会在集成环境的信息窗口显示出来。再来经由我们自己脑补道：“varA 的输出结果是 1，那么开发板的 LED 是点亮的”。

到目前为止，笔者也只是大概讲述一下“调试”的内容而已，事实上“仿真”的内容比起“调试”还要麻烦许多，如果读者要确确实实理解调试与仿真之前的实际区别，读者

就必须从语言的本质开始理解。

在笔者的眼中,一些高级语言如 C, C++, Java 等都视为顺序语言,此外还有古老的 Basic 和汇编也是一样。为什么笔者之将它们称为顺序语言呢?原因很单纯,因为这些语言是按着顺序的步骤在执行操作,举个例子:

```
(一)  main()
(二)  {
(三)      varA = 1;
(四)      varB = 2;
(五)      varC = 3;
(六)  }
```

代码 1.2.2

代码 1.2.2 的 3~5 行是 varA, varB 与 varC 的赋值操作,先是 varA 赋予 1 值,然后 varB 再赋予 2 值,最后 varC 赋予 3 值。在此之前, varB 的赋值操作需要等待 varA 的赋值操作完毕之后才能执行,同样 varC 的赋值操作需要等待 varB 赋值完毕之后才被执行。varA, varB 与 varC 之间的赋值操作延迟也称为“步骤差”。

故名思议,顺序语言会让我们普遍将代码以步骤的单位去认为,如果步骤 1 不完成执行,步骤 2 就无法继续,其它以此类推。笔者再将代码 1.2.2 换成另一个形式看看:

```
1.  main()
2.  {
3.      varA = 1; varB = 2; varC = 3;
4.  }
```

代码 1.2.3

代码 1.2.3 是将代码 1.2.2 的 3 行集为 1 行,然后我们会这样解读代码:“第 3 行, varA 赋予 1 值, varB 赋予 2 值, varC 赋予 3 值。”,笔者再顽皮一点,继续胡搞代码 1.2.3:

```
1.  main()
2.  {
3.      varC = 3; varB = 2; varA = 1;
4.  }
```

代码 1.2.4

代码 1.2.4 是将代码 1.2.3 的赋值操来回颠倒,然后我们会这样解读代码:“第 3 行, varC 赋予 3 值, varB 赋予 2 值, varA 赋予 1 值。”,无论笔者怎样胡搞 varA, varB 与 varC 的位置,我们都无法割舍步骤这个单位去解读代码中 varA 与 varB 还有 varC 的赋值过程。因为只要失去“步骤”我们就会迷失解读的方向,这是顺序语言的特征,我们也可以说“步骤就是支撑整体顺序语言最基本的结构”。

在此读者必须理解，“平常我们就是太习以为常使用步骤这个单位去解读代码，不知不觉习惯就成为理所当然”，但是事实却好相反，这种想法还有这种思路也仅限于“调试”这个框架而已。此外，读者还必须理解“步骤”只是可视的宏观单位而已，然而不可视的微观单位却是“指令”。假设  $\text{varA} = 1$  的赋值操作，可视步骤也只有 1 个，但是在隐藏中， $\text{varA} = 1$  这样简单的赋值操作到底需  $N$  个指令完成，完全取决与编译器的编译质量。

对于我们这些只会认为肉眼看见才是事实的小白而言，指令就像不存在与人界的幽灵般，时而增多时而减少，尽是虚幻也难以捉摸。除此之外，处理指令所需的时钟，也会根据指令的版本，处理器的工艺等因素产生改变。讲白点，调试只会输出步骤的结果却无视指令的内容，它也会无视时钟的消耗数。

因此读者必须理解，调试一般离不开顺序语言，宏观上顺序语言需由步骤这个可视单位支撑。微观上，步骤则是由无数的指令在后面支撑着，然而指令的内容还有时钟的消耗数都是隐藏内容。总结说，调试只在乎步骤在表面上产生的结果而已，又或者说是追求单向结果。

---

笔者一直以来都在不停思考，为什么 Verilog HDL 不使用“调试”而是“仿真”这词呢？有些朋友可能会认为笔者有点过于钻牛角尖了，对此笔者不敢否认，但是那种违和感时时刻刻都在折磨笔者，笔者也十分焦急想将它揪出来。我们知道“调试”与顺序语言有切不断的关系，然而“仿真”却与并行语言有着强烈的羁绊。

所谓的并行语言有如 VHDL 或者 Verilog HDL 等各种描述语言，描述语言不像顺序语言有步骤支撑整体的顺序结构，结果描述语言显得结构自由，甚至称为没有结构的地步。顺序语言每一个关键字都在暗示处理器的处理行为，然而描述语言每一个关键字只是描述手段，我们用它在白纸上绘出我们想要的“形状”。

顺序语言与并行语言之间最大的差异就是，顺序语言每一个时钟只能执行一个步骤（如果这个步骤在一个时钟内完成的话），然而并行语言可以在一个时钟内执行千千万万个步骤，上限是没有尽头的。顺序语言之所以不关心时钟，那是因为时钟不仅隐藏而且无法控制。反之，并行语言的时钟不仅开放也能控制。

仿真有“功能仿真”还有“行为仿真”两大分类，如表格 1.2.1 所示。功能仿真与行为仿真的差别，即前者是下线，后者则是上线。一般所谓的仿真就是“功能仿真”，而不是行为仿真，为了避免浑浊，往下内容笔者皆用“仿真”来表示“功能仿真”。至于行为仿真已经超出本书的范围，恕不解释。

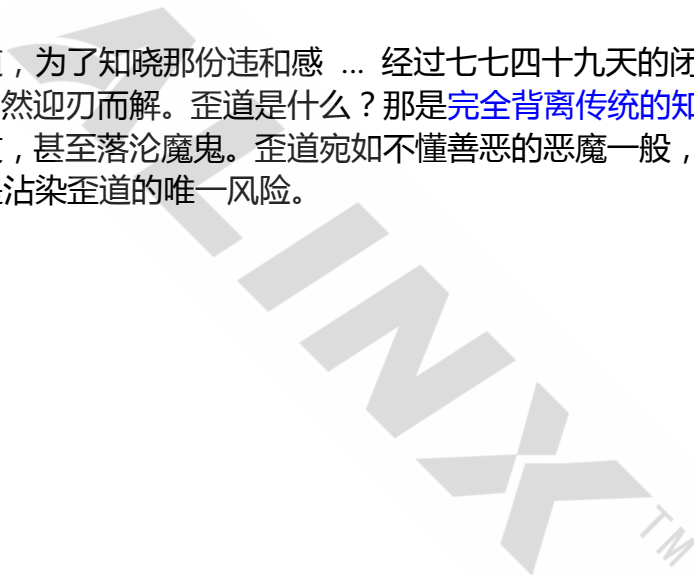
仿真是一件很麻烦的事情，而且“仿真”也不像“调试”那样，只要轻松按下 <F5> 或者 <F6> 等调试热键，信息就会像水一样，花啦啦地打印出来。换之，执行仿真之前必须经历许多准备工作，如创建仿真对象，编辑仿真环境，仿真之间我们必须一边追踪过程，一边解析信息。

笔者曾说过调试是追求单向结果，如果用仿真来比拟调试，调试是用来断定某个信号在某个时钟的某个结果而已，如：信号 B 在时钟 T10 的结果是逻辑 0，又或者数据 C 在时钟 T12 的结果是 8'hAA。反之，仿真是追求多向过程，亦即 N 个信号在所有个时钟发生里什么结果，如：模块 A，有信号 A 与数据 B，而且模块 A 的一次性操作需要耗时 10 个时钟，那么仿真会用来观察信号 A 与数据 B 在 10 个时钟内的结果变化。

笔者曾被仿真杀死过许多次，如果不是它在笔者最绝望的时候拉笔者一把，如今笔者就是一只怨气十足的冤魂了。学习仿真就像处于金庸所描述的江湖般，那里存在许多各种帮门流派，其中一种称为传统流派，也是网络流传已久的右翼硬派，门徒最多，死人也是最多。初落平阳的笔者为了寻求照应，就这样糊里糊涂加入其中。

期间笔者至少死过十余来次，最后终于支持不住，一心来到崖边寻求解脱。那是笔者的人生当中最黑暗的一刻，就在跳下的瞬间，一直温柔却有力的右手揪主笔者，然后让笔者感动不已的声音传遍全身：“孩子，千万别做傻事，明天总有希望！”，此刻是笔者最痛哭的一次。

为了寻找仿真之道，为了知晓那份违和感 ... 经过七七四十九天的闭关以后，歪道终于开窍，所有问题自然迎刃而解。歪道是什么？那是完全背离传统的知识，它像毒瘾一般让人深入无法自拔，甚至落沦魔鬼。歪道宛如不懂善恶的恶魔一般，只会给予方法却不会承担后果，这是沾染歪道的唯一风险。





### 1.3 理想与物理

理想与物理就像梦想与现实之间的关系 ... 我们知道现实世界（物理）是充满病痛，鄙视，欺骗还有杀戮等各种悲剧的复合空间，然而传统流派就是基于它们。笔者第一次踏入传统流派的大门，一股恶寒经由脚根直达脊椎，全身也不禁颤栗起来。反之，理想世界却是现实的相反，那里不存在任何悲剧。

闭关期间，笔者曾经穿梭诸神逗留的乌托邦，眼前出现的一切不经让笔者目瞪口呆。在那里，生命都有黄金比例的结构，大伙都是协调相处，啊！多么理想的世界 ... 对！这就是笔者向往的世界，也是仿真应该演变的方向，而不是那坑坑爸爸的物理世界。我们知道**时序就是寄存器还有组合逻辑产生的活动（信号）**，传统流派强调**时序应该接近物理**，亦即**物理时序**，反之笔者强调**时序是理想完美**，亦即**理想时序**。

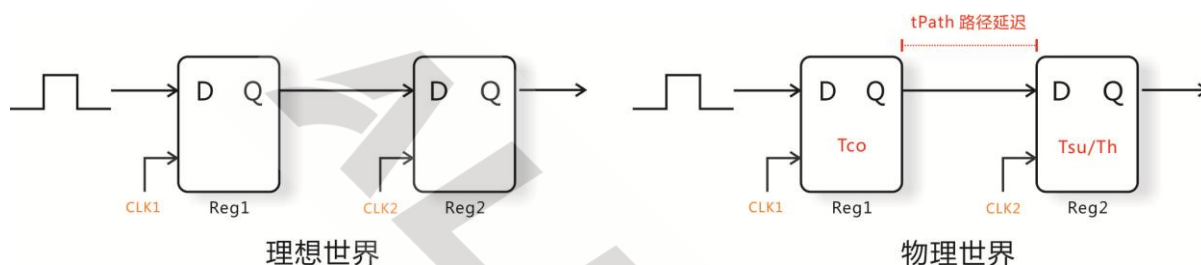


图 1.3.1 理想世界与物理世界的寄存器。

图 1.3.1 显示有两个世界的居民，左图是理想世界，右图是物理世界。理想世界是非常**整洁**的世界，不像物理世界存在许多**物理元素**如： $T_{co}/T_{su}/T_h$  等寄存器特性以外，还有  $t_{Path}$  等物理延迟。初学的朋友可能会问：什么是寄存器特性？什么又是物理延迟？，朋友可以将它们想象为**阻碍寄存器沟通的障碍**。

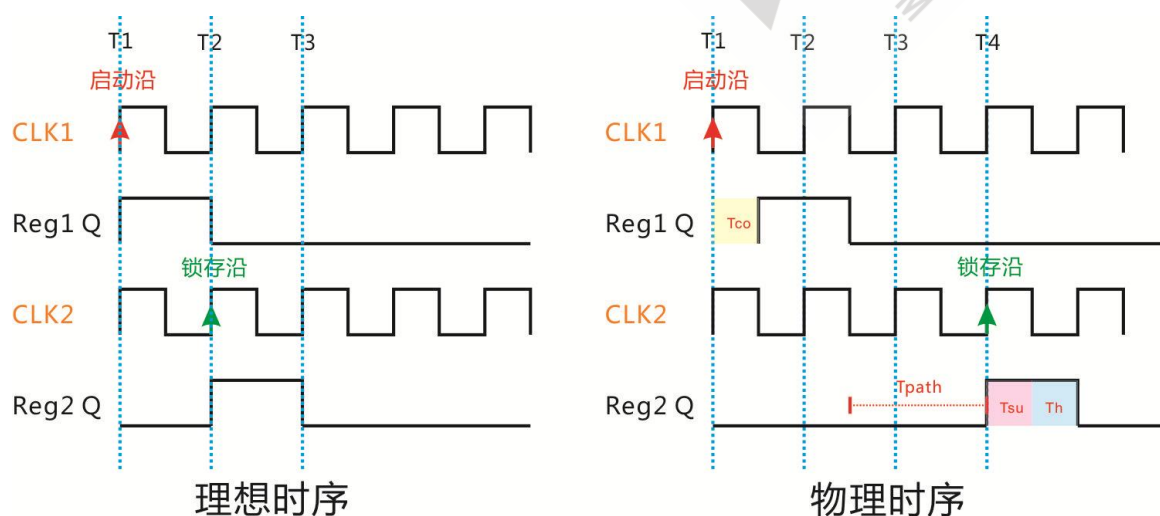


图 1.3.2 理想时序与物理时序。

图 1.3.3 是寄存器之间的活动记录（沟通记录），亦即**时序**，也是**该世界唯一可视的信息**。

左图是理想世界产生的理想时序，右图是物理世界产生的物理时序。根据左图所示，寄存器 1 在 T1 的时候向寄存器 2 发送（启动）数据，接着寄存器 2 便在 T2 接收（锁存）并且输出。根据右图的表现，寄存器 1 在 T1 向寄存器 2 发送数据，可是数据遭受  $T_{co}$  的妨碍之余，还被  $T_{path}$  拖后腿，最终数据在 T4 被寄存器 2 读取，不过寄存器 2 在读取之间还要考虑  $T_{su}$  与  $T_h$  是否满足。

左图是非常整洁又高效的沟通过程，然而右图是沟通过程非常烦乱。试问读者，哪一个时序图更加顺眼更加容易解读？答案理想是理想时序，不是吗？在此，心机重的读者可能会反驳道：“叫兽说过时序不能缺少物理元素，如果物理元素被吃掉，时序还是时序吗？”可怜的读者，叫兽就是风水师，骗人骗到祖宗十八代也不奇怪。

首先我们必须理解，所谓仿真就是在虚拟的环境下运行模块，测试功能是否按照预期执行。为此，我们为何不取最理想的结果呢？举例而言，假设模块 A，要求 T1 拉高输出，然后 T2 拉低输出。为此，仿真仅是单纯地观察它是在 T1 拉高输出，然后在 T2 拉低输出——这是理想状态。而不是观察模块 A 在 1ps 拉高的电平是多少 V，2ps 拉低电平多少 V——这是物理状态。

再者，笔者也强调过，Modelsim 只是一台电视机而已，功能就是播放波形（时序），这种情形宛如读者看电视，要享受当然选择高清节目，而不是走色的崩坏节目。理想时序就是高清节目，物理时序就是崩坏节目，仿真就是享受高清节目这么一回事，读者能理解吗？

在此，有些同学可能会不安道：“物理时序该怎么办？物理元素该怎么办？”。为此，先让笔者帮忙消除不安：

1.  $T_{co}/T_{su}/T_h$  寄存器特性，或者  $T_{path}$  等物理延迟，理想时序都会无视。
2. Modelsim 只是一台电视机而已，它可以播放理想时序也可以播放物理时序，不过没有傻子会喜欢崩坏的节目。
3. 物理时序 Modelsim 虽然可以播放但是无法解决。此外，各大 FPGA 厂商早已经为物理时序准备好各种仿真和纠正的工具，如 TimeQuest。
4. 笔者也准备好物理时序的教程。

读者用不着担心理想时序多难学习，事实上理想时序相较物理时序更加容易掌握。再者，笔者爱用的建模技巧，仿真技巧，整合技巧，甚至静态时序分析，都有应用理想时序。理想时序作为设计是非常重要的概念，尤其是仿真的环节上，它不仅可以减少仿真的难度，也可以减轻激励文本的编辑工作，还有内容的解读。

它曾说过：“理想的开始就是成功的一切”，这句话暗喻心情的重要性，理想或者美丽的东西会舒缓心情，结果高产。反之，瑕疵还有丑陋的东西会搞坏心情，结果难产。这种感觉好比新年图新，什么都是新，华人相信新东西除了示意好开始之外还有圆满和理想的含义，因为新东西没有肮脏和瑕疵。

## 1.4 综合与验证

描述语言专业分类有综合语言和验证语言，**一般认为综合语言用来设计，验证语言用来仿真**，说实话**那是放屁**！笔者还记得第一天学艺的时候，师兄当下给笔者递过一本厚厚的书籍，封面写着“验证语言”，笔者随意一翻，蛋蛋立即落在地上。因为内容仅是意义不明的关键字还有语法。

师兄最后还说道：“今天给老子啃完，不然明天把你干掉 .... ”

许多新手曾是那样，**综合语言还没有掌握又要立即学习验证语言**，不然仿真就无法开工。老实说，别开本大爷的玩笑了！拜托了，笔者是人不是吸尘机，不可能在短时间内吸收那么多东西。学习和恋爱一样，不能同时一脚踏两船，不然进度会进入两头不到岸的窘境 .... 直到最后，不管综合语言还是验证语言，半桶水的程度也没有达到。

此外，**验证语言也会随着年份拉长，内容也会不断增加**，如 Verilog 1993 进化到 Verilog 2001（据说未来还会继续增长）。只要稍微打开手册一看，我们立即就会发现验证语言占满全量的 4/5，这点无疑是一起厚重的压力。闭关期间，笔者一直苦思冥想：“**仿真的定义是什么？为什么仿真离不开验证语言呢？**”不知不觉，笔者的意识再度穿梭诸神逗留的乌托邦。

它告诉笔者：“**物理世界有资源却有法则束缚，理想世界没有资源也没有法则束缚**，真是一言惊醒梦中人。仿真是**利用虚拟的环境取得假想的结果**。然而，这个虚拟环境，假想空间，没有所谓的物理限制，如：理想的 FPGA 有数不尽的逻辑资源，开发板要什么硬件就有什么硬件。但是**这个虚拟环境却没有实际的资源，如时钟信号什么的**。为此，我们需要利用验证语言产生虚拟的时钟信号。

上面的内容告诉我们一个非常重要的信息，亦即**仿真也是建模，不过是虚拟建模**，它虽然不会局限于硬件，但是需要模拟实际资源，为此需要用到验证语言。为此，笔者得到这样一个问题，**如果验证语言可以用来描述虚拟的资源，为什么综合语言不能用来描述虚拟的资源呢？**。

实际上，仿真只要最小利用验证语言而已，例如产生时钟信号什么的，之余**其余的描述工作，我们都可以交由综合语言去搞定**。这是笔者最活跃的仿真思想，**把仿真当成建模来玩**。反之，传统流派的仿真概念好像被堵塞的臭水沟一样非常死非常臭，仿真和建模有绝对的分割线划开，建模就是综合，仿真就是验证，**两者没有深切的关系**。反之笔者却认为，仿真与建模不仅**关系深切**，而且两者之间只有**概念的差别**而已，即**一个是虚拟建模，一个则是实际建模**。

## 1.5 激励文本 ( Testbench )

激励文本或称激励文件，英文名为 Testbench，常见的后缀名有 .vt 与 .tb。笔者曾问过师兄，激励文件是什么，师兄却怒吼道：“激励文件就是激励文件啦，怎么！想死吗！？”，这是传统流派给予的回答。根据笔者的妄想，激励文本宛如绘出虚拟世界的一张白纸，亦即仿真环境。然而，我们就是创建这个环境的神明。

身为神明，我们有 5 项重任：

- 产生环境输入。
- 建立仿真对象。
- 产生虚拟输入。
- 产生虚拟输出。
- 观察世界，更正世界。

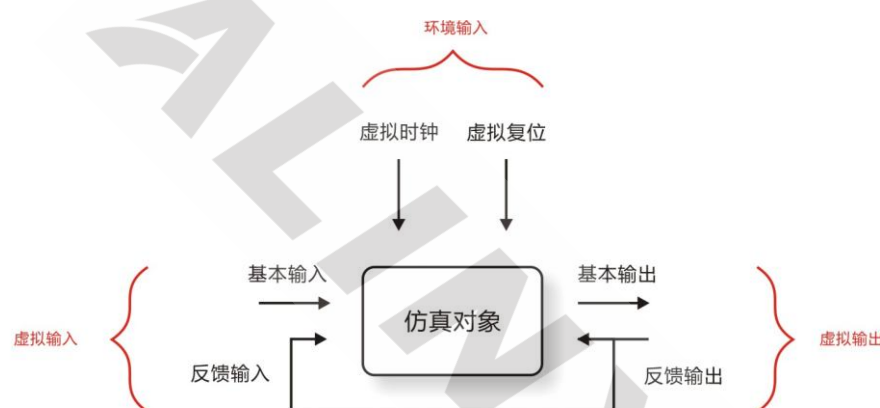


图 1.5.1 神明的任务之一。

如图 1.5.1 所示，那是仿真环境最基本的概念。首先是环境输入，环境输入一般则是模块所要的时钟信号还有复位信号，它们都是最基本的需要，这种感觉好比水源，空气等 ... 任一缺少仿真环境也无法成立。仿真对象好比居住在仿真环境的生物，它一般先在集成环境建模，然后实例化在激励文本当中，我们当然也可以直接在激励文本中描述仿真对象。

虚拟输入又指刺激，这种感觉好比生物的生存危机，如外敌如入侵，资源干枯等。仿真对象除了需要环境输入，仿真对象也要虚拟输入刺激才行。虚拟输入又分为基本输入与反馈输入，基本输入可以视为第一刺激，反馈输入则是第二刺激，形象点说：假设外星人入侵地球，此为人类的第一刺激。事情发生以后，人类不仅没有团结，而且还出现叛徒，这是人类的第二刺激。

除了，环境输入还有虚拟输入以外，激励文本还有虚拟输出。虚拟输出又指反应，这种感觉好比生物对应危机的反应。虚拟输出也有基本输出与反馈输出之分，它们也称为第一反应与第二反应。这种感觉好比外星人入侵地球以后，有些人会绝望，有些人会反抗，有些人宁愿成为走狗，此为人类的第一反应。为了解决那些叛徒，联合国实现全名监控，

此为人类的第二反应。

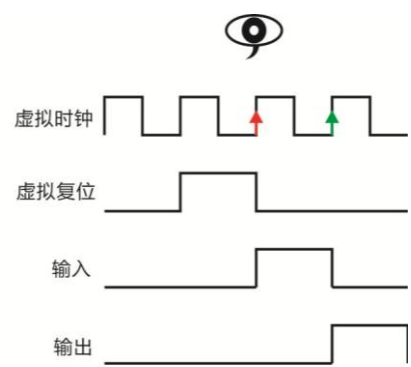


图 1.5.2 神的任务之二。

仿真环境对神来说不过是心血来潮的实验场所而已，神闲来无事创建了仿真环境 A，不久之后生物 B 便诞生。神为了刺激生物 B 进化，神召唤外星人攻击生物 B，此刻生物 B 会出现各种抉择。如图 1.5.2 所示，那是仿真环境的演化过程，也是仿真环境的运动，亦即时序。作为神，我们的眼睛“全能之眼”时常处在高处窥视一切，如果觉得那里不顺眼就插手哪里。

例如神觉得外星人入侵太无趣了，于是顺便召唤陨石下来（更动虚拟输入）... 又或者神觉得生物 B 太弱了，然后强化它们（更动仿真对象）。再假设神觉得演化步伐太慢了，结果神加快时间的流失（更动环境输入）。在此，读者可能会觉得这个神太可恶了，把生物当成玩具来玩 ... 嘛，别激动朋友，仿真本来就是那么一回事。

笔者说过，仿真既是虚拟建模，为此笔者开始自问：“激励文本是不是也要结构？”。答案是肯定的，激励文本有两种结构性，其一是布局的结构性，还有激励内容的结构性。

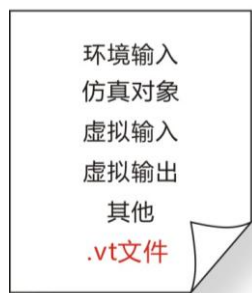


图 1.5.3 布局的结构性。

如图 1.5.3 所示，那是笔者根据习惯，然后为激励文本所建立的布局结构。环境输入一般都是置于激励文本的最顶端，余下是仿真对象的实例化，接下则是虚拟输入还有虚拟输出，最后就是其它。好奇的同学可能会怀疑布局结构的重要性，这位同学试想一下，如果世界万物失去结构会是怎样的场景呢？是不是无法想象呢？根据笔者的认识，激励文本之所以需要布局结构，其一是为了维护激励文本内容，其二是为了节能。

除了布局结构以外，激励文本还有激励内容的结构性。笔者一般都将虚拟输入还有虚拟输出称为**激励内容**又或者**激励过程**。那是因为虚拟输入还有虚拟输出原本就是一组操作，而且操作都是经由无数步骤组成。笔者曾经说过，描述语言是自由结构的语言，步骤又是顺序操作的单位，默认下它是没有结构它的，为此笔者应用了低级建模的用法模板。

```
1. reg [3:0]i;
2.
3.     always @ ( posedge CLOCK or negedge RESET )
4.         if( !RESET )
5.             begin
6.                 i <= 4'd0;
7.                 Start_Sig <= 1'b0;
8.                 WrData <= 8'd0;
9.             end
10.        else
11.            case( i )
12.
13.                0:
14.                    if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
15.                    else begin WrData <= 8'd8; Start_Sig <= 1'b1; end
16.
17.                1:
18.                    if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
19.                    else begin WrData <= 8'd9; Start_Sig <= 1'b1; end
20.
21.                2:
22.                    if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
23.                    else begin WrData <= 8'd10; Start_Sig <= 1'b1; end
24.
25.                3:
26.                    begin i <= i; end
27.
28.            endcase
29.
30.    /*******/
```

代码 1.5.1

如代码 1.5.1 所示，那是虚拟输入应用用法模板以后的例子。其中我们可以看见步骤 i 指向步骤，指向操作。用法模板除了为激励内容**提供最基本的顺序结构以外**，用法模板还会**帮助我们节能**。因为，如果仿真对象还有激励内容都有相同的用法模板，那么两者



之间都能应用相同的思路，还有相同的习惯。



## 1.6 仿真流程

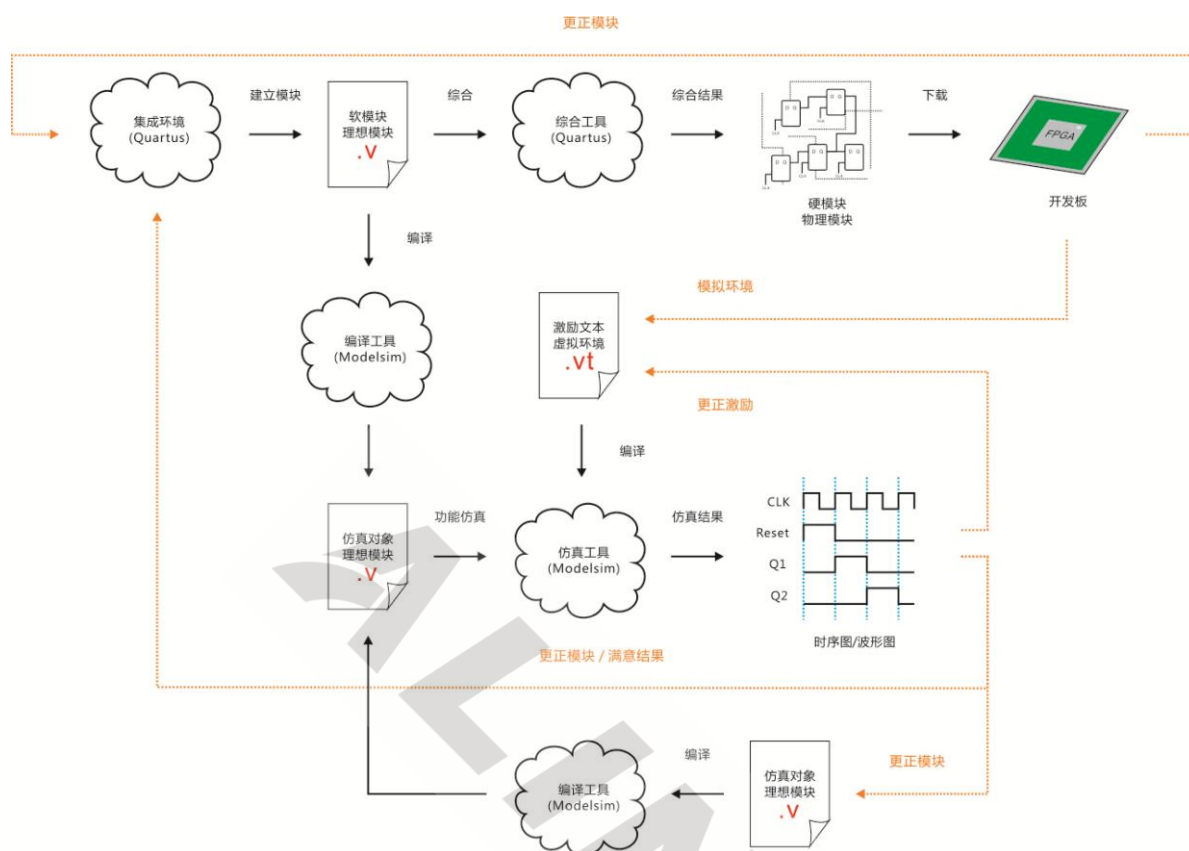


图 1.6.1 仿真流程图。

为了帮助小白扫盲，笔者绘出简单明了的仿真流程图。如图 1.6.1 所示，里边拥有许多流程与分支，然而一切流程与分支都起源于“集成环境”。接下来，让我们从“集成环境”开始，然后了解各个流程与分支。

### 流程 1：

- 集成环境生成软模块，也是俗称的建模。
- 软模块经过综合工具成为硬模块，也是俗称的综合或者编译。
- 硬模块生成以后便下载到开发板观察输出，如果输出结果不理想，就返回步骤 1。

软模块和硬模型都是笔者的专用语，软模型是意思是指[综合之前或者下载到开发板之前的理想模块](#)，[没有实际的逻辑资源](#)。反之，硬模型意思是指[物理模块](#)，[拥有实际的逻辑资源](#)。流程 1 也称为上线调试，亦即典型的调试方法，只要结果不符合预期，步骤就会返回开始，然后重复流程。该调试方式虽为最笨但也是最管用，不管调试对象是什么都适合。

### 流程 2：

1. 集成环境生成软模型。
2. 软模型经过仿真工具编译成为仿真对象。

3. 创建激励文本。
4. 激励文本作用仿真对象经由仿真工具输出波形（理想时序图）。

流程 2 是仿真最基本的仿真步骤，流程 2 相较常规的仿真流程，只有细节上的大同小异而已。在此有些读者可能会觉得疑惑，软模块与仿真对象都是理想模块，两者之间的差异究竟在哪里？理论上来说，软模块是未经加工的生肉，而且本质理想。换之，仿真对象是经过仿真工具加工过的正肉，本质也是理想。

流程 2 之后会产生 3 条通往不同分支。

#### **流程 2，分支 1（仿真结果符合预期）：**

1. 仿真结果符合预期以后，流程会返回集成环境。
2. 软模型经过综合成为硬模型。
3. 硬模型下载到开发板观察输出，如果输出结果不理想返回步骤 1。

流程 2，分支 1 基本上是流程 1 的翻版，亦即仿真结果符合预期，但不等于实际效果是否理想，所以需要进一步将软模块综合成为硬模块，再下载到开发板观察输出是否达到预想效果？如果是，流程结束；如果不是，重复流程 1。

#### **流程 2，分支 2（仿真结果不符合预期，仿真对象有问题，返回集成环境）：**

1. 仿真结果不符合预期以后，流程返回集成环境更正软模型。
2. 更正以后的软模型，再经过仿真工具编译成为仿真对象。
3. 仿真对象经由仿真工具输出波形（理想时序图）。

#### **流程 2，分支 3（仿真结果不符合预期，仿真对象有问题，返回仿真工具）：**

1. 仿真结果不符合预期以后，流程返回仿真工具更正软模型。
2. 更正以后的软模型，再经过仿真工具编译成为仿真对象。
3. 仿真对象经由仿真工具输出波形（理想时序图）。

流程 2，分支 2 与 3 直接的差距就是第二次更正的软模型是经过集成环境还是仿真工具。在此，可能会有同学觉得疑惑它们之间的差异何在？集成环境拥有较强的更正能力，但也更加耗时耗力；相反，仿真工具的更正能力虽然不及集成环境，但是耗时耗力相对较小。不管选择哪一种分支，都是见仁见智的事情。

此外，还有一个关键点，一些仿真对象可能会携带官方插件模块，许多时候仿真工具都对官方插件模块的支持不怎么友善，主要问题是编译手段还有仿真库的问题。安全起见，那些携带官方插件模块的仿真对象返回集成环境会比较妥当。

#### **流程 2，分支 4（仿真结果不符合预期，激励文本有问题）：**

1. 仿真结果不符合预期以后，流程重返更正激励文本。
2. 更正以后的激励文本再作用仿真对象输出波形图（理想时序图）。

流程 2，分支 4 是激励文本有问题，除了最基本的语法错误以外，读者还要注意一下。

仿真工具的编译器比较别扭，**必须遵守先声明后调用这个规则**。相比之下，集成环境的编译器比较醒目，声明还有调用的次序上下颠倒也没有问题。所以说，集成环境编译成功并不代表仿真工具一定编译成功，其中一定出现声明调用的次序问题。

总结来说，图 1.6.1 只是自定义的仿真流程而已，实际流程会因人而异。图 1.6.1 是笔者的经验总结，也是笔者的想象力爆发。它曾说过：“**流程会根据平衡发生变化 ...**”，这句话足让笔者深思许久，流程充其量只是便利的指南而已，活物不是跟死流程的机械人，所以笔者非常建议，流程看看以笑笑就好，不要过度纠结。



## 1.7 建模的切糕

切糕是什么？切糕是梦幻般的硬通货，传说投资切糕比起金银还有房地产更实在。市价好的时候可以换钱，灾难来的时候可以充饥，此外也有切糕达人一夜巨富的故事 ... 啊哈哈，以上纯属恶搞而已。切糕是新疆的传统食物，既是玛仁糖，也是体积庞大的饼干，**模块好比切糕**，其实这种比喻一点也不夸张，还不如说再适合不过。**一个系统模块的份量，差不多是一座 200 公斤重的切糕。**

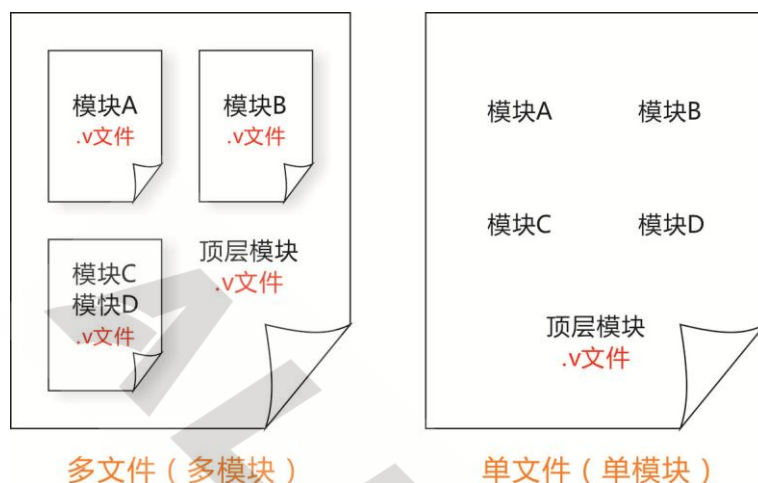


图 1.7.1 传统建模，单文件（单模块），多文件（多模块）。

传统建模有**单模块与多模块之分**。如图 1.7.1 所示，假设有四个模块 A, B, C 与 D，左图是多模块建模，模块 A 与模块 B 各有独自的 .v 文件，模块 C 与模块 D 则共用一个 .v 文件。右图则是单模块建模，也是笔者常常讽刺的单文件主义，这里所有模块共处于一室。传统建模都有一贯的致命缺点，就是**没有结构性可言**。接下来，让笔者逐个分析它们的缺点吧。

首先是单模块建模，也是**初学者最常犯的问题**，**所有模块都强挤在一起**，虽然在编辑方面有过人之处，作为代价，**单模块建模却为后期工作带来许多麻烦**。单模块建模宛如所有代码集于一身的 Main 函数一样，不过别忘了，并行语言不是顺序语言，它没有“步骤”这个最小的单位去支撑。此外，单模块建模不仅拖累模块的表达能力，单模块建模也不适合仿真。这种建模过度集中的情况，最终会演变成一块大切糕。

仿真还没有开始之前，我们的**心情就被切糕一样大的模块搞砸了**，200 公斤绝对不是普通人可以应付的重量，到头来，我们也只能傻乎乎看着它而已。所以说，**要一口气搞定切糕一样大的模块绝非易事**，于是人们开始动脑思考对策 ... 如果整座切糕无法处理，我们是否可以分块处理呢？就这样“多模块”的思想开始流行起来。

传统的多模块建模如图 1.7.1 左图所示，模块 A, B, C, D 分别分散在各别的.v 文件里，**模块虽然有分化，但是模块却没有规则分化**。这种感觉好比没有规则切整座切糕，结果每块小切糕都高矮不一。人类是一种自欺欺人的生物，**人们以为只要分散整座切糕，压力就会不复存在**，可是事实确是如此嘛？在此之前，先听笔者讲个故事：

某天，小明不幸在森林里迷路，衰到极点的小明被一只饿狼追杀，小明只有不停向前逃命而已。忽然间，眼前的状况把小明愣住了，因为眼前出现 2 条以上的分叉路，就在小明犹豫的一瞬间，饿狼就把小明推到了，可怜的小明死因却是**一瞬的犹豫**。可能读者会吐槽小明笨，干嘛停下犹豫，总之先跑再说 ... 旁人总是在旁冷言冷语，因为它们不是当事者也无法知晓小明，同时它们也忽略一个重要的关键，那就是“小明当下的状态”。

小明之所以失去冷静，是因为焦急，为什么焦急，因为被饿狼追赶，然后最该死的关键是突如其来的分叉路，它害死小明出现片刻的犹豫。当我们在执行仿真的时候，我们的处境就好比小明，不 ... 应该说是比小明更糟，因为小明起码被一只饿狼追杀而已。**一只饿狼的好比一个过程在运行。**

**并行语言发生问题绝不仅一处**，换句话说 Verilog 绝非一个时间执行一个过程，而是同一个时间有无数过程在同步执行。这种情况好比 N 只饿狼同时在追赶读者，普通人类根本无法同时应付 N 只饿狼。再者，**无规则分散模块会将仿真搞成像迷宫一样**，如果读者同时被 N 只饿狼在迷宫里追杀，想必读者的结局比小明更加惨不忍睹。为什么呢？原因很简单，无规则分散模块会将**无数过程搞成错综复杂**，在此**犹豫的时间一定会多过小明**，人一旦犹豫就会忘记如何前进。

曾经何时，笔者也像小明一样，同时被 N 只饿狼追赶，然而无规则分散模块的后果，仿真相似迷宫一样让足笔者**琢磨不定，失去方寸，找不到仿真的切入口**。仿真从模块 A 开始？还是从模块 B？除此之外，**激励文本的编辑工作也非常混乱**。模块 A 与模块 B 应该共享一个激励文本？还是模块 A 与模块 B 拥有各自的激励文本呢？想着想着，就觉得头好疼，蛋蛋好不舒服。

圣者说过：“**胡乱分散就是混乱**”，不规则分散模块会导致**模块变成迷宫，然后丢失仿真的方向**。既然如此，我们还不如不分散为好，可是整座模块的压力实在太大了，不分散又不行。结果分也不是！不分也不是！左右为难，还不如死去算了！冷静点我的朋友，**切糕一定要分，不过要有规则还有标准。**



图 1.8.1 低级建模的基本模块。

这种标准就是“**功能**”。模块按功能分化，这种感觉好比切糕按公斤划分。模块除了按“功能”**量化**以外，模块也根据按照“功能”**类化**。如图 1.8.1 所示，那是低级建模的基本模块，分别有控制模块，和功能模块。控制模块有正方形的外观，属于“控制类”；功能模块有长方形的外观，属于“功能类”。无论是哪一种模块，它们都遵守低级建模的准则，亦即“**一个模块，只有一个功能**”而已。

假设一套进食动作有以下几个步骤：



1. 举起食物
2. 张开嘴巴
3. 送入嘴中
4. 闭上嘴巴
5. 咬碎食物
6. 吞入肚子

一套完成的进食动作，基本上由 6 个步骤组成，每个步骤可以视为“功能类”。然而，左边的数字 1~6 却可是视为“控制类”。

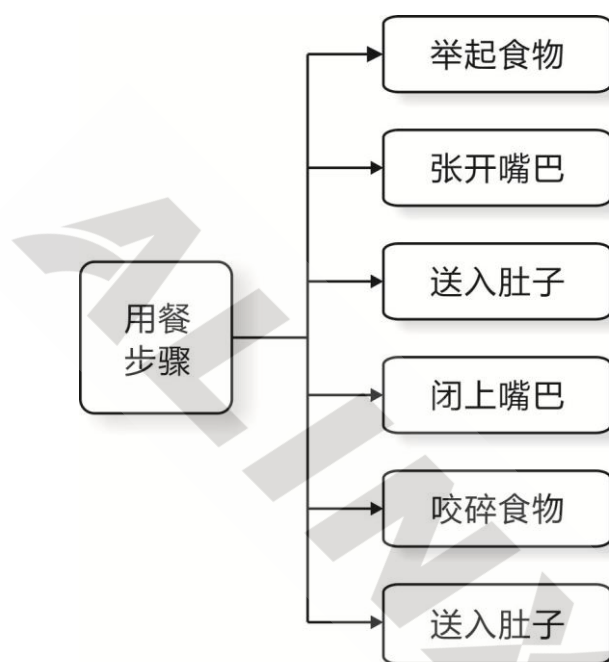


图 1.8.2 用餐模块化。

一套进食动作按照“功能”分化以后，结果如图 1.8.2 所示。每一个模块仅有一个功能类，如：举起食物，张开嘴巴，送入肚子，闭上嘴巴，压碎食物，送入肚子都是拥有一个模块”。至于用餐步骤，它属于“控制类”，它也有一个模块。

如果一座大切糕可以品均又有规划，分散成为每块小切糕的话，那么每块小切糕好比独立的个体一样 ... 这种感觉好比整座切糕被分散成为无数可以承受的最小份量，最后再由耐心将全部慢慢吃掉。模块有规则分化以后，功能的复杂度也会跟着简化，形象点说就是一座复杂的大迷宫，无数分散成为简单的小迷宫，如此一来仿真就不容易迷路了。此外，控制模块就好比迷宫的入口一样，一切从它开始。为此，我们可以这样说：有规则分化模块除了可以简化迷宫以外，它也为仿真建立入口。

除此之外，该准则还有一个好处就是分散功能，分散过程，举例而言：一个大功能好比 10 只饿狼的狼群，面对它们我们不仅没有胜算，而且压力也很大了。不过，大功能经过分散以后，10 只饿狼的狼群也会因此分散成，我们虽然不能一次单挑 10 只饿狼，但是我们可以 10 次单挑 1 只饿狼。

总结来说,有规划分散模块除了上述的好处以外,有规划分散模块也会为人带来好心情。这种感觉好比切糕太大我们会觉得反胃,切糕切成稀巴烂我们也会觉得恶心,不管哪一种情况都会影响我们进食的心情,仿真也是一样的道理。还有一点读者必须好好记住,建模还有仿真有千丝万缕切不断的关系,前期有好的建模,后期就有轻松的仿真。



## 1.8 仿真的切糕

笔者曾在 1.8 小节当中说过，**单模块或者没有规则的多模块建模会将仿真搞成迷宫，让仿真失去方向，最糟还会搞坏心情**。反之，有规则的多模块建模，不仅会简化迷宫也会给仿真前带来好心情。切糕除了出现在建模以外，切糕也会出现在仿真之前还有仿真之间。不过切糕又会以什么形式出现在仿真当中呢？这个问题就让笔者来慢慢长谈吧，好让读者有个深刻的认识。

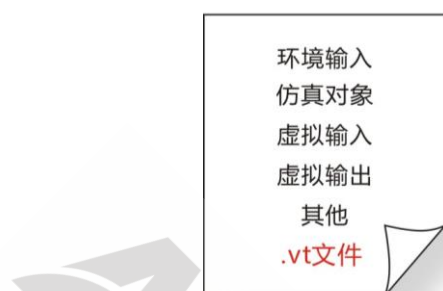


图 1.8.1 激励文本。

笔者曾在小节 1.5 介绍过激励文本的作用。根据笔者的习惯，激励文本可以按着图 1.8.1 所示的次序插入相关激励内容。处于激励文本的顶端是环境输入，其次就是仿真对象。仿真对象一般都是先在集合环境里完成建模，然后再经由激励文本将其实例化成为仿真对象，如：

```
function_module.v
module function1( input CLK, input RESET, input SigD, output SigQ );
...
endmodule

function_module.vt
module function1_simulation();
function1 U1( .CLK(CLK), .RESET(RESET), .Sig(SigD), .SigQ(SigQ) ); // 仿真对象实例化，出入端声明
...
endmodule
```

代码 1.8.1

笔者先在集成环境里创建名为 function1 的模块，为了仿真 function1 模块，笔者必须先建立仿真环境。为此，笔者建立一个名为 function1\_simulation 的仿真环境名，然后将 function1 实例化为 U1，过程如代码 1.8.1 所示。

根据代码 1.8.1 所示，function1 有 4 个出入端，亦即 CLK 信号，RESET 信号，SigD 信号，还有 SigQ 信号。在此，读者尝试想象一下，像整座像切糕一样大的模块到底有多少出入端呢？答案是肯定的，亦即非常多，而且还多到不可想象。**过多的出入端无疑会**

拉长激励文本，结果导致激励文本成为臃肿的大肥，这是切糕出现在仿真中的第一种形式。

根据笔者的经历，20 几个出入端还算是小切糕的程度而已，随着建模层次不断提升，出入端的数量也会不断增加，一个系统模块有 50 来个出入端也是非常普遍的事情。分散模块就是为预防这个问题，逻辑而言，模块愈是分散，出入端的数量理应也会分散，而且实例化也会变的更加轻松，激励文本因此也会变得更加苗条。浏览苗条的激励文本，好比我们观察苗条的美女一般，骨干的美丽还有完美的曲线（简洁直观的内容），不仅令我们流失压力，也让我们的心情变得愈加愉快，情绪越来越有干劲。

知晓切糕的第一种形式以后，接下来让我们寻找切糕的第二种形式。

出入端过多除了拉长激励文本以外，它也会影响激励过程的编辑工作。激励文本有两种激励内容，亦即虚拟输入还有虚拟输出，尤其是虚拟输入，出入端的影响会是更重。

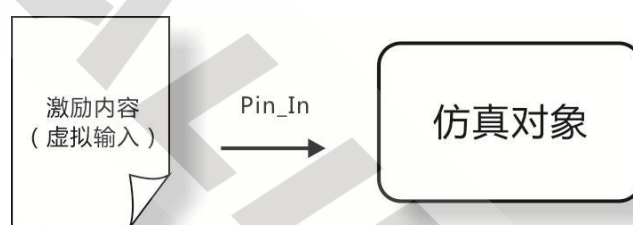


图 1.8.2 虚拟输入。

```
1.    reg [3:0]i;
2.
3.    always @( posedge CLK or negedge RSTn )
4.        if( !RSTn )
5.            begin
6.                Pin_In = 1'b1;
7.                i <= 4'd0;
8.            end
9.        else
10.            case( i )
11.
12.                0,1:
13.                    begin Pin_In <= 1'b1; i <= i + 1'b1; end
14.
15.                2:
16.                    begin Pin_In <= 1'b0; i <= i + 1'b1; end
17.                .....
18.
```

虚拟输入有基本输入与反馈输入，图 1.8.2 还有代码 1.8.2 则是基本输入最简单的例子。如图 1.8.2 所示，激励内容经由 Pin\_In 产生基本输入刺激仿真对象。

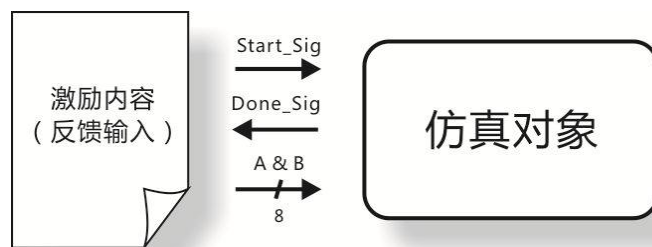


图 1.8.3 反馈输入的假想图。

```

1.    reg [3:0]i;
2.    always @ ( posedge CLK or negedge RSTn )
3.        if( !RSTn )
4.            begin
5.                i <= 4'd0;
6.                A <= 8'd0;
7.                B <= 8'd0;
8.                Start_Sig <= 1'b0;
9.            end
10.        else
11.            case( i )
12.
13.                0:
14.                    if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
15.                    else begin A <= 8'd2; B <= 8'd4; Start_Sig <= 1'b1; end
16.
17.            endcase

```

代码 1.8.3 虚拟输入。

图 1.8.3 还有代码 1.8.4 则是反馈输入简单的例子。首先，仿真对象会根据基本输入的激励产生相关的反应，即基本输出。如代码 1.8.3 所示，在步骤 0 当中，A 赋值 8'd2，B 赋值 8'd4，Start\_Sig 则拉高，这是基本输入。仿真对象完成操作以后，它会经由 Done\_Sig 反馈完成信号，这是基本输出。虚拟输入接收 Done\_Sig 以后，便会拉低 Start\_Sig，这是反馈输入。

眼睛犀利的同学想必已经猜想到切糕处的第二种形式了吧！没错，**仿真对象的功能越是复杂，出入端的数量就越多，期间激励内容的编辑工作不仅会变得繁重，而且激励文本**

也会越来越臃肿，内容变得非常杂乱。当简单的内容剧增到某种程度以后也会变成枯燥乏味，这种感觉好比读者被老师罚写名字，假设读者有“王一二”般的简单名字，假设老师罚写 500 次，就算名字再怎么简单，读者也会哭死。激励内容的编辑工作也是同样的道理。

笔者相信解读时序信息是最头疼问题 ... 但是，让真正笔者担心是没完没了的时序。我们知晓顺序语言如果当前的步骤无法奏效，那么问题一定是上一个步骤有地方弄错了，顺序语言好比单行通道一样，就算通道再长问题再多，只要努力的走下去，黎明一定会出现。

红苹果敲响牛顿的头，让他发现万有引力，不过野心勃勃的牛顿预想挑战万有定律，牛顿最后发现万物只能按照时间演变下去，这种感觉好比我们走在又长又暗的单行通道里，单调即无趣。牛顿虽然克服万有定律，但是他的人生观宛如牛顿力学一般非常消极，因为人生只能按照时间一直走下去而已 ... 这种情况，我们称为单向。

牛顿死后几个世纪，量子邪说也随之兴起，量子力学之所以称为邪说是因为主意完全违背牛顿力学，举例来说：一只猫将其关在盒子里始终不打开，根据邪说，世界很可能会分支成为两条不同演化的世界线，亦即并行世界。世界有可能走向猫已死亡的世界线 A，世界也有可能走向猫还活着的世界线 B，其中猫死亡与否是造就世界线 A 与 B 的变数。这种情况，我们称为多向。

仿真可是单向，也是多向。多向的仿真好比量子邪说般，信号亦即变数，仿真对象的功能越多，信号的数量理应也是一样多，结果变数也会变多。变数会衍生分支，无数的变数最终会衍生数之不尽的分支，从而海量仿真信息。

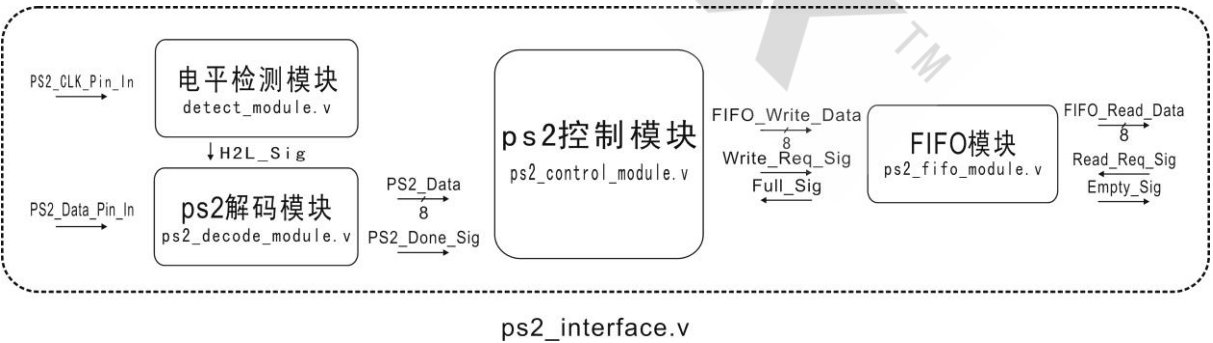


图 1.8.4 ps2 接口模块的建模图。

为了给予读者一个感知的认识笔者就用一张建模图来解释。如图 1.8.4 所示，那是一幅 ps2 接口的建模图，ps2 接口差不多有一座小切糕的级别，它有 3 个功能模块，1 个控制模块，信号的数量总和为 10，亦即：PS2\_CLK\_Pin\_In，PS2\_Data\_Pin\_In，PS2\_Data，PS2\_Done\_Sig，FIFO\_Write\_Data，Write\_Req\_Sig，Full\_Sig，FIFO\_Read\_Data，Read\_Req\_Sig，还有 Empty\_Sig。



ps2 接口模块的激励过程大致如下：

我们必须先产生 2 个主要的虚拟输入，亦即 PS2\_CLK\_Pin 和 PS2\_Data\_Pin\_In。电平检测模块接收输入就会产生输出 H2L\_Sig 刺激 ps2 解码模块，ps2 解码在 PS2\_Data\_Pin\_In 还有 H2L\_Sig 双重的刺激下就会产生 PS2\_Data 还有 PS2\_Done\_Sig 的输出进一步刺激 PS2 控制模块。

PS2 控制模块接收刺激之后，会经过 FIFO\_Write\_Data 和 Write\_Req\_Sig 等输出刺激 FIFO 模块，FIFO 模块也会使用 Full\_Sig 反馈 PS2 控制模块。最后 FIFO 模块会经由顶层信号 Read\_Req\_Sig 刺激接而输出 FIFO\_Read\_Data 还有 Empty\_Sig 等顶层信号。

读者千万别忘了，上述流程只是简单的文字描述而已，此外上述流程仅讲述一条成功的流向而已。单文字流程，笔者的头脑已经快要发疼了，假设仿真流程既不是文字描述，也不仅一条的话 ... 没错，在实际的仿真当中，流程不仅没有成功的保证，流程也不可能只有一条而已，此外信息也不是文字描述，而是错综复杂的信号。多向仿真说过，信号就是变数，信号越多变数越多，变数所衍生的流程也会越多，许多流程的信息相乘以后会产生令人窒息的信息量。

没错，这就是糕的第三形式，为了让读者深刻体验切糕的威力，笔者继续用图 1.8.5 来举。假设有两条信号 H2L\_Sig 与 PS2\_Data\_In 刺激 ps2 解码模块，此刻会产生以下几个可能性：

可能性一：H2L\_Sig 先刺激 ps2 解码模块，然后 PS2\_Data\_In 再刺激 ps2 解码模块；  
可能性二：PS2\_Data\_In 先刺激 ps2 解码模块，然后 H2L\_Sig 再刺激 ps2 解码模块；  
可能性三：H2L\_Sig 与 PS2\_Data\_In 同时刺激 ps2 解码模块；

假设 1 个可能性仅为 1 条流程，然后一条流程需要仿真 1 次。上述有 3 个可能性表示有 3 条流程，因此需要仿真 3 次。读者请仔细一想，上述 3 条流程仅是 ps2 解码模块的局部情况而已，试问 ps2 接口有几个模块？结果又会产生多少可能性呢？为了取得正确的结果，我们又要仿真多少次呢？没错，这就是仿真最可怕的地方，亦即无限可能性，海量仿真信息！

在此，读者的蛋蛋是不是开始在发颤呢？人不仅时间有限，而且精力也不多，庄子一直劝告我们，“别用有限的身体去追求无限的目标”。当模块过度集中以后，出入端增加仅是小事一桩，真正让人觉得害怕的是，无数信号相乘以后做照成的仿真信息。为了避免仿真变成悲剧，我们必须想尽办法分散模块，分散功能，分散信号，控制变数，减少流程 ...

---

最后，让我们来总结一下，这个小节究竟有几种形式的切糕？

1. 仿真对象的功能越复杂，出入端越多，激励文本越臃肿。

2. 仿真对象的功能越复杂，出入端越多，激励内容越混乱。
3. 仿真对象的功能越复杂，信号的数量越多，变数也会越多。

切糕除了小节 1.8 的介绍以外，还有本节的 3 种形式。切糕之所以给人绝望是因为天生俱来的**压倒性份量** ... 描述语言不同与高级语言，那些许多烦人又猥琐的底层工作高级语言都交由后台处理，我们只要关心代码的正确性即可。反之，描述语言的**后台能力很弱**，结果**许多底层工作必须交由我们自己承担**，但是人的精力是有限的 ...

最后笔者再次强调，**仿真真正最可怕的地方就是数之不尽的流程，还有解析不玩的海量信息**。世界线还有平行世界虽然听起来感觉非常浪漫，可是浪漫的背后往往都是无限残酷，这种感觉好比 Stein : Gate 的男主，为了拯救青梅竹马，却无法避免世界线的收束，结果无数次目睹她的死亡。为了避免仿真发生悲剧，我们应该尽可能减少功能数量还有变数，以最小的力气引导仿真流程走向预想的途径。

有些同学可能还在幻想自己成为勇者挑战恶龙然后征服所有世界线，但是笔者还是告诫读者**千万不要过分高估自己的承受能力**，就算读者有再好的吞吐量，吃了第一块大切糕，绝对没有能耐吃第二块大切糕，这点笔者可以拍胸保证，因为笔者就是过来人。如今，笔者已经放弃当初作为勇者的身份，因为笔者发觉自己的身心已经被切糕搞到破烂不堪。



## 1.9 自动思想的简介

传统流派虽然手段强硬而且暴力，不过它们相信气势可以闯出一片天，是典型自信的主攻派，然而，笔者本质柔弱还有节能，宛如不停往下游走的水资源，是典型怕事的撤退派。面对宛如切糕一样大的模块，[传统流派有可能会选择消耗战，誓死吃掉整座切糕](#)。换之，笔者没有那种自信，也没有那种胆量，所以笔者选择[分化切糕到最小数量，然后再逐个吃掉](#)。

笔者当然晓得过度分散模块的缺点，亦即[重复性的工作量](#)，举例而言：假设一块切糕有 10 个功能，切糕经过分散以后成为 10 块小切糕，因此笔者必须重复十次仿真的工作。很不巧这是[马死落地走的最佳方法](#)，鬼叫笔者的承受能力差，无法一口吃掉切糕。此外，模块过度分散也会导致[仿真失去整体性与连接性](#)，亦即我们[只能看见个体的局部仿真状况，而不是整体的仿真状况](#)。这个问题让足笔者纠结过一阵子 ...

师兄曾说过：“仿真是允许强者淘汰弱者的残酷世界！”，不管笔者怎样否定传统流派，这是唯一一句受到笔者赞同的话，仿真本来是挑战一块又一块的切糕，那些承受能力差的弱者，根本没有仿真的资格。这句话听起来，虽然让人觉得不舒服，然而[这就是现实](#)。不是仿真拒绝我们，而是弱小让我们失格。

笔者是弱者，所以笔者没有能耐承受大切糕，为此必须将大切糕分化为无数小切糕。然而大切糕经过分化以后就会失去连接性与集中性，原先的完整性就会受到破坏。笔者只要一天找不到[连接小切糕的关键，回复切糕的完整性](#)，笔者不仅没有资格仿真，笔者也没有资格去挑战切糕 ...

为了寻找那份关键的东西，笔者苦思冥想了好久，想着想着 ... 意识不知不觉中又来到诸神逗留的乌托邦，然而它却在那里等待笔者。

“在这里，生命不受负责也不受管理，[每个生命既是个体也是整体](#)”，它说道。

“既是个体也是整体的生命，这不矛盾吗？”，笔者反驳道。

它摇摇头，然后指向前方继续说道。

“那里有一处花田，花草虫蚁都在生活着，生存压力让它们无暇顾及其它，这是个体表现。从旁观望，无数个体存造就眼前的花田，这是整体表现 ... 理解了吗，孩子？”

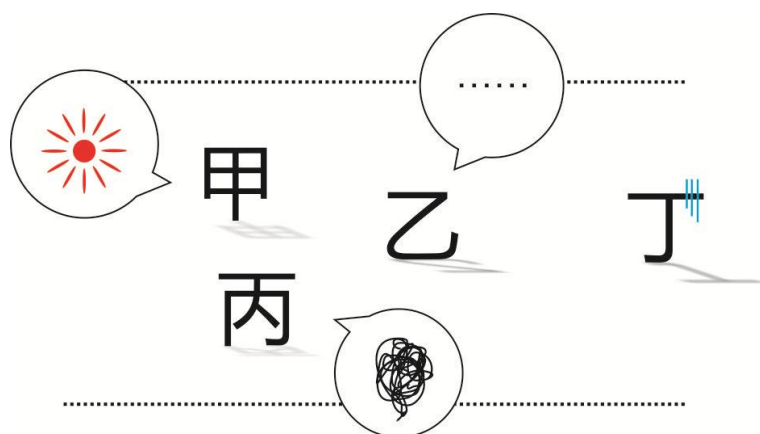


图 1.9.1 主动设计概念。

神的话永远都是充满寓意，为了了解它的话语，我们必须换个角度去思考问题。如图 1.9.1 所示，假设有甲乙丙丁四只家伙在交谈，小丁因为不擅长交流所以被排斥在外。如果站在局外者的角度观察这起交谈，小丁则会认为**这起谈话必须甲乙丙三人无间断合作才能完成**。反观，如果站在当局者的角度去观察这起交谈的话，事实上**小甲只说想说，小乙只听想听，然而小丙边听边说，每个人只作自己想作的而已**。

假设小丁不小心错过交谈内容却又想了解详情 ... 此刻，如果小丁是传统流派的门徒，它认为**最佳方法莫过再现现场还有执行实时监控**。换之，如果小丁换作笔者，它会认为实时监控和再现现场**太耗神耗力**了，此外小丁还要**同时接受三人份的谈话信息**。所以小丁选择逐个拜访小甲，小丙，还有小乙 ... 小丁**了解片断的详情后**，然后将**无数片断信息都交由想象力去结合**。

并行语言是一门偏硬的语言，由于本质限制的关系，它无法实现顺序语言所拥有的函数调用还有传递参数等能力，不过并行语言可以建立仿顺序结构的效仿顺序操作。但是，没有步骤的弱点，导致并行语言不适合过多内容集于一身，因为会降低解读能力，所以多模块建模就流行起来。低级建模也是多模块建模之一，不过是有准则的多模块建模。多模块建模却有一个**缺点，亦即模块分化以后，模块会失去集中性与连接性**。

为此，建模图的作用就非常重要了，建模图不仅**可视化模块之间的关系**，也能**加强个体整体的存在感**，过后**经由想象力再现整体效果**。建模阶段，**个体模块只做想做，只听想听，不用顾忌其他**。完后，**再运用想象力联系所有个体情况**，整体情况就会出现在脑海当中。这种思想笔者称为**主动思想**，应用这种思想的技巧也称为主动设计。

人类有两种记忆能力，左脑拥有清晰记忆也擅长处理具体的信息；右脑拥有模糊激励也擅长处理抽象的信息（想象）。主动思想的作用就是**将左脑的记忆压力，还有处理压力分担给右脑**，好让**用脑保持平衡而不是一面倒**。人类的头脑是一件不可思议的工具，右脑虽然没有左脑的强劲分析能力，但是右脑的想象能力比起左脑可优秀许多。例如记忆陌生人的面貌，如果用左脑记忆的话，左脑竟可能会多收集更多的面部数据，但是面部信息远远超越左脑的承受能力，所以用不了多久，左脑就会当机。

反之右脑只会**收集少数面部特征，其余细节任由想象力填充**，亦即模糊能力。假设将两

张相似的明星脸交由电脑对照，不管两张脸给人的感觉再怎么相识，电脑始终会给予否定的答案。如果将明星脸交托左脑分析，左脑也会给予相同的否定，因为左脑和电脑不会给予“好像，似乎，相识”等暧昧的回应，所以说明星脸是右脑引起的浪漫现象。

根据图 1.9.1 的谈话情况，小丁在不知不觉之间发挥了主动思想的能力。首先它无意识记忆甲乙丙最有特征性的谈话内容，假设甲在这次谈话中说了，“海边”，乙说了“美女”，丙则说了“BBQ”。小丁经由脑补以后，小丁会认为甲乙丙在谈论“美女在海边 BBQ”，又或者“美女在海边被 BBQ”。主动思想除了应用在日常生活种以外，主动思想也能应用在仿真当中。

当我们将整体模块分化为无数个体模块以后，为了应用主动思想充当个体模块之间的连线，为此个体模块必须露出特征，给人留下强烈的局部印象。这种感觉好比笔者记忆美女 A 的脸庞，不过笔者很懒，所以笔者不会记忆所有脸部细节，为此笔者会记忆她怜人的泪痣，性感的粉唇，还有可爱的娃娃头。最后在笔者的记忆中，美女 A 的脸庞除了泪痣，嘴唇，还有娃娃头特别清晰以外，其它细节怎样也无所谓。

仿真也有同样的道理，如果个体模块有强烈的特征，经过仿真以后，它就会成为清晰的局部信息，然后深深烙印在我们的记忆中。无数清晰的局部信息刻入记忆以后，再由想象力将所有细节结合起来，一幅完整的整体信息就会出现在我们的脑海当中。换句话说，我们利用想象力充当个体模块之间的黏糊剂，但是作为前提，个体模块必须清晰，而且印象要非常强烈，不然的话我们作不到脑补时序。



## 1.10 不可仿真对象的简介

这个世界上存在一些危及生命的仿真对象，例如切糕级别的模块（功能过多的模块），但是还有一些仿真对象**比切糕更危险**，笔者称为不可仿真对象 ... 不可仿真对象实际上**不是不能仿真，而是没有仿真意义，或者阻碍仿真**。根据笔者的认识，不可仿真对象有三种，亦即“超乱模块”，“超烦模块”还有“超傻模块”。

超乱模块顾名思义就是模块**内容过度杂乱以致无法解读**，说白了就是**解读能力极差的模块**。这类模块好比，**官方插件模块，还有就是失去结构的模块**。官方为了保护商业秘密，结果会故意搅乱模块的内容，这是一种非常恶心的行为，不过它们起码也做好善尾的工作，确保插件模块即时健全，还有附加说明书，所以它们情有可原。

仿真本来就是处在虚拟的环境下模拟模块的功能状态，**如果模块内容本身也无法确切表达，仿真就会失去原来意义**。此外，仿真也是**用来测试不健全的模块**，然而官方插件模块不仅健全，而且还有许多资料支持。仔细一想，我们为何还要浪费无谓的精力在它们的身上呢？为了节能，除非是特殊的情况，否则笔者是不会仿真官方插件模块。

“超烦模块”是指**用时过于沉长**的模块，**让人等到烦心**。这个问题一般主要是由**计数器或者定时器灌水过多**引起的悲剧。仿真的最小单位不是现实中 1 秒或者 1ms 等物理时间，而是 N 个时钟。例如典型的流水灯实验，流水间隔至少都是 100~1000ms 之间，结果灌水会是非常严重。

根据笔者的惨痛经历，20Mhz 为例的仿真时钟，**仿真时间一般不推荐操作超过 1 秒**，1ms~10ms 之间最为理想，**过长的仿真时间会拉长 wave 界面**。曾经何时笔者也是傻子一名，为了验证正确的计时结果，笔者常常在仿真界面上拖来拖去，实在累人 ... 天生节能的笔者绝对不能容许这种滑稽的事情发生在自己身上，于是笔者左思右想最终想到整合技巧。整合技巧有各种作用，其中一项功能就是精密控时，任何长时间计时都能在代码上搞定，而不是经由仿真。

最后还有一个不可仿真对象就是“超傻模块”，故名思议超傻模块是指**功能过度简单明了**，这种功能简单到一眼就能辨出是园是扁，难道我们还会大费周章，敲锣打鼓去仿真它吗？当然不会啦。超傻模块典型例子有：多路选择器，加码，解码等小组合逻辑还有小功能模块。

最后让笔者来个简单的总结吧：

超乱模块：模块内容极乱，解读不能，例子：官方插件模块，无结构的模块。

超烦模块：模块内容水分，极度费神，例子：定时器，计数器。

超傻模块：模块内容单纯，浪费气力，例子：多路选择器，输出器等小组合逻辑。



总结：

1. 虽然尽是毫无里头的扫盲文，不过扫盲文的作用究竟有多大读者应该心中有数。

“仿真有建模三倍的负担”这句话一点也夸张 ... 建模亦即实际建模，除了建模还是建模；然而，仿真除了建模以外，还有创建激励还有解读时序信息等猥琐的工作要我们去干。说实话，同时兼学建模和仿真很容易两头不到岸，这也是学习最忌遇见的窘境。笔者希望读者可以透过扫盲文做好最起码的思想准备。

仿真是什么？虽然在字面上“调试”还有“仿真”相较有同义的嫌疑，不过“调试”不等价“仿真”，“调试”为求单向结果，换之“仿真”为求多向过程。简单说，调试是顺序语言的测试手段，然而仿真是并行语言的测试手段，后者相较前者不仅不单纯，而且让人匪夷所思甚至难以捉摸。

调试与仿真之间也是顺序语言与并行语言之间的思路问题，错用思路宛如遇上一堵杀人偿命的隐形巨墙。笔者已经无数目睹同学翻墙不成反被摔死，实在可怜 ... 为什么他们要模仿飞蛾寻求绝命之火呢？事实上这些悲剧，传统流派实在功不可没，它们手段强硬而且暴力，强迫学习从来不说明，无辜之人才会接续遭殃。

“仿真结果一定是物理时序！”

“仿真需要验证语言！”

“仿真和建模是两回事！”

“我们才是正道！”

正是这些花言巧语骗尽傻子卖猪仔，曾经何时笔者也是一只傻傻上当的猪仔 ... 解脱不遂之后，笔者才渐渐领悟，传统流派只会传授蜻蜓点水的表面功夫，许多重要细节都会潇洒带过。仿真不当容易受伤，不过仿真真正让人恐惧的是——切糕。切糕是隐藏在仿真背后的绝望，切糕会改变形态躲在黑暗处，然后无时无刻等待时机扑食希望。切糕一般是指压倒性的工作量还有信息量，主要是仿真手段不当所产生的副产物。传统流派不知真傻还是装傻，那么大的切糕它们既然可以视而不见？

笔者被切糕施虐的惨痛的经历最终浓缩成为这本书。这个世界（仿真）太大了，传统手段只会沦落切糕的猎物，为求生存异常手段是必须的。这本书是这个世界（仿真）的地图，也是这个世界（仿真）的生存手册 ... 此外，这本书也能给予出发前的心里准备。读者好好切记，这个世界（仿真）是由并行法则支配的空间，以往的顺序手段已经一去不返，心存半点也会引来悲剧。

“**仿真是会死人的学习 ...**”，笔者道。

## 第二章 Modelsim 就是电视机

### 2.1 连接 Modelsim

传统 CRT 电视给人印象笨重不过操作却非常简单，现代液晶电视给人印象轻便不过操作却非常麻烦。如果传统 CRT 电视与现代液晶电视不小心结合起来，结果就成为现在的 Modelsim。许多朋友常认为[学习 Modelsim 等价与学习仿真](#)，不过这也是美丽的误会而已。Modelsime 不过是仿真所需的重要工具而已，然而 Modelsim 常用的功能如下：

- (一)编译软模块成为仿真对象。
- (二)播放波形图。

根据笔者的认识，不管模块是否综合成功，[只要一天未下载到开发板都视为本质理想的软模块](#)。仿真之前，[软模块必须经过 Modelsim 编译成为仿真对象](#)才行。此外，[播放波形图 Modelsim 最为重要的任务](#)，也是学习 Modelsim 最为重要的课题。基本上只要掌握上述两项功能，就已经足够了。不过。首先我们需要搞懂启动 Modelsim 的方法。

Altera 公司的集成环境——Quartus II，传闻说[版本 10.0 以前自带官方的仿真工具](#)，但是自认不如的它已经心灰意冷回老家去了。Quartus II 升级为版本 10.0 以后，Modelsime 自然而然就取代它的位置。这个故事告诉我们，Modelsime 是一位外来者，预想使用 Modelsim 之前，[Modelsime 必须做好连接 Quartus II 的工作](#)。

Modelsime 启动文件名是 modelsim.exe，一般藏身在以下路径：

C:\altera\12.1\modelsim\_ase\win32aloem\modelsim.exe //第三方免费版本

笔者所使用的 Modelsim 是 Altera 自定义的版本，亦即 Modelsim Altera Started Edition，简称 ase，源目录一般都放置在 altera 的目录之下。除了 Started Edition 以外，Altera 还有自定义的付费版本，亦即 Modelsim Altera Edition，简称 ae，然而默认路劲如下：

C:\altera\12.1\modelsim\_ae\win32aloem\modelsim.exe //第三方付费版本

好奇的同学可能想问道“[免费版本付费版本究竟差异何在？](#)”。根据笔者的理解，两者之间的差别就在于付费版本的“仿真库”远胜免费版本，而且付费版本也没有行数限制。不过笔者却认为，只要[巧用理想时序与仿真技巧](#)，免费版本已经足够应用了，因为逝去的老嬷嬷一直告诉笔者“[东西够用就好，多了就是浪费](#)”，所以笔者不会烦恼什么付费版本，再加上[付费版本一年要九百多刀](#) ... 真是吓死人。那么，什么又是仿真库？在此先留个悬念。

抱歉，笔者不知不觉又离题了，让我们切回话题吧。首先找到 Modelsim 启动文件的藏在目录，笔者的话是：

C:\altera\12.1\modelsim\_ase\win32aloem

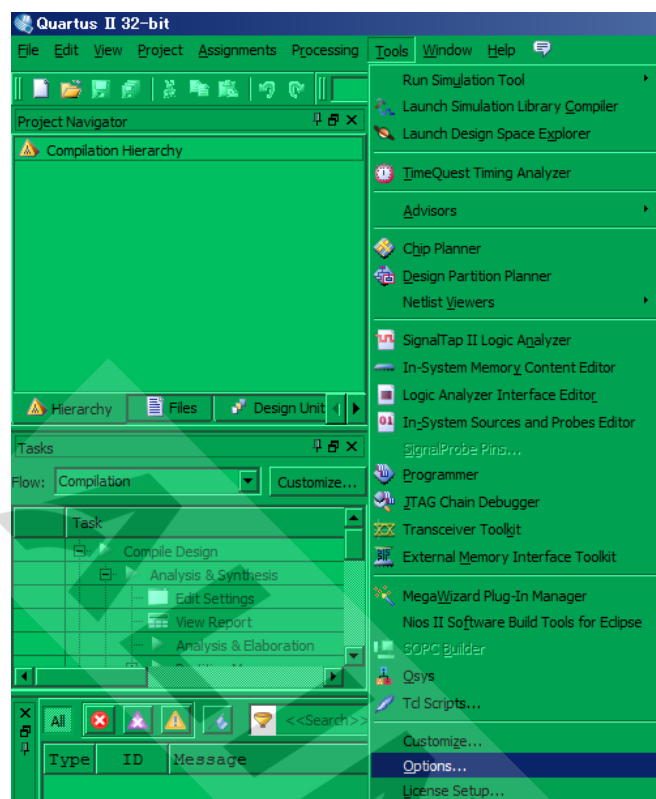


图 2.1.1

然后打开 Quartus II，接着调出 Tool 菜单，往下打开 Option，过程如图 2.1.1 所示。

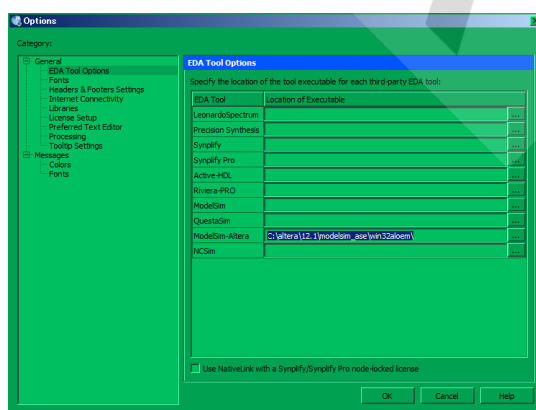


图 2.1.2

随后，点击右边菜单树下的 Eda Tool Options，再将原先的路径拷贝到 Modelsim-Aletra 的空白处，完后点击 Ok 生效。就这样，Modelsim 连接 QuartusII 的工作就完成了，真是可喜可贺！

## 2.2 自动编译，半自动编译，手动编译

Modelsim 开始执行仿真之前需要执行一堆让人觉得猥琐的准备工作，然而这些准备工作到底又包含那些步骤呢？

1. 创建仿真设计
2. 加载 .v 文件
3. 加载 .vt 文件
4. 生成仿真对象
5. 生成仿真环境
6. 启动仿真
7. 加载仿真信号
8. 播放仿真

一般，准备工作包含上述 8 个步骤，具体解释往下继续 ...

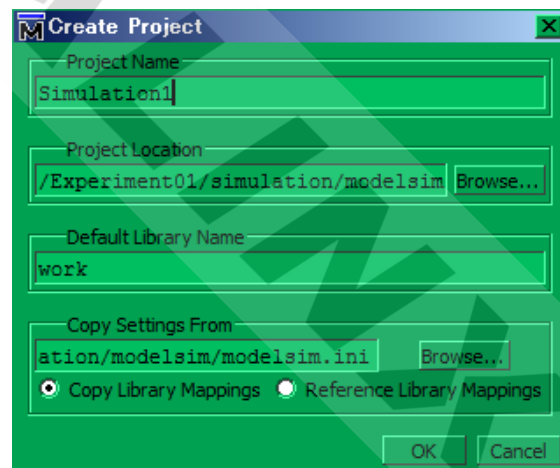


图 2.2.1 创建仿真设计

其一，创建仿真设计如图 2.2.1 所示，有时候也称为创建仿真项目，自动编译与半自动编译会简化该步骤。

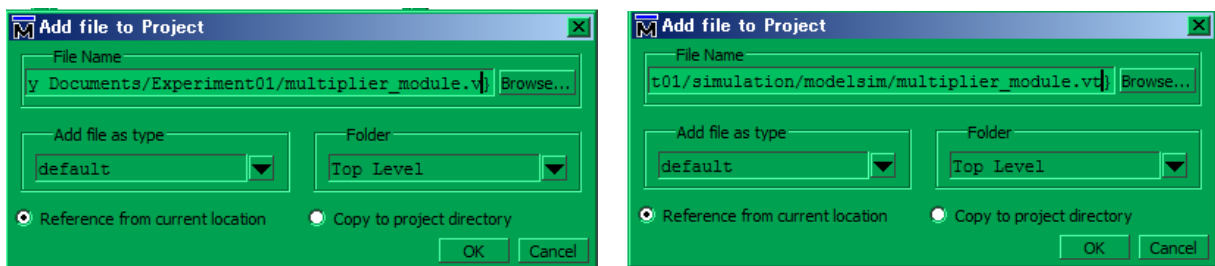


图 2.2.2 加载 .v 和 .vt 文件。

其二，其三如图 2.2.2 所示，既加载 .v 文件和 .vt 文件，自动编译与半自动编译会简化

该步骤。

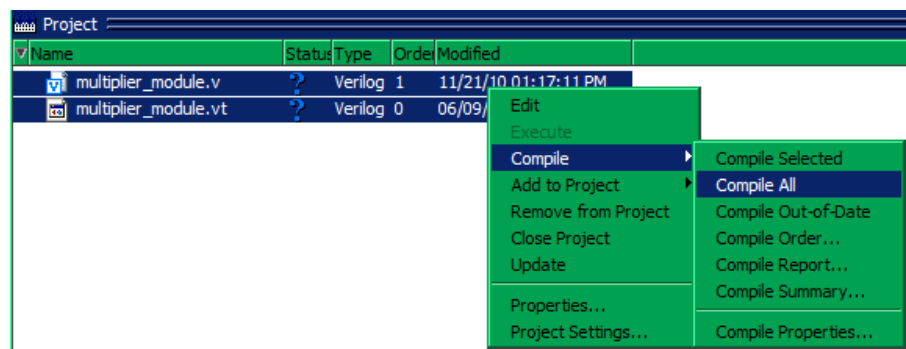


图 2.2.3 生成仿真对象和生成激励文本。

其四，其五皆如图 2.2.3 所示，既生成仿真对象还有仿真环境，俗称编译，自动编译会简化该步骤。

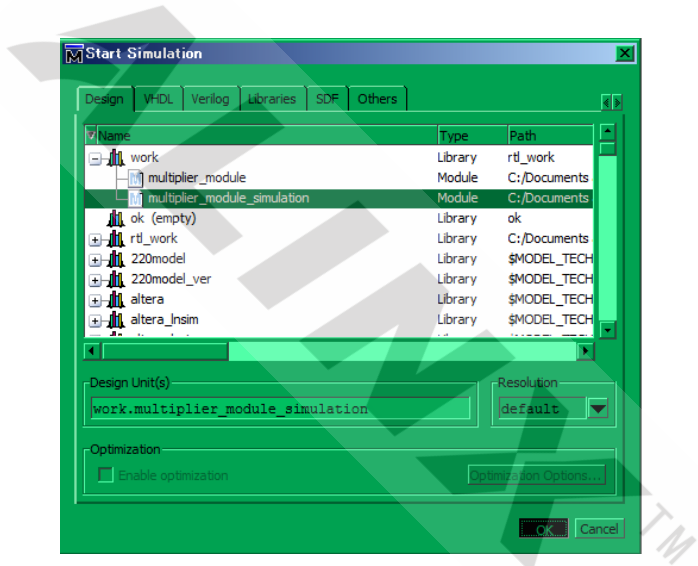


图 2.2.4 启动仿真。

其六，如图 2.2.4 所示，既是启动仿真，用笔者的话说就是激活仿真环境，自动编译会简化该步骤。

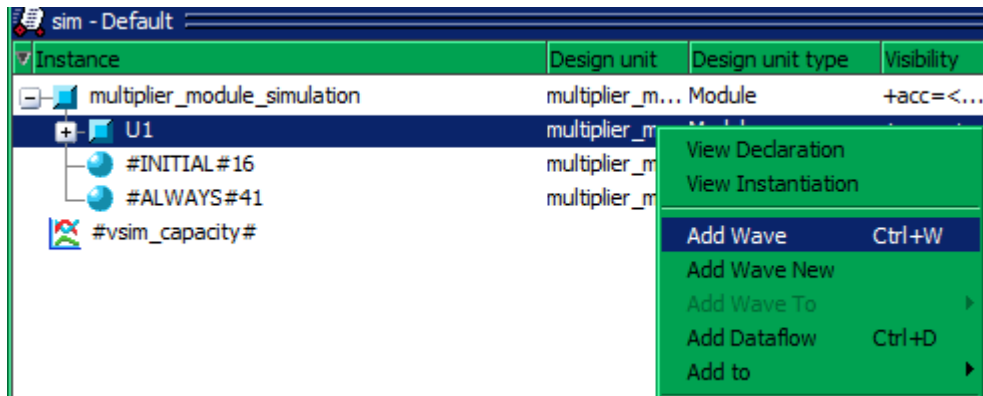


图 2.2.5 加载仿真信号。

其七，如图 2.2.5 所示，既添加仿真信号，自动编译会简化该步骤。

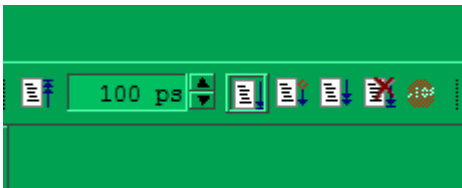


图 2.2.6 播放仿真。

其八，如图 2.2.6 所示，即播放仿真时间，自动编译会简化该步骤。

每起仿真执行之前,我们就必须重复上述 8 个令人厌烦的步骤,不过黑暗总是存在光明。根据笔者的认识,上述这些准备工作有分为 3 个类,亦即自动编译,半自动编译,还有手动编译。笔者是一位节能意识非常强的男人,所以特别喜欢自动编译,还有半自动编译,些许讨厌手动编译。三种编译方法之间的差异,如表 2.2.1 所示。

表 2.2.1 各种编译办法。

准备工作	自动编译	半自动编译	手动编译
创建仿真设计	自动	自动	手动
加载 .v 文件	自动	自动	手动
加载 .vt 文件	自动	自动	手动
生成仿真对象	自动	手动	手动
生成仿真环境	自动	手动	手动
启动仿真	自动	手动	手动
加载仿真信号	自动	手动	手动
播放仿真	自动	手动	手动

笔者之所以那么爱死自动编译,原因如表 2.2.1 所示,所有准备工作 Modelsim 都会自动完成,啊 ... 多么节能的手段。换之,半自动编译除了前 3 个步骤以外,亦即创建仿真设计,加载 .v 与 .vt 文件以外,余下的 5 个步骤都必须手动执行。半自动编译虽然不想自动编译那么省事,不过笔者也爱它。反之,笔者却有点讨厌手动编译,因为上述所有准备工作都必须人为执行,话说死不死呀。



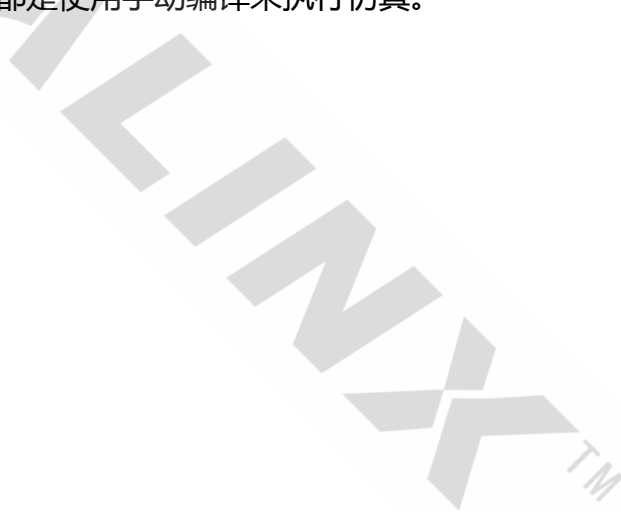
图 2.2.7 自动创建仿真项目。



一寸光阴一寸金，寸金难买寸光阴，伟大的祖辈教诲后代要珍惜时间，创建仿真设计还有加载 .v 与 .vt 文件这些个步骤往往是整个准备工作当中最浪费时间的，因为我们必须人为创建仿真项目，然而自动编译还有半自动编译会将仿真项目添加在设计项目的目录下，如图 2.2.7 所示，这是一个非常节能的手段。

虽然在便利的范畴上，自动编译无疑是最好的选择，不过自动编译也有局限的地方，亦即自由度不大，这句话是什么意思呢？假设第一次的仿真结果无法符合预期的预想，结果读者必须重新启动 Modelsim，不然那些更动以后的文件都无法生效。如果读者的计算机条件很好，哪倒没什么问题，重启 Modelsim 也是小片刻的等待而已。但是，那些计算机条件不好的同学，等待可谓是一场漫长的恶梦。

相反的，自动编译还有手动编译的自由度可大了，两者虽然用不着重启 Modelsim 便可生效更动过的文件。但是手动编译相较自动编译，手动编译较为费力。经过种种的排除以后，最后我们会发现半自动编译会是我们最好的选择。站在节能的角度上，笔者当然推荐自动编译还有半自动编译。不过作为初学，笔者还是建议读者先习惯手动编译为好，因为接续的例子，笔者都是使用手动编译来执行仿真。



## 2.3 自动编译的预先设置

如果 Modelsim 经由集成环境（Quartus II）启动，这个过程就称为自动编译，但是我们在**执行自动编译执行之前**，我们必须**预先设置编译信息**，好让笨蛋的集成环境知晓一切。首先将 Experiment01 的项目打开，那是笔者在以前设计过的传统乘法器，不过那是一只实验性的小白鼠而已。

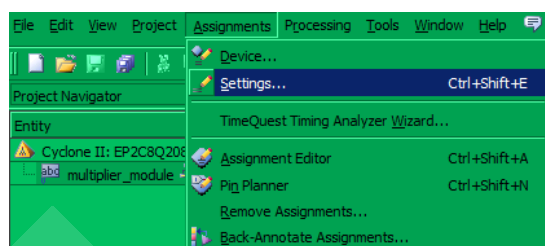


图 2.3.1 菜单 Assignment，还有 Settings 选项。

接着，调出 Assignment 菜单，然后进入 Settings 界面。如图 2.3.1 所示。

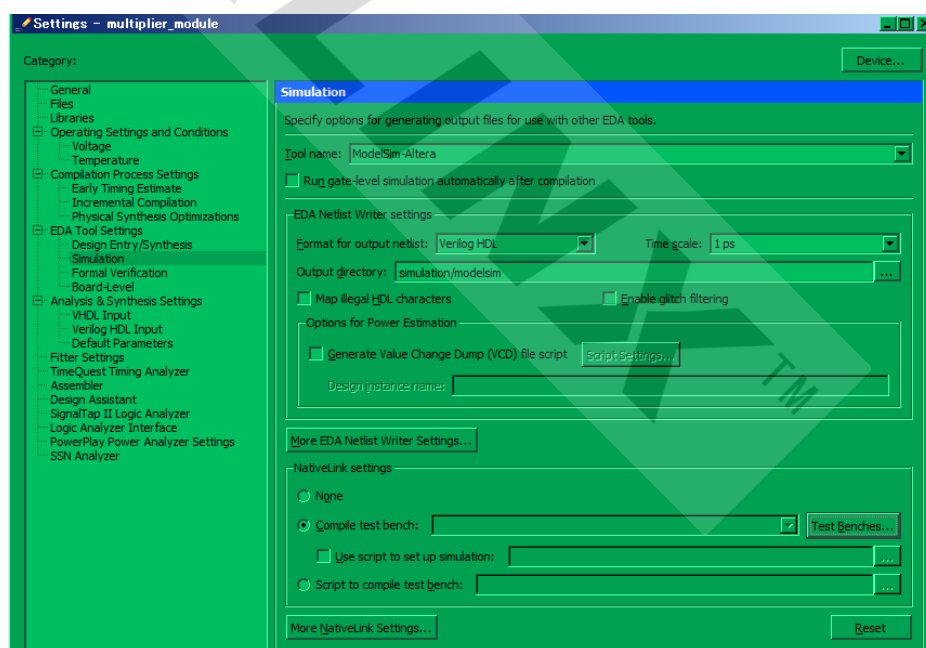


图 2.3.2 界面 Settings。

打开 Settings 界面后，我们会看见各种意义不明的选项，暂时无视它们，然后读者只要展开 EDA Tool Settings，再点击 Simulation，Simulation 界面随之就会浮现在右边的窗口。沿着下方我们会看见 NativeLink Setting，顺手点击 Compile test bench 左边的圈圈，结果如图 2.3.2 所示。最后点击 Test benches 按键即可。

（确保 Tool name 设置为 Modelsim-Altera）

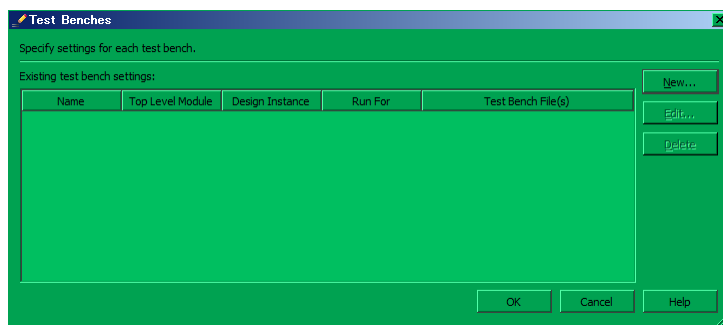


图 2.3.3 窗口 Test Benches。

事后，Test Benches 窗口会浮现在眼前，如图 2.3.3 所示。此刻窗口空空如也，因为我们还没输入相关的编译信息。沿着右边，点击 New 按键。

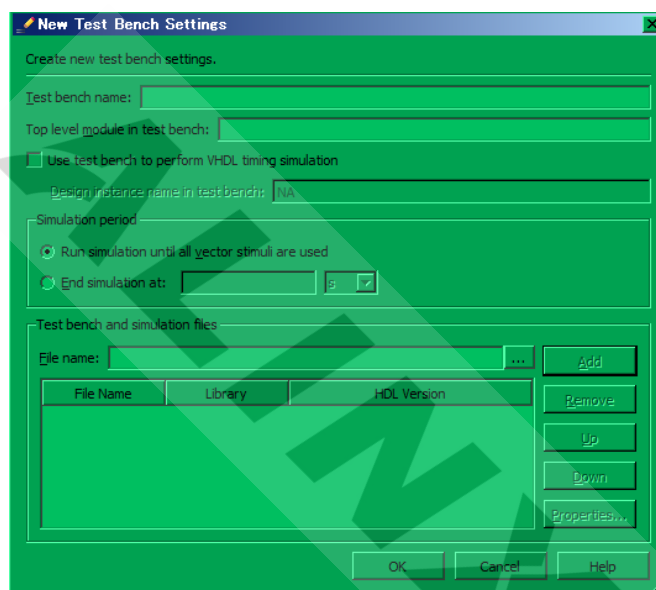


图 2.3.4 窗口 New Test Bench Settings。

如图 2.3.4 所示，New Test Bench Settings 窗口会浮现在眼前，该窗口有 3 个部分，其一是激励文件名（上段），最大仿真时间（中段），还有激励文件的具体路劲（下段）。实验一的模块名为 multiplier\_module.v，然而激励文本则名为 multiplier\_module.vt。在此，Test bench name 是指激励文本名，亦即 multiplier\_module（省略后缀 .vt）。

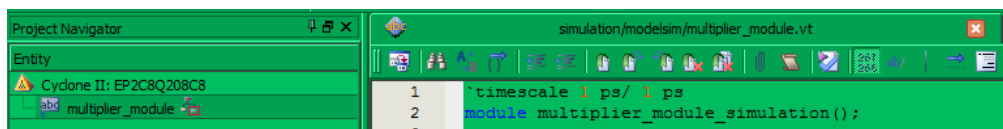


图 2.3.5 激励文件 multiplier\_module.vt 的仿真环境取名。

Top level module in test bench 中文译名为顶层激励模块，任笔者怎么翻译都觉得怪怪的，所以笔者索性称呼它为仿真环境。如图 2.3.5 所示，名为 multiplier\_module.vt 的激励文本，内容所属的仿真环境取名为 multiplier\_module\_simulation。

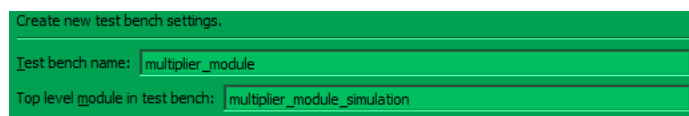


图 2.3.6 激励文件名设置完成。

完后，激励文件信息的基本设置如图 2.3.6 所示，也就是 Test bench name 输入为 multiplier\_module，Top level module in test bench 输入为 multiplier\_module\_simulation。

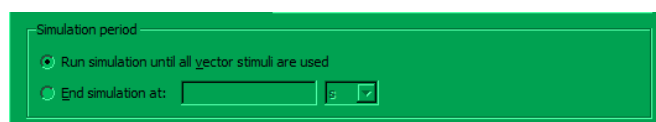


图 2.3.7 设置仿真时间。

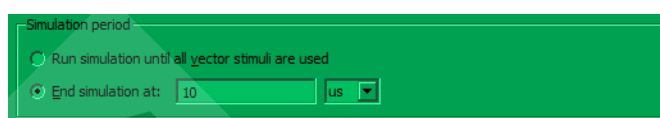


图 2.3.8 最大仿真时间为 10us。

New Test Bench Settings 窗口中段是仿真时间的设置，如图 2.3.7 所示。不管默认选项是什么东西，读者只要使能 End simulation at 左边的圈圈，然后在右边的文本框中输入 10us 即可，既是最大仿真时间为 10us。完后，如图 2.3.8 所示。

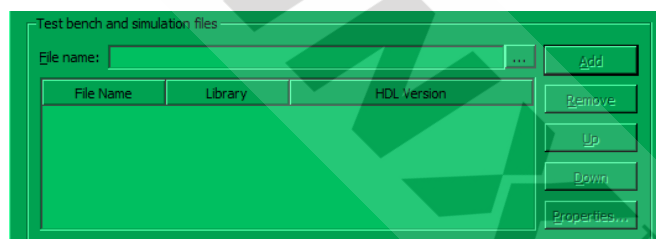


图 2.3.9 设置激励文件路径。

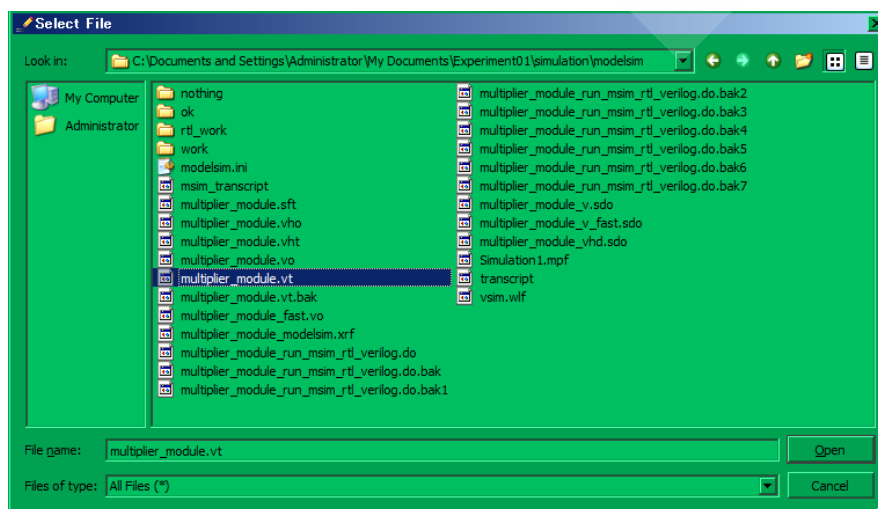


图 2.3.10 资源管理窗口——选择文件。

如图 2.3.9 所示 ,New Test Bench Settings 窗口的下段是设置路径。沿着 File name 右边 , 点击 <...> , 然后资源管理窗口就会浮现在眼前, 在此寻找 multiplier\_module.vt 文件, 随后点击 Open 按钮更新选择, 过程如图 2.3.10 所示。

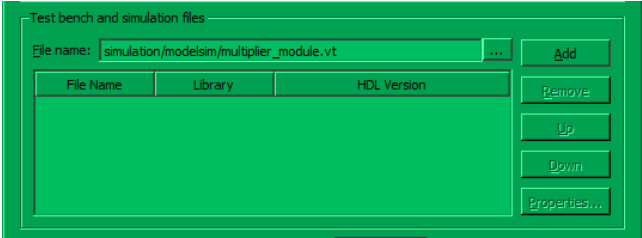


图 2.3.11 激励文件路径显示。

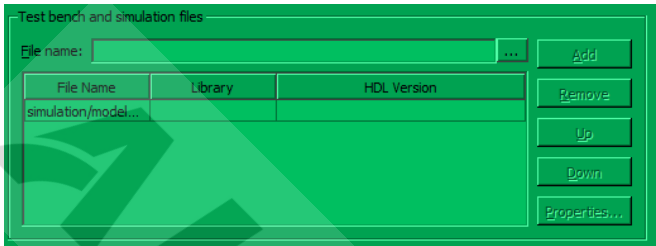


图 2.3.12 激励文件路径添加。

过后 , File name 会显示之前选择的路径, 如图 2.3.11 所示。接着 , 沿着右边点击 Add 按钮添加路径, 如图 2.3.12 所示。

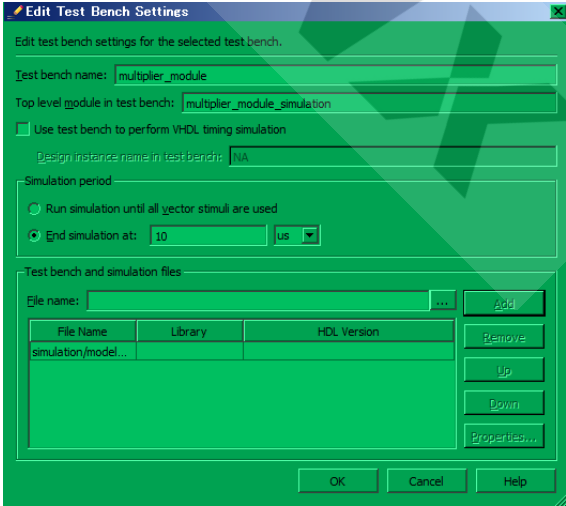


图 2.3.13,整体效果。

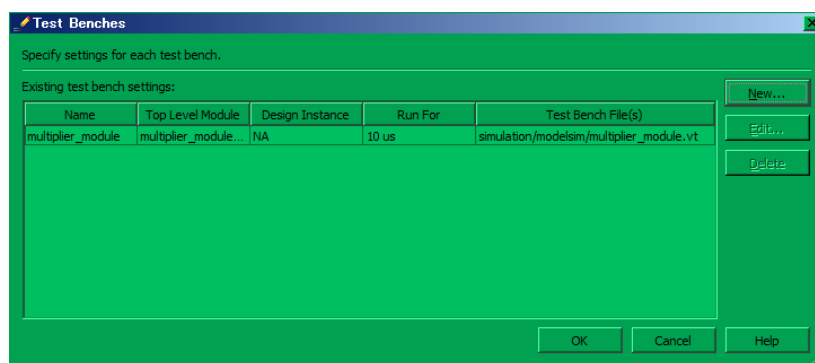


图 2.3.14 激励信息设置完成。

设置完后,整体效果如图 2.3.13 所示,接着点击 OK 按键生效设置,然后返回 Test Bench 窗口。此刻,我们已经完成激励信息的输入,作为证明,方才的设置内容都会显示在 Test Bench 窗口之中,如图 2.3.14 所示。最后点击 OK 按键更新设置。

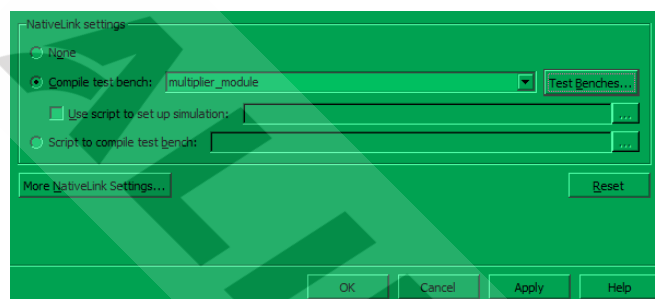


图 2.3.15 编译信息输入完毕。

接着我们会返回 Settings 界面,沿着 Compile test bench 的右边 multiplier\_module 的激励信息会显示在文本框中,以此证明编译信息输入完毕,结果如图 2.3.15 所示。最后点击 OK 更新设置。

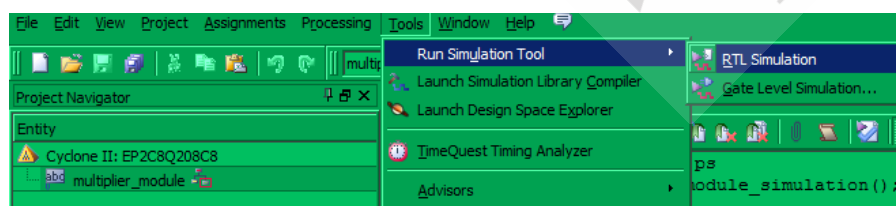


图 2.3.16 Modelsim 启动,自动编译执行。

确认一切设置办妥以后,打开 Tools 菜单,沿着 Run Simulation Tool 选项的右方,点击 RTL Simulation,过后 Modelsim 会自行启动,而且自动编译也会开始执行。过程如果 2.3.16 所示。





图 2.3.17 仿真项目自动创建。

对于自动编译与半自动编译而言，设计库（工作库）的默认名为 work，结果 work 文件夹会自动添加在设计项目的目录下。期间，自动编译需要一定的执行时间，然而快与慢完全根据个人的计算机配置。

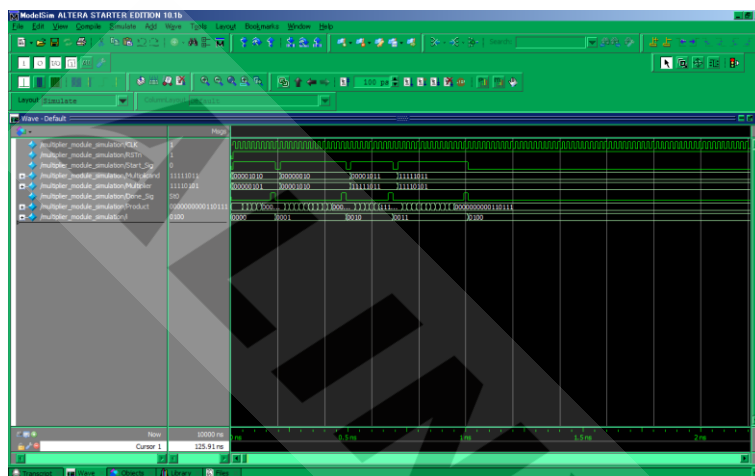


图 2.3.18 Modelsim 启动成功，自动编译执行成功。

如果一切错误皆没有发生的话，Modelsim 的界面会成功显示在电脑屏幕上，结果如图 2.3.18 所示。

## 2.4 常用界面

Modelsim 是一款功能强大的电视机，除了播放属于 Verilog 的节目（波形图）以外，Modelsim 还可以播放其它节目，为此 Modelsim 旗下如果拥有多个界面，我们一点也不用觉得奇怪。那些不擅长软件的朋友，可说是一场恶梦，然而很庆幸的是，我们用不着学习全部。根据笔者的理解，仿真仅需要以下几个界面而已：

1. Project 界面
2. File 界面
3. Library 界面
4. Transcript 界面
5. Simulation 界面
6. Wave 界面

接着，让笔者逐个为大伙介绍。

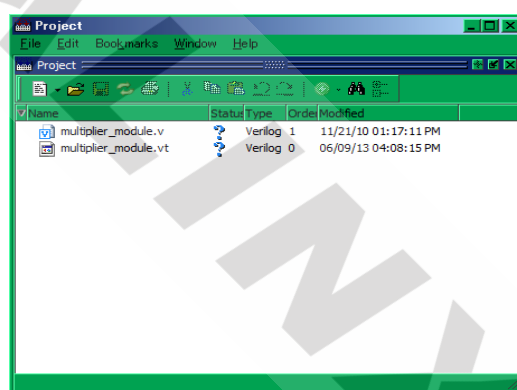


图 2.4.1 Project 界面

如图 2.4.1 所示，Project 界面顾名思义就是仿真项目界面，然而 Project 界面的使用程度并不频繁，因为自动编译还有半自动编译会省略创建仿真设计的步骤，余下也只有手动编译才会使用 Project 界面。总之，读者先看个大概，暂时把 Project 界面的功能想象成为[收纳东西的文件夹](#)一样。往后，笔者自然会在手动编译的小节介绍它。

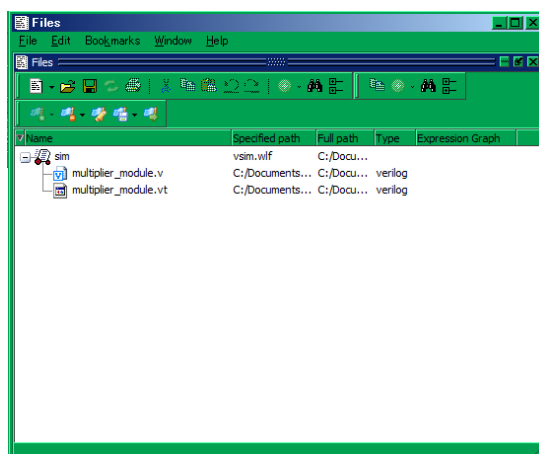


图 2.4.2 File 界面

如图 2.4.2 所示，File 界面有时候也称为文件界面，任何与仿真有关的文件都会显示在这个界面上。以实验一为例，multiplier\_module.v 文件还有 multiplier\_module.vt 文件都展示在资源树之下。File 界面除了展示仿真相关的所有文件以外，什么作用也没有。顺便一提，不管自动编译还是手动编译，File 界面都会使用到。

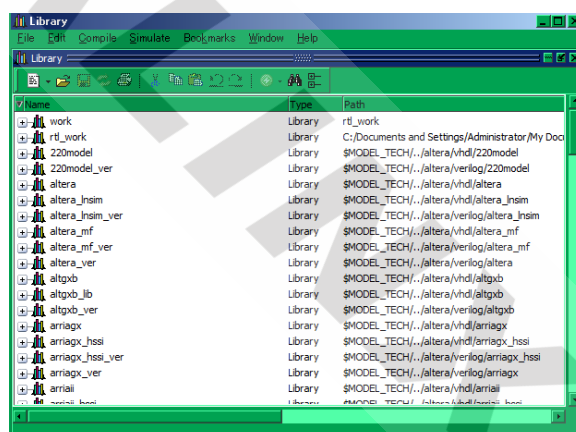


图 2.4.3 Library 界面

如图 2.4.3 所示，Library 界面有时候也称为仿真库界面，然而 Library 界面的作用却不怎么单纯。仿真库界面一般有两个作用，其一就是储存自定义 IP 所需的仿真信息，例如自定义关键字信息，自定义功能，延迟等信息。所谓的**仿真库质量就是仿真信息的支持程度**。此外，Library 界面一般都是有**两大分类**，亦即设计库（工作库）与资源库。不管怎么样，详细内容往后详谈。

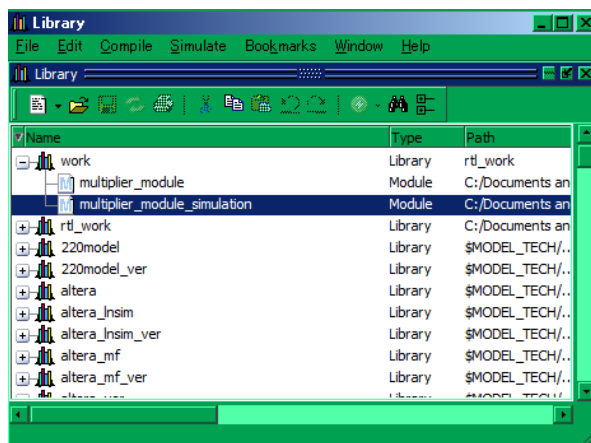


图 2.4.4 自动编译创建名为 work 的仿真设计。

设计库别名也有工作库。设计库好比仿真所需的暂存空间，所有相关的仿真信息都会储存在这里。但是自动编译会擅自将它取名为 work，以实验一为例，实验一的仿真对象是 multiplier\_module，仿真环境则是 multiplier\_module\_simulation，结果两者信息就这样储存在 work 里边（如图 2.4.4 所示）。

仿真库界面看似麻烦不过却非常有用，我们可以直接在这里更新（编译）仿真对象又或者更新（编译）仿真环境，还是启动仿真也可以。

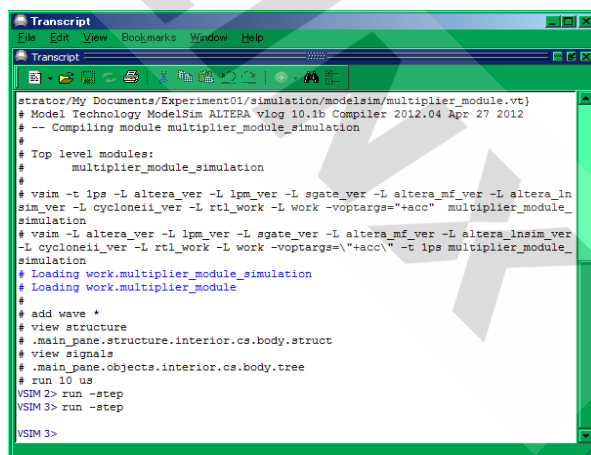


图 2.4.5 Transcript 界面。

如图 2.4.5 所示，Transcript 界面也是终端（Console）界面，故名思议 Transcript 界面的作用就是显示打印信息还有接受 TCL 命令（工具命令语言）。Modelsim 虽然可以支持相关的 TCL 命令实现相关的操作，然而笔者却不建议学习 TCL 命令。笔者认为，只要了解界面，善用界面，即使不学习 TCL 命令也可以实现一样的操作。实际上这写都是笔者犯懒的借口而已，因为 TCL 命令学起来很麻烦。

结果而言，笔者比较喜欢将 Transcript 界面称为信息界面，一些有用的错误反馈信息或者打印信息都会显示在其中。不过，笔者偶尔也会被海量的信息打败，所以要好好控制信息的打印数量，不然结果就会自讨苦吃。

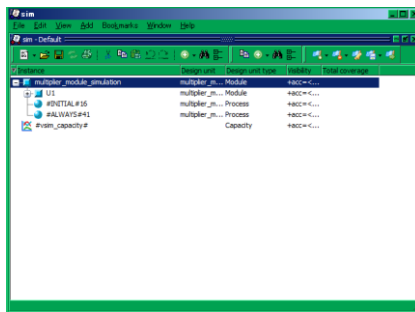


图 2.4.6 Simulation 界面。

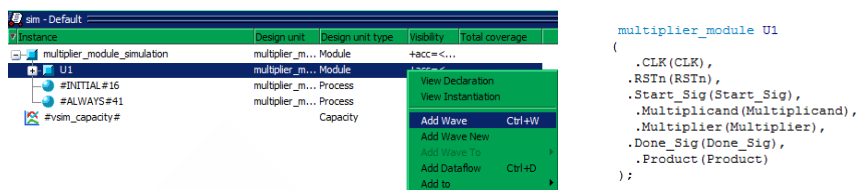


图 2.4.7 添加仿真信号。

如图 2.4.6 所示，sim 界面是 simulation 界面的缩写，然而笔者却称为仿真信号界面。一些可以仿真的信号都会放置在这里。此外，我们也可以手动添其它仿真信号，以实验一为例：假设，笔者想要观察仿真对象 U1，U1 是仿真对象 multiplier\_module 的实例名，因此，笔者需要先展开仿真环境 multiplier\_module\_simulation，接着右键点击 U1，然后选择 Add Wave，此刻仿真对象 U1 的内部一切将成为仿真信号，过程如图 2.4.7 所示。sim 界面虽然有用但是自动编译却会忽略它。换之，半自动编译还有手动编译比较受到爱戴。

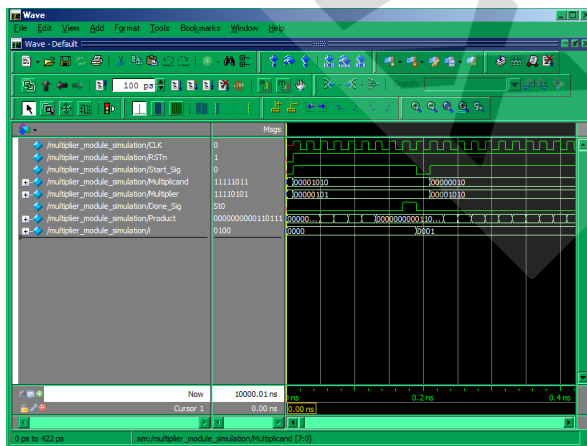


图 2.4.8 Wave 界面。

如图 2.4.8 所示，Wave 界面也称为波形图界面，也是最常用，最重要的界面，而且“学习 Modelsim 就是学习 Wave 界面”——这种说法一点也不过分。Wave 界面的作用就是显示仿真信号，Wave 界面虽然看似复杂但是操作却非常傻瓜，学起来也非常轻松。我们知道仿真对象不会使用人类的语言述自述，换之仿真对象会经由波形图显示活动记录，学习 Wave 界面真正的困难之处就是如何解读这些波形图。

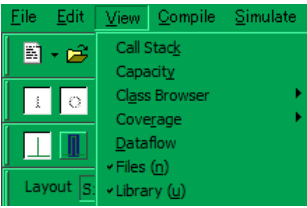


图 2.4.9 显示相关界面。

最后，让笔者做个简单的总结，上述 6 个界面是仿真最常用的界面，我们可以透过 View 菜单打开或者关闭它们，过程如图 2.4.9 所示。然而，根据编译方法的不同，界面的使用程度也会跟着不同，因此笔者绘出表 2.4.1：

表 2.4.1 仿真常用界面。

仿真常用界面	自动编译	半自动编译	手动编译
Project 界面	无用	无用	有用
File 界面	有用	有用	有用
Library 界面	有用	有用	有用
Transcript 界面	有用	有用	有用
Sim 界面	无用	有用	有用
Wave 界面	有用	有用	有用

虽然 Modelsim 还存在许多界面，但是对笔者而言，只要上述掌握上述 6 个界面就足够应付仿真了。学习仿真的过程是非常耗神耗气的，因此节能学习才是学习仿真的上上之道。



## 2.5 自动编译与半自动编译

在这个小节里，我们会透过实验一去理解各种编译方法的不同之处。笔者曾在小节 2.3 讲解过自动编译的预先设置，事实上自动编译是上天给予懒人的礼物，我们可以透过预设将麻烦的准备工作交由集成环境劳动。

自动编译一般如流程如下：

预先设置 → 启动功能仿真 → 自动创建仿真项目 → 自动加载 .v 与 .vt 文件 → 自动编译 .v 与 .vt 文件 → 自动启动仿真 → 自动添加仿真信号 → 自动播放仿真

除了预先设置可视之外，直至 Wave 界面显示波形图，其余步骤都是集成环境在暗地里执行。自动编译最大的好处就是节能，不过自动编译也有坏处，每当我们更新 .v 与 .vt 文件以后，如果我们想要更新波形图，我们必须重新启动 Modelsim 才行。重启 Modelsim 至少需要几个呼吸的时间，笔者则认为那是短暂放松，然而那些性格比较焦急的朋友，可能是痛苦的等待。由此，半自动编译就诞生了。

半自动编译一般如流程如下：

预先设置 → 启动功能仿真 → 自动创建仿真项目 → 自动加载 .v 与 .vt 文件 → 自动编译 .v 与 .vt 文件 → 自动启动仿真 → 自动添加仿真信号 → 自动播放仿真

更动 .v 与 .vt 文件以后，预想更新波形图，流程如下：

手动编译 .v 与 .vt 文件 → 手动启动仿真 → 手动添加输出 → 手动播放仿真

半自动编译是经由自动编译执行一番以后，再手动执行几个步骤。具体过程如下：

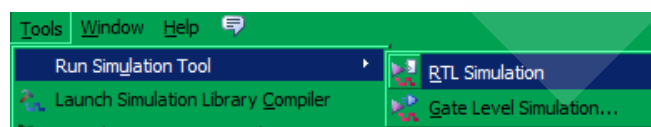


图 2.5.1 执行功能仿真（启动自动编译）。

首先是经过 Quartus 的 Tools 菜单，沿着 Run Simulation Tool 选项的右边，点击 RTL Simulation，然后自动编译就会执行，如图 2.5.1 所示。这样作的目的是经由自动编译，让集成环境做好仿真前的工作，好使我们节省气力。

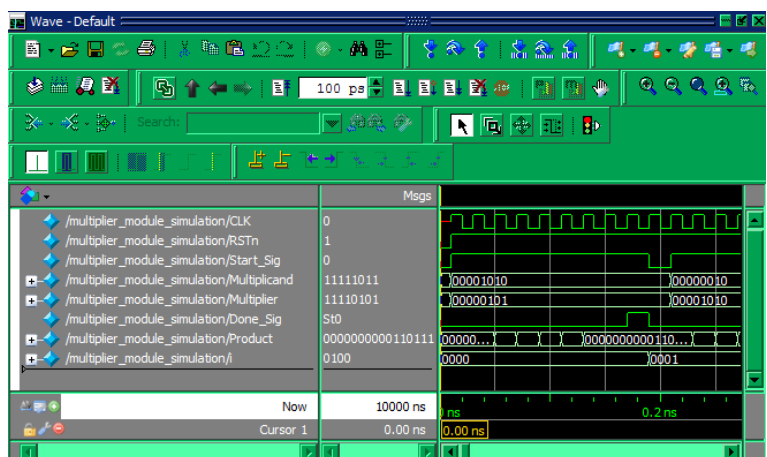


图 2.5.2 波形图出现在 Wave 界面。

经过一段小时间以后，Wave 界面会呈现波形图，亦即自动编译执行成功，如图 2.5.2 所示。当然前提条件必须确保 .v 文件还有 .vt 文件没有语法错误。此刻，一切准备工作已经经由自动编译执行完成。

```

52      0: // Multiplicand = 10 , Multiplier = 2
53      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
54      else begin Multiplicand <= 8'd10; Multiplier <= 8'd8; Start_Sig <= 1'b1; end
--

```

图 2.5.3 更动 .vt 文件内容。

假设笔者手痒，更新 .vt 文件其中一段内容，如图 2.5.3 所示。但是笔者又不想重启 Modelsim，于是半自动编译发生了。同样，必须先确保更改内容的语法是正确无误。

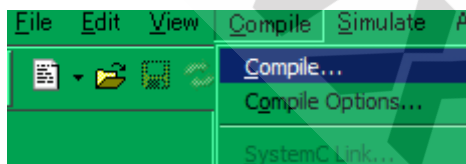


图 2.5.4 打开 Compile 窗口。

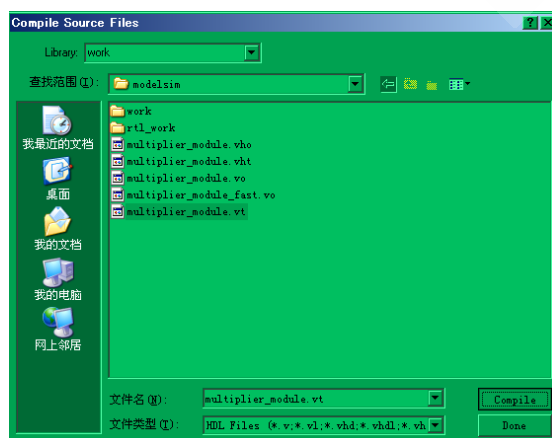


图 2.5.5 选择更新文件，手动编译。

鼠标指向 Compile 菜单，然后单击 Compile 选项，如图 2.5.4 所示。接着，资源管理窗口会浮现在眼前，选择要更新的文件，在此是 multiplier\_module.vt，确定以后沿着右下方依序单击 Compile 随之单击 Done 即可。此刻，最新的 multiplier\_module.vt 内容就会生效。（步骤手动编译 .v 文件与 .vt 文件完成）

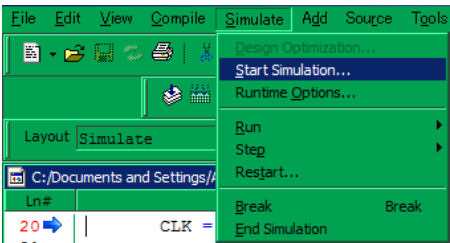


图 2.5.6 打开 Start Simulation 窗口。

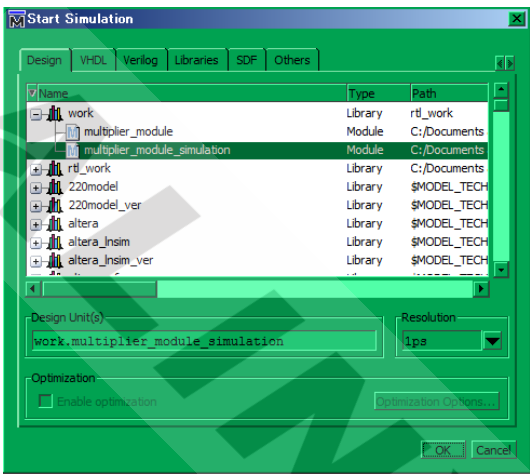


图 2.5.7 Start Simulation 窗口，启动仿真。

事后，沿着 Simulate 菜单，单击 Start Simulation 选项，如图 2.5.6 所示。如图 2.5.7 所示，Start Simulation 窗口接着会浮现在眼前，展开 Design 子窗口下的设计库 work，选择仿真环境 multiplier\_module\_simulation，然后鼠标致右下方单击 OK，仿真就开始启动。（步骤手动启动仿真完成）

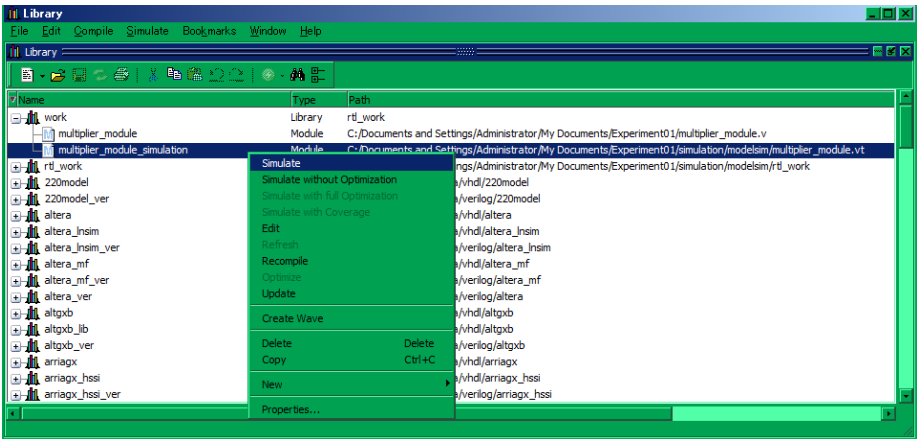


图 2.5.8 Library 界面，手动编译文件还有手动启动仿真。

事实上，上述两个步骤（手动编译文件还有手动启动仿真）都可以经由 Library 界面完成，如图 2.5.8 所示，展开仿真库 work 以后，会有仿真对象 multiplier\_module 还有仿真环境 multiplier\_module\_simulation。右键点击仿真环境，其中小窗口就有 Recompile（再编译）和 Simulate（启动仿真）等选项，只要依序点击 Recompile 还有 Simulate，同样的操作也会发生。

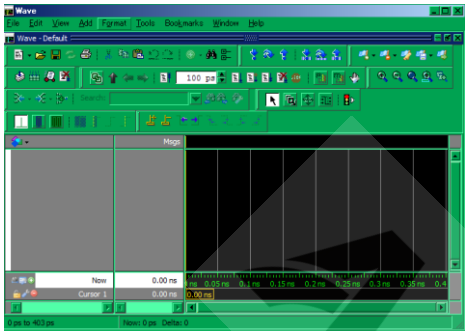


图 2.5.9 Wave 界面什么也没有。

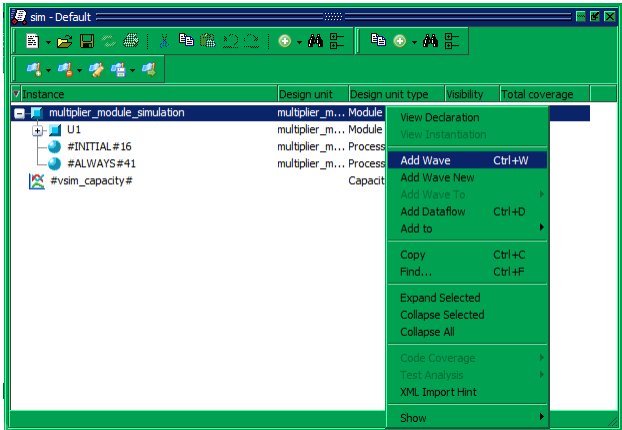


图 2.5.10 Sim 界面，手动添加仿真信号。

一段等待以后，仿真就启动完毕。不过，此刻 wave 界面什么也没有，结果如图 2.5.9 所示。因此，我们必须手动添加仿真信号，首先打开 Sim 界面，仿真环境 multiplier\_module\_simulation 会是最高的父选择。如果，读者仅需要显示仿真对象的出入端而已，那么右键点击仿真环境，然后选择 Add Wave，过程如图 2.5.10 所示。

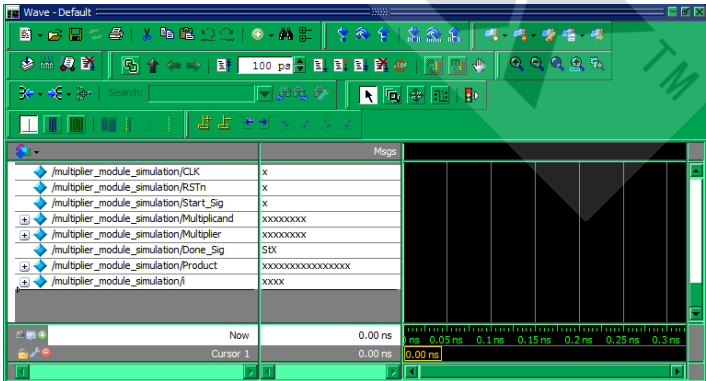


图 2.5.11 手动添加仿真信号。

如图 2.5.11 所示，仿真信号手动添加成功后，那么仿真信号就会显示在 Wave 界面的左框中。但是，此刻 Wave 界面（右框）还没有显示任何波形图，为什么呢？原因很单纯，因为播放键还没有按下。（步骤手动添加仿真信号完成）



图 2.5.12 手动播放仿真。

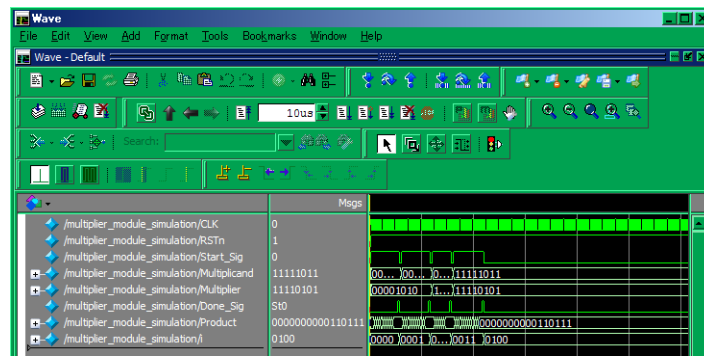


图 2.5.13 手动播放仿真完毕。

读者是否还记得，我们曾经在预设当中将仿真最大时间设置为 10us，然后自动编译会自动播放仿真直至 10us。同样，我们也可以手动播放仿真时间，如图 2.5.12 所示，输入预想播放时间在播放文本框当中。在此，笔者输入 10us，然后沿着右方点击“↓”按键，稍等一会后，波形图就会播放完成，结果如图 2.5.13 所示。我们可以一次播放最大仿真时间之余，我们也可以多次播放片刻仿真时间。（步骤手动播放仿真完成）

到目前为止，半自动编译的所有步骤我们已经执行完成，真是可喜可贺。在此，稍微让我们来思考一下：

自动编译虽然方便，不过它也有不足之处，即自由度很小。其一必须重启 Modelsim 不然更新文件无法生效；其二不能随意添加仿真信号，因为自动编译仅针对仿真对象的出入端而已。为此，半自动编译就是为了弥补自动编译这两点不足之处才会降临到这个世界上。讨论完结自动编译还有半自动编译以后，接下来让我们来讨论手动编译。

手动编译一般流程：

手动创建仿真项目 → 手动加载 .v 与 .vt 文件 → 手动编译 .v 与 .vt 文件 → 手动启动仿真 → 手动添加仿真信号 → 手动播放仿真

正如上述流程所示，手动编译相较自动编译只是少了步骤“预先设置”之余，还有自动改为手动而已。况且在效果上，半自动编译有点类似手动编译，因此不得不让人思考，我们为何还要手动编译呢？

假设笔者是一粒穷光蛋，没有钱购买 Quartus II，因此笔者无法实现自动编译还有半自动编译。此刻 Modelsim 就会成为另一个集成环境，为此让我们暂时抛开 Quartus II，尝试使用 Modelsim 执行手动编译好让自己有个深刻的理解。

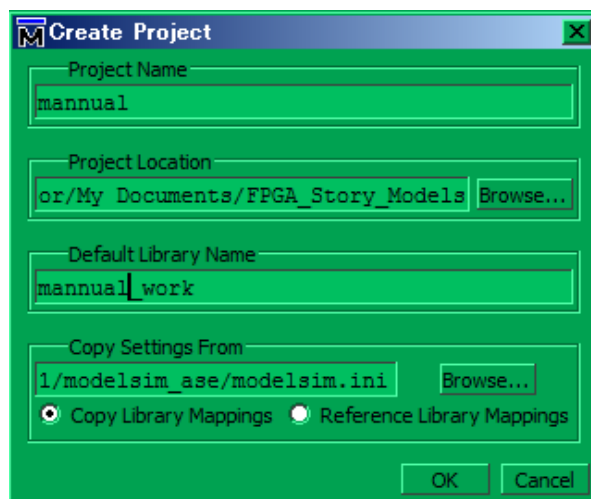
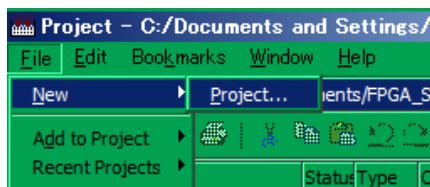


图 2.5.14 手动创建仿真项目

首先打开 Modelsim，然后切换到 Project 界面，随之选择 File 菜单，沿着 New 选项右边再点击 Project，过程如图 2.5.14 所示。过不了一会，Create Project 窗口就会浮现在眼前：

1. Project Name 的作用不大随意输入就好，笔者取名为 manual；
2. Project Location 是仿真设计的路径；
3. Default Library Name 是设计库的取名，默认为 work 笔者取名为 manual\_work；
4. 点击 OK 生效。（步骤手动创建项目完成）

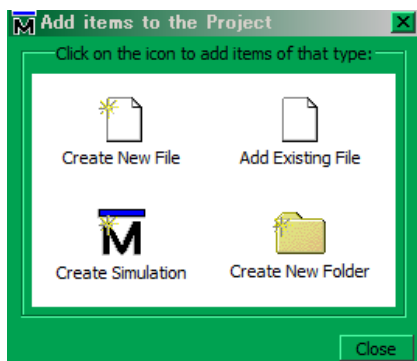


图 2.5.15 Add item to the Project 窗口

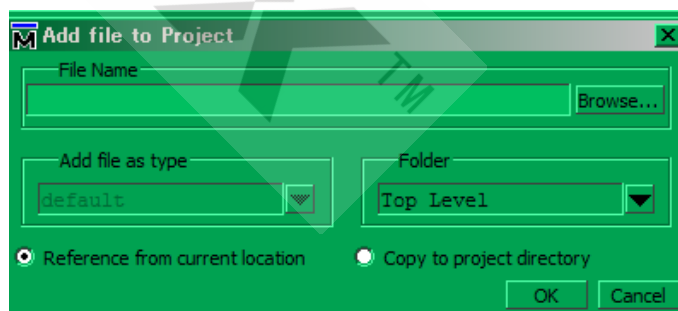


图 2.5.16 Add file to the Project 窗口



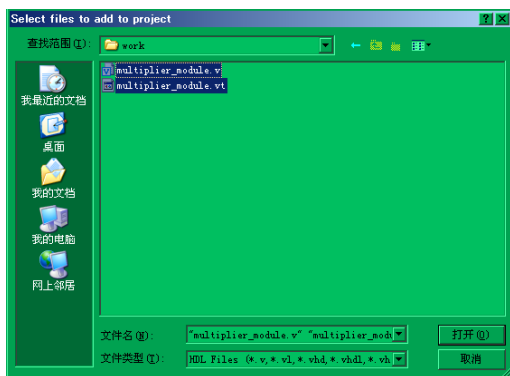


图 2.5.17 添加以后文件。

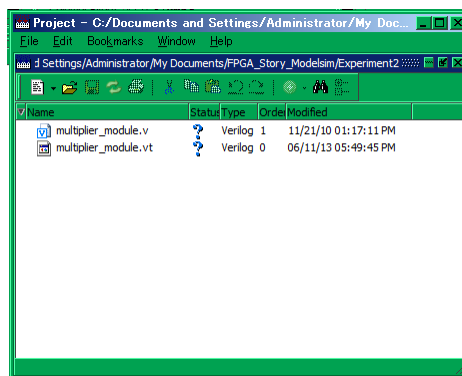


图 2.5.18 文件添加完成。

手动创建项目完成以后，Add item to the Project 窗口会浮现在眼前。如图 2.5.15 所示，其中有 4 个选项，而且意义也很直接，在此笔者就不解释了。点击 Add Existing File，Add file to Project 窗口就会浮现在眼前，结果如图 2.5.16 所示，沿着右方点击 Browse 按键，将 Multiplier\_module.v 还有 multiplier\_module.vt 这两个已有文件添加进来，过程如图 2.5.17 所示。时候，Project 界面就会出现方才添加进来的两个文件，结果如图 2.5.18 所示。（步骤手动添加文件完成）

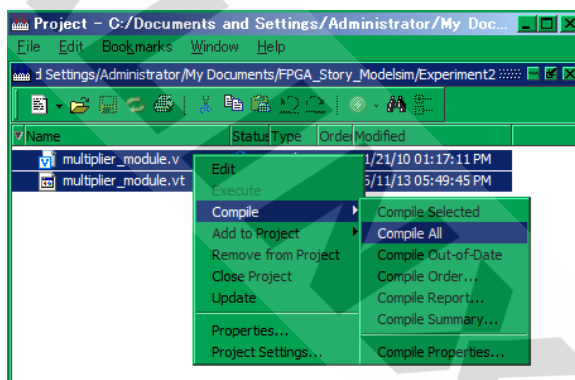


图 2.5.19 手动编译。

如图 2.5.19 所示，右键点击任意文件，然后沿着 Compile 选项选择 Compile All，编译所有文件（步骤手动编译文件完成）。

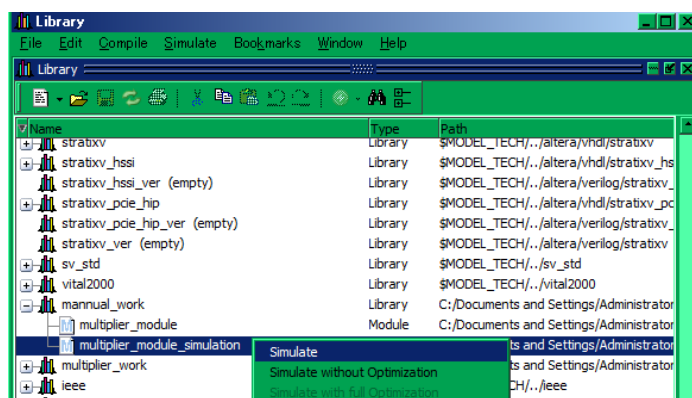


图 2.5.20 手动启动仿真。

事后，切换到 Library 界面，向下拉动滚条直至找到 manual\_work 这个自定义的设计库，暂开之后右键点击仿真环境 multiplier\_module\_simualtion，接着选择 Simulate，然后仿真就会启动。（步骤手动启动仿真完成）

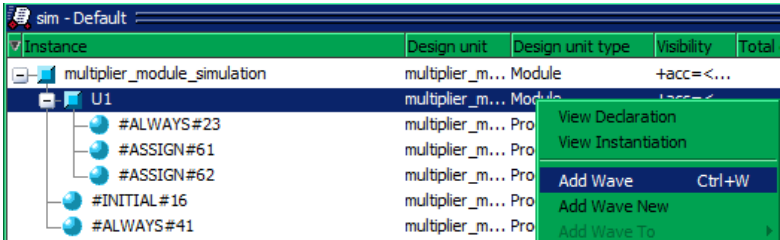


图 2.5.21 手动添加仿真信号。

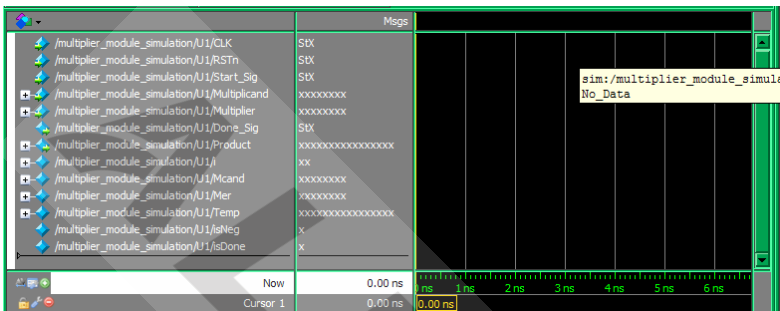


图 2.5.22 仿真信号手动添加完毕。

稍等一会后，仿真就会启动完毕，借此 Sim 界面也会跟着浮出水面。假设笔者想观察仿真对象 multiplier\_module 的全体输出，笔者可以展开仿真环境，右键点击 U1，接着选择 AddWave，过程如图 2.5.21 所示。切换至 wave 界面，仔细观察一下 wave 界面的左框，已经出现密密麻麻的仿真信号，结果如图 2.5.22 所示。在此我们可以断定，手动添加仿真信号已经完成。（步骤手动添加仿真信号完成）

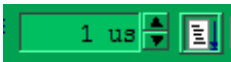


图 2.5.23 手动播放仿真。

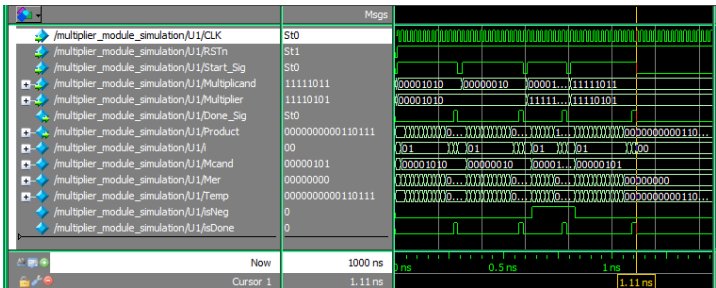


图 2.5.24 手动播放仿真完毕。

10us 的仿真时间对于实验一来说过于充足了，在此笔者输入 1us 的仿真时间，结果如图 2.5.23 所示，然后沿着右方点击 “↓” 按钮，开始播放仿真。经过几个呼吸的时间以后，

wave 界面的右框就会出现波形图，结果如图 2.5.24 所以。此刻我们可以断定，手动播放仿真已经完毕。（步骤手动播放仿真完成）

手动编译一般是辛苦的劳动，啊！不是 ... 是缺少集成环境（如 Quartus II）的情况下，才会选择的下下策。手动编译与半自动编译虽然相似，但是笔者却一直喜欢不上它，是否笔者太懒，还是手动编译太麻烦了？不管怎么样，初学者一般建议使用自动编译，然后再选择性使用半自动编译。至于手动编译，除非是学习作用，不然就无视它。



## 2.6 操作 Wave 界面

Wave 界面是 Modelsim 所有界面之中最有用的界面，wave 界面好比电视屏幕，用来显示节目，亦即波形图。操作 wave 界面近似操作老式 CRT 电视，又似操作现代液晶电视，这句话听起来虽然觉得有点矛盾。举例而言，如果读者只是“看爽”波形图而已，亦即看看形状然后傻笑自认看懂，结果就是老式 CRT 电视，操作程度非常傻瓜。反之，如果读者要仔细观察每一个时钟具体的时序情况，那么就是现代液晶电视，操作程度非常细腻。

操作 wave 界面主要分为两种，针对老式 CRT 电视称为傻瓜操作；针对现代液晶电视称为细腻操作。傻瓜操作没什么好谈的，换之细腻操作是仿真技巧不可缺少的手段之一。不过，读者别担心，细腻操作界面也没有印象中那么困难，前提条件除了耐心以外，还是耐心，因为细腻操作 wave 界面非常耗费精力还有时间，稍微一个不留心也会自乱阵脚。

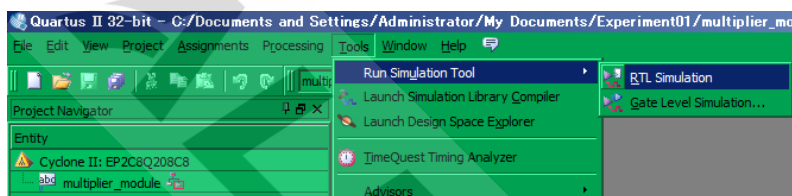


图 2.6.1 自动编译实验一。

好了，开场白就讲到这里。首先经由自动编译启动实验一的仿真，然后再切换到 wave 界面作为本节的开端，过程如图 2.6.1 所示。

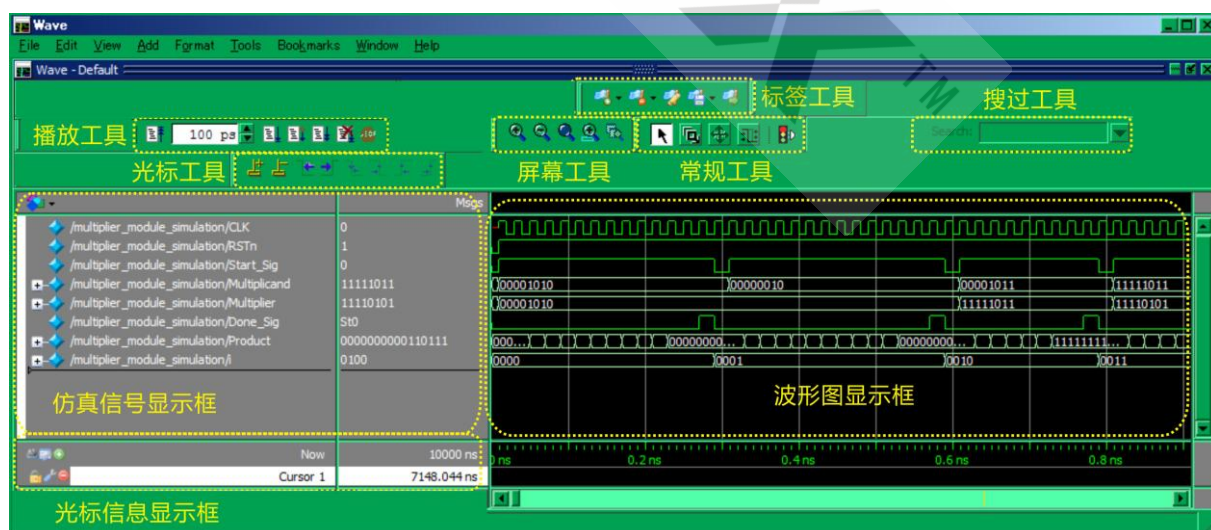


图 2.6.2 Wave 界面简介。

为了减轻读者的学习压力，图 2.6.2 是经过笔者打码以后的 wave 界面。Wave 界面一般有 3 大显示框：

1. 波形图显式框
2. 仿真信号显示框
3. 光标信息显示框

其一，波形图显示框顾名思义就是用来显示波形图的地方，亦即 Modelsim 的图形输出；其二，仿真信号显示框主要是用来显示仿真信号，还有仿真信号当前时钟的结果；其三，光标信息显示框是用来指示光标信息，如光标名还有光标位置。显示框不同，工具也不同，常用的工具如图 2.6.2 所示，表 2.6.1 则是工具的简单归类。

表 2.6.1 各种工具归类。

其他	波形图显式框	光标信息显示框	仿真信号显示框
播放工具	常规工具 屏幕工具 标签工具	光标工具	搜索工具

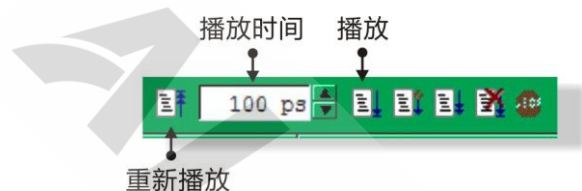


图 2.6.3 播放工具。

如图 2.6.3 所示，那是读者再熟悉不过的播放工具，不过笔者还是循例来讲一下。有作为的功能如图 2.6.3 所示，亦即重新播放，播放时间还有播放。

7. 重新播放如字面上的意思就从 0 时间开始起播放仿真，而不是重启整个仿真；
8. 播放时间也是播放进度；
9. 播放亦即开始播放；

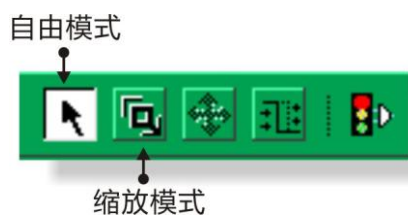


图 2.6.4 常规工具。

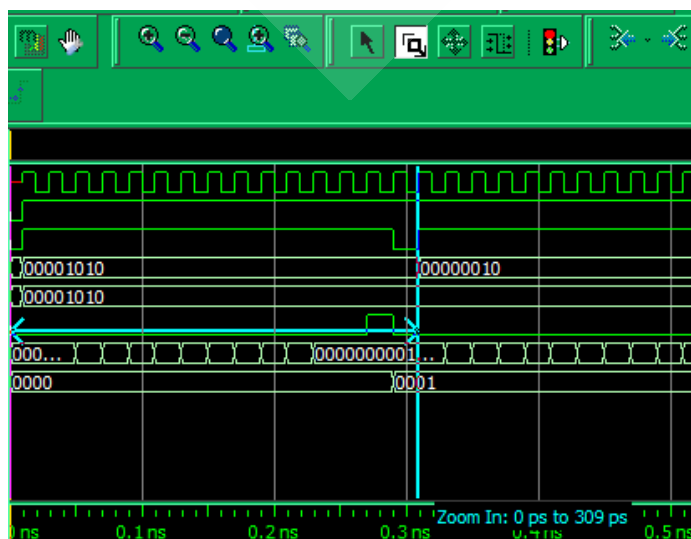


图 2.6.5 缩放模式。

图 2.6.4 是常规工具也称为模式工具。常用的模式除了自由模式以外还有就是缩放模式。自由模式也是默认模式，当自由模式启动以后我们可以任意点击波形图显示框任意地方，缩放模式主要用来放大某个时间段。如图 2.6.5 所示，笔者启动缩放模式以后，笔者放大 0~309ps 的时间段，事后 0~309ps 时间段就会充溢整个波形图显示框。然而缩放模式，笔者也是偶尔的情况下才会使用而已。

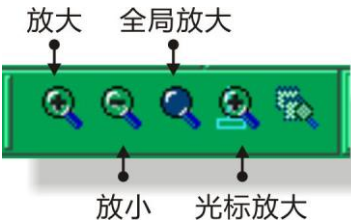


图 2.6.6 屏幕工具。

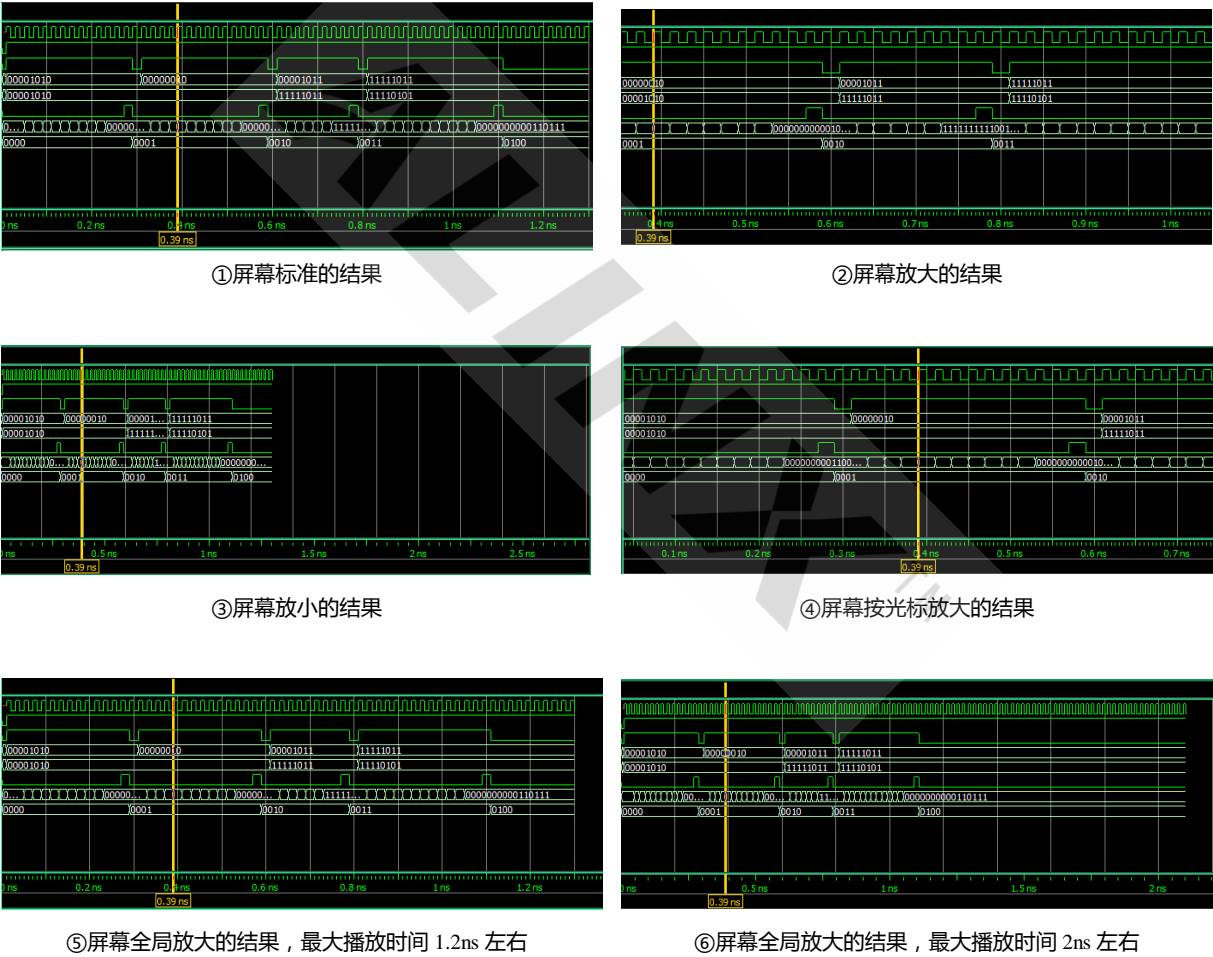


图 2.6.7 各种显示结果。

图 2.6.6 所示是屏幕工具，常用功能有屏幕放大，屏幕放小，按光标放大，还有全局放大，各种显示结果如图 2.6.7 所示。①是标准显示；②是标准显示根据放大以后的结果；③是标准显示放小以后的结果；④是标准显示按光标放大以后的结果，然而②与④的不同之处就在于④是基于光标为中心的位置——3.9ns 放大显示；



⑤与⑥都是标准显示按全局放大以后的结果，然而⑤的最大播放时间只有 1.2ns 左右，反之⑥的最大播放时间只有 2ns 左右，根据最大仿真时间的不同，全局放大也会产生不同的显示结果。反观笔者，究竟那种功能才是最常用的呢？笔者很懒，笔者比较喜欢按着 <Ctrl> + 滑鼠滚动键，缩放屏幕，因为这样作比较省力。

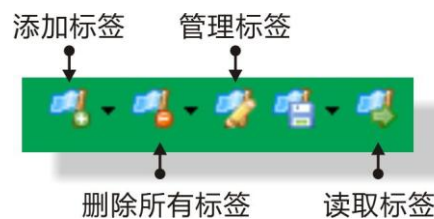
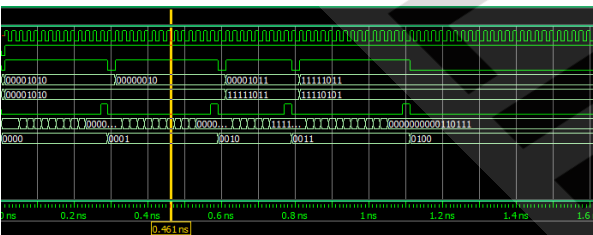


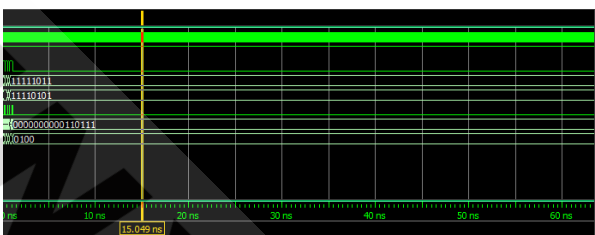
图 2.6.8 标签工具。

标签是保存当前波形图显示框状态最好的工具，如图 2.6.8 所示，常用的标签功能有：

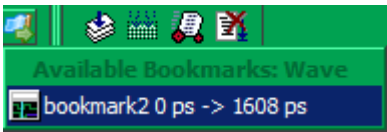
- 2. 添加标签，亦即为当前波形图显示状态插入标签；
- 3. 删除所有标签，顾名思义就是清楚所有标签记录；
- 4. 读取标签，则是读取标签记录；
- 5. 管理标签，如字面上的意思，也有标签综合功能之称。



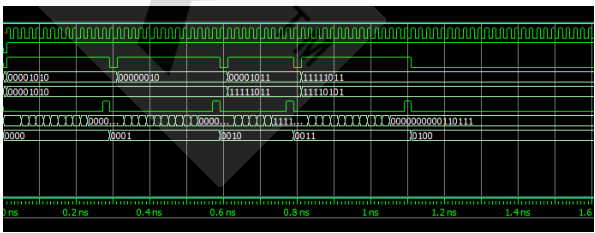
①预想保存当前显示状态。



②图 2.6.10 显示状态经搅乱以后。



③读取标签记录



④显示状态读取完毕

图 2.6.9 标签工具使用过程。

图 2.6.9 是标签工具使用过程，假设①是笔者预想保存的显示状态，然后笔者可以点击“添加标签”功能将当前显示状态记录起来。②是笔者不小心搅乱显示状态的后果，不甘心的笔者，如③所示点击“读取标签”功能，调回①的显示状态。④是读取标签的结果，①与④相较之下并没有差别。

对于波形图显式框而言，标签工具是非常方便的工具，有时候时序记录过于庞大或者过度复杂，由于波形图显式框的长度有限，这时候标签工具就派上用场了。善用标签工具会节省将波形图显式框拖来拖去的劳动，我们可以将某个时间段的显示状态记录起来，

又或者保存屏幕缩放结果。所以说，标签工具和笔者的相性是非常好。

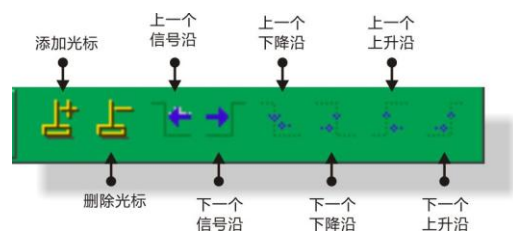


图 2.6.10 光标工具。

光标是一个非常好用的工具，光标的作用好比辅助线，参考线或者对齐线，然而光标也可以充当计算时间个数的好帮手。光标工具的常用功能如图 2.6.10 所示：

- 4. 添加光标，就是添加新光标；
- 5. 删除光标，就是删除当前选择光标；
- 6. 上一个信号沿，就是将当前光标移至当前信号的上一个信号沿；
- 7. 下一个信号沿，就是将当前光标移至当前信号的下一个信号沿；
- 8. 上一个下降沿，就是将当前光标移至当前信号的上一个下降沿；
- 9. 下一个下降沿，就是将当前光标移至当前信号的下一个下降沿；
- 10. 上一个上升沿，就是将当前光标移至当前信号的上一个上升沿；
- 11. 下一个上升沿，就是将当前光标移至当前信号的下一个上升沿。

信号沿是时序基础的基础，所谓触发沿，有时候也称为触发沿，亦即信号发生状态变化的那一刻。下降沿，意指由高变低的信号沿；换之，上升沿意指由低变高的信号沿。不管物理时序还是理想时序，信号沿的概念也是相同的。为了让读者有感知认识一下光标工具的作用，笔者稍微示范几个光标工具常用的例子。

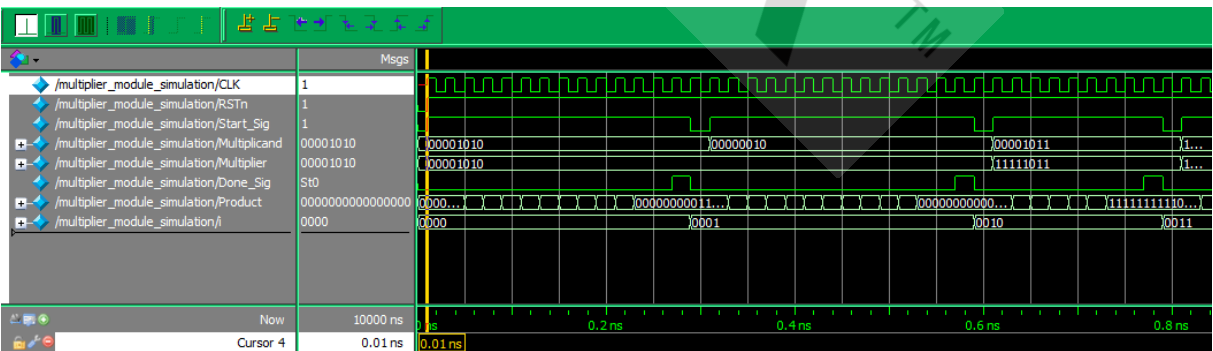


图 2.6.11 添加光标，选择信号，光标信息。

如图 2.6.11 所示，笔者新添加一个光标，光标信息显示框表示该光标名为 Cursor4 (简称 C4——塑料炸弹~笑)，指向位置为 0.01ns。单条光标的作用下，C4 作为信号的对齐辅助线。

说点题外话。人的大脑是非常喜欢偷懒的东西，其中有这样的实验说过：当一个人步入一间四方空间以后，大脑为了省事，就会记录四方空间 8 个对角，作为平衡校准。同样

的道理发生在光标的身上，时序图是一种并行可视化的记录，在此对齐作用的辅助线可以减轻眼睛还有大脑的负担。

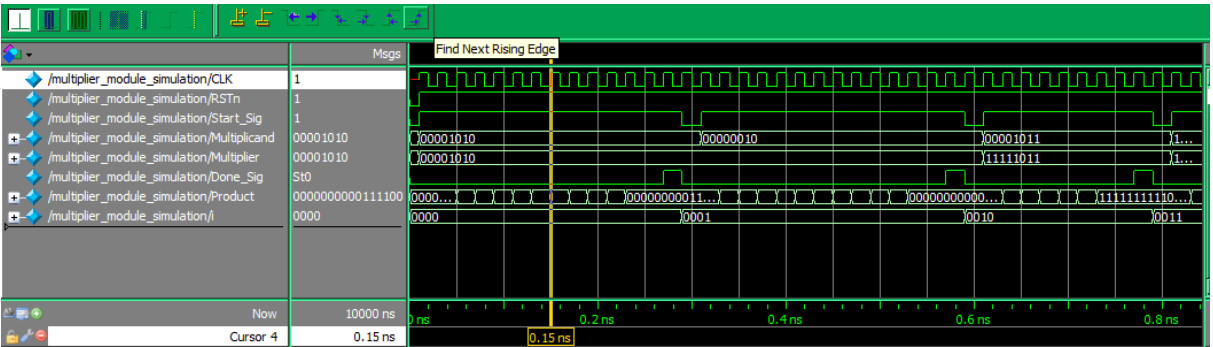


图 2.6.12 随着信号的上升沿移动光标

举例 CLK 信号与 C4 光标同时被选中。我们知道 CLK 信号一般将上升沿作为触发沿，如果笔者预想来回移动在于 CLK 信号之间的上升沿，笔者可以执行上一个上升沿还有下一个上升沿功能，结果如图 2.6.12 所示，C4 已经移动至 0.15ns 的位置。

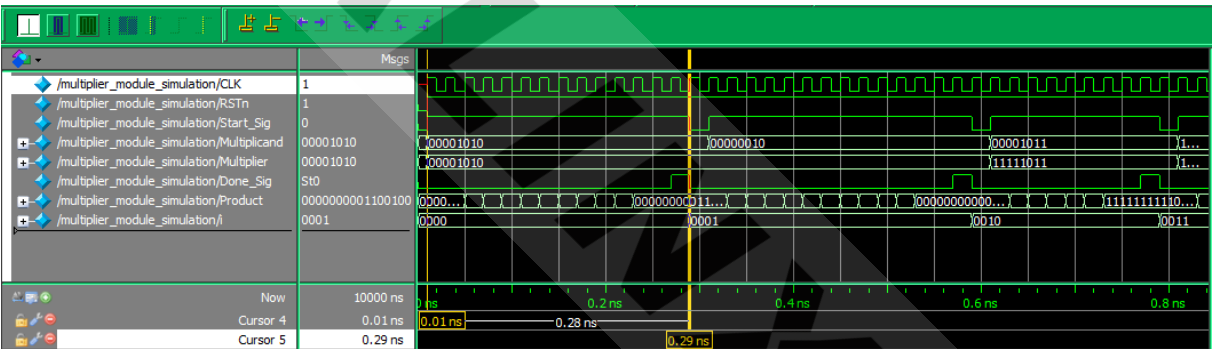


图 2.6.13 计算信号的周期时间。

假设笔者预想结出 Start\_Sig 信号的周期，于是笔者再新添加一个光标，名为 Cursor5(简称 C5)。C4 移至 0.01ns 的位置，然而 C5 移至 0.29ns 的位置，然后光标工具之间会自动求出时间差。如图 2.6.13 所示，Start\_sig 信号位于 C4~C5 之间一共占有 0.28ns 时间。如果笔者为进一步求出 Start\_Sig 信号位于 C4~C5 之间一共占用多少个时钟，笔者可以这样计算：

Start\_Sig 信号周期 / 时钟周期 = 0.28ns / 0.02ns  
= 14

结果而言，Start\_Sig 信号位于 C4~C5 之间，一共占用 14 个时钟。

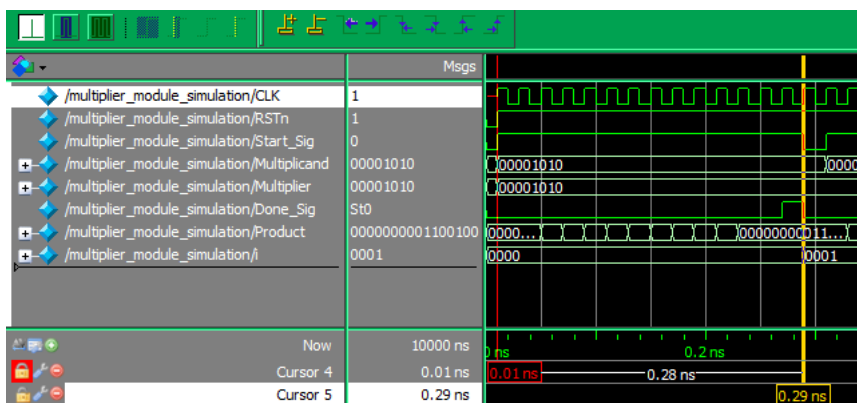


图 2.6.14 锁定光标。

光标是一种非常顽皮的工具，时常跑来跑去，因此我们可以配合光标信息显示框的锁定功能来固定光标。如图 2.6.14 所示，笔者沿着左下方点击锁定按键以后，C4 就这样被固定住了，换之 C5 却没有。除了锁定功能以外，光标信息框还有两个像极“把手”还有“禁止进入”的按键，前者是配置光标，后者是删除光标，总之是非常单纯的功能，用不着笔者特意解释吧？

锁定功能算是笔者比较常用的光标功能之一，像笔者这种喜欢作记录的男人，有时候为了标记详细的时序过程，动不动就会用上十几个光标。如果每个光标都跑来跑去的话，笔者会直接发疯的，因此锁定功能在某种意义上已经多次拯救笔者的小命。

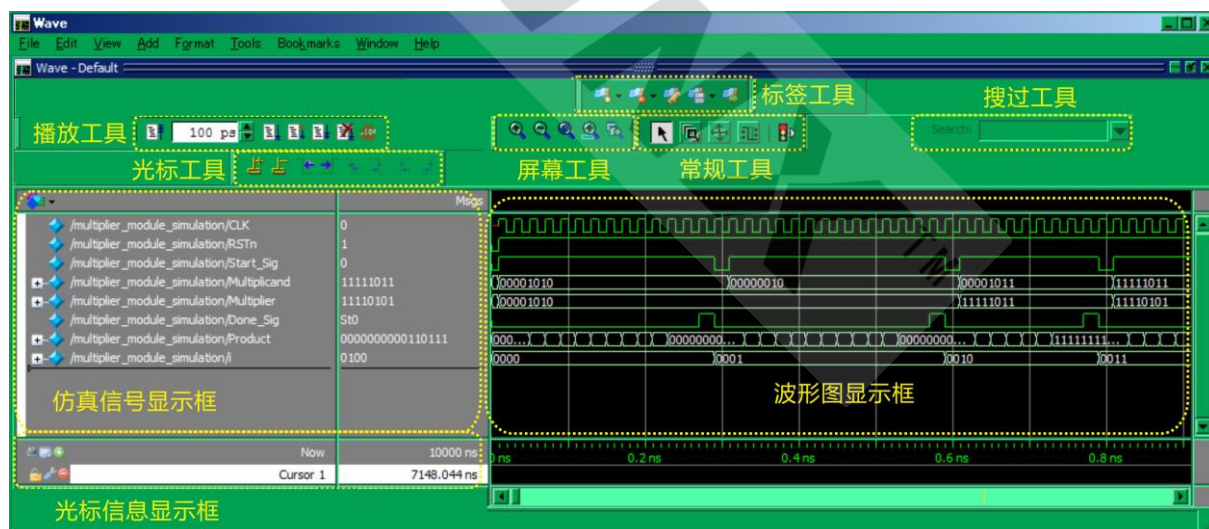
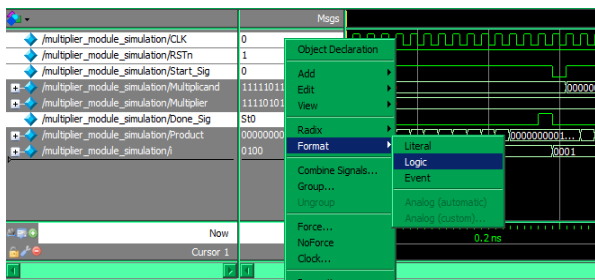
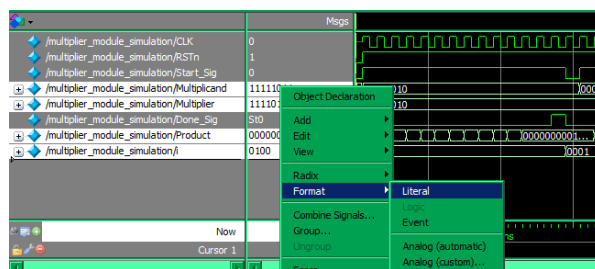


图 2.6.2 Wave 界面简介。

图 2.6.2 显示，除了波形图显示框或者光标信息显示框以外，还有一个名为仿真信号显示框的东西。笔者曾在 2.5 小节演示过，仿真信号可经由自动编译自行添加又或者人为后期添加都行。仿真信号显示框相较其它，自身隐藏的功能不仅丰富而且也非常实用。仿真信号显示框的左边是仿真信号的命名，右边则是信号处于当前时钟的结果。前者可以更动，后者则不行。



①单位宽信号



②多位宽信号

图 2.6.15 单位宽与多位宽仿真信号。

仿真信号一般分为两种，亦即单位宽信号还有多位宽信号，如图 2.6.15 所示。单位宽信号例子有 CLK 信号，RSTn 信号，Start\_Sig 信号还有 Done\_Sig 信号，一般格式皆为 Logic 逻辑。多位宽信号例子有 Multiplicand 信号，Multiplier 信号，Product 信号还有 i 信号，一般格式皆为 Literal。

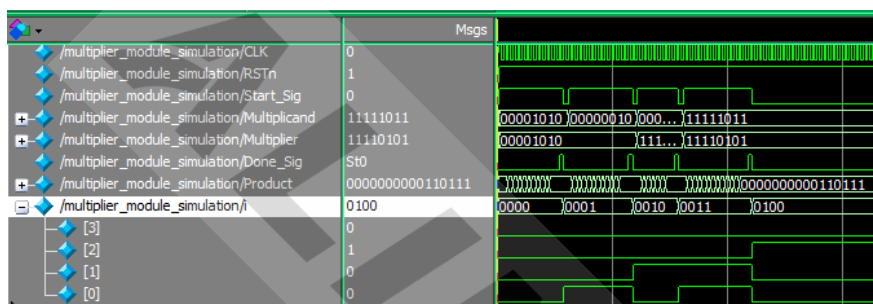


图 2.6.16 展开多位宽信号。

我们可以经过右键点击任意信号，然后沿着 Format 的右方更换格式也有可能，不过一般却不会这么作。Logic 顾名思义不是零既是一，然而 Literal 在此则是添加的意思。所谓的多位宽仿真信号是由 N 个单位宽仿真信号叠加而成，如图 2.6.16 所示，笔者故意展开仿真信号 i，然而里边都是单位宽仿真信号。

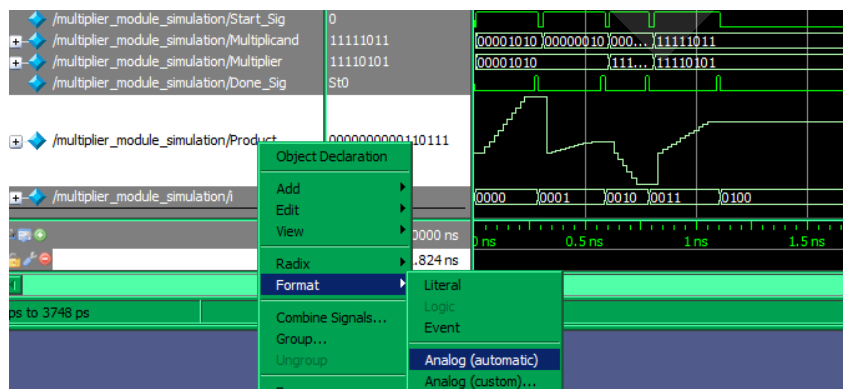
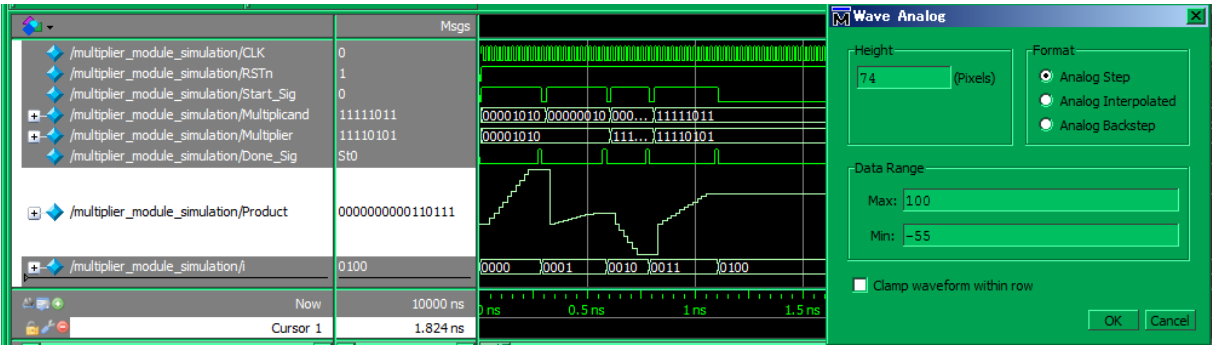


图 2.6.16 多位宽仿真信号的模拟格式。

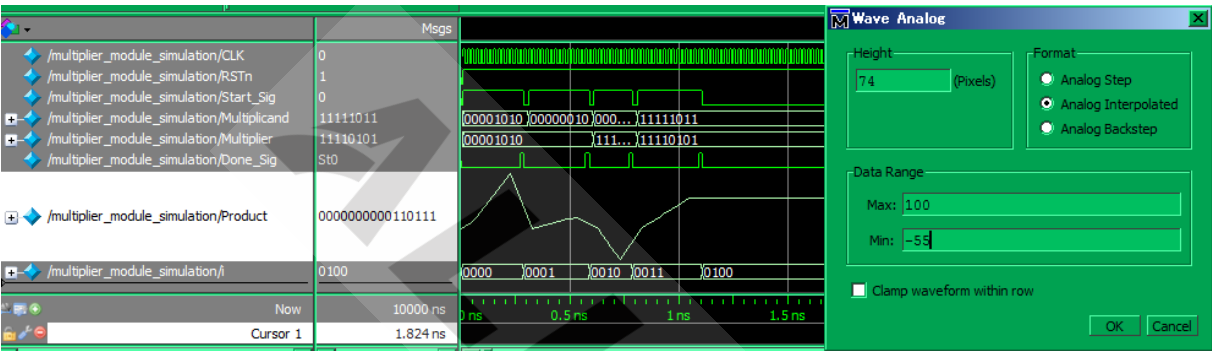
多位宽仿真信号还有另一个有趣的显示格式，亦即模拟格式（Analog）。如图 2.6.16 所示，笔者选择 Product 信号作为小白鼠，然后更改模拟格式。紧接着，Product 信号的



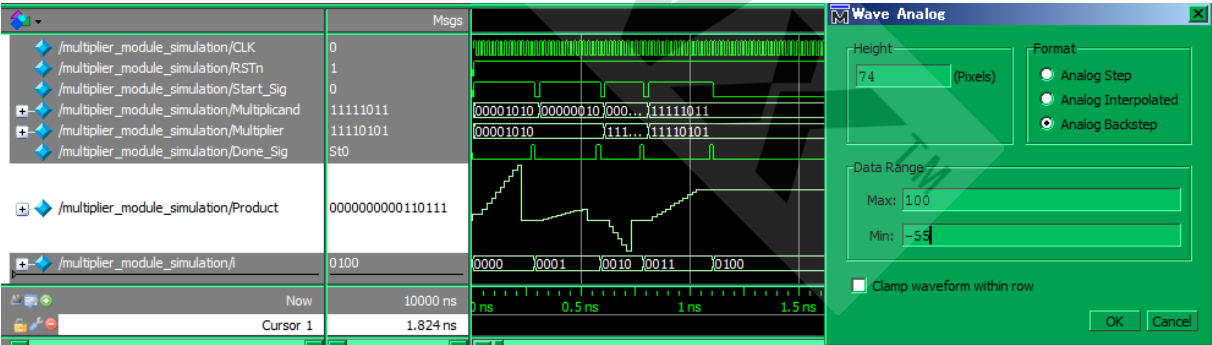
波形图再也不是叠加格式而是模拟格式。模拟格式有 Automatic ( 自动 ) 与 Custom ( 自定义 )。



① 模拟格式——阶梯式。



② 模拟格式——加入式。



③ 模拟格式——后退式。

图 2.6.17 各种模拟格式。

自动模拟格式会执行设置，显示高度（Height），还有最大值（Max）和最小值（Min）的范围，阶梯式作为默认模拟格式，反之自定义模式必须手动设置上述几个选项。此外，除了默认的阶梯式①以外，模拟格式还有加入式②，或者后退式③（预想知道什么式请自行谷歌），各种效果如图 2.6.17 所示。模拟格式的程度仿真 AC/DC 或者波形算法以外才使用而已，普通情况下很少用上。

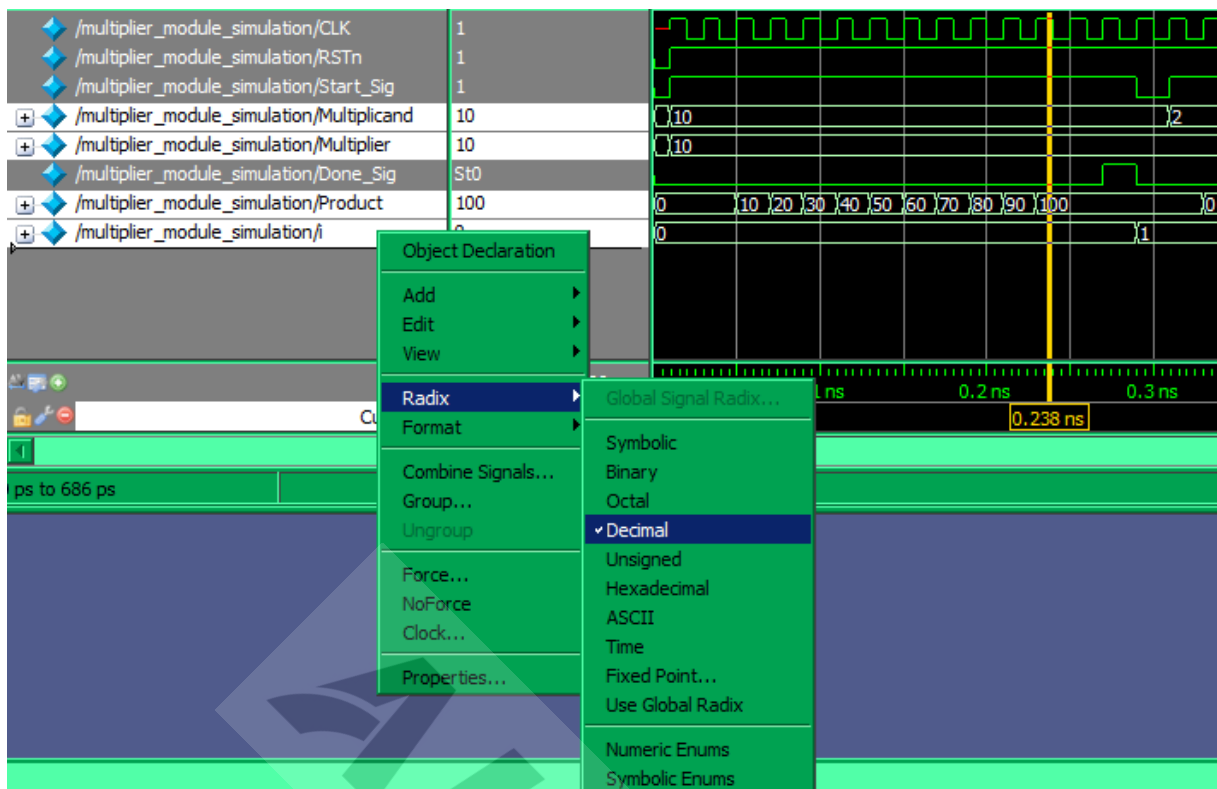


图 2.6.18 更改进制。

多位宽仿真信号除了可以更改格式 ( Format ) 以外, 还能更改基数 ( Radix )。如图 2.6.18 所示, 笔者将 Multiplicand 信号, Multiplier 信号, Product 信号, 还有 i 信号更改为十进制。此刻, 仿真信号显示框, 还有波形显示框的文字信息都为十进制表示。曾经有同学问过笔者, 如何批量更改仿真信号的显示基数? 方法非常简单, 只要按住<Ctrl>键或者<Shift>键多项选择仿真信号, 然后右键点击其中一个选中信号再更改基数, 结果其它被选中的仿真信号也会跟着变动。

Modelsim 可以支持的基数如表 2.6.1 所示：

表 2.6.1 Modelsim 可以支持的显示基数。

Symbolic 作用不明	Binary 二进制	Octal 八进制	Unsigned 无符号位
Hexadecimal 十六进制	ASCII 字符	Time 时钟单位	Fixed Point 定点
UseGlobalRadix 作用不明	Numeric Enums 作用不明	Symbolic Enums 作用不明	

如果表 2.6.1 所示, 尽是常见的基数, 不过如 Time 或者 Fixed Point 它们是比较特殊的基数, 此外还有一些作用不明的基数, 怒笔者知识有限不懂说明, 有兴趣的朋友可以自行研究看看。

有时候我们会遇见仿真信号过多, 结果逼不得已不分组仿真信号。根据笔者的习惯, 仿真信号一共有 3 种分组方法, 亦即 New Window Pane 分组, Add Divider 分组, 还有 Group 分组。



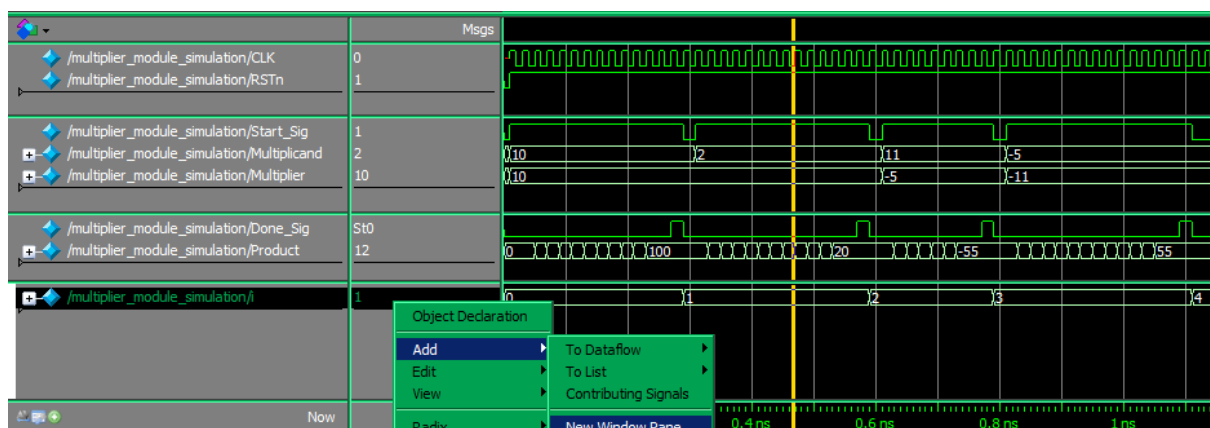


图 2.6.19 New Window Pane 分组。

如图 2.6.19 所示，随便右键任意仿真信号，然后沿着 Add 选择 New Window Pane，然后 Modelsim 会自动向下添加新 Window Pane，接着利用滑鼠手动拖拽仿真信号进入即可。

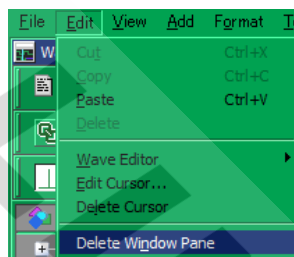


图 2.6.20 删除 Window Pane。

New Window Pane 分组有一个非常头疼的问题，Window Pane 虽然容易添加，不过删除 Window Pane 却比较麻烦。如图 2.6.20 所示，我们可以经由 Edit 菜单选择 Delete Window Pane 来删除当前选择的 Window Pane，但是连同组内信号都会一同删除。New Window Pane 分组看似作用不大，不过如果解读时序比较随意，好比使用卫生纸擦完既丢的话，New Window Pane 分组不差是一个好方法。

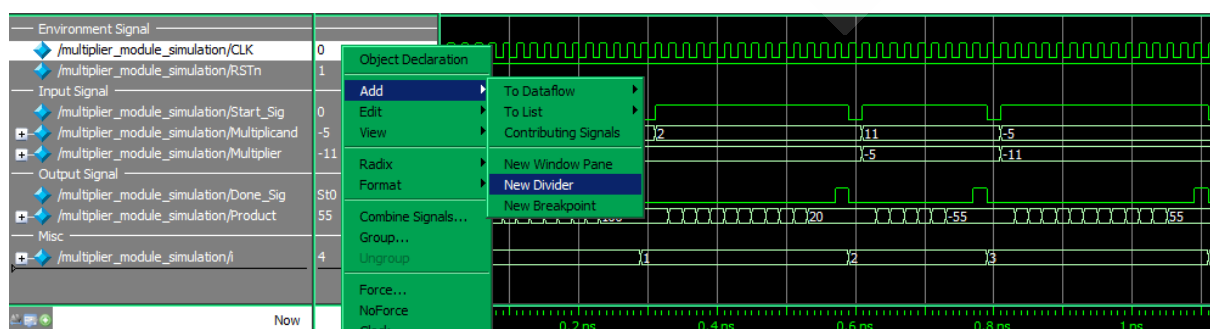
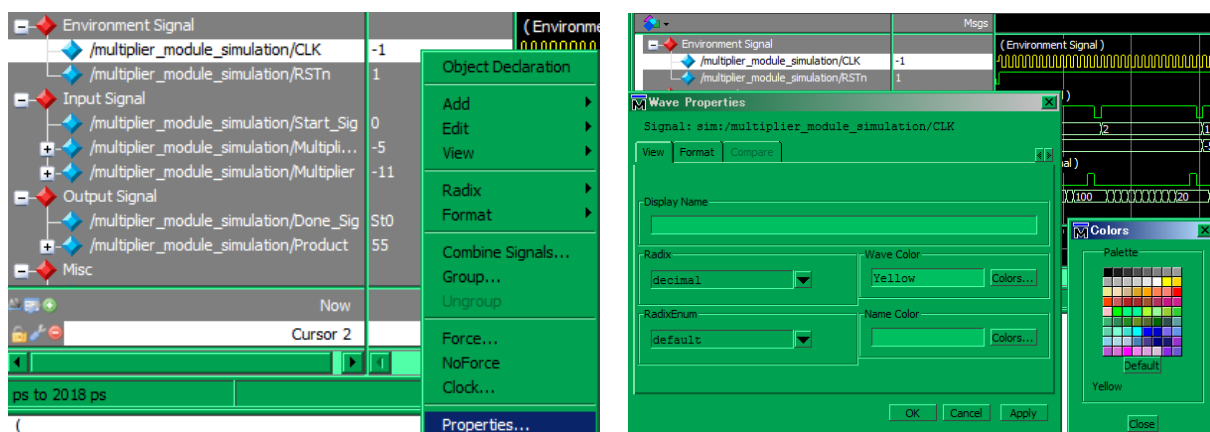


图 2.6.21 Add Divider 分组。

Add Divider 分组是笔者常用的分组方法之一。如图 2.6.1 所示，右键点击任意仿真信号，沿着 Add 然后选择 New Divider。Add Divider 分组可以注释组名，图中笔者将仿真信号分为四组，亦即 Environment Signal，Input Signal，Output Signal，还有 Misc。如果解读





①右键点选择 Properties。

②选择颜色。

图 2.6.25 信号选择颜色。

仿真信号显示框还有较为花俏的功能，亦即更改信号波形的颜色。如图 2.6.25 所示，①笔者右键点击 CLK 信号，然后选择 Properties。不一会儿，②Wave Properties 窗口就会浮现在眼前，沿着 Wave Color 点击 <Colors>按键，结果又会跳出颜色小窗口，笔者选择黄色。事后，CLK 信号的波形图颜色就会让上黄色，过程诶图 2.6.25 所示。

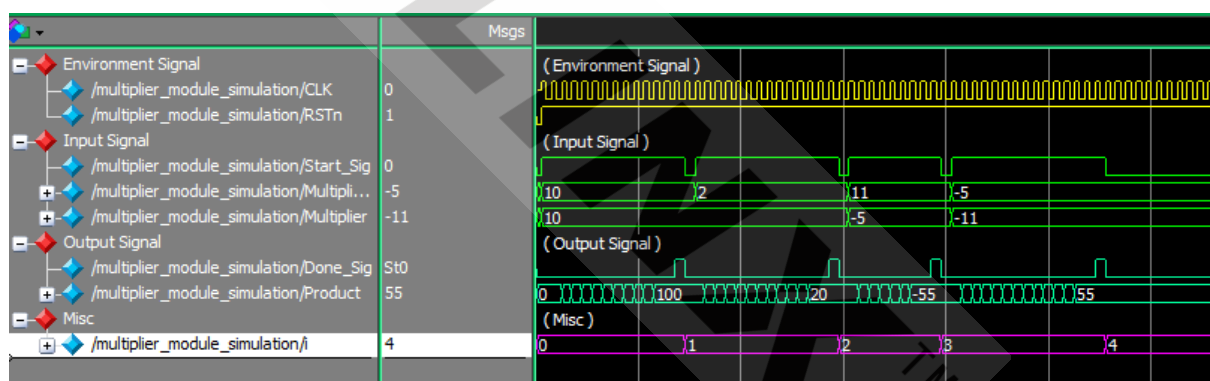


图 2.6.26 酷爆的波形图。

只要读者存有几分心思为各组信号染上不同的颜色，最终会产生预想之外的效果。如图 2.6.26 所示，是笔者心血来潮的艺术成果，读者是不是觉得很酷呢？只要读者有心，专业波形图其实一点也不难做出来。

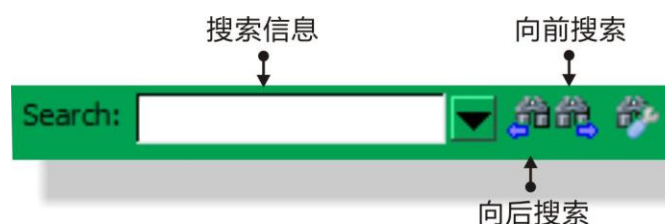


图 2.6.27 搜索工具。

如图 2.6.27 所示是搜索工具，然而常用的功能有：

12. 搜索信息，亦即寻找信息；
13. 向后搜索，让当前光标在当前信号上向左寻找搜索信息；
14. 向前搜索，让当前光标在当前信号上向右寻找搜索信息。

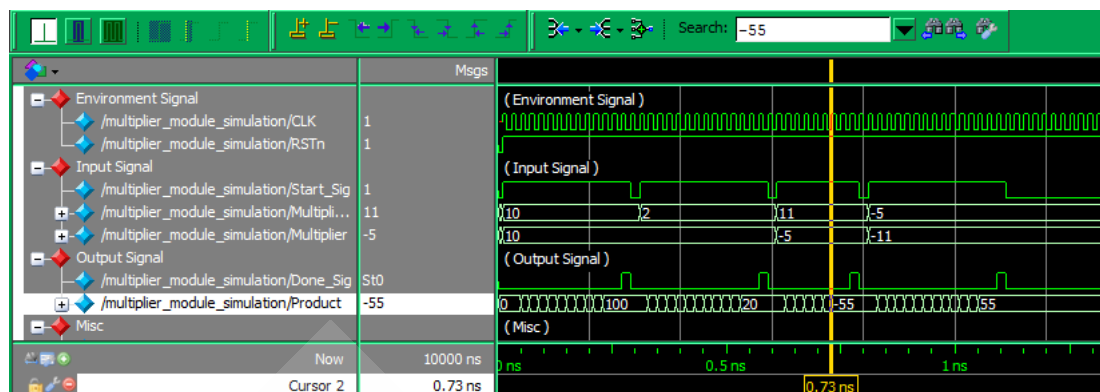


图 2.6.28 使用搜索工具。

搜索工具虽说是综合工具，但是也只有仿真信号显示框使用而已。如图 2.6.28 所示，笔者同时选择 Product 信号还有光标 C2，接着笔者又在搜索信息输入框中写入“-55”，然后按回车键。回车键默认是向前寻找搜索信息，结果光标 C2 停留在 Product 信号 -55 的地方。搜索工具的作用一般都是在海量信息中寻找某个结果。

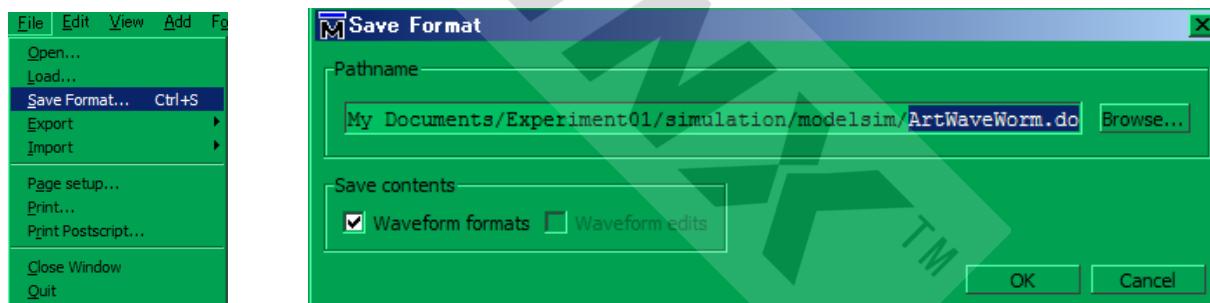


图 2.6.29 保存 Wave 界面。

当我们完成 Wave 界面的最终设置以后，我们可以将当前 wave 界面保存下来。沿着 File 菜单点击 Save Format，不一会 Save Format 窗口就会浮在眼前。左下方记得勾选 Waveform formarts，然后输入任意名字，.do 文件就会保存在仿真项目的目录下。在此，笔者将图 2.6.26 的 wave 界面取名为 ArtWaveWorm，并且保存下来。这样作有两个目的，其一就是为未来做好复习的准备；其二就是与人分享。

虽然还有许多工具未曾涉及其中，不管初学也好还是熟练也好，只要掌握上述方法，基本上已经足够应付仿真了。享受节目当然要选择清晰的画面啦，然而操作 Wave 界面好比调节电视机的显示状态，只要 wave 界面足够显示时序细节，那么操作就到此为止。余下，我们应该保留更多精力用来解读时序，难道会有傻子使用电视不是观赏节目，而是研究电视机的构造或者产生原理吗？

## 总结：

Modelsim 是否等价仿真？答案当然是否定的，原因很单纯，因为 Modelsim 只是一架性能比较优秀的电视机而已。然而，如何摆弄这架电视机使它播放最有用的节目才是学习 Modelsim 的真正目的。不过，观赏节目之前我们必须学会如何打开电视机，根据笔者的习惯，Modelsim 有三种启动方法，亦即：自动编译，半自动编译还有手动编译。

作为初学或者已经熟透 Modelsim 的朋友，不管怎么样，笔者还是强烈推荐使用自动编译，然后再选择性使用半自动编译，至于手动编译除非有特殊因数，不然就无视。仿真本来就是非常费劲耗力的差事，而且启动 modelsim 的过程也非常猥琐，我们不应该耗费过多的精力在作用性不大的地方。因此，自动编译无疑是我们最好的节能手段，不过自动编译也有局限性的弱点，结果选择性使用半自动编译就是弥补这方面的不足。

当 Modelsim 启动成功以后，各种作用界面也会跟着浮现在我们的面前。常见的界面有，Project 界面，File 界面，Library 界面，Transcript 界面，Simulation 界面，还有 wave 界面。自动编译会省略 Project 界面的执行过程，然而编译成功以后的信息会保存在默认设计库——Work 里边。Library 界面有，除了临时作用的设计库以外，也有相关的资源库。此外 Library 界面也可以充当“更新文件（编译文件）”，还有“启动仿真”的快捷入口。

Simulation 界面需要启动仿真成功以后才会浮现的仿真界面，Simulation 界面一般省略名为 sim 界面。Sim 界面的里边隐藏大量仿真信号，不过自动编译仅将仿真对象的输入端作为仿真信号而已，换之半自动编译还有手动编译可以随意添加仿真信号。仿真信号添加成功以后，便会移至 wave 界面的仿真信号显示框当中。

如说 Wave 界面是 Modelsim 最重要的界面，话语一点也过不过分。Wave 界面的作用就是可视化 Verilog HDL 语言，也是俗称的波形图播放。Wave 界面主要分为三个显示框，亦即波形图显示框，仿真信号显示框，光标信息显示框。波形图显示框最为重要，而且操作工具也是最多；仿真信号显示框，作用仅此与波形图显示框，它虽然没有丰富的操作工具，不过本身自带许多操作功能；光标信息显示框，顾名思义就是用来显示光标的信息还有管理光标，作用程度虽然不及前面两者，但是也很重要。

如何操作 wave 界面，就是如何摆弄以上 3 个显示框而已，然后再善用操作工具还有显示框自带的操作功能，以致将时序细节有多清晰就多清晰般显示出来。根据笔者的习惯，对齐信号是非常重要的，因为可以节约眼睛的注意力还有大脑的焦距力，对此我们可以使用光标充当辅助线。

除此之外，分组信号也非常重要，分组信号不仅造就管理仿真信号变成更加省力，而且解读时序也能更加集中。然而，分组信号最根本的原因是让波形图变得更加整洁和美观，因为顺眼的东西可以柔化焦急的心情，打造仿真的好心境，好心情自然会产生好结果，这是千古不变的事实。

# 第三章 理想就是美丽

## 3.1 理想时序

笔者虽然已经写过无数关于理想时序的用法，然而笔者却未曾思考它的本质？理想时序究竟是什么，想必读者非常好奇吧！？首先，让我们先来理解一下理想的概念，理想不等价完美，举例而言：假设旅游 A 的一人费用是 1000 元，世界上也只有傻子准备 1000 元正而已，换做常人的准备金额通常都是费用的好几倍。

如果读者准备 5000 元，那么这笔 5000 元就是理想的准备金额，至于完美准备金额当然是越多越好。结果而言，理想时序并不是完美时序，完美时序好比无穷无上限的准备金额一样，根本不可能存在在这个世界上，完美顶多只是幻想中的一滴浪漫，或者盲目的一丝自负而已。

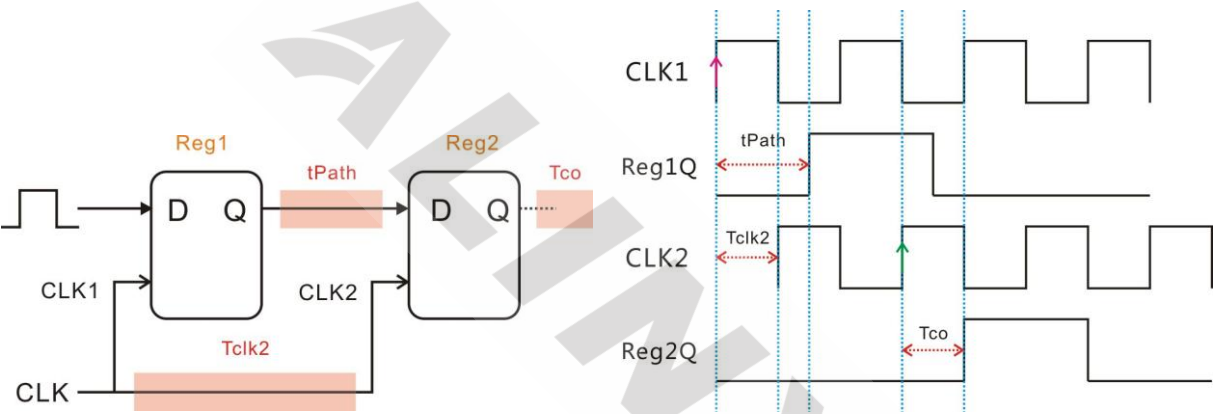


图 3.1.1 一对寄存器与物理延迟。

图 3.1.2 物理时序。

笔者曾近还是传统流派的门徒之时，一位臭脸的师兄恰好负新弟子的指导工作，师兄非常不厌烦般在黑板上绘出两幅意义不明的涂鸦，如图 3.1.1 与 3.1.2 所示，然后命令我们自己想办法用 Verilog 表达出来，一个时辰以后提交答案。就这样，一群师弟们便吵吵嚷嚷起来。

抱怨的抱怨，瞎搞的瞎搞，总之每个人泥菩萨过江就是了。沉思将喧闹的环境隔开，笔者注释黑板嘀咕道，图 3.1.1 当中的 tPath，Tclk2 还有 Tco 都是物理延迟，然而图 3.1.2 就是物理延迟造成的物理时序，物理时序有一个特征，那就是信号失去对齐性。图 3.1.1 姑且还能使用 Verilog 表达出来，结果如代码 3.1.1 所示：

```
1. reg Reg1,Reg2;
2.
3. always @ ( posedge CLK )
4. begin
5.     Reg1 <= Sig;
6.     Reg2 <= Reg1;
```



但是 Verilog 就没有办法表达图 3.1.2，因为 Verilog 不能描述  $t_{Path}$ ,  $T_{clk2}$  还有  $T_{co}$  等物理延迟。想着想着，于是笔者便结出“不可能”三个字的结论，不可能的事情就是不可能完成。节能的本质绝对不允许无用功的行为，与其浪费气力笔者还不如睡觉好。不知不觉之间周围的吵杂声渐渐逝去，原来是师兄回来了，各个师弟便紧张兮兮坐回原位，师兄来回扫视一样，眼见人人拿着代码，唯有笔者两手空空，一阵迅风忽然划过耳边，脸颊来不仅疼痛，笔者整个人便落在地上 ... 死了，这次笔者的第一次死亡。

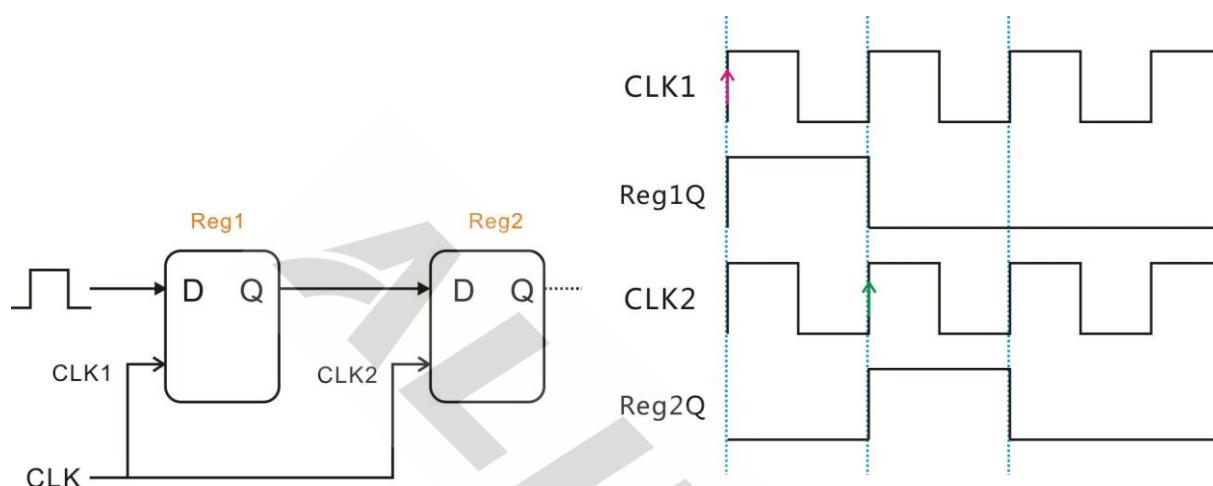


图 3.1.3 一对理想的寄存器。

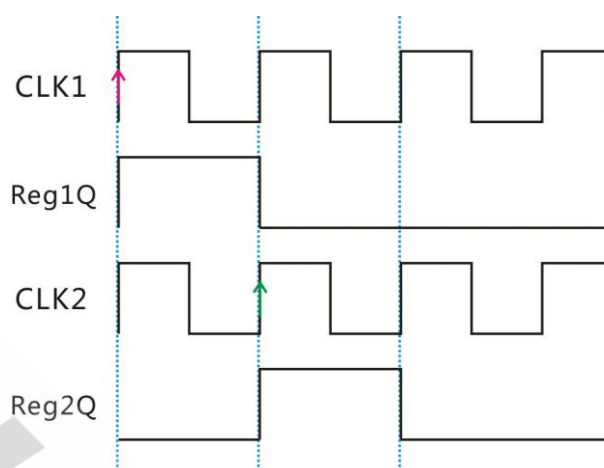


图 3.1.4 理想时序。

从大神殿复活回来以后，笔者便一直被关在地牢里面壁，打从第一次笔者充分感觉到传统流派的无理暴力，笔者没有折服暴力的理由。于是，笔者将记忆当中的图 3.1.1 还有图 3.1.2 按自己的感觉重绘出来，结果如图 3.1.3 还有图 3.1.4 所示。图 3.1.3 是的一对寄存器的理想模型，理想的东西当然产生的理想时序，结果就是 3.1.4。

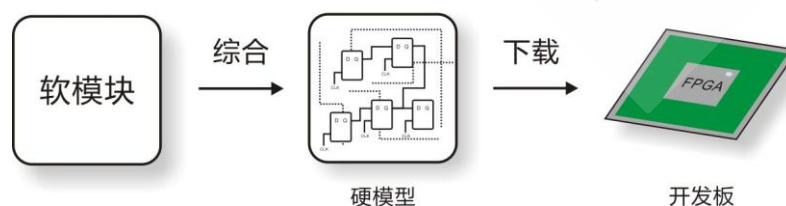


图 3.1.5 硬模型。

好奇的同学可能会问：“那些物理延迟消失在哪里了？”，物理延迟并不是消失，而是打从一开始它们根本不存在。根据笔者的想象，一般软模块成功编译以后会产生模型，接着模型就可以下载到开发板当中，如图 3.1.5 所示。笔者之所以模型为硬模型，是因为综合器给予模型相对应的物理信息，不过这些相对应的物理信息只是表面估计，实际上却与下载以后的物理信息持有一定差别。



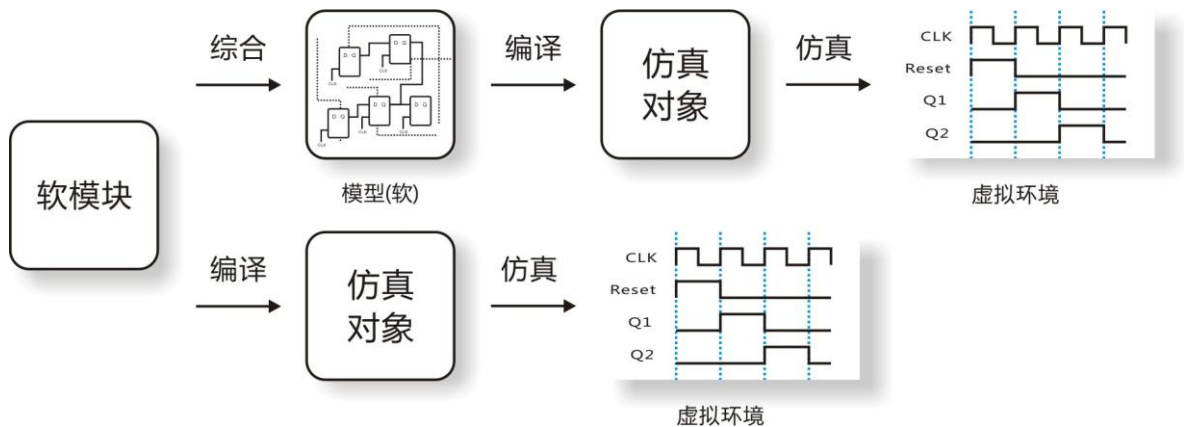


图 3.1.6 软模型。

经过综合以后的模型，也有选择不下载到开发板的权利，而是经过 Modelsim 编译成为仿真对象，最终下载到虚拟环境，此刻模型称为软模型。软模型自身虽然也有携带综合器给予的物理信息，但是使不使用就是另外一回事。事实上，软模块也可以省略综合这个步骤，直接交由 Modelsim 编译成为仿真对象直接仿真，过程如图 3.1.6 所示。结果，笔者可以这样总结道：

图 3.1.5 与图 3.1.6 的共同点就是起源与软模块，然而软模块却是未综合未编译的原始种子，它是理想，它没有污垢。换之，综合以后的软模块可以选择自己的演化方向，成为硬模型还是软模型。硬模型没有权利拒绝物理信息，相反软模型可以接受或者拒物理信息。如果软模型不接受物理信息，这意味着它继承软模块的理想本质。因此笔者可以断言，物理信息是后生加上去的枷锁，好让时序失去原先无垢的本质。

物理信息好比沉重的包袱，压在背上增加负担，这种感觉完全表达 Verilog 描述物理时序的痛苦。Verilog 本属理想，产生的时序理所当然也是理想，可是为什么传统流派却不这么认为呢？鬼才知道那般家伙的脑子在思考什么，不然也不会突然一脚把笔者踢死。就这样，反叛的念头才在心底深处逐渐萌芽。

笔者选择理想时序不仅因为是理想时序可以节能，因为理想时序是仿真的真理。领悟理想时序让笔者失眠三个夜晚，期间笔者一直在思考理想时序究竟拥有怎样的特性？

1. 其一理想时序必须拥有理想时钟源。
2. 其二理想时序不会失去对齐性。
3. 其三理想时序拥有 2 个触发事件（时序表现）。

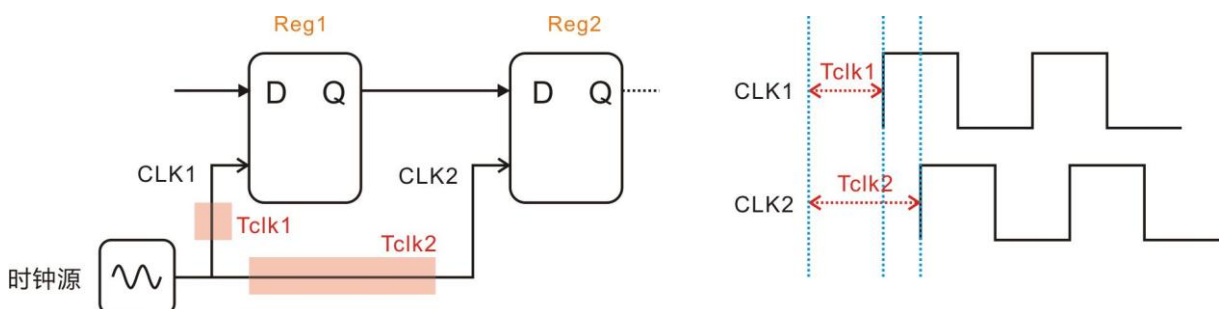


图 3.1.7 时钟路径延迟。

理想的时钟源不存在时钟路径的延迟，如图 3.1.7 所示是存在时钟路径延迟的物理模型还有物理时序。时钟源所产生的时钟源，经过不同物理延迟的  $T_{clk1}$  与  $T_{clk2}$  以后，Reg1 与 Reg2 的时钟源被失去对齐性。 $T_{clk2}$  与  $T_{clk1}$  之间的差别也称为时钟差——Clock Skew。

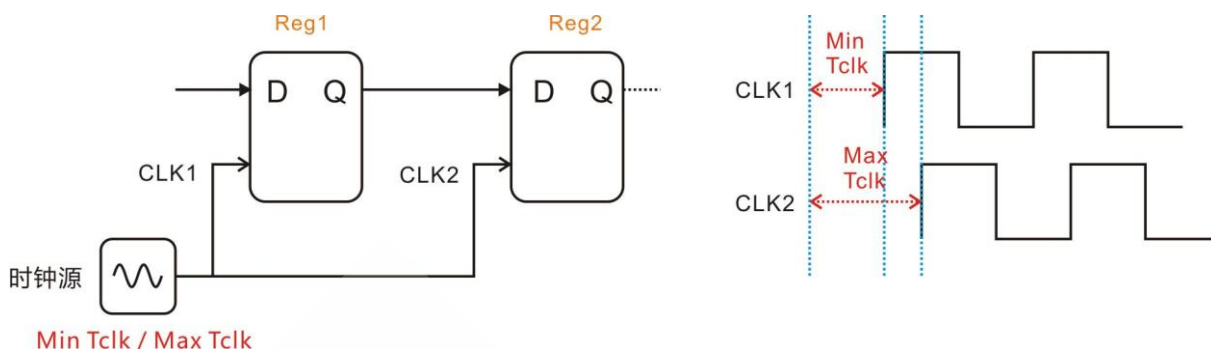


图 3.1.8 时钟源抖动。

另外一种产生时钟差的缘由不是时钟路径延迟而是时钟源的输出抖动。如图 3.1.8 所示，假设 CLK1 与 CLK2 的路径并不存在延迟，但是时钟源的抖动会导致 CLK1 与 CLK2 失去对齐性。所谓的时钟抖动就是时钟源的输出延迟并不一致，有时大（Max），有时小（Min），一般是时钟源劣质或者老化引起。

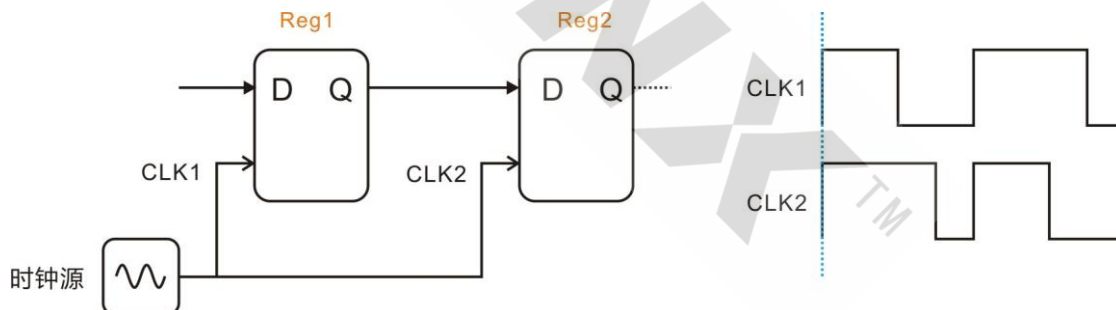


图 3.1.9 时钟周期不一致。

另一种较为严重的问题是时钟周期不一致，如图 3.1.9 所示，虽然时钟源没有输出抖动，而且时钟路径也不存在任何物理延迟，可是时钟源却非常不稳定，结果导致时钟周期有时大有时小。这种现象一般有两个凶手，其一就是时钟源劣质或者老化；其二就是寄存器的时钟供源端出现问题。

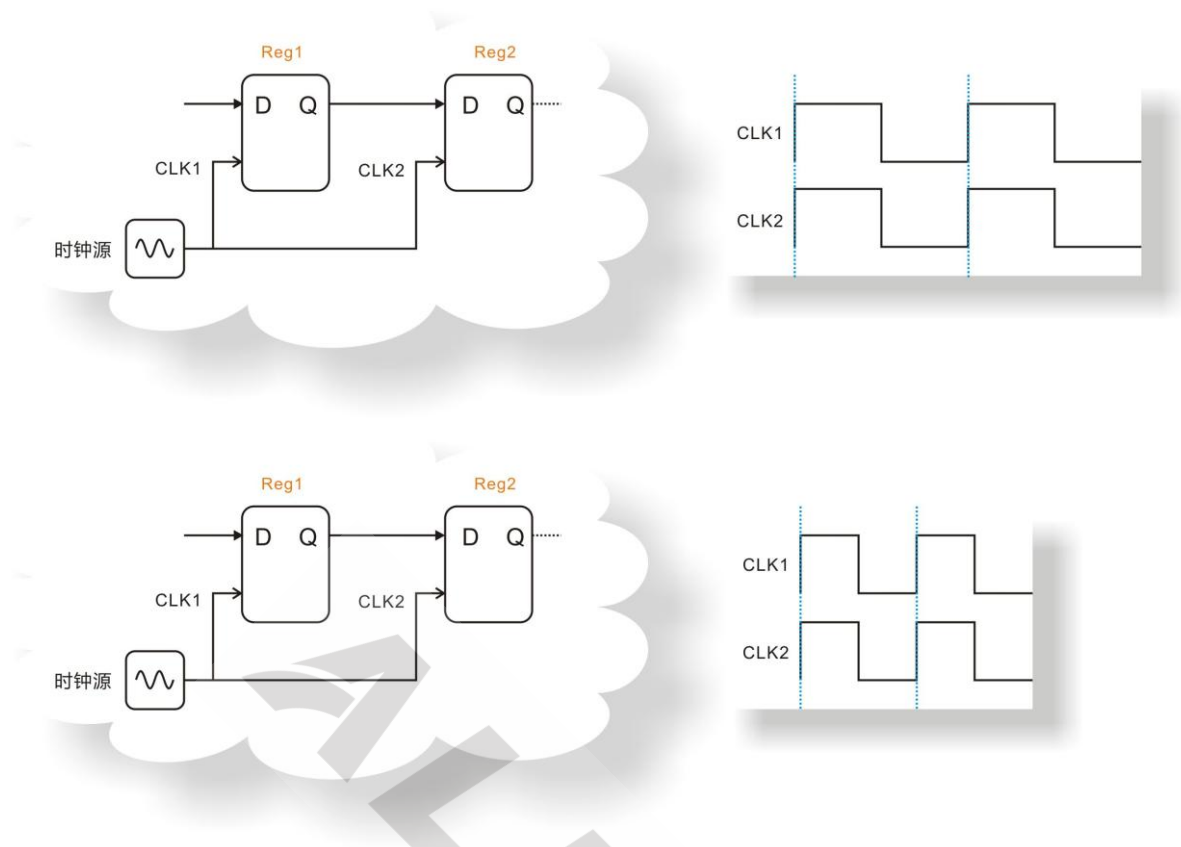


图 3.1.10 区域性的固定时钟周期。

虽然建模允许一个以上的时钟源出现，不过不同的时钟源只允许出现在各种的区域而已。如图 3.1.10 所示，图中有两个区域，而且各个区域都拥自己固时钟源，而且还是固定性周期。除非有特别的机制，不然不同时钟源的区域，信号不能随意往来。

为什么理想的时钟源如此重要？因为时钟源在仿真当中视为“环境输入”，环境在生态学是指生物赖以生存的需求，如空气，土地，食物，水源等。如果环境状态不佳，生命也不会好到那里去。所以说，时钟源是仿真首屈一指的需求，不仅仿真对象需要它，而且激励内容同样也需要它。

建模一般分为无时钟源的组合逻辑建模，另一种则是有时钟源的时序逻辑建模，也是俗称的 RTL 级建模。笔者曾经所过，仿真既是虚拟建模，而且也是虚拟的时序逻辑建模，时钟源除了提供模块活动的心跳以外，时钟源也领导所有模块的同步性。既然时钟源如此重要，时钟源怎么可以不理想呢？

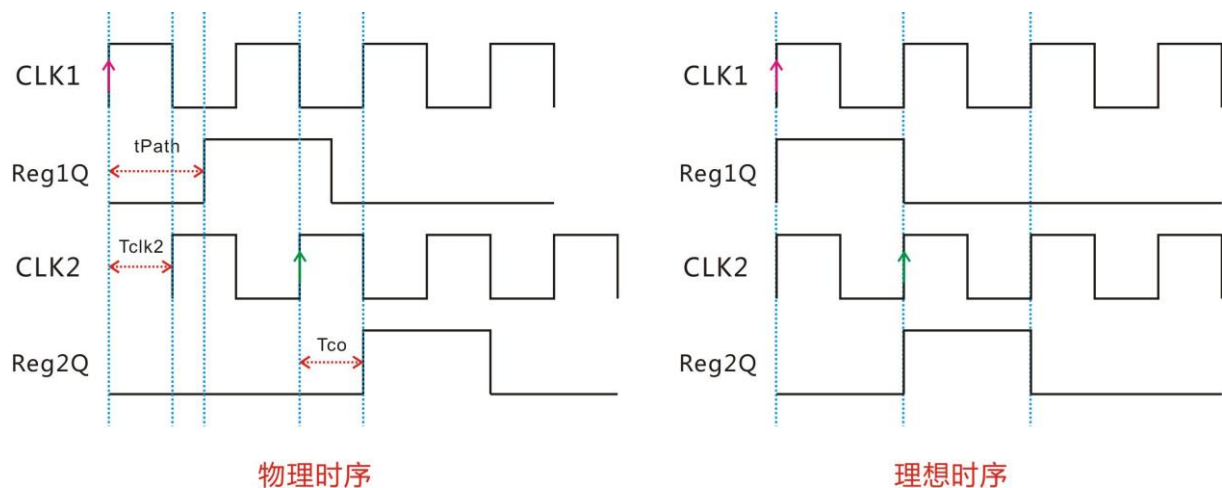


图 3.1.11 物理时序与理想时序。

理想时序的特性除了理想时钟源以外，理想时序的信号不会失去对齐性，亦即不存在物理延迟。如图 3.1.11 所示，Verilog 本质所属理想，所以描述不了左边的物理时序，换之右边则是 Verilog 可以描述的理想时序。所谓对齐性，既是信号不仅不会跳来跳去之余，信号还会依准时钟沿划出一块一块。

宏观上，信号对齐就是上述这么一回事；微观上，信号对齐除了拥有更高的表达能力以外，时序细节也更加清晰。然而最为重要是信号对齐以后，两种时序触发事件，亦即“时间点事件”还有“即时事件”才能有效划分特征。那么，[什么又是时序触发事件呢？](#)

```
<= // 非阻塞赋值
= // 阻塞赋值
```

Verilog 有 2 种赋值操作符，亦即 <= 非阻塞赋值操作符，还有 = 阻塞赋值操作符，然而，关于两种赋值操作符的差别，传统流派总是一笑而过或者干脆装傻不知，所以笔者才讨厌它们。

<= 非阻塞赋值操作符，一般用于时序逻辑建模（设计），如：

```
always @ ( posedge CLK )
    begin Reg1 <= Sig_In; Reg2 <= Reg1; end
```

= 阻塞赋值操作符，一般用于组合逻辑建模（设计），如：

```
always @ ( * ) // 输出选择器
    if( Start_Sig_A ) rLED = LED_U1;
    else if( Start_Sig_B ) rLED = LED_U2;
    else rLED = x;
```

又或者用于连线或者输出驱动，如：

```

wire LineA;
wire LineB;
assign LineB = LineA; // LineA 连线 LineB , 亦即 LineA 驱动 LineB 连线

```

```

assign LED = rLED; // 寄存器 rLED , 驱动 LED 输出端。

```

以上纯属两种赋值操作符的建模表达方式而已，然而它们真正的价值还有差别只有在于理想时序之间才能凸显出来。

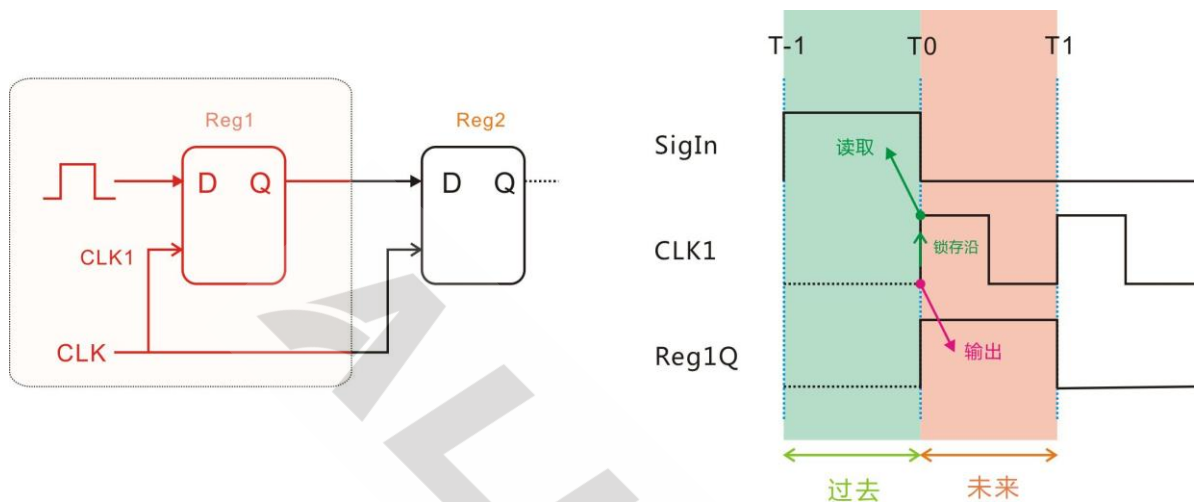


图 3.1.12 时间点事件 T-1~T0。

首先，让我们来了解一下什么是时间点事件。图 3.1.12 的左图有一对寄存器，期间 Reg1 从读取信号到输出信号的瞬间，时序何种变化是我们必须理解的地方。图 3.1.12 的右图是 Reg1 的时序活动，时序上边标示的 T-1，T0，还有 T1 是时钟的经过，然而 T 负值除了表示初始化还有复位化以外，T 负值还表示准备时刻。

在 T-1 的时候，SigIn 送入一块值为逻辑 1 的数据，然后该时间点结束。在 T0 的时候，Reg1 被 CLK1 的上升沿触发，该触发沿也称为锁存沿，然后 Reg1 会锁存该时间点的过去值——逻辑 1，接着决定发送逻辑 1。T1 时间点结束以后，Reg1 便输出未来值——逻辑 1。图 3.1.12 对应的时序行为描述如下：

```

input SigIn; // 输入端声明
reg Reg1;    // 寄存器声明
always @ ( posedge CLK1 )
    begin Reg1 <= SigIn; ..... // 时序行为描述

```

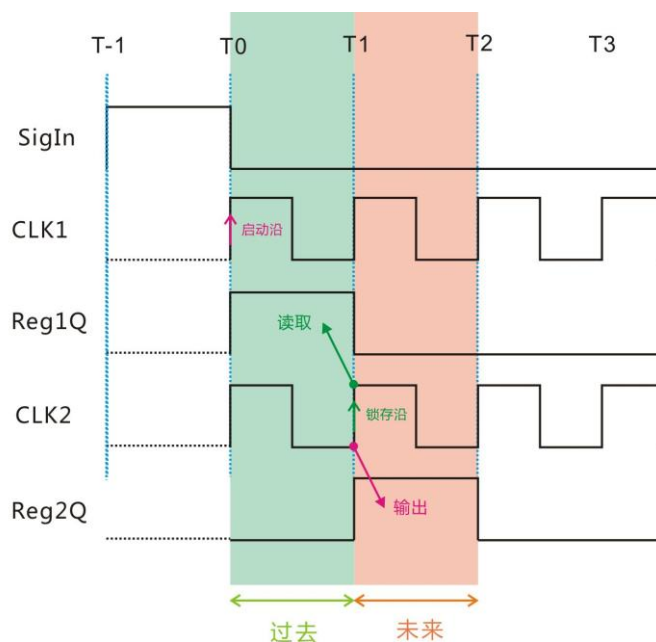
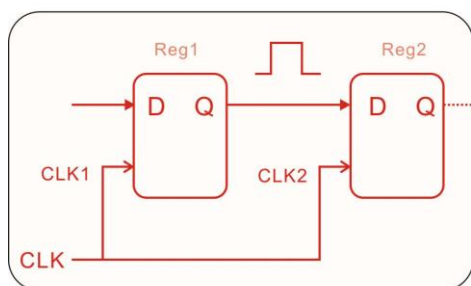


图 3.1.13 时间点事件 T0~T1。

当 Reg1 读入逻辑 1 的 SigIn 以后，SigIn 便会游走 Reg1 与 Reg2 之间，然后再经由 Reg2Q 输出，过程如图 3.1.13 的左图所示。图 3.1.13 的右图则是 Reg1 与 Reg2 的时序活动，在 T0 的最后（图 3.1.12 的结果）Reg1 输出逻辑 1 的未来值，然而时间的巨轮现在已经转向 T1。在 T1 的时候，CLK2 的锁存沿造就 Reg2 读取 Reg1Q 的过去值（亦即 T0 的未来值），然后 Reg2 在 T1 的结束之际决定输出逻辑 1 的未来值。

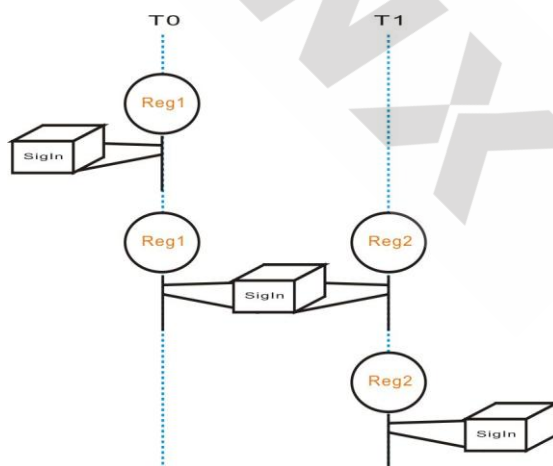


图 3.1.14 时间点事件形象化。

从旁观看，整体时序具体一点程好似 Reg1 将一块 SigIn 数据搬入然后又搬给 Reg2，Reg2 接收之后又将输出搬出，过程如图 3.1.14 所示。然而，此刻 Reg1 和 Reg2 想干什么或者该怎么干都是发生在时间点那个瞬间。图 3.1.13 对应的时序行为描述如下：

```
input SigIn; // 输入端声明
reg Reg1;    // 寄存器声明
always @ ( posedge CLK1 )
```



```
begin Reg1 <= SigIn; Reg2 <= Reg1; end // 时序行为描述
```

至于 Reg2 在时间点 T1 之后想将数据版给谁，这点笔者就不知道了，毕竟 Reg1 与 Reg2 的行为描述就是那么丁点而已，寂寞的朋友可以自行添加 Reg3 或者 Reg4 看看。总结来说，时间点事件，分为：“时间点”，亦即此刻谁谁想干什么，操作标志为  $\leq$ ；“过去值”，亦即此刻送来的东西；“未来值”，亦即此刻送去的东西。就这样，一刻又一刻的时间点连接起来，成为理想的时序图。

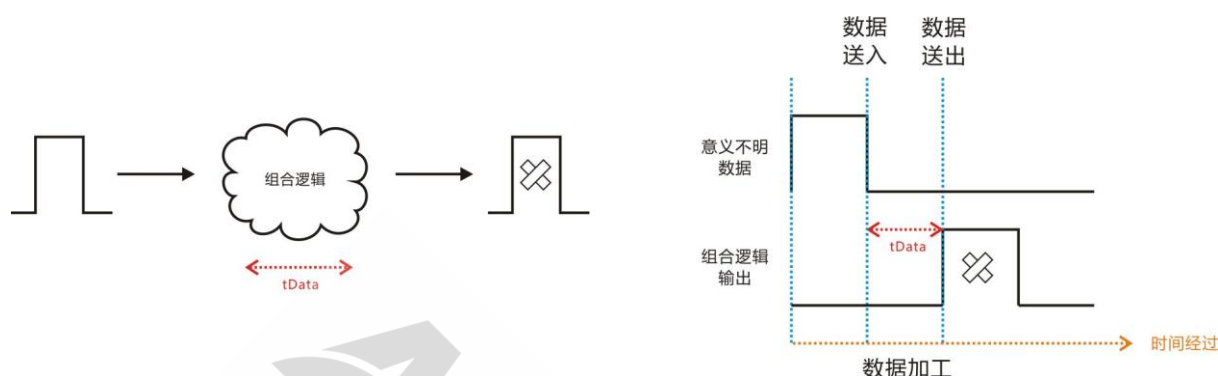


图 3.1.15 物理延迟 tData。

除了时间点事件以外，理想时序还有即时事件，顾名思义**即时就是立即**，马上，英文叫做 immediately。不过，为什么即时称为即时？而且，即时的由来又是什么？如图 3.1.15 的左图所示，读者尝试将**组合逻辑想象成为数据加工工厂**，然而在现实世界当中，无论是怎么优秀的加工技术也需要一定的加工时间，因此数据加工时间称为 tData。

假设有一块意义不明的数据送入组合逻辑加工，经过 tData 的时间以后，数据贴上创可贴表示加工完毕。图 3.1.15 的右图是整体过程的时序图，意义不明的数据被送入组合逻辑以后，会经过 tData 的延迟，然后再经由组合逻辑输出。

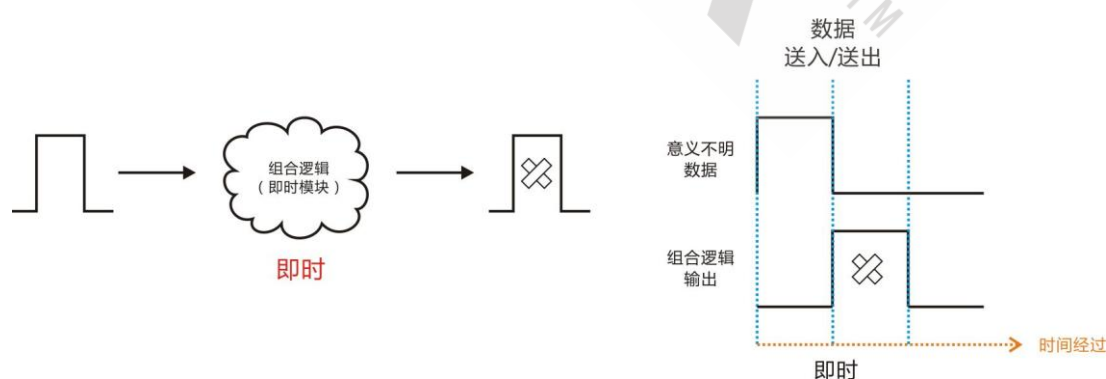


图 3.1.16 即时效果。

**理想时序不是现实世界，而是梦境中的美丽世界**，那里不存在任何物理延迟也不存在 1ns 或者 3.2ns 等丑陋的数值。如图 3.1.16 所示，意义不明的数据送入不久就立即加工完毕并且输出，期间加工即时完成，如此模块笔者称为即时模块。



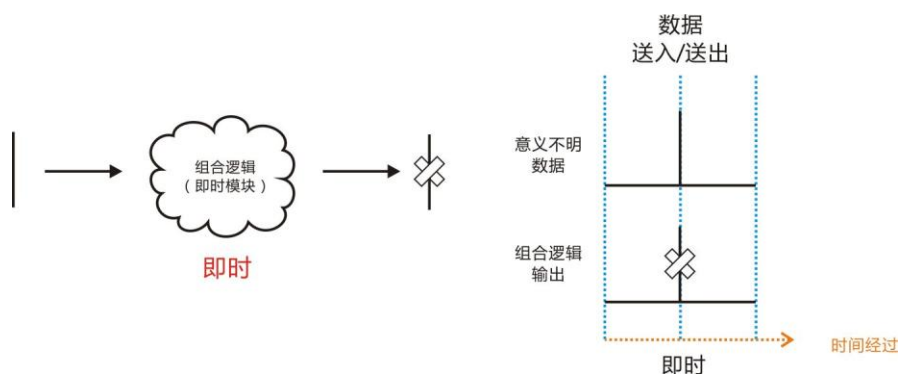


图 3.1.17 瘦骨如柴的数据。

假设数据是一支瘦骨如柴的东西，而不是一块肥嘟嘟的东西，那么即时效果更佳显眼，过程如图 3.1.17 所示。注意图 3.1.117 右图，此刻时间经过在也没有意义，因为数据加工好比变魔术般瞬间完成。这个道理也告诉我们，即时结果在物理时序当中是**无视时钟**，**无视时间**的犯规存在，读者是不是觉得很神奇呢？因此称为即时事件。

即时事件一般的时序行为描述如下：

```
rData = Temp + 1; //组合逻辑的加法器
rData = Temp * 2; //组合逻辑的乘法器
```

有些同学可能会非常担心的问：“即时事件如此无视时钟还有无视物理延迟，真的没有问题吗？”，笔者也没有法子呀，理想时序的即时事件本来就是这么一回事，鬼叫笔者只是探索者而不是创作者，笔者看见什么就写什么而已。笔者稍微强调一下，即时事件无视时间也无视时钟，所以有没有时钟源驱动即时模块还是照样跑照样跳。

举例而言，常见的选择器芯片，都有表 3.1.1 所示的大致功能：

表 3.1.1 常见的选择器功能。

输出 / 输入	D1	D2
Q1	0	0
Q2	0	1
Q3	1	0
Q4	1	1

表 3.1.1 当中表示有两个输入——D1 与 D2，根据两者取值不同输出通道也会经由不同，然而表 3.1.1 也告诉我们一个事实，既是输出会根据输入不同而即时生效，这是即时事件也是选择器最理想的功能状态，所以没有必要考虑数据加工的物理时间。（芯片手册当然会标注实际的时间参数，但是如果没有必要的话，用户一般会无视）

即时事件与时间点事件不同，前者无视时钟后者必须时钟支援，假设即时事件不小心掉入时间点事件当中，它们又会产生怎么样的花火呢？有趣，实在有趣。

## 3.2 时间点事件还有即时事件的时序表现

首先让我们瞧瞧两种相关语法，亦即 `begin ... end` 与 `fork ... join`。

参考书曾经说过 `begin ... end` 是顺序块，然而 `fork ... join` 是并行块，然而类似说法只是语法方面的规定和说明而已。先不管它们是顺序块或者并行块 根据笔者的理解 `begin ... end` 是综合语言的括号功能，类似 c 语言的 `{}`；反之，`fork ... join` 则属于验证语言，不过笔者不太喜欢验证语言，所以暂时无视它。

事实上，**操作是否并行执行还是顺序执行并不是取决语法本身的规定或者说明，而是取自语言的本质**。Verilog 打从一出生就是并行性质，不管我们要不要。换句话说，不用我们去管，**并行操作就是 Verilog 的默认操作** ... 很遗憾的是 Verilog 天生并不支持顺序操作，所以我们才需要建立仿顺序结构，好让 Verilog 支持顺序操作。

相反的道理，C 语言一出生就是顺序（步骤）性质，也就说顺序操作是默认操作，并行操作确实 C 语言一辈子的痛，为此才会出现**任务调度这种机制，好让 C 语言模仿并行操作**。因此，Verilog 语言的“`begin ... end`”等价 C 语言的“`{}`”，这样的理解绝对有理。

*exp01\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp01_simulation();
3.
4.     /*****/ // environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #10 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [3:0]Reg1,Reg2;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            begin
20.                i <= 3'd0;
21.                Reg1 <= 4'd0;
22.                Reg2 <= 4'd0;
```

```

23.         end
24.     else
25.         case( i )
26.
27.             0:
28.                 begin Reg1 <= 4'd2; Reg2 <= 4'd3; i <= i + 1'b1; end
29.
30.             1:
31.                 i <= i;
32.
33.         endcase
34.
35.         /*****/
36. endmodule

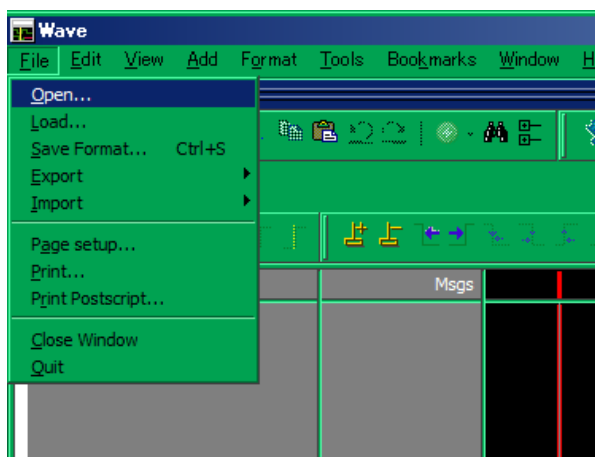
```

exp01\_simulation.vt 是笔者建立的激励文本，话是这么说 ... 不过里边却不包含仿真对象，所以可以称为虚拟建模或者单纯的仿真环境而已。exp01\_simulation.vt 保存在 Experiment02 的目录下，读者再根据自己的喜好用 notepad 软件代开或者其他什么都行。我们暂时不管激励文本的建立过程，在此读者只要明白笔者在第 4~12 行产生时钟信号还有复位信号，它们皆是环境输入。

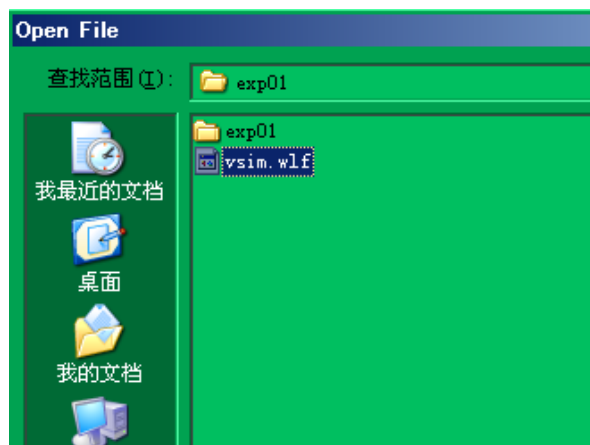
接着，笔者在第 14 行建立寄存器 i，用于指向步骤或者时；第 15 行是寄存器 Reg1 与 Reg2 的声明，用于暂存结果；第 17~23 行是相关的复位操作；第 25~33 行是仿顺序操作用法模板。如果读者无法上述几行代码的语中意思也是理所当然的事情，不打紧，详细内容会在后面的章节补充。

步骤 0 ( 第 27~28 行 )，Reg1 赋值 4'd2 而 Reg2 赋值 4'd3，根据时间点事件的解释，这个时间点 Reg1 会输出 4'd2 的未来值，而 Reg2 会输出 4'd3 的未来值。接下来，我们就要启动 Modelsim 来验证这个事实。

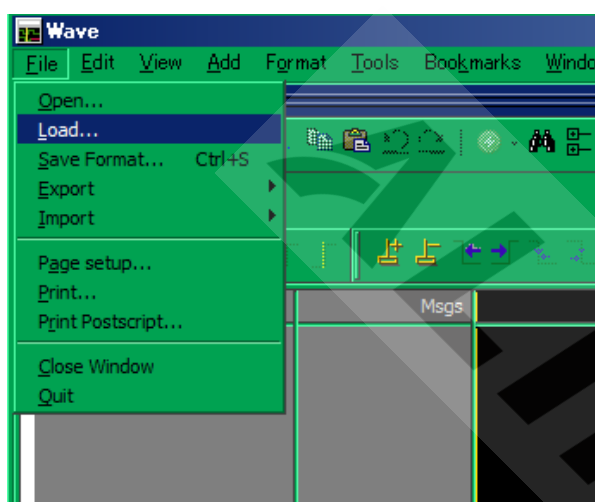
exp01\_simulation.vt 并不是依赖集成环境建立而成的 .vt 文件，所以必须经过手动编译才能启动仿真，不过贴心的笔者已经事先完善了，读者只要独立启动 Modelsim，然后再调出 wave 界面即可。( wave 界面只要沿着 View 菜单选择 wave 选项即可 )。



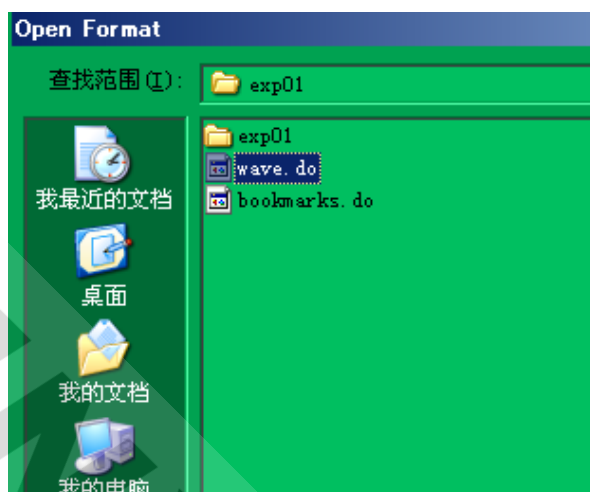
① 沿着 File 菜单选择 Open。



②选择 .wlf 文件并且打开。



③ 沿着 File 菜单选择 Load。



④ 选择 .do 文件并且打开。

图 3.2.1 打开预先建立的 wave 界面。

当我们打开 wave 界面，wave 界面则是什么东西也没有，朋友别慌 ... 这是正常的现象。如图 3.2.1 所示，①沿着 File 菜单选择 Open；②选择 .wlf 文件并且打开，不过这时候 wave 界面还是空空如也，因为方才读者只是读入笔者预先保存的 wave 界面配置文件而已；③在此沿着 File 菜单选择 Load；④选择 .do 文件并且才开，此刻读者已经加载成功笔者预先设置好的 wave 显示状态。

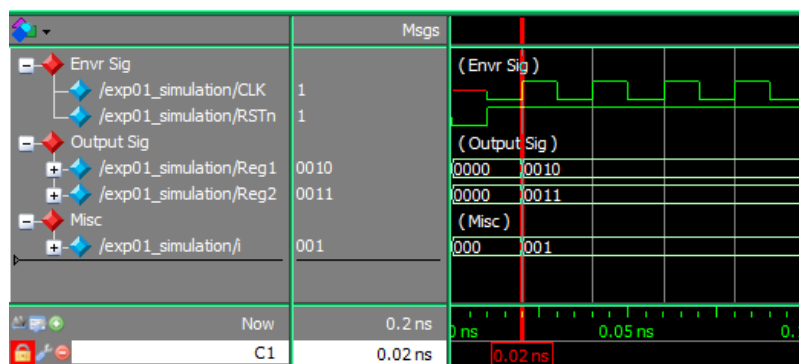


图 3.2.2exp01\_simulation.vt 的仿真结果。

图 3.2.2 是 exp01\_simulation 的仿真结果，其中光标 C1 指向的地方既是步骤 0 或者说时间点 T0。仿真结果完全符合预期的猜测，因为在 T0 的未来，Reg1 和 Reg2 输出未来值 4'b0010 亦即 4'd2，还有 4'b0011，亦即 4'd3。

*exp02\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp02_simulation();
3.
4.     /*****// environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #10 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [3:0]Reg1,Reg2;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            begin
20.                i <= 3'd0;
21.                Reg1 <= 4'd0;
22.                Reg2 <= 4'd0;
23.            end
24.        else
25.            case( i )
26.
27.                0:
28.                    begin Reg1 <= 4'd2; Reg2 <= 4'd3; i <= i + 1'b1; end
29.
30.                1:
31.                    i <= i;
32.
33.            endcase
34.
35.        /*****/
36.    endmodule
```

假设顽皮的笔者故意更改一下 exp01\_simulation.vt 文件的 28 行的 Reg2 <= 4'd3 成为 Reg1 <= 4'd3，然后再仿真看看会发生什么一回事。想必许多同学一定会认为这是语法错误，然后编译无法通过是否？答案是否定的，第 28 行的代码不仅无法没有错误，而且编译也成功通过 ... 感觉是不是很蹊跷呢？

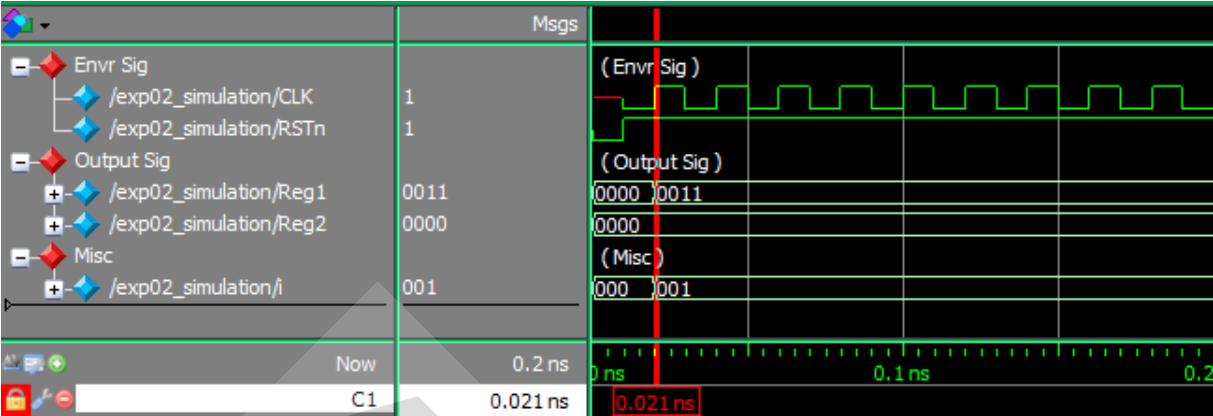


图 3.2.3 exp02\_simulation.vt 的仿真结果。

重复图 3.2.1 的动作，打开笔者预先保存好的文件，显示以后的 wave 界面如图 3.2.3 所示。光标 C1 同样指向步骤 0，也是时间点 T0 ... 此刻，Reg2 因为没有赋值操作，所以没有任何未来值。反之，此刻的 Reg1 并不是输出 4'd2 的未来值，而是 4'd3 的未来值，为什么会这样呢？

```
(一)  begin
(二)      Reg1 <= 4'd2;
(三)      Reg1 <= 4'd3;
(四)      ...
```

有人会认为这是纯粹的语法 Bug，不过笔者却认为这是编译次序的灰色特性。我们知道不管是什么语言的编译器，都是一行一行按着次序编译的代码。根据笔者的理解，编译器会先编译第 2 行 Reg1 <= 4'd2，接着再编译第 3 行的 Reg1 <= 4'd3，随之第 3 行的编译结果会覆盖第 2 行之前的编译结果，因此 Reg1 最终输出未来值 4'd3 而不是 4'd2。

```
exp03_simulation.vt

1. 'timescale 1 ps/ 1 ps
2. module exp03_simulation();
3.
4.     /*****// environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
```

```

9.      RSTn = 0; #10; RSTn = 1;
10.     CLK = 0; forever #10 CLK = ~CLK;
11. end
12. /*****/
13.
14.     reg [2:0]i;
15.     reg [3:0]Reg1,Reg2;
16.
17.     always @ ( posedge CLK or negedge RSTn )
18.         if( !RSTn )
19.             begin
20.                 i <= 3'd0;
21.                 Reg1 <= 4'd0;
22.                 Reg2 <= 4'd0;
23.             end
24.         else
25.             case( i )
26.
27.                 0:
28.                     begin Reg1 = 4'd2; Reg2 <= 4'd3; i <= i + 1'b1; end
29.
30.                 1:
31.                     begin Reg1 = 4'd4; Reg2 <= 4'd6; i <= i + 1'b1; end
32.
33.                 2:
34.                     i <= i;
35.
36.             endcase
37.
38.         /*****/
39. endmodule

```

笔者之前曾经说过，即时事件是无视时钟的操作，如果即时事件不小心被时钟源波及会是什么样的结果呢？步骤 0（第 27~28 行）Reg1 用 = 操作符赋值 4'd2，Reg2 则用 <= 操作符赋值 4'd3；步骤 1（第 30~31 行）Reg1 用 = 4'd4，Reg2 则用 <= 赋值 4'd6。



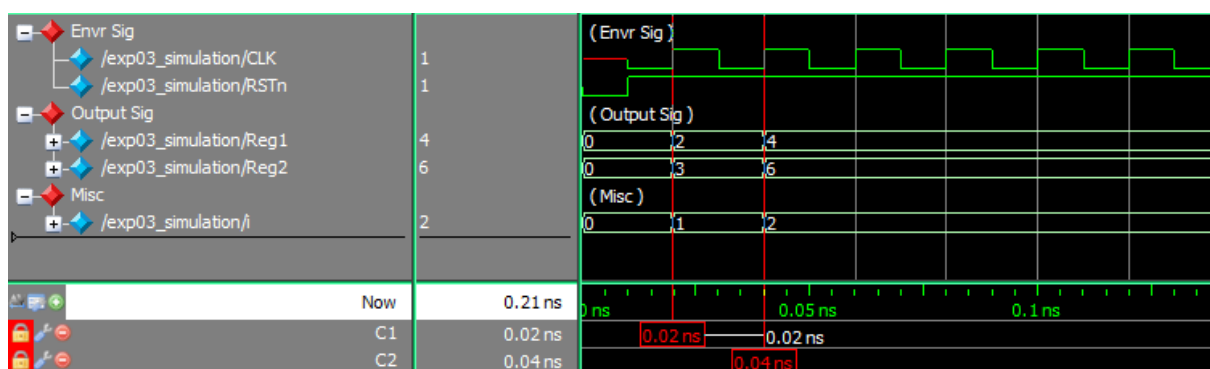


图 3.2.4 exp03\_simulation 仿真结果。

重复图 3.2.1 的方法，打开原先保存好的 wave 界面显示状态。图 3.2.4 是 exp03\_simulation.vt 的仿真结果，图中光标 C1 指向时钟点 T0，然而光标 C2 指向时间点 C2。根据代码第 27~31 行所示，Reg1 与 Reg2 没有直接的关系，T0 的时候 Reg1 用 = 操作符赋值 4'd2，而 Reg2 用 <= 操作符赋值 4'd3，所以 Reg1 输出即时值 4'd2，Reg2 则输出未来值 4'd3。在 T1 的时候，Reg1 用 = 操作符赋值 4'd4，而 Reg2 用 <= 操作符赋值 4'd6，结果 Reg1 输出即时值 4'd4，Reg2 则输出未来值 4'd6。

好奇的同学可能会问：“怎么两个赋值操作符的结果都是一样？”的确，表面上却是这么一回事，不过细微来讲的话，就算即时结果再怎么无视时钟，但是 Reg1 的输入源被时间点（时间沿）掌控着，所以即时结果不得不屈服。

```

(一)  'timescale 1 ps/ 1 ps
(二)  module exp04_simulation();
(三)
(四)      /*****/ // environment signal
(五)      reg CLK, RSTn;
(六)
(七)      initial
(八)      begin
(九)          RSTn = 0; #10; RSTn = 1;
(十)          CLK = 0; forever #10 CLK = ~CLK;
(十一)     end
(十二)     /*****/
(十三)
(十四)     reg [2:0]i;
(十五)     reg [3:0]Reg1,Reg2;
(十六)
(十七)     always @ ( posedge CLK or negedge RSTn )
(十八)         if( !RSTn )
(十九)             begin
(二十)                 i <= 3'd0;
(二十一)                 Reg1 <= 4'd0;

```

```

(二十二)          Reg2 <= 4'd0;
(二十三)          end
(二十四)      else
(二十五)          case( i )
(二十六)
(二十七)              0:
(二十八)                  begin Reg1 = 4'd2; Reg2 <= Reg1; i <= i + 1'b1; end
(二十九)
(三十)              1:
(三十一)                  i <= i;
(三十二)
(三十三)          endcase
(三十四)
(三十五)      /*****/
(三十六)  endmodule

```

为了体现即时事件的能力，exp04\_experiment.vt 的步骤 0（第 27~28 行）笔者将 Reg1 与 Reg2 建立关系，Reg1 先用 = 操作符赋值 4'd2，然后 Reg2 则用 <= 赋予 Reg1 的过去值。终究 Reg2 是否会读取 Reg1 的过去值还是读取 Reg1 的即时值，答案在仿真结果当中。

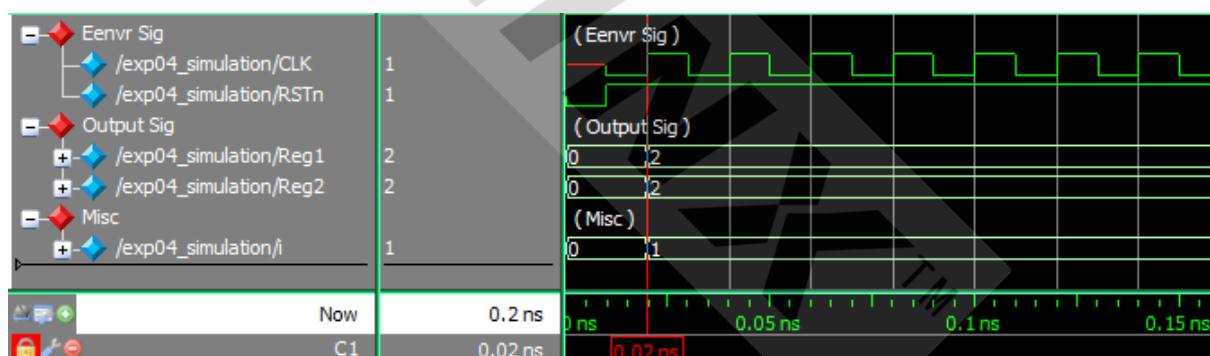


图 3.2.5 exp04\_simulation 仿真结果。

重复图 3.2.1 的方法，打开 exp04\_simulation 原先保存好的 wave 界面显示状态。如图 3.2.5 所示，光标 C1 指向时间点 T0，此刻 Reg1 与 Reg2 建立关系，根据时间点事件的作用，Reg2 应该读取此刻 Reg1 的过去值，亦即 0 才是，然后输出未来值 0。这个事实告诉我们，Reg2 没有读取 Reg1 的过去值 0，而是读取 Reg1 的未来值 2，结果 Reg2 才会输出未来值 2。

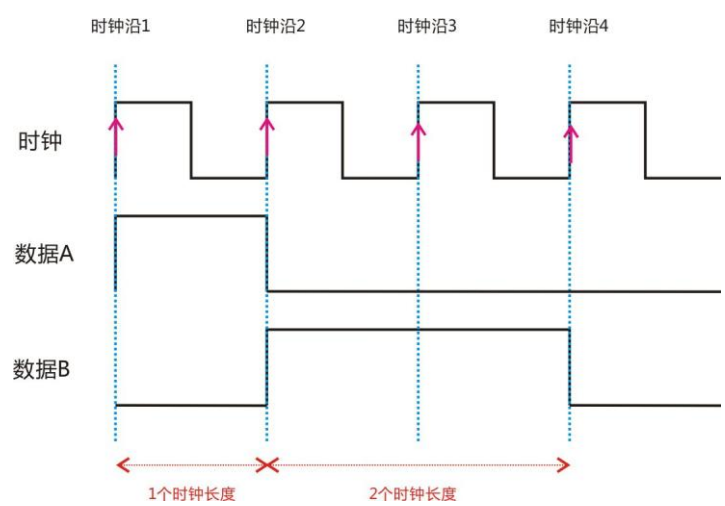


图 3.2.6 时钟沿，时钟块，数据块。

整合技巧有这样一个想法，为了实现紧密控时，时钟还有数据必须拥有“**具体单位**”。一般参考书常说 N 个时钟是指时钟沿，然而这种概念是非常抽象和空虚的。为此，时钟块，还有数据块的概念就诞生，所谓**时钟块是指两个时钟沿的长度**，举例第一个时钟块，以时钟沿 1 算起，长度介于时钟沿 1~2 之间；第二个时钟块，以时钟沿 2 算起，长度介于时钟沿 2~3 之间，其它以此类推，过程如图 3.2.6 所示。

**所谓数据块是指时钟沿划分的数据长度**，如图 3.2.6 所示，数据 A 有一块数据在于时钟沿 1~2 之间，长度是 1 个时钟（时钟块）；数据 B 也有一块时钟位于时钟沿 2~4 之间，长度则是 2 个时钟（时钟块）。（注意：块也可以看成周期）

上述内容告诉我们一个事实，**数据的长度是由时钟来决定**。笔者曾经说过即时事件无视时钟，就算即时结果**没有数据长度也没有关系**，或者说数据长度应该越短越好，然而理想状态当然是 0 个时间。

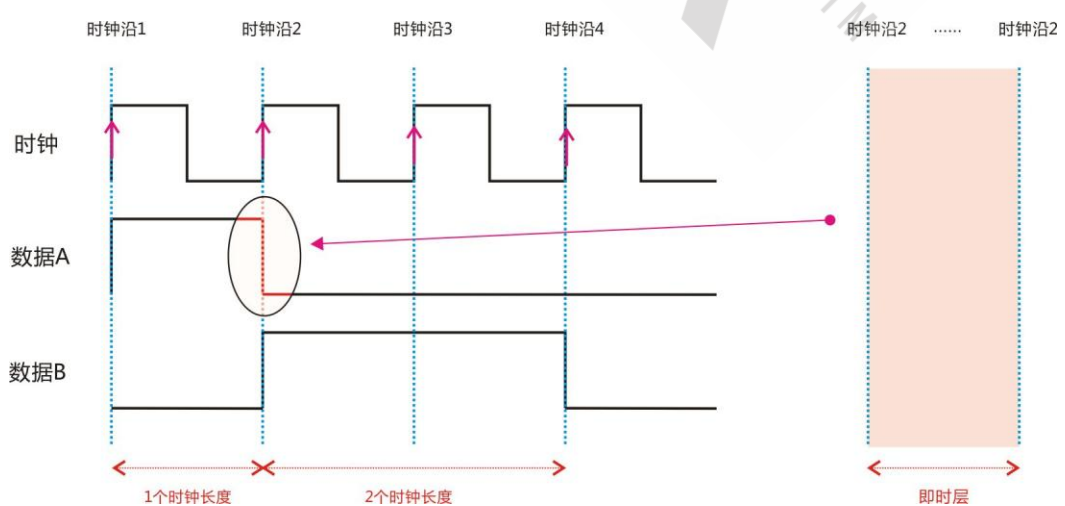


图 3.2.7 即时层的概念。

即时结果不是没有显示在理想时序图当中，而是**即时结果是在太窄**，窄到肉眼看不见。即时结果往往发生在时钟的细缝当中，如果**使用未来科技将它放大几万倍**，假设放大对

象是时钟沿 2，结果我们可以看到位于时钟沿 2~2 之间有一个称为即时层的存在，而且即使结果就在哪里产生，结果如图 3.2.7 所示。

即时层也有其它命名，如：时钟无效空间或者时间停止空间等非常科幻的取名。根据笔者的妄想，即时层可以发生无穷无尽的即时事件，也可以添加没有上限的小即时层 ... 笔者承认自己是歪歪看多了所以才会脱离现实。

如图 3.2.5 所示，光标 C1 指向的时钟沿当中，Reg1 在即时层当中被赋予即时值 2，即使结果无视时钟，所以即时生效，然后 Reg2 又被赋予 Reg1 的即时值 2，最后 Reg2 输出未来值 2。

*exp05\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp05_simulation();
3.
4.     /*****// environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #10 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [3:0]Reg1,Reg2;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            begin
20.                i <= 3'd0;
21.                Reg1 <= 4'd0;
22.                Reg2 <= 4'd0;
23.            end
24.        else
25.            case( i )
26.
27.                0:
28.                    begin Reg2 <= Reg1; Reg1 = 4'd2; i <= i + 1'b1; end
29.
30.                1:
```

```

31.             i <= i;
32.
33.             endcase
34.
35.             /*****/
36. endmodule

```

exp05\_simulation.vt 与 exp04\_simulation.vt 相比，语法方面虽然没有改变，但是在第 28 行当中 Reg1 = 4'd2 与 Reg2 <= Reg1 的执行次序被笔者修改过。exp04\_simulation.vt 是先执行 Reg1 = 4'd2 然后再执行 Reg2 <= Reg1；换之，exp05\_simulation.vt 则是先执行 Reg2 <= Reg1，接着再执行 Reg1 = 4'd2。

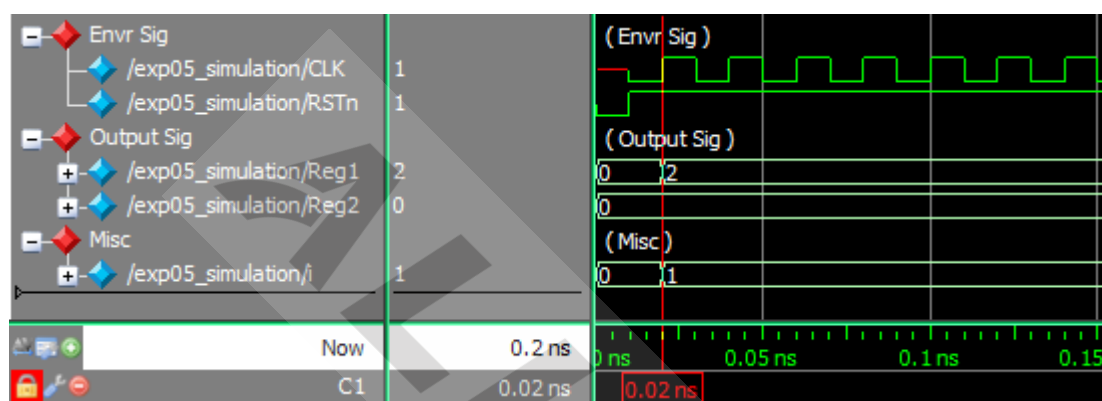


图 3.2.8 exp05\_simulation 仿真结果。

重复图 3.2.1 的方法，打开 exp05\_simulation 原先保存好的 wave 界面显示状态。如图 3.2.8 所示，笔者只是更改执行次序而已，然而仿真结果却不一样。光标 C1 指向时间点 T0，此刻 Reg2 读取 Reg1 的过去值 0，而 Reg1 则赋予即时值 2，结果 Reg2 输出未来值 0，而 Reg1 输出即时值 2。

同学可能会觉得奇怪，当更换执行次序而已，输出结果却不一样，为何呢？其实这是编译器的灰色特性，编译器在编译的时候会数顺序执行，Reg2 <= Reg1 先被编译，然而该被操作视为时间点事件。在 T0 的时候，此刻 Reg1 的过去值是 0，所以 Reg2 读取到 0 值。换之，Reg1 = 4'd2 只是重复 exp03\_simulation 的现象而已。

#### exp06\_simulation.vt

```

1. 'timescale 1 ps/ 1 ps
2. module exp06_simulation();
3.
4.     /*****/ // environment signal
5.     reg CLK, RSTn;
6.
7.     initial

```

```

8.      begin
9.          RSTn = 0; #10; RSTn = 1;
10.         CLK = 0; forever #10 CLK = ~CLK;
11.     end
12.     /*****/
13.
14.     reg [2:0]i;
15.     reg [3:0]Reg1,Reg2,Reg3;
16.
17.     always @ ( posedge CLK or negedge RSTn )
18.         if( !RSTn )
19.             begin
20.                 i <= 3'd0;
21.                 Reg1 <= 4'd0;
22.                 Reg2 <= 4'd0;
23.                 Reg3 <= 4'd0;
24.             end
25.         else
26.             case( i )
27.
28.                 0:
29.                     begin Reg1 = 4'd2; Reg2 = Reg1 + 4'd2; Reg3 <= Reg2; i <= i + 1'b1; end
30.
31.                 1:
32.                     i <= i;
33.
34.             endcase
35.
36.         /*****/
37.     endmodule

```

exp06\_simulation.vt 是在 exp04\_simulation.vt 的基础上再添加一行即时操作，如代码行 29 所示，笔者新添另一个寄存器 Reg3，然而 Reg1 赋予即时值 4'd2，Reg 被赋予的即时值是由 Reg1 的即时结果再加上 2，Reg3 则赋予 Reg2 的即时值。

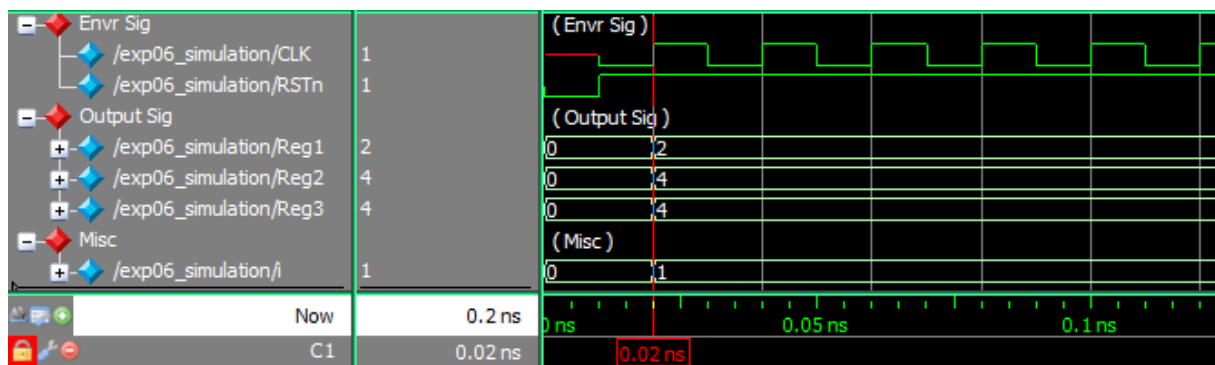


图 3.2.9 exp06\_simulation 的仿真结果。

重复图 3.2.1 的方法, 打开 exp06\_simulation 原先保存好的 wave 界面显示状态。如图 3.2.9 所示, 光标 C1 指向时间点 T0, 此刻 Reg1 被赋予即时值 2, 而 Reg2 同样也是赋予即时值, 但是赋值对象是 Reg1 的即时结果再加上 2, 至于 Reg3 则赋予 Reg2 的即时结果。结果再 T0 的未来, Reg1 输出即时值 2, Reg2 输出即时值 4, Reg3 则是输出即时值 4。

为什么会发生这样的事情呢? 其实这是, 编译器灰色特性惹的祸, 根据编译器的编译次序:

4.  $\text{Reg1} = 4'd2$ ; 视为即时事件。
5.  $\text{Reg2} = \text{Reg1} + 4'd2$ ; 视为即时事件, 期间 Reg1 的即时结果则是上一层的即时值 2。
6.  $\text{Reg3} \leq \text{Reg2}$ ; 虽然视为时间点事件, 但是赋值源则是 Reg2 的即时结果。

以上现象告诉我们一个事实, 亦即“**即时层可以插入无限即时事件**”。因为在时间点 T0 之上已经发生两起即时事件, 即  $\text{Reg1} = 4'd2$  和  $\text{Reg2} = \text{Reg1} + 4'd2$ , 而且  $\text{Reg2} = \text{Reg1} + 4'd2$  当中的 Reg1 也是即时生效的即时值。最后 Reg3 再赋予 Reg2 的即时结果 3, 而不是 Reg2 的过去值。

exp07\_simulation.vt

```
1. 'timescale 1 ps/ 1 ps
2. module exp07_simulation();
3.
4.     /*****/ // environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #10 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [3:0]Reg1,Reg2,Reg3;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            begin
20.                i <= 3'd0;
21.                Reg1 <= 4'd9;
```



```

22.             Reg2 <= 4'd0;
23.             Reg3 <= 4'd0;
24.         end
25.     else
26.         case( i )
27.
28.             0:
29.                 begin Reg2 <= Reg1; Reg1 = 4'd2; Reg3 <= Reg1; i <= i + 1'b1; end
30.
31.             1:
32.                 i <= i;
33.
34.         endcase
35.
36.         /*****/
37. endmodule

```

最后让我们再看一段较为匪夷所思的现象，exp07\_simulation 的第 21 行 Reg1 的复位值为 9，然而根据第 29 行，先是执行 Reg2 <= Reg1，然后 Reg1 赋予即时结果 2，最后再执行 Reg3 <= Reg1。读者是不是开始觉得有点头晕了？不打紧，我们只要根据编译次序还有时间点事件与即时事件之间的关系作出推断，一切结果都是符合逻辑的。

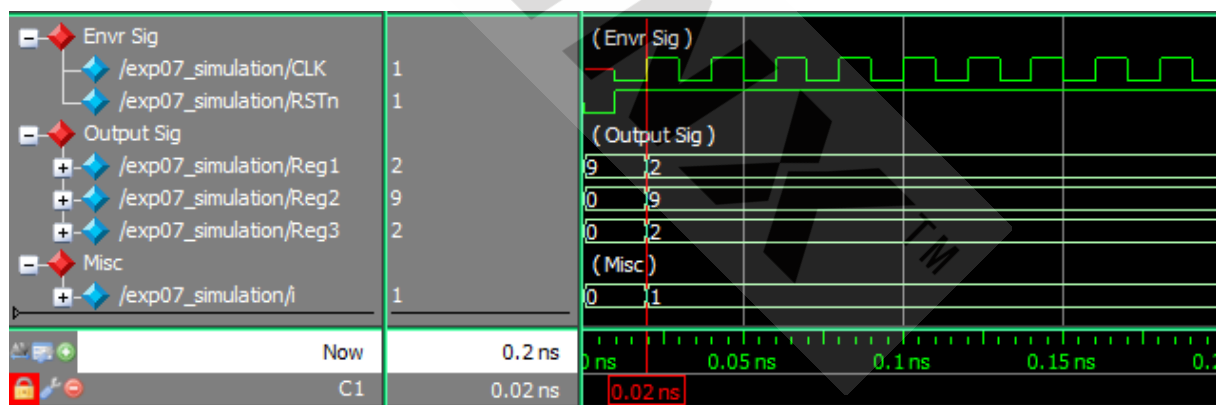


图 3.2.10 exp07\_simulation 的仿真结果。

重复图 3.2.1 的方法，打开 exp07\_simulation 原先保存好的 wave 界面显示状态。如图 3.2.10 所示，光标 C1 指向时间点 T0，此刻 Reg1 的过去值（复位值）是 9，而且 Reg2 先是读取 Reg1 的过去值 9，然后输出未来值 9。不一会儿，此刻即时事件突然发生了，Reg1 被赋予即时值 2，结果 Reg1 输出即时值 2；此外，此刻 Reg3 再也不是赋予 Reg1 的过去值 9 而是即时值 2，因此 Reg3 也输出即时值 2。

为了理清思路，我们可以理解这一切都是编译器惹的祸：

1. Reg2 <= Reg1；视为时间点事件，Reg2 赋予 Reg1 的过去值。
2. Reg1 = 4'd2；视为即时事件，Reg1 赋予即时值。

3.  $\text{Reg3} \leq \text{Reg1}$  ; 虽然也是时间点事件，但是赋予结果则是  $\text{Reg1}$  的即时值而不是过去值。

看完这么多有关时间点事件还有即时事件在理想时序上产生的现象，读者很可能会疑惑道：“知道这些秘密我们终究又能得到什么好处？”当然好处可多得是。我们知道 HDL 拥有并行的本质，亦即单个时钟可以执行无数个时间点事件。不过根据笔者的妄想，即时事件会产生即时层，然而即时层可以插入无数次即时事件。这个事实告诉我们，时间点事件和即时事件虽然都是并行操作，但是前者需要时钟后者则不需要时钟。

举例而言，假设笔者为了求出 D 的结果，然而 D 求得的过程必须经过以下 4 个步骤：

```
A = 1;
B = A + 1;
C = B + 1;
D = C + 1;
```

如果笔者用时间点事件求得 D，笔者至少需要消耗 4 个时钟，而过程发生如下：

```
T0 执行 A <= 1;
T1 执行 B <= A + 1;
T2 执行 C <= B + 1;
T3 执行 D <= C + 1;
```

换之，如果笔者使用即时事件求得 D，笔者仅需要消耗一个时钟而已，过程发生如下：

```
T0 执行 A = 1; B = A + 1; C = B + 1; D <= C + 1;
```

这个事实告诉我们，原本求得 D 必须消耗 4 个时钟，不过即时事件却将 D 的求得压缩至 1 个时钟而已。因此我们可以断定，即时事件拥有“偷时钟”的作用。读者千万别小看偷时钟的作用，偷时钟可以减少时钟消耗至于，例如一些时序非常紧张的设计也有可能多一个时钟也不行，因此即时事件就会派上用场。

当然，即时事件的好处也不仅局限于偷时钟而已，即时事件可以实现真正意义上的精密控时，此外即时事件在仿真当中也有很大的用处。然而，使用即时事件的真正目的是建立即时模块。读者目前很可能暂时无法理解笔者所说的这番话，但是读者必须记住，理想时序除了时间点事件存在以外，也有同等身价的即时事件存在。

区分它们不仅仅只是  $\leq$  阻塞操作符与  $=$  非阻塞操作符之间的识别而已，我们还要配合编译器的灰区使用才能实现它们的意义。为了帮助读者建立大概的概念，读者可以这样理解：先执行即时事件，再执行时间点事件，HDL 描述如下：

```
begin
    Reg2 = Reg1;    // 即时事件
```

```
Reg3 <= Reg2;  // 时间点事件
end
```

所以，千万不要搞错即时事件还有时间点事件的执行顺序噢！

=====

刮着刮着，整间地牢已经刻满笔者的想法，笔者不禁感叹道，理想时序既然存在如此宝贵，如此重要的信息，为什么传统流派难视若无睹？是真瞎还是假傻，鬼才知道。不管怎么样，笔者不禁怀疑，传统流派所说的一切是否属实或者纯粹坑人？笔者想做点什么改善一下现状，可是传统流派人多势众笔者只是寡人一名，寡不敌众，而且有谁又会相信笔者的话语呢？



### 3.3 指向时钟的 i

西元二零一三年的今天，我们到处都可以看到 i 叉叉，如著名的 iPhone，黑金的 iBoard 等等，i 叉叉真是无奇不有，于是乎笔者也顺应流行，创建 i——仿顺序操作。什么是仿顺序操作？仿顺序操作是低级建模当中非常重要的一环，本质上 Verilog 是没有结构去支持顺序操作，因此仿顺序操作才会诞生。仿顺序操作最基本的认识就是使用 Verilog 创建可以支持顺序的操作结构。

其中 i 就是仿顺序操作的标志，为什么是 i 而不是 j 或者 k 呢？不知道，只是直觉性这么选择而已。i 有许多功能，其中一项功能就是指向步骤，如代码 3.1.1 所示：

```
1. always @ ( posedge CLOCK ... )
2.     ....
3.     case( i )
4.
5.         0:
6.             begin reg1 <= reg2; i <= i + 1'b1; end
7.
8.         1:
9.             begin ... i <= i + 1'b1; end
10.
11.        ...
12.
13.        9 :
14.            begin ... i <= 4'd0; end
15.
16.    endcase
```

代码 3.3.1

代码 3.3.1 很简单，always 底下声明 case ... endcase，然后括号中就是 i。默认起始步骤是 0，当步骤完成以后 i 就会递增以示下一个步骤，其他如此类推，直到所有步骤操作完毕就返回步骤 0。换之仿真，i 除了指向步骤以外还有指向时钟的功能，如代码 3.3.2 所示：

```
1. always @ ( posedge CLOCK ... )
2.     ....
3.     case( i )
4.
5.         0:
6.             begin Reg1 <= 4'b1; i <= i + 1'b1; end
7.
8.         1:
```

```

9.      begin Reg2 <= 4'b2; i <= i + 1'b1; end
10.
11.     2:
12.     begin Reg3 <= 4'b3; i <= i + 1'b1; end
13.     ...

```

代码 3.3.2

代码 3.3.2 的操作非常简单，从步骤的角度上我们可以这样解读，即 Reg1 在步骤 0 赋值 4'b1，Reg2 在步骤 1 赋值 4'b2，Reg3 在步骤赋值 4'b3。换之理想时序则可以这样解读，即 Reg1 在时间点 T0 决定赋值 4'b1，Reg2 在时间点 T1 决定赋值 4'b2，Reg3 在时间点 3 决定赋值 4'b3。

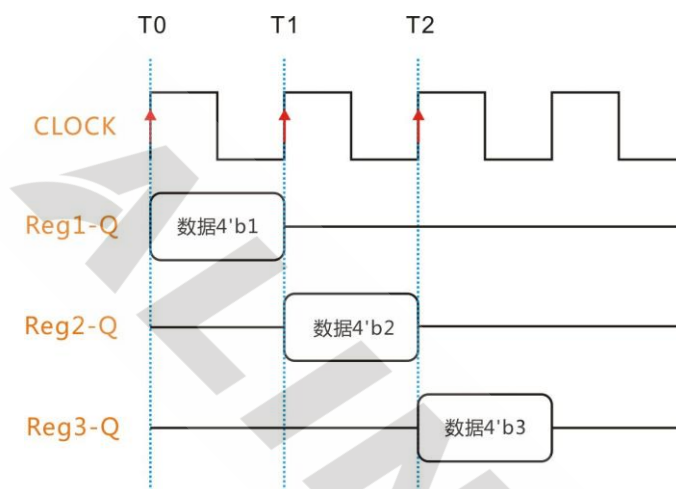


图 3.3.1 代码 3.3.2 对应的理想时序图。

代码 3.3.2 根据理想时序的解读结果，最终会产生图 3.3.1 所对应的理想时序图。如图 3.3.1 所示，在 T0 的时候，由于 Reg1 决定赋值 4'b1，结果 Reg1 输出未来值 4'b1；在 T1 的时候，Reg2 决定赋值 4'b2，结果输出未来值 4'b2；在 T2 的时候，Reg3 决定赋值 4'b3，结果输出 4'b3 的未来值。

代码 3.3.2 只要换个方式解读就会产生不同的结果，读者是不是觉得很神奇呢？接着，再让我们看一段代码 3.3.3：

```

1.  always @ ( posedge CLOCK ... )
2.      .....
3.      case( i )
4.
5.          0:
6.          begin Reg1 <= 4'b2; i <= i + 1'b1; end;
7.
8.          1:
9.          begin Reg2 <= Reg1; i <= i + 1'b1; end;

```

代码 3.3.3 常规说道，即步骤 0 Reg1 赋值 4'b2，步骤 1 Reg2 赋值 Reg1 的结果。如果换做理想时序解读代码 3.3.3，亦即在时间点 T0 的时候，Reg1 决定赋值 4'b2，结果 Reg1 输出未来值 4'b2；在时间点 T1 的时候，Reg2 决定读取 Reg1 的过去值，结果 Reg2 赋值 4'b2，然后 Reg2 输出 4'b2 的未来值。

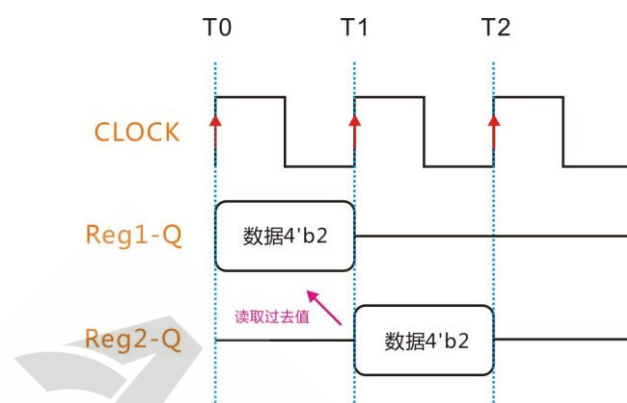


图 3.3.2 代码 3.3.3 对应的理想时序图。

图 3.3.2 是代码 3.3.3 用理想时序去解读所对应的时序图。如图所示，在 T0 的时候，Reg1 输出 4'b2 的未来值；在 T1 的时候，Reg2 决定读取 Reg1 的过去值，结果 Reg2 读取 4'b2 并且输出未来值 4'b2。我们知道理想时序并不存在 1ns~7ns 等物理延迟，再加上代码 3.3.2 或者 3.3.3 一个步骤只有一个操作而已，结果步骤与步骤之间敲好是一个时钟块，两个时间沿。

```
always @ ( posedge CLK )
.....
case( i )

0:
begin Reg1 <= 4'b2; i <= I + 1'b1; end

1:
begin Reg2 <= Reg1 ; i <= I + 1'b1; end

endcase
```

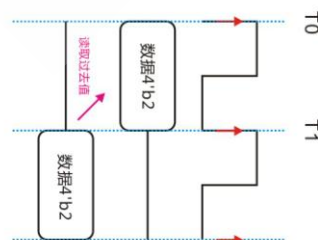


图 3.3.3 i 指向步骤又指向时钟。

如图 3.3.3 所示，左边当中的步骤 0 与步骤 1 只停留一个时钟，因此我们只要将脑袋向时钟转 90°角，我们就会看见效果。笔者需要强调一下，[右图是脑海想象的理想时序图](#)，而不是仿真结果。“为什么会那么神奇？”，读者可能会这样问及 ... 这就是理想时序融合 i 以后所有的效果，[模块会提升一倍的表达能力](#)，这也是主动设计最基本的基础——[清晰的代码](#)。

C 语言一般都是代码越精简，代码就会越清晰，因为人类的左脑是线性处理，减少代码就好比缩短操作线程一般。换之 Verilog，曾经何时笔者也是如此认为，不过笔者却为自己的天真付出惨痛的带价。事实上，Verilog 并不是代码越精简就会越清晰，即使我们将整体代码精简到 1 到 2 成，代码不仅不会清晰度，反之还有可能弄巧成拙，为什么呢？那是因为 Verilog 是并行语言。

二十一世纪的今天 科技高度发展人类的文明也随之飞速成长 然而悲剧多过喜剧 ... 智能科技没有节制发展造就人们贪图便利，大步倾向左脑的使用，什么都要快！什么都要傻瓜！什么都要轻松！真是不知所谓的效率标准，尤其搞电子的人们，问题更是严重。许多人之所以那么容易适应顺序语言，那是因为环境影响再加上后天性的用脑习惯。

**顺序语言有线性的本质**，这种感觉宛如**游走在单向通道里**，通道的距离好比操作线程的劳动程度，亦即代码越精简通道就越短，通道越短操作线程就越快结束。换句话说，距离越短，整体印象更加容易放入左脑。此外，我们只要遵循通道的流向，亦即代码行段（线性地址），如果出现障碍物妨碍前进，我们就要设法移除障碍即可。

相反的，Verilog 恰恰好是相反的东西，Verilog 有并行的本质，这种感觉宛如**游走多项通道的迷宫**，又或者说 Verilog 是由无数单向通道组成的大迷宫。我们知道在迷宫直走就会迷路，左脑最大程度只能处理相接的单向通道而已，但是这样作对走出迷宫只有很小的帮助而已，在此我们就要指望右脑了。

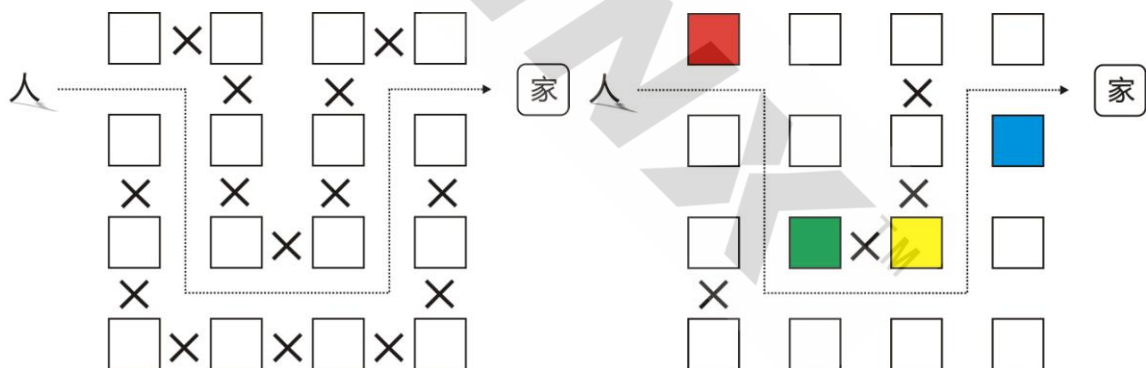


图 3.3.4 笔者要回家——单向道。

假设小小的笔者要回家，如图 3.3.4 所示，叉叉是妨碍物，小格子是建筑物，左图是一段单向到路程，没有什么好解释，笔者只要**一直向前冲**笔者就能回家。换之右图是多向道的回家路，小时候笔者必须**借助特定的建筑物**才能平安回家，如图 3.3.4 的右图所示，首先经过红色建筑物以后右转；经过绿色建筑物以后左转；经过黄色建筑物以后左转；经过蓝色建筑物以后右转，笔者就能回家。

在此，**充满颜色的建筑物使笔者认识回家路途更加清晰**，当然其中还有左转右转等小动作，但是大体上有颜色的建筑物就是回家路的关键。并行性质好比图 3.3.4 右图的回家路，线性行为有可能回永远回不了家，因为这种  $4 \times 4$  布局会产生  $4! (4 \times 3 \times 2 \times 1) 24$  个可能性的回家路。当然我们可以使用排除法寻找回家路，如果是布局是更加复杂或者有



更多可能性的话，使用排除法无疑是自杀的行为因为人的经历是有限的，更何况是小时候的笔者？

笔者需要稍作一下补充，**记忆有色建筑物，笔者回家绝非 100% 成功**，然而有色建筑物确实**提升清晰度将迷路降低**（或者说缩变数/可能性），好让回家更加有效和准确。假设读者是一位顺序（线性），惯用左脑的小孩，读者就会尝试每一条回家路直到成功而已，如果运气好读者就能早点回家，反之运气稍差的话就会成为报纸的头条新闻（失踪）。

继续话题，如果 **i 有能力指向步骤，而且又能指向时钟的话，代码的清晰度无疑会大大增加**，即使不用实现仿真，我们只要稍微想象一下，时序图就会浮现在我们的脑海中。不过，前者所诉都是单个步骤停留一个时钟作为前提，如果单个步骤停留超过 1 个时钟的话，i 是否又能指向时钟呢？答案是肯定的，在此我们有几种选择：

```
1.  always @ ( posedge CLOCK )
2.      ...
3.      case( i )
4.
5.          0,1,2,3:
6.              begin Reg1 <= 4'b2; i <= i + 1'b1; end
7.
8.          ....
9.
10.     endcase
```

代码 3.3.4

代码 3.3.4 表示，Reg1 赋值 4'b2 等操作足够执行用足 4 个步骤，如果使用理想时序来解读的话，就是时间点 T0~T3 Reg1 决定赋值 4'b2。

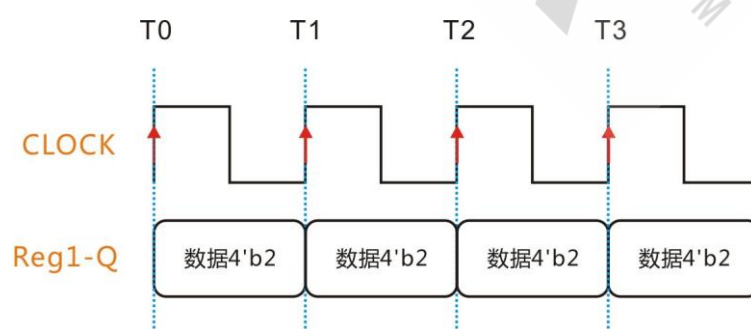


图 3.3.5 代码 3.3.4 对应的理想时序图时序图。

图 3.3.5 是对应代码 3.3.4 的理想时序图。如图所示，时间点 T0 Reg1 输出未来值 4'b2；时间点 T1 Reg1 输出未来值 4'b2；时间点 T2 Reg1 输出未来值 4'b2；时间点 T3 Reg1 输出未来值 4'b2。这是一种直接又简单的多时钟指向方法，即时步骤停留多少时钟，i 的数值也跟着增加多少，如代码 3.3.4 所示，同样 Reg1 <= 4'b2 操作共占据步骤 1，2，3，4。这种方法虽然好用，但是仅适合小数量的多时钟指向而已，如果我们遇见大数量

的多时钟指向，假设 100 个时钟，同样操作既不是要占据步骤 1 致 100，如代码 3.3.5 所示，如果当中有误，我们既不是一牵动全山吗？为此我们需要另一个多时钟指向技巧。

```
1. case( i )
2.
3.     0 ..... 100:
4.         begin Reg1 <= 4'b2; i <= i + 1'b1; end
5.
6. endcase
```

代码 3.3.5

```
(一)  always @ ( posedge CLOCK )
(二)      ...
(三)      case( i )
(四)
(五)          0,1:
(六)          begin
(七)              Reg1 <= 4'b2;
(八)
(九)              if( C1 == 4 - 1 ) begin C1 <= 4'd0; i <= i + 1'b1; end
(十)              else C1 <= C1 + 1'b1;
(十一)          end
(十二)
(十三)      endcase
```

代码 3.3.6

代码 3.3.6 是整合技巧的一项功能，其中 `if( C1 == 4-1 )` 控制步骤时钟逗留的数量，而 `Reg1 <= 4'b2` 既是该步骤的操作，如果用理想时序解读的话，我们则可以这样表达：在 T0 的时候，操作 `Reg1 <= 4'b2` 逗留 4 个时钟；在 T1 的时候，操作 `Reg1 <= 4'b2` 又逗留 4 个时钟。

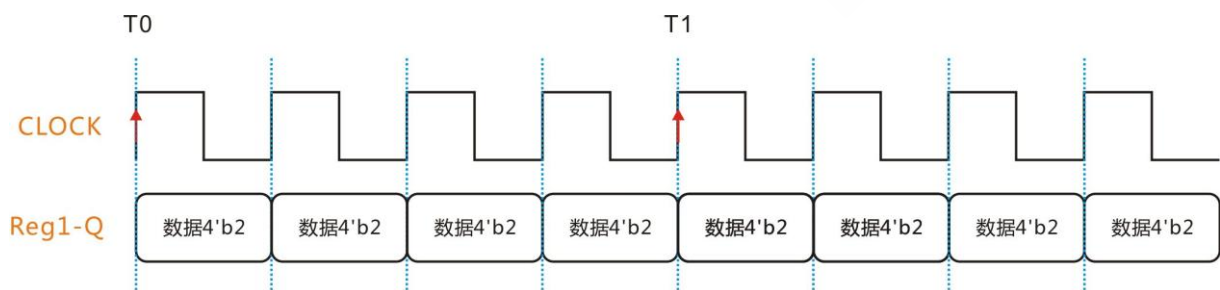


图 3.3.6 代码 3.3.6 对应的时序图。

如图 3.3.6 多事是对应代码 3.3.6 的理想时序图，时间点 T0 决定 Reg1 在步骤 0 执行 4 个时钟的 `Reg1 <= 4'b2` 赋值操作，于是 T0 接续 4 个时钟都会输出未来值 4'b2；时间点 T1 决定 Reg1 在步骤 1 执行 4 个时钟的 `Reg1 <= 4'b2` 赋值操作，于是 T1 接续 4 个时钟

都输出未来值 4'b2。

笔者曾在扫盲文说过，[仿真不推荐执行对象拥有多时钟](#)，亦即“超烦模块”。既然不仿真，为什么还要创建针对多时钟指向的整合技巧呢？整合技巧本来就是为了避免过度依赖仿真才会被创建，即仅凭代码表达就能脑补理想时序图。归根究底，不管是哪一种方法，目的也是为使 i 指向时钟，好让时钟有个标志可以记录和追踪，最终提升代码的表达能力。

说了那么多，不管 i 再怎么厉害，既然 i 身在时序它就要遵守时序的表现规则，然而 i 究竟如何在理想时序上指向时钟，这是非常有学习的价值。

*exp09\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp08_simulation();
3.
4.     /*****// environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #5 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [3:0]Reg1;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            begin
20.                i <= 3'd0;
21.                Reg1 <= 4'd0;
22.            end
23.        else
24.            case( i )
25.
26.                0:
27.                    begin Reg1 <= 4'd1; i <= i + 1'b1; end
28.
29.                1:
30.                    begin Reg1 <= 4'd2; i <= i + 1'b1; end
```

```

31.
32.             2:
33.             begin Reg1 <= 4'd3; i <= i + 1'b1; end
34.
35.             3:
36.             begin Reg1 <= 4'd4; i <= i + 1'b1; end
37.
38.             4:
39.             i <= i;
40.
41.         endcase
42.
43.         /*****/
44. endmodule

```

exp08\_simulation 是一段非常简单的代码，我们先不管部分验证语言，Reg1 先在步骤 0 在执行赋值 4'b1；步骤 1 赋值 4'b2；步骤 2 赋值 4'b3；步骤就 3 赋值 4'b4；步骤 4 停止操作。如果用理想时序解读，即 Reg1 在时间点 T0 赋值 4'b1；时间点 T1 赋值 4'b2；时间点 T2 赋值 4'b3；时间点 T3 赋值 4'b4；

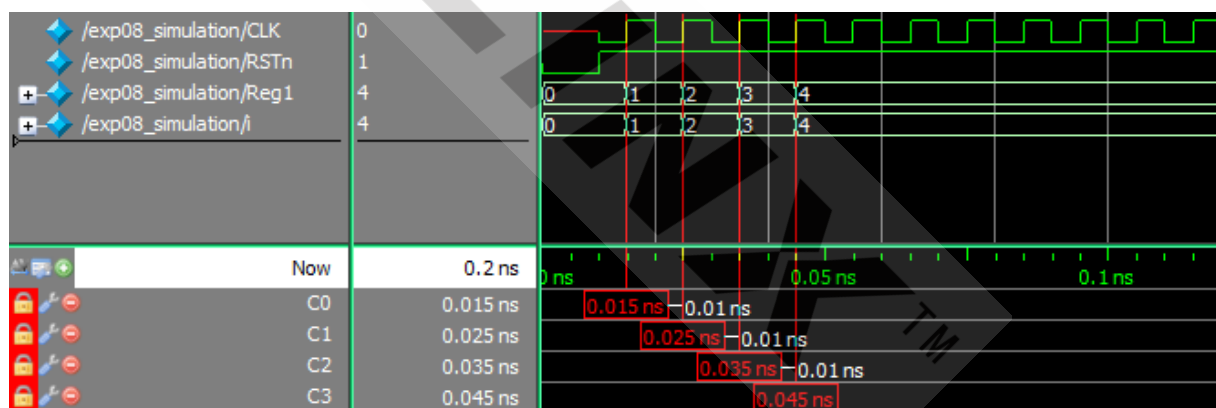


图 3.3.7 exp09\_simulation 仿真结果。

重复图 3.2.1 的方法，打开事先准备好的 wave 状态记录。如图 3.3.7 所示，i 一般建议放置在信号的最下方，此外光标 C0~C1 分别指向时间点（时钟沿）T0~T3，完后我们就可以开始分析图中的时序活动。首先是时间点 T0，此刻 i 的过去值（复位值）是 0，所以 i 当前指向时间点 T0，时间点结束 i 会自动递增。同样时刻，Reg1 决定赋值 4'b1，所以 Reg1 输出 4'b1 的未来值。

当时间点来到 T1 的时候，此刻 i 的过去值是 1，因此 i 当前指向时间点 T1，时间点结束会使 i 递增。同一时刻，Reg1 决定赋值 4'b2，所以 Reg1 输出未来值 4'b2；而时间点 T2 的时候，此刻 i 的过去值是 2，因此 i 指向时间点 T2，时间点结束会使 i 递增。同一时间点，Reg1 决定赋值 4'b3，所以 Reg1 输出未来值 4'b3；最后时间点 T3，i 的过去值是 3，所以 i 指向是时间点 T3。时间点 T4，Reg1 决定赋值 4'b4，因此 Reg1 输出未来值 4'b4。

读着读着，读者是否觉得有点眼花缭乱呢？起初笔者也是如此，实际上 Modelsim 并没有提供任何指向时钟的工具，当然我们可以手动添加光标，如图 3.3.7 所示那样。笔者手动添加光标 C0~C3 以示指向 T0~T3，但是 Wave 界面以外光标就不能使用了，关于这点多少让人觉得遗憾。因此，笔者才要利用 Verilog 的代码本身实现时钟指向功能，为此笔者可曾思考一段时间，不知巧合还是必然，只要结合仿顺序操作 i 还有理想时序，i 自然而然会指向时钟。

话虽如此，i 实际上并不是指向时钟，而是 i 此刻的过去值正好吻合时钟数，因此我们才会错认 i 确实指向时钟。结局不管怎么样，死马当活马医，[节能主义教导笔者要充分使用资源](#)。最后笔者还是强调一下，i 是否指向时钟，实际上是 i 此刻的过去值正确吻合时钟经过的个数，此外操作也必须是使用一个时钟执行，虽然也有其他技巧指向多时钟 ... 但是，一般都是一个 i 指向一个操作，指向一个时钟。



### 3.4 指向过程的 i

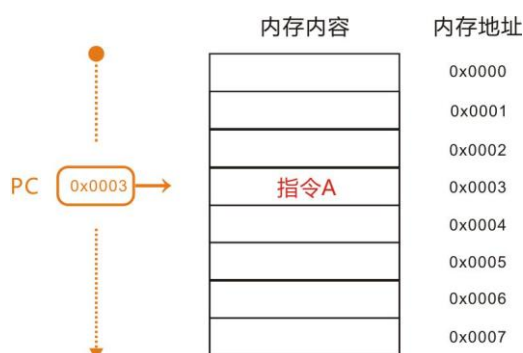


图 3.4.1 PC 的概念。

i 除了指向步骤还有时钟以外，i 还能[指向过程](#) ... 不过“指向过程”这句话又是什么意思呢？我们事先追忆一下大一所学过的“计算机结构”，传统处理器结构都有一个 PC——Program Counter，亦即计数器。PC 的功能是指向处理器下一位要执行的指令地址，然而指令又经常储存在内存当中，因此 PC 有时候也成为指向内存地址，概念如图 3.4.1 所示。处理器每执行一位指令，PC 就会自行递增，除非我们使用特殊指令妨碍 PC 或者更动 PC，否则 PC 永远的工作就是指向还有递增。

机器语言就是顺应这个传统的 PC 处理器器件，因此机器语言[才有顺序或者线性的本质](#)。结果而言，C 语言也好，汇编语言也好，C++ 还是 Java 等高级语言也好，本质上都一样，既是顺序语言。对于顺序语言而言，代码之间最粗是行数差，步骤差，最细就是指令的地址差，不管它们有什么“差”，顺序语言[永远都会伴随一个指向工具](#)，实时指向下一个要处理的代码，步骤，还是指令。然而，时钟频率既是处理器的执行频率，也是 PC 的更新频率，简单说就是差与差之间的切换间隔。

不管顺序语言再怎么庞大，操作再怎么复杂，[它始终都是单向通道](#)，指向工具就是引导主角走出通道的利器。换句话说，只要前面出现障碍，指向工具就会[停止更新地址](#)，指向工具亦即指向错误行数，步骤，还是指令地址。笔者究竟是为了什么才会长篇大论述说指向工具？在此，笔者想表明，顺序语言之说以拥有那么强的排错能力，八九不离十都是指向工具在作怪，换做并行语言呢，结果又是如何？

Verilog 语言是描述语言的家族之一，天生就有并行的性质，同时它也是一只天然呆。Verilog 的脑中除有时钟以外，[它不曾知道“步骤”是什么？“过程”是什么？但它更加不知道自己有能力描述“步骤”，还有“过程”](#)。不过，步骤还有过程对于 Verilog 来说没有时钟来得重要，因为失去时钟的它顶多只能描述组合逻辑而已。

好奇的读者可能会问“笔者为什么要强调过程的概念呢？”，回答这个问题之前，我们必须知道“一个过程是由 N 个步骤组合而成”，然而“一个步骤则是由 N 个时钟组合而成”，结果而言：

过程 = N 步骤，如果过程等价 N 步骤  
步骤 = N 时钟，如果步骤等价 N 时钟

过程 = N 时钟，那么过程既等价 N 时钟

因此我们可以这样表达：

时钟 => 步骤 => 过程

上述表达式则表示“时钟产生步骤，步骤产生过程”，这个表达式指明了时钟，步骤还有过程之间的层次关系，亦即没有时钟，步骤不能产生，没有步骤过程不能产生。同时也表示时钟比起后面两者更加重要。

笔者使用 i 指向时钟是为了强化“时钟”的印象；笔者使用 i 指向步骤是为了加强“步骤”的印象；笔者使用 i 指向过程是为了强化“过程”的印象。这些行为归根究底就是增加代码的清晰之余，**还有就是增加 Verilog 的排错能力**。就这样，Verilog 既有能力指向过程当中的错误。

但是并行语言相较顺序语言之间，指向工具还存在根本性的性质差别。笔者曾在前面说过，顺序语言不仅没有时钟概念，而且也是单向通道。换之，并行语言不仅重视时钟，而且也是多向通道。顺序语言的指向工具是系统自行创建，然而并行语言的指向工具必须手动创建。此外，**顺序语言的指向工具非常智能，相比之下并行语言的指向工具就略显逊色了**。

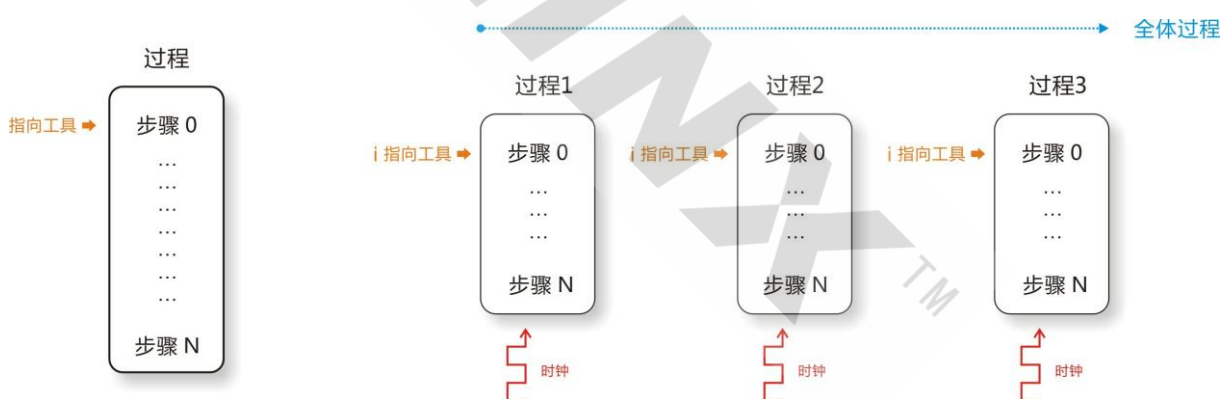


图 3.4.2 单向过程与多项过程。

图 3.4.2 表示单向过程与多项过程的示意图。左图是单向过程，聪明的指向工具正在指着当前执行步骤；右图则是多向过程，举例中有 3 个过程，每一个过程都有各自的笨蛋指向工具 i。此外，各个过程也有各自的步骤还有时钟供源，3 个过程形成全体过程。多向过程相较单向过程，要确保的东西太多，简直要喊救命。

多向过程还有一个比较蛋疼的问题，相较单向过程一个步骤卡死，全体过程就会崩溃，但是多向过程既是出现一个过程奔溃，也不会发生全体奔溃，即使输出结果亦也不是预想所要。看到这里，读者的蛋蛋是否在颤抖呢？心想仿真不仅不单纯而且还如此猥琐？没错这就是仿真不为人知的一面，此刻只是恐怖切糕略显一角而已，往后仿真还有更多黑暗等待我们绝望。如果读者不小心被吓着，笔者衷心谢罪 ... 仿真就是因为猥琐，所以我们才要事事增加模块的表达能力，增加代码的清晰度，增加时序的控制能力。



这个章节的目的就是要讨论如何让 i 实现指向过程？笔者爱用的低级建模，有一种称为用法模板的书写习惯，如下所示：

```
1.  always @ ( posedge CLOCK ... )
2.      case( i )
3.
4.          0:
5.              begin ... i <= i + 1'b1; end
6.
7.          1:
8.              ...
9.
10.     endcase
```

always 下创建一个 case ... endcase，然后 case 条件是 i。不管建模大事小事，还是有事无事，笔者都非常建议使用上述用法模板，这样作除了可以稳定风格之余，我们还可以细化仿真，助长后期建模，总之好处就是多得数不完。i 指向过程的原理非常简单，如果过程出现问题，那么 i 就会停留在该处有问题的步骤当中。总之还是使用实验来说话比较直白。

*exp09\_simulation.vt*

```
1.  'timescale 1 ps/ 1 ps
2.  module exp09_simulation();
3.
4.      /*****/ // environment signal
5.      reg CLK, RSTn;
6.
7.      initial
8.      begin
9.          RSTn = 0; #10; RSTn = 1;
10.         CLK = 0; forever #5 CLK = ~CLK;
11.     end
12.     /*****/
13.
14.     reg [2:0]i;
15.     reg [2:0]j;
16.     reg [3:0]Reg1;
17.
18.     always @ ( posedge CLK or negedge RSTn )
19.         if( !RSTn )
20.             begin
```

```

21.         i <= 3'd0;
22.         Reg1 <= 4'd0;
23.     end
24.     else
25.         case( i )
26.
27.             0:
28.                 begin Reg1 <= 4'd1; i <= i + 1'b1; end
29.
30.             1:
31.                 begin Reg1 <= 4'd2; i <= i + 1'b1; end
32.
33.             2:
34.                 if( j == 3) i <= i + 1'b1;
35.
36.             3:
37.                 begin Reg1 <= 4'd3; i <= i + 1'b1; end
38.
39.         endcase
40.
41. /*****/
42.
43. reg [3:0]Reg2;
44.
45. always @ ( posedge CLK or negedge RSTn )
46.     if( !RSTn )
47.         begin
48.             j <= 3'd0;
49.             Reg2 <= 4'd0;
50.         end
51.     else
52.         case( j )
53.
54.             0:
55.                 if( i == 2 ) j <= j + 1'b1;
56.
57.             1:
58.                 begin Reg2 <= 4'd1; j <= j + 1'b1; end
59.
60.             2:
61.                 j <= j;
62.
63.             3:

```

```

64.             begin Reg2 <= 4'd2; j <= j + 1'b1; end
65.
66.             endcase
67.
68.             /*****/
69.
70. endmodule

```

exp09\_simulation 有两个 always 块，亦即拥有两组仿顺序操作。第 18~39 行是由 i 指向步骤，Reg1 执行操作；第 45~66 行则由 j 指向步骤，Reg2 执行操作。为了方便理解，读者可以暂时将两者看成两组过程。由 i 指导的过程（简称 i 过程），先是步骤 0 把 Reg1 赋值 4'b1，然后步骤 1 赋值 4'b2；至于步骤 3 必须先满足 j 等于 2 的条件才能继续执行操作。

换之，由 j 指导的过程（简称 j 过程），先是步骤 0 判断 i 是否等于 2，如果是就继续执行操作；步骤 1 则是 Reg2 赋值 4'b1；步骤 2 是人为卡死。按照常规解读，i 过程先行一步，直到步入步骤 2，j 过程才会开始执行，此刻 i 过程必须等待过程执行一定程度才能继续执行。至于 j 过程，开始执行不久就会卡死在步骤 2 当中，因此全体过程也随之奔溃起来。

相反的，换做时序方法解读，i 过程与 j 过程是并行发生，而且也没有所谓过程奔溃的概念，顶多就是操作停留在某个步骤而已。不管增氧，我们还是先看 exp09\_simulation 仿真结果再下定论。

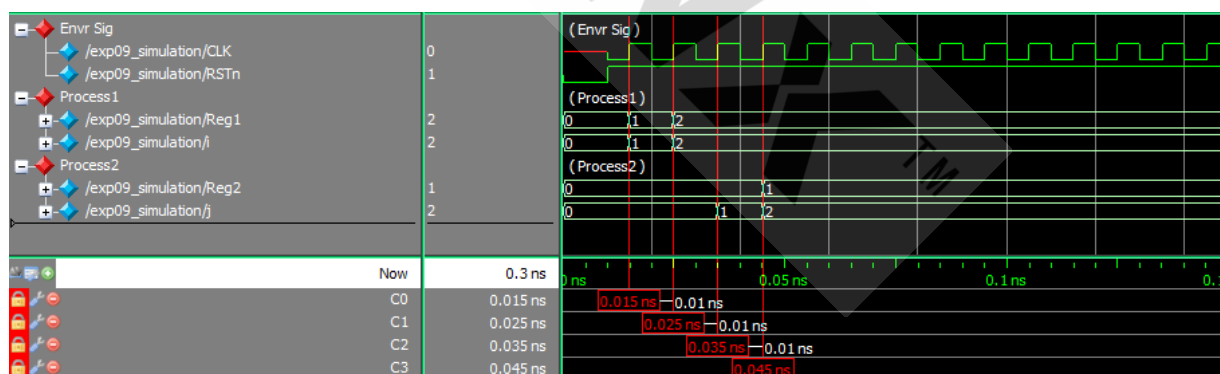


图 3.4.3 exp09\_simulation 仿真结果。

重复图 3.2.1 的方法，打开事先保存好的 wave 界面显示状态。如图 3.4.3 所示，左边笔者分类两组过程，其中指向工具 i 与 j 皆是放在组内的最尾端。此外还有四位光标 C0~C3，它们分别标志时间点 T0~T3。时间点 T0 的时候，此刻 Reg1 在步骤 0 决定赋值 4'd1，然后输出未来值 4'd1，i 也因此递增。同一个时刻 j 过程在步骤 0 判断 i 的过去值是否 4'b2，结果不是，所以没有动作发生。

时间点 T1 的时候，此刻 Reg1 在步骤 1 决定赋值 4'd2，然后输出未来值 4'd2，i 也因此递增。同样时刻，j 过程在步骤 0 判断 i 的过去值是否 4'd2，结果不是，所以什么动作也没有发生。时间点 T2 的时候，此刻 i 过程步入步骤 2 判断 j 的过去值是否为 4'd3，结果

不是，所以没有动作发生。同一时刻，j 过程在步骤 0 判断 i 的过去值是否为 4'd2，结果是，Reg2 决定赋值 4'd1，j 也因此递增。

时间点 T3 的时候，此刻 i 过程在步骤 2 判断 j 的过去值是否为 4'd3，结果不是，所以没有动作发生。同样时刻，j 过程因为笔者的坏心眼，所以卡在步骤 1，结果没有动作发生。由于 j 过程卡在步骤 1，结果 i 过程也跟着卡在步骤 2，在接续的时间点之内，i 过程和 j 过程任然没有动作发生。

常规上整体过程已经可以称为崩溃，但是 Verilog 不是顺序语言，而且**时序也没有崩溃的概念，时序最多结果不是预期所要而已**。那么重点来了 ... 读者尝试想象一下，如果此刻没有 i 指向过程 i，没有 j 指向过程 j，读者究竟会沦落当何等的窘境呢？因为没有标志指向问题所在，所以读者必须逐个时钟分析所有过程，试问自己会不会吓死不偿命呢？嗯姆姆（品尝声），这就是切糕的味道，美味吗？

换个情况，如果存在指向工具可以指向有问题的位置（步骤停留的地方），我们就可以发挥单向寻道的能力。j 过程停留在步骤 2，因此步骤 2 就是 j 过程的问题所在。换之，i 过程停留在步骤 2，因此步骤 2 就是 i 过程的问题所在。多向过程与单向过程指的向工具的排错过程大同小异，单向过程是指向工具来回游走一个过程，而多向过程则是指向工具来回游走多个过程。



图 3.4.4 单步调试工具。

事实上，**Modelsim 默认下也有指向工具，不过称为“单步调试”**，如图 3.4.4 所示。不过非常遗憾的是，“单步”这词不仅显示它是直走撞墙的傻瓜，而且“调试”这词更加暴露它是偏向顺序操作的家伙。笔者曾在前面说过，**调试不等价仿真**，因为仿真是并行发生，追求多向过程，然而调试是顺序发生，追求单向结果。结果而言，单步调试工具根本无用武之地，笔者不是说它不好，只是它不适合仿真而已。

究竟是否笔者不了解它，还是笔者的仿真风格与它相性不好呢？答案是见仁见智的，因为**笔者的仿真风格基于理想时序之余，而且时序还和代码之间拥有非常强的联系**。单步调试同一此刻只能指向一个步骤（代码行）而已，但是最为人痛心的是它没有能力指向时钟。时钟是仿真的一切，因为没有时钟无法产生步骤，没有步骤不能产生过程。

不过非常讽刺的是，传统流派把它当成宝贝来看护，就算未曾认真使用过它。想必传统流派打从一开始就**错当仿真不是追求多向过程，而是像极顺序语言那样追求单向结果**，其实**这是一座大坑**，数以万计的初学者早就已经入坑身亡了，笔者的第二次死亡就是掉入这个大坑当中。不管时间流逝多久，那种无尽黑暗逐渐吞噬全身的恶心体验，笔者时时刻刻都无法释怀 ...

### 3.5 激励的好帮手是 i

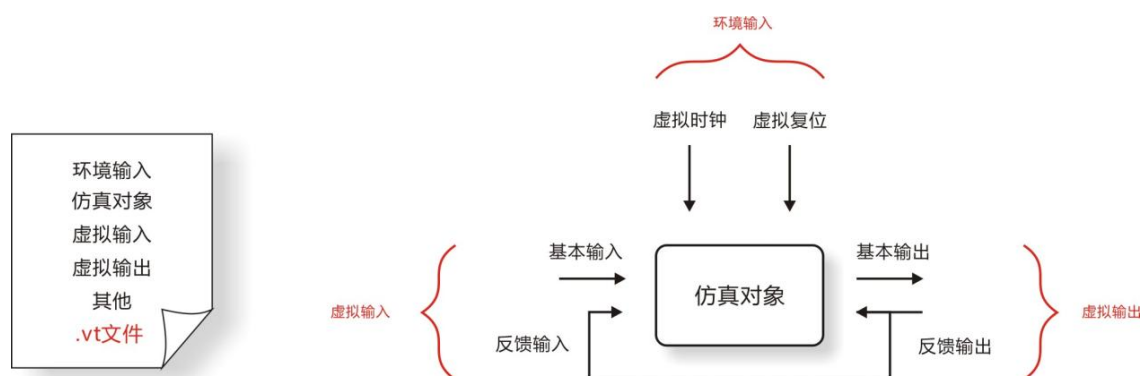


图 3.5.1 激励文件。

i 除了指向时钟，步骤，还有过程以外，i 还是激励的好帮手。虽然我们还未曾认真学习过激励，不过作为扫盲文的经验，激励基本上可以分为 5 个部分，最上端是环境输入，亦即虚拟时钟信号与复位信号；第二段是仿真对象的声明；第三段是，虚拟输入也是基本输入还有反馈输入；第四段是虚拟输出，亦即基本输出还有反馈输出；第五段则是其他，结果如图 3.5.1 所示。

我们知道 i 可以指向过程，换句话说，i 也是仿真对象的活动标志，简单而言就是仿真对象不管偷懒还是装病都会一一反映在 i 当中。这句话未完全理解之前，先让我们了解一下，所谓“反馈”究竟是激励何等定义？

基本输入还有基本输出的定义非常单纯，基本输入是给与仿真对象的刺激，然而基本输出就是仿真对象接受刺激以后所产生的反应。所谓反馈输入又名第二次输入，它不是基本输入，因为它需要根据基本输出才能做出相对应的刺激，常见反馈信号如 Done\_Sig 就是其中之一。至于什么是反馈输出呢？



图 3.5.2 模式一：单纯的功能输出。

功能仿真一般有两种模式，其一就是观察仿真对象的功能输出，亦即只有基本输出，如图 3.5.2 所示。

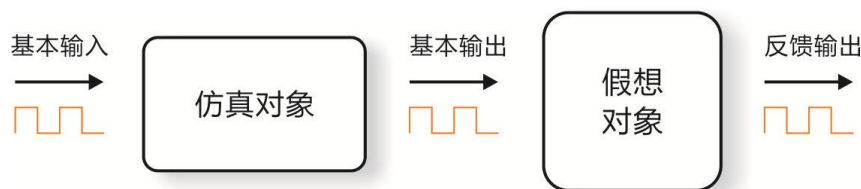


图 3.5.3 模式二：假想对象

其二，我们不仅需要观察仿真对象的基本功能，我们还要使用基本输出刺激假想对象，然而假想对象所产生的反应就是反馈输出，结果如图 3.5.3 所示。所谓假想对象实际是存在脑海当中的虚拟硬件，它是妄想所以并不存在与仿真对象或者激励内容当中，但是测试仿真对象非使用它不可。

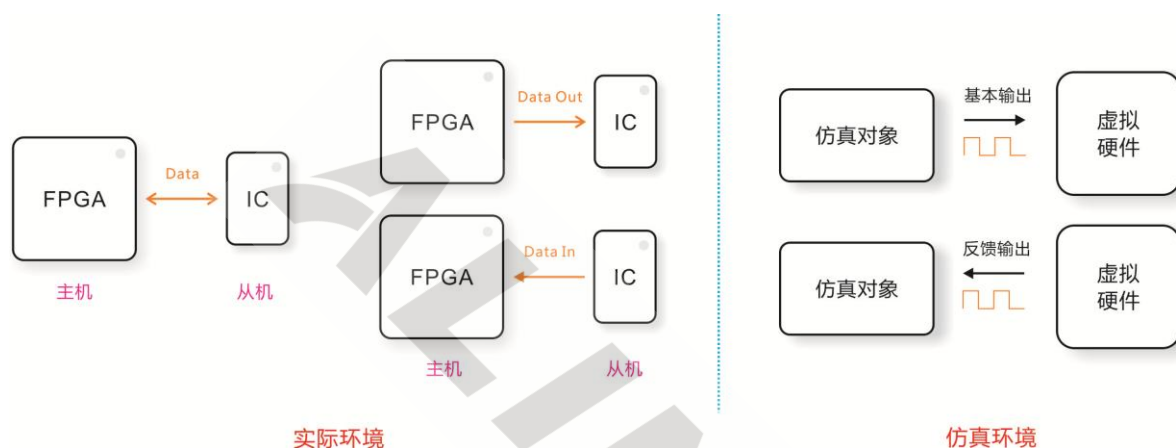


图 3.5.3 反馈输出示意图。

假设有个实际环境，我们想要使用 FPGA 驱动某个储存 IC，此刻 FPGA 是主机而 IC 从机，其中连接双方是双向 IO 的 Data 信号，如图 3.5.3 的左图所示。FPGA 驱动 IC 的的驱动程式就是我们的仿真对象，然而 IC 在仿真环境中却不存在，如图 3.5.4 的右图所示。实际环境中，如果 FPGA 为 IC 写数据的，相对之下仿真环境的仿真对象会产生基本输出，基本输出可以是一串 8'hA0, 8'hE3 等数据信号，然后以时序方式呈现在 wave 界面当中。

换之，如果实际环境中的 IC 给 FPGA 读数据 ... 相较之下，仿真环境并不存在任何实际 IC，而且虚拟 IC 也是想象的产物，此刻我们必须模拟虚拟硬件所读出的数据，这种人为的输出也称为反馈输出。实际硬件（IC）接受命令以后，知道命令的意义，然后产生相关的输出。反之，虚拟硬件宛如痴呆一般，它不仅没有命令的概，它更没有自律的能力，没叫它动它就不动，可是叫它下午 3 时吃饭，才却会准时在下午 3 时吃饭 .... 实在是在是搞不懂的家伙。

虚拟 IC 虽然痴呆但却晓得吃饭，但是重点是必须告诉它“准确时刻”。那么问题来了，何为“准确时刻”？是夕阳西落还是满月悬挂？不是，这些都不是！准确时刻是指“虚拟硬件什么时候该产生什么输出”，然而准确时刻必须根据仿真对象的内部过程来决定。Yes！我们终于找到重点了！前期，我们用 i 指向过程不仅仅是加强仿真的排错能力，实际上我们为了此刻做好准备，话说 i 强不强呀！？

不过，这里也有一个遗憾的消息，亦即我们必须手动将 i 从仿真对象当中牵引出来。这是非常麻烦的活儿，假设一个仿真对象里边包含 N 个功能，指向信号自然而然也有 N 个，结果我们必须将全部都牵引出来，感觉如代码 3.5.1 所示。

```
(一)  module abc_funcmod
(二)  (
(三)      input CLOCK, RESET,
(四)      ...
(五)      output [3:0]SQ_i, SQ_j, SQ_k ... // 出入端声明指向信号
(六)  );
(七)
(八)      reg [3:0]i , j , k ; // 建立指向工具
(九)      ...
(十)      assign SQ_i = i ; // 牵引指向信号
(十一)      assign SQ_j = j ;
(十二)      assign SQ_k =k ;
(十三)
(十四) endmodule
```

代码 3.5.1

代码 3.5.1 显示一个模块 abc\_funcmod，里边至少有 3 指向信号，亦即 i , j 与 k。我们先在出入端声明 SQ\_i ,SQ\_j 和 SQ\_k ,它们分别对应 i ,j ,k 的输出 ,至于 SQ 是 Simulation Output 的缩写，意思是指仿真输出。

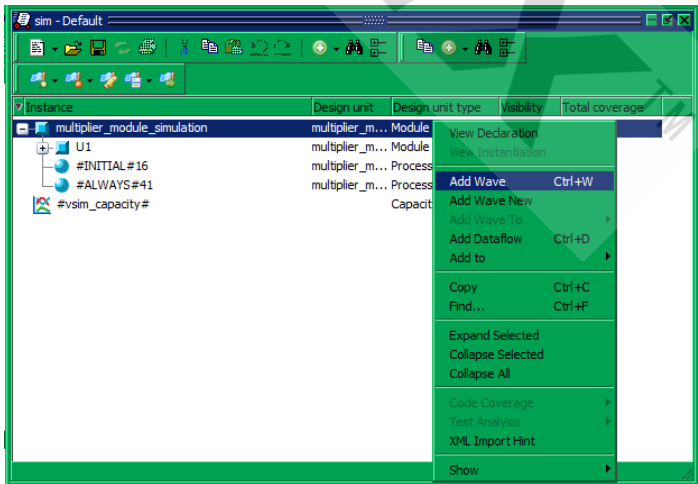


图 3.5.4 手动添加仿真信号。

虽然自动编译还有手动编译不用故意引出信号，我们只需几个简单步骤就可以将它们添加成为仿真信号。如图 3.5.4 所示，位于 sim 界面右键选择仿真对象，这样就可以将所有信号添加成为仿真信号。不过，这种临时抱佛脚的手段，仿真信号仅有观察作为，只能用来显示过程是否卡死，绝对不能引用在激励文本当中。



所以说，我们必须执行如同代码 3.5.1 的劳动，如此一来 i, j, k 这些指向信号才能用作激励。那么，究竟如何使用指向信号呢？方法很简单，如代码 3.5.2 所示。

```
1. module abc_funcmod_simulation();
2.     .....
3.     wire[3:0]SQ_i, SQ_j, SQ_k; // 指向信号
4.     .....
5.     abc_funcmod U1 // 仿真对象声明
6.     (
7.         .CLOCK( CLOCK ),
8.         .RESET ( RESET ),
9.         .....
10.        .SQ_i( SQ_i ), // 指向信号 i 引出
11.        .SQ_j( SQ_j ), // 指向信号 j 引出
12.        .SQ_k( SQ_k ) // 指向信号 k 引出
13.    );
14.    .....
15.    always @ ( posedge CLOCK )
16.        .....
17.        if( SQ_i == 3 && SQ_j == 4 && SQ_k == 5 ) ... // 使用指向信号
18.        .....
19. endmodule
```

代码 3.5.2

先为仿真对象牵引指向信号以后，然后在激励文本当中实例化仿真对象 abc\_funcmod，最后又在激励文本当中使用这些指向信号，结果如代码 3.5.2 所示。忧心的同学可能会担心，如果仿真对象的过程数量过多，那么指向信号亦也不是很多？如此一来建模是不是很麻烦？而且实例化以后，激励内容会不会不美观呢？同学就别担心了，因为只要好好遵守低级建模的准则，基本上指向信号最多只是 2~3 个而已。

不知不觉中话题又忽然扯远了 ... 如何使用指向信号实际是件麻烦事，内容涉及太多基础知识了，所以笔者就不多做实例了，详细内容往后我们再谈也不迟。在此，读者只要好好记住，如果 i 有能力指向过程，这也表明 i 有能力“**标志过程**”，例如 i 等于 0 的时候仿真对象 A 正在初始化；i 等于 1 的时候仿真对象在执执行加法等。

我们就是使用这些“**过程标志**”来“**描述不存在的虚拟硬件**”。实际上，**我们不可能为了仿真一段小功就费神去创建一个虚拟硬件**，因为这是一件非常不明智的举动，就算再现实硬件的 10% 功能，也不是说创建就能创建的程度。笔者曾经尝试创建某 IC 的虚拟硬件，不过后来却放弃了 ... 虽然 10% 的功能仿真起来可以正常运作，但是激励容量已经达到臃肿的程度，如此一来激励内容的清晰度就会大打折扣。

此外，节能本性也不允许笔者去干一些“大力完小事”的劳动，因此笔者才会动歪脑筋，借用指向信号来描述虚拟硬件，至于功能再现几%完全是根据需要所定。如此一来，这

样才符合笔者“小力完大事”的节能主义，好让自己拥有更多足够的精力用来解读时序图。



### 3.6 协调的时序。

我们知道  $i$  有能力指向时钟，步骤，过程，甚至还能描述“虚拟硬件”用作产生反馈输出。我么之所以要求  $i$  指向这个又指向那个，其实背后就是为了实现“同步”这个重要任务。一般情况下，同步是指两组对象使用相同频率发生动作，例如一起哭一起笑，换做时序的话，就是数个模块使同时钟频率的意思。不过在此，同步除了上述意义以外，笔者认为“同步”还有另一个更具有意义的意思，那就是“协调”。

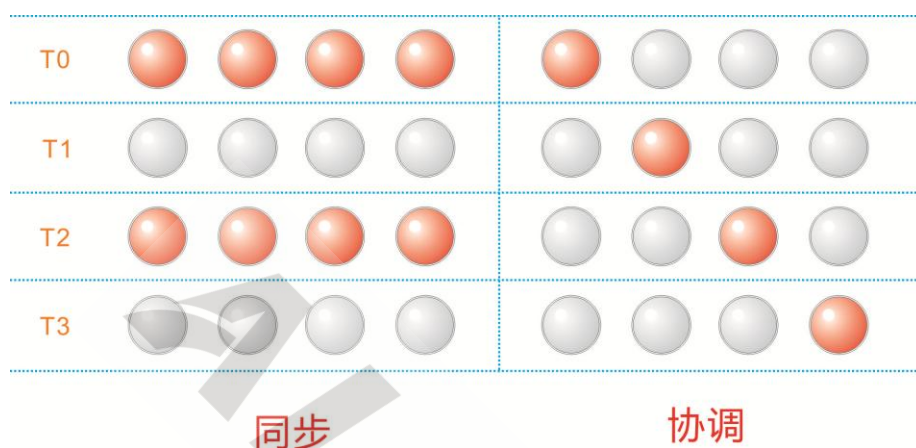


图 3.6.1 同步与协调。

举例而言，假设有两组四位的 LED 灯，左边一组，右边又一组，结果如图 3.6.1 所示。两组 LED 分别使用 4 个时间发生动作，红色表明亮，灰色表明灭。T0 的时候，左边的 LED 群一起亮，然而右边的 LED 群是最左边的一颗 LED 点亮而已；T1 的时候，左边的 LED 群一起灭，换之右边的 LED 群从左边数起第二颗 LED 点亮而已；T2 的时候，左边的 LED 群又一起亮，右边的 LED 群则是左数第三颗 LED 点亮而已；T3 的时候，左边的 LED 群又一起灭，反而右边的 LED 群是左数第四颗 LED 点亮而已。

换句话说，左边的 LED 群呈现闪耀功能也是同步操作，而右边的 LED 群展示流水功能也是协调操作。好奇的朋友可能会纳闷道：“协调就协调嘛！协调那里相似同步呢？”  
，这位同学真是提出一个好问题，接着让我们仔细分析一些图 3.6.1 的两组 LED 群。首先左边的 LED 群只有一个动作，即时亮灭而已，换之右边的 LED 群除了亮灭动作以外，还有流水动作。

朋友尝试想象一下，如果[没有节拍流水动作是否会实现](#)？答案当然是不可能，那么再请朋友想象一下如果动作不协调，流水效果还是流水效果吗？答案也是肯定的。在此，我们得到一个重要的信息，亦即同步在某种意义上有如“固定间隔的节拍”，然而[协调就是建立在这些“固定间隔节拍”之上的一系列动作（操作）](#)。

在自然界当中“同步”和“协调”是息息相关的好兄弟，就拿人体来举例。心跳供源全身可变性的固定节拍，别名又指动物时钟。紧接着，全身上下的细胞军团便依赖生物时钟发出一系列的协调操作，如喉咙——食道：

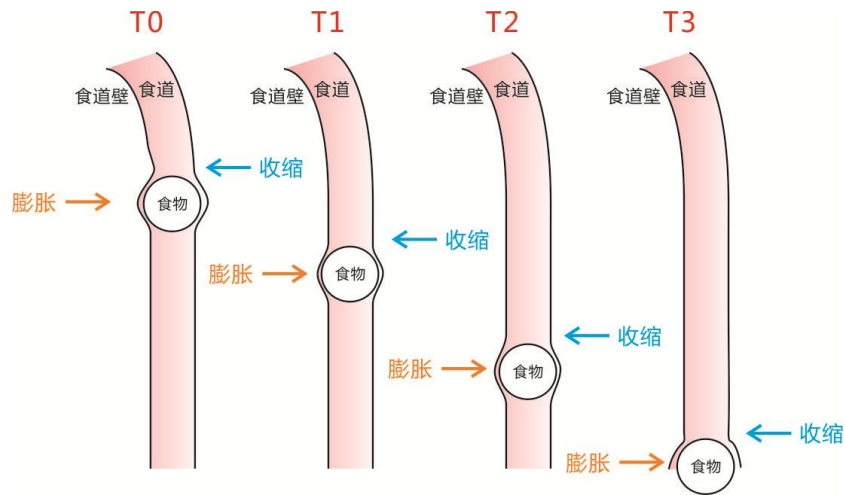


图 3.6.2 食道的协调操作。

图 3.6.2 是食道运送食物的示意图，首先我们非常清楚死人的是不会吞食物，因为死人没有心跳，因此组成食道的细胞军团并没有时钟供给，结果失去节拍 ...（血液停止循环，细胞得不到资源滋润也是其中一个原因）所以图 3.6.2 所示是活人吞咽食物的正常过程。如图 3.6.2 所示，食物完全送到胃部至少需要 4 个时间，除了地心引力拉扯食物向下以外，其实更为重要的是食道壁——收缩和膨胀的协调操作，食物最终才能成功抵达胃部。

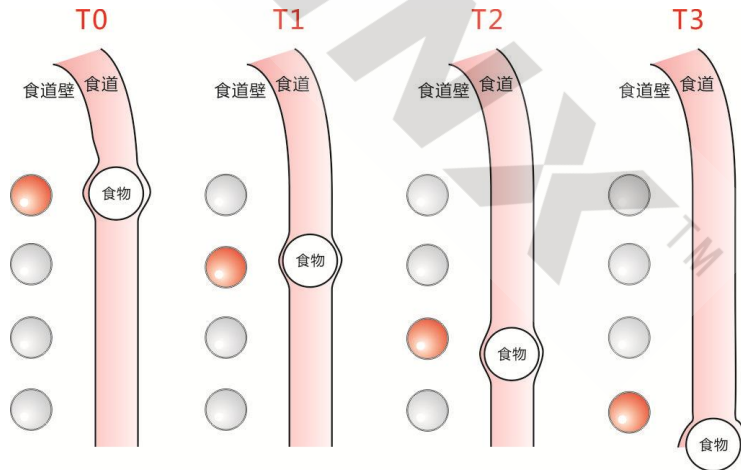


图 3.6.3 流水效果比拟食道的协调操作。

假设食道的长度恰好是 4 颗 LED 灯的数量，其中 LED 点亮代表食道壁膨胀状态，灭灯表示食道壁收缩状态。如图 3.6.3 所示，LED 群协调般一亮一灭，结果食道送入胃部完全吻合 LED 灯的流水效果。笔者承认自己 YY 想太多了，细胞军团的协调操作实际上比流水效果还要复杂一万倍以上，笔者这样做只是为了让读者有个感知的认识而已，所以情有可原。归根究底，人体的协调操作和仿真究竟有又何关系呢？

道家说过，太空是大宇宙，人体是小宇宙，或者说人体是宇宙的缩影，所以道家相信天星变化会直接或者间接影响人类。如此类推，**仿真也可以是人体的缩影**，所以仿真存在

协调操作一点也不奇怪，可是为何协调对仿真来说是如此重要呢？如果仿真有时钟用量不超过 10 个，或者单一对象还是单一操作，其实有没有协调都没有关系。反之，如果仿真有时钟用量超过 10 个，多个对象或者多组操作的话，结果就会凸显协调的重要性。

协调简单而言就是，在准确的时候，发生准确的动作，然后产生预想所要的结果，然而协调必须是“之上两个对象在一定的时间用量内”才能产生的现象。其中如何断定“准确的时候”还有“准确的动作”都是仿真应该追求的最终目的。所以说，仿真绝对不像传统流派所言那样，随便哈拉几下波形图就可以收工。那么协调又如何具体表现在仿真当中呢？

我们必须事先满足两个首要条件：其一，理解理想时序的时序表现；其二，明确指向时钟，步骤，过程等。读者不要怀疑，前面所有章节都是为此而铺垫 ...

*exp10\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp10_simulation();
3.
4.     /*****// environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #5 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [2:0]j;
16.    reg [3:0]Reg1;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            begin
21.                i <= 4'd0;
22.                Reg1 <= 4'd0;
23.            end
24.        else
25.            case( i )
26.
27.                0:
28.                    begin Reg1 <= 4'd1; i <= i + 1'b1; end
```

```

29.
30.             1:
31.             i <= i;
32.
33.             endcase
34.
35.     /*****/
36.
37.     reg [3:0]Reg2;
38.
39.     always @ ( posedge CLK or negedge RSTn )
40.     if( !RSTn )
41.     begin
42.         j <= 3'd0;
43.         Reg2 <= 4'd0;
44.     end
45.     else
46.     case( j )
47.
48.         0:
49.             begin j <= j + 1'b1; end
50.
51.         1:
52.             begin Reg2 <= Reg1; j <= j + 1'b1; end
53.
54.         2:
55.             j <= j;
56.
57.     endcase
58.
59.     /*****/
60.
61. endmodule

```

第 25~33 行是 i 过程,第 39~57 行是 j 过程。首先是 i 过程的步骤 0 执行 Reg1 赋值 4'b1 操作,然后停留在步骤 1。j 过程的则是现在步骤 0 偷懒,然后才在步骤 1 读取 Reg1 的值,最后停留在步骤 2。exp10\_simulation 虽然是习以为常的代码,但是当中却包含有趣的信息,为什么 j 过程必须先放空一个步骤才然后在步骤 1 读取 Reg1 的值,而不是一开始在步骤 0 读取 Reg1 的值。

那些不清楚的朋友当然会如此认为,反之那些已经理想时序表现的朋友非常清楚,j 过程是为了在时钟点 T1,读取 Reg1 此刻的过去值 4'b1 ( T0 的未来值 ) .. 是不是如此,就让我们一同瞧瞧仿真结果。

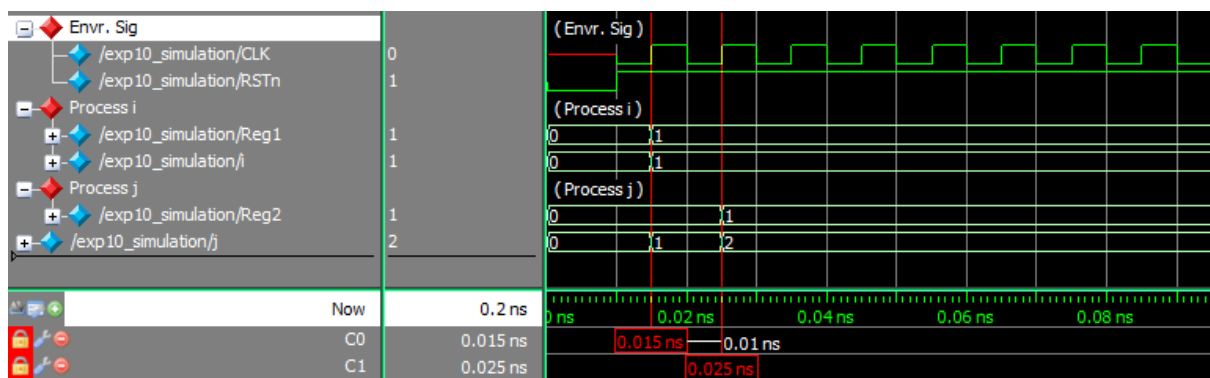


图 3.6.4 exp10\_simulation 仿真结果。

重复图 3.2.1 的方法打开事先保存好的显示状态。如图 3.6.4 所示，这里有两组信号亦即 i 过程与 j 过程，然后又有光标 C0~C1 分别指向时钟点 T0~T1。在 T0 的时候，i 过程的 Reg1 决定在步骤 0 赋值 4'b1，所以 Reg1 输出 4'b1 的未来值。同一时刻，j 过程却在步骤 0 发呆，所以什么事情也没发生。

在 T1 的时候，i 过程已经结束操作，所以什么事情也没发生。同一时刻，j 过程的 Reg2 决定读取 Reg1 的过去值 4'b1，结果 Reg2 输出 4'b1 的未来值。至于时间点 T2，i 过程和 j 过程已经没有动作发生，往后时间点也是如此。

果真，我们的猜测是正确的 ... j 过程之所以故意在步骤 0 放空，原因是为了等待 Reg1 在 T0 更新值为 4'b1。然后，j 过程在时间点 T1 亦即步骤 1，再读取 Reg1 的过去值，因此 Reg2 成功输出 4'b1 的未来值。这种外表看似非常孩子气的仿真结果，可是里边却隐藏大量值得让人思考的价值信息。

Q1：为什么 j 过程的 Reg2 会在步骤 1 成功读取 Reg1 的过去值？

Q2：其二，为什么 j 过程又会知道步骤 1 才能最快读取 Reg1 在 T0 的更新值 4'b1？

Q3：其三，这种慢一拍的数据传输又是什么现象？

首先读者必须理解，exp10\_simulation 改如代码 3.6.1 所示，我们也可以得到一样的结果。

```

1.         case( i )
2.
3.             0:
4.                 begin Reg1 <= 4'b1; i <= i + 1'b1; end
5.
6.             1:
7.                 begin Reg2 <= Reg1; i <= i + 1'b1; end
8.
9.             2:
10.                i <= i;
11.
12.        endcase

```

代码 3.6.1



但是笔者也曾经说过，相识 Verilog 这种描述语言是并行性质，习惯并行操作才能真正发挥它的能力。代码 3.6.1 与 exp10\_simulation 相较，前者好比顺序操作，因为只有一个过程，然而后者就是如假包换并行操作，因为两组操作同时发生。这是一种非常奇怪的情形，exp10\_simulation 只要换做代码 3.6.1 的形式，不知为何代码更容易理解，反之将代码 3.6.1 拆分为 exp10\_simulation 般两组过程，头就开始发疼。笔者认为倾向左脑使用的后遗症，因为左脑是线性运作，遇上多线运作左脑会忽然短路几下也不成。

当操作或者过程有两组以上的时候，协调就会发挥比金银贵的重要性。在此，可以这样回答以上 3 个问题：

A1：这是因为（理想时序）时间点事件的关系。

A2：这时因为我们有 i 指向步骤还有时钟。

A2：数据传输之间存在一个延迟的硬道理。

答案结果中，A1 与 A2 完全符合协调的事先条件，因此我们可以断定 exp10\_simulation 已经却却事实在协调操作，不然 Reg2 是不会成功在 T1 的时候读取 Reg1 在 T0 发送的未来值 4'b1。

*exp11\_simulation.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module exp11_simulation();
3.
4.     /*****/ // environment signal
5.     reg CLK, RSTn;
6.
7.     initial
8.     begin
9.         RSTn = 0; #10; RSTn = 1;
10.        CLK = 0; forever #5 CLK = ~CLK;
11.    end
12.    /*****/
13.
14.    reg [2:0]i;
15.    reg [2:0]j;
16.    reg [3:0]Reg1;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            begin
21.                i <= 4'd0;
22.                Reg1 <= 4'd0;
23.            end
```

```

24.         else
25.             case( i )
26.
27.                 0:
28.                     begin Reg1 = 4'd1; i <= i + 1'b1; end
29.
30.                 1:
31.                     i <= i;
32.
33.             endcase
34.
35.         /*****/
36.
37.         reg [3:0]Reg2;
38.
39.         always @ ( posedge CLK or negedge RSTn )
40.             if( !RSTn )
41.                 begin
42.                     j <= 3'd0;
43.                     Reg2 <= 4'd0;
44.                 end
45.             else
46.                 case( j )
47.
48.                     0:
49.                         begin Reg2 <= Reg1; j <= j + 1'b1; end
50.
51.                     1:
52.                         j <= j;
53.
54.                 endcase
55.
56.         /*****/
57.
58. endmodule

```

exp11\_simulation 与 exp10\_simulation 相比，更改的地方不多。i 过程的步骤 0，Reg1 赋值 4'b1，不过是使用 = 赋值操作符。j 过程则是没有放空步骤 0，而是 Reg2 在步骤 0 读取 Reg1 的过去值。在此，我们知道 i 过程在步骤 0 当中，Reg1 引起即时事件，j 过程的 Reg2 则是引起时间点事件。我们知道即时事件无视时钟，不过不同的是 Reg1 与 Reg2 是两个过程的居民。此刻 Reg2 能否成功读取，i 过程当中 Reg1 的即时值呢？

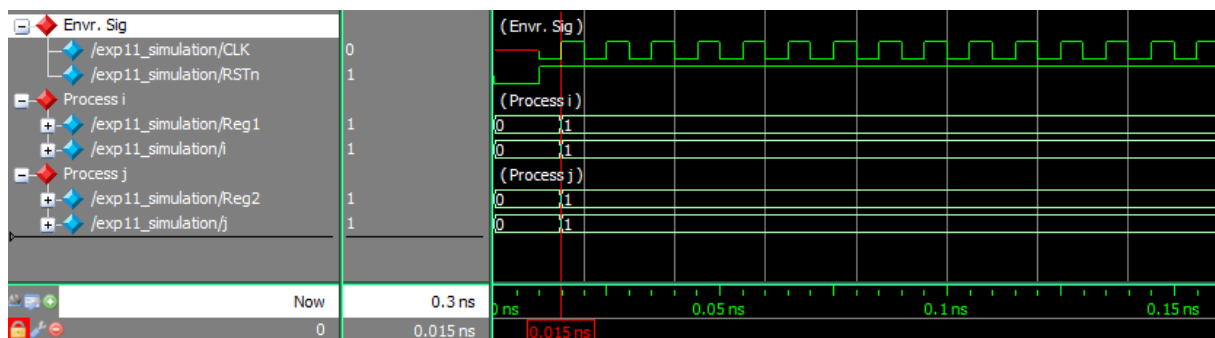


图 3.6.5 exp11\_simulation 仿真结果。

重复图 3.2.1 的方法，打开事先保存好的显示状态。如图 3.6.5 所示，仿真结果有两组过程，亦即 i 过程与 j 过程，此外还有光标 C0 表示时间点 T0。在 T0 的时候，i 过程的 Reg1 在步骤 0 赋予即时值 4'b1。在同一个时刻，j 过程的 Reg2 在步骤 0 读取得到 Reg1 的即时值而不是过去值 4'b1，结果输出未来值 4'b1。

```

13.         case( i )
14.
15.             0:
16.                 begin
17.                     Reg1 = 4'b1;
18.                     Reg2 <= Reg1
19.                     i <= i + 1'b1;
20.                 end
21.
22.             1:
23.                 i <= i;
24.
25.         endcase

```

代码 3.6.2

代码 3.6.2 的效果虽然等价 exp11\_simulation 的仿真结果，但是读者必须注意，既是结果相同，两者还是完全不同的东西。代码 3.6.2 的 Reg1 与 Reg2 是在同一个步骤发生即时事件，然而 exp11\_simulation 的 Reg1 与 Reg2 则间隔两组过程发生即时事件。不管怎么说，两者的区别是非常清楚的，代码 3.6.2 不用考虑协调的问题，然而 exp11\_simulation 则需要 ...

如果读者这样问：

Q1：Reg1 的值 4'b1 为什么会即可生效？

Q2：Reg2 怎么又知道，在步骤 0 就可以读到 Reg1 的更新结果 4'b1？

Q3：数据传输之间没有时钟延迟？

A1：因为 Reg1 使用 = 赋值操作符，结果引发即时事件。

A2：这是因为 i 不仅指向步骤，而且还有即时事件在作怪。

A3：即时事件没有时钟概念，尽管读取对象不同过程。

其中 A1 还有 A2 完全符合协调的前提条件 ... 在此有一个重点我们必须认真思考一下，如果 Reg1 赋予不是即时值！？**如果 i 没有指向时钟或者步骤的话，实际上协调操作是不会发生的。**

协调只是一个概念，协调也可以是一种代名词，如果说它存在，它却没有形体，协调就是这种暧昧的存在。仅有一组过程也不会发现协调的痕迹，当两组过程正在有效互动的时候，协调就会露出尾巴。举个而言，当读者寂寞一人在跳舞的时候，协调是不存在的。相反的，如果读者参与 10 人以上的千手观音舞蹈，协调的重要性就会凸显出来，协调发挥到极致的千手舞动，宛如华丽的流水般，一起一落都是如此优美，好似马陆虫的百只美腿一起移动。

“顺序语言有没有协调的概念？”，好奇的朋友可能会这样问 ... 顺序语言仅有一组过程，所以不存在协调的概念，不过这种情形也有限定在“任务调度机制”出现之前。

“任务调度”说得难听一点就是顺序语言模仿并行操作，也可以称为仿并行操作，其中 Thread（线程）或者 Task（任务）用作表示独立的一组过程，只要两组任务或者线程同时活动，协调自然而然就会出现。

不过，顺序语言的比起描述语言，前者没有后者般的细腻而且也不优美。不管顺序语言再怎么模仿并行操作，顺序语言时钟存在致命的步骤差，或者指令差（亦即步骤或者指令之间的延迟），这种感觉还比马陆虫的美腿移动起来一卡一卡般怪异。高频时代的今天，虽然 Ghz 级别的频率时钟可以缩短（平滑）步骤差或者指令差的间隔时间，以致卡位现象不那么明显，但是指令差或者步骤差还是存在的。

协调真的那么重要吗？很重要，非常重要，重要到不得了，尤其是仿真这一环！仿真的作用除了可视化时序以外（波形图），还要准确指明“哪一个信号在什么时候发生什么”，而且还要准确联系“这个信号之所以会那样，一定是什么代码在作怪”。除此之外，仿真为建模提供一个虚拟的模拟环境，换句话说仿真是再现功能活动的实际环境。实际环境也是现实空间，我们之所以可以同时运动 10 支手指，那是因为现实空间有“并行”这一机制。

当我们使用 10 支手指巧妙地透过键盘敲出 apple 五个英文字，其中手指敲击键盘这一系列动作称为“过程”，输出 apple 五个英文字称为“结果”（目的）。apple 五个英文字之所以可以成功输出，其实人们的手指还有键盘在不知不觉间，已经同时发生协调了。反过来说，仿真无疑是为了测试模块是否有效输出结果，输出结果之前必须经过一系列过程，为了得到正确的结果，过程必须正确无误发生才行。

假设时序表现不了解，过程没有指向，步骤没有指向，甚至时钟没有指向，这种情况“协调”又何处谈起呢？因为一切尽是模糊不堪 ... 协调对于仿真之所以那么重要，因为时序是非常铭感的东西，各种信号一起协调运作才会得到我们预想所要的仿真结果，多一个时钟或者少一个信号也有可能照成仿真结果出现偏差。

如果时序要求允许容差的话，小程度的偏差的仿真结果不会影响整体大局。虽然这个章节笔者只有两个举例而已，不过不容置疑的是，两小例子已经表明协调的重要性。假设读者试想写入 8 个数据到 ram 当中，协调性良好的情况之下，发送方只要只要 8 个时钟或者 8 个步骤即可，然而 ram 需要 9 个时钟或者 9 个步骤即可，如代码 3.6.3 所示：

```
1. reg [3:0]rData,rAddr;
2.
3. always @ ( posedge Clock ) // 发送方, i 过程
4.     .....
5.     case ( i )
6.
7.         0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 :
8.             begin rAddr <= i; rData <= i; i <= i + 1'b1; end
9.
10.    endcase
11.
12. reg [3:0] ram [3:0]
13.
14. always @ ( posedge Clock ) // 接收方, j 过程 ( ram )
15.     .....
16.     case( j )
17.
18.         0:
19.             j <= j + 1'b1;
20.
21.         1 , 2 , 3 , 4 , 5 , 6 , 7 , 8:
22.             begin ram[ rAddr ] <= rData; j <= j + 1'b1; end
23.
24.     endcase
```

代码 3.6.3

代码 3.6.3 有两组过程，其中 i 过程是发送方，然而 j 过程 ram 则是接收方。发送方只有 8 个时钟，或者 8 个步骤向 ram 发送 8 个地址数据 rAddr，还有 8 个信息数据 rData。换言之，接收方使用 9 个时钟，或者 9 个步骤接收来之发送方的 8 个地址数据，还有 8 个信息数据，其中步骤 0 是用来等待沟通延迟，因为模块与模块之间沟通至少需要一个时钟；步骤 1~8 则是接收 8 组数据的操作。

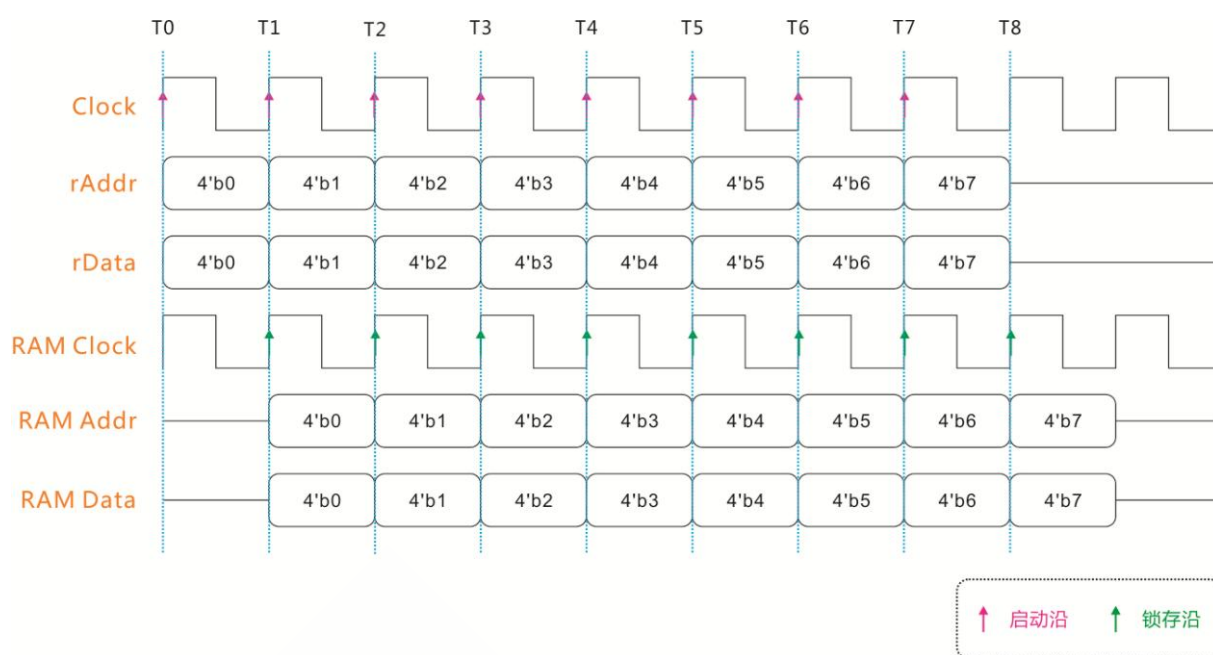


图 3.6.6 i 过程发送 8 组数据，j 过程接收 8 组数据的理想时序图。

理想的时序过程如图 3.6.6 所示，其中红色箭头称为启动沿，别名也是发送沿，主要是输出此刻（时间点）的未来值，绿色箭头称为锁存沿，别名也是读取沿，主要是用来读如此刻（时间点）的未来值。如图 3.6.6 所示，i 过程在 T0~T7 分别发送 8 组数据，然后 j 过程则在 T1~T8 接收 8 组数据。在此我们可以断定，协调已经发生了，因为 i 过程是如此有效发送 8 组数据，然而 j 过程又是如此准确接收 8 组数据。

代码 3.6.3 或者说 ram 传输时序是比较经典的例子，常常会有同学问道“为什么数据慢一拍”，等奇葩的问题，然而笔者却会回答道“这是因为时序不协调”。上述所诉，算是比较初级的问题，随着建模程度增加，信号的数量也会跟着增加，随之代码的清晰度越显重要，因为清晰的代码，时钟指向，步骤指向，过程等指向表示却是如此直接了当。再加上理想时序在背后作祟，信号之间的协调操作就会简单起来。

传统流派没有不仅没有理想时序的概念，也没有任何指向工具，所以传统流派自负的物理时序往往都是不协调的，光是看着笔者就想死。不协调的物理时序已近无数次伤害笔者那玻璃般脆弱的心灵 ... 痛苦，怨念，还有复仇等负面感情最终引发奇迹般的歪道思想。又一次，师兄召集所有初学师弟聚在一堂，然后吩咐所有人在一个时辰以内完成黑板所示的时序要求。

那是笔者人生第一次遵守歪道，写出自己专属的时序表达式 ... 瞬间笔者成为纵目注视的对象，师兄看着意义不明的描述代码，然后问道：“这是什么东西？”，笔者昂然的回答道：“这是协调的理想时序 ... ”。此外，协调也可以看是必然性，因为协调还有必然性共同指向“什么东西在什么时候发生什么”。至于笔者为什么会怎样说，往后自然会知晓。



总结：

不知不觉间，这个小节也写完了 ... 这篇小节勾起笔者无数的回忆还有感情波动，笔者既是越写越激动，内容虽然属实不过也有几分夸张的成分。小节 3 是学习仿真最为重要的内容，因为笔者所用的仿真技巧都是基于这些。其中，最为核心当然是理想时序，理想时序是非常对齐又整洁的时序，相较物理时序它太美丽了。另一方面，理想时序也是强调语言的理想本质还有功能的理想状态。

虽然笔者选择理想时序的缘由是犯懒的关系，不过理想时序却为笔者开启仿真的另一道大门。理想时序有两种时序表现，亦即时间点事件还有即时事件 ... 前者，有分此刻时间点，此刻过去值，还有此刻未来值；后则则是无视时钟的犯规存在。不过而言，整体时序一般都是时间点事件发生而成，即时事件虽然只是部分的小插曲而已，单它（即时事件）偶尔偷一下时钟就心满意足了。

此外，理想时序还非常强调清晰度，亦即代码的清晰度，还有时序的清晰度，为了达到这个目的 ... 我们必须手动使用 `i` 指向时钟，步骤，还有过程。指向时钟虽然简单不过却是非常重要的基础，许多初玩仿真的朋友认识时钟，是非常模糊而且又陌生。它们不能有效控制时钟，最大的原因是时钟不清晰，时钟也没有标志，一句盖过就是没有指向时钟的工具。

`i` 有能力指向时钟以后，随之 `i` 又可以指向步骤，还有指向过程。指向步骤也是仿顺序操的核心基础，其次也称为用法模板，传统的状态机太死板又不好控制，所以笔者使用步骤取代状态机，虽然本篇没有详细解释，不过指向步骤是很容易明白的概念。如果一个步骤停留一个时钟，就称为单步骤（时钟）指向；如果一个步骤停留多个时钟，就称为多步骤（时钟）指向。

时钟产生步骤，随之步骤又会产生过程。简单而言，过程是一系列的步骤，Verilog 相较顺序语言，指向过程是多向道，亦即无数过程同时运行。为什么我们要指向过程？原因有两个，其一就是增加仿真的排错能力，其二就是为了表述不该存在的虚拟硬件。虚拟硬件是脑海的产物，仿真期间我们不会刻意创建它，而是选择最简单的方式，借用过程的指向信号描述它们。虚拟硬件的标志是激励内容的反馈输出。

我们之所以用 `i` 指向这个，又指向那个，完全是为了协调在铺垫。协调的简单意义除了“准且的时候”或者“准确的动作”以外，还有“准确的结果”，因此代码和时序不仅不能不清晰，而且信号也不能失去指向标志。协调是建立在同步之上，同步是建立在并行之上，亦即协调至少需要两组过程同时活动才能显露痕迹。协调虽然是抽闲的概念，但它确实存在，仿真需要它，仿真依赖它，如果时序不协调，仿真结果也会出现偏差。



## 第四章 激励文本就是仿真环境

### 4.1 验证语言是什么？

一般描述语言分为两个部分，亦即“综合”还有“验证”，也是俗称的[综合语言](#)还有[验证语言](#)。为了方便，笔者时常把综合语言说为“建模语言”，然而在笔者的概念之中，验证语言的价值是可有可无，为什么呢？验证语言虽然给人看似“很强”，但是许多关键字用处都不大。此外，验证语言也是[初学 Verilog 的负担](#)，不管怎么说，笔者就是欢喜不起来 ...

传统流派时常说道：“建模用综合语言，仿真用验证语言”，其实这种说法是非常不负责任的，建模还有仿真之间的差异有如笔者曾说那样，前者是实际环境的建模，后者则是虚拟环境的建模，亦即实际建模还有虚拟建模。实际建模会受限与实际环境，虚拟建模则是相反的情况。由此可见，两者拥有大同的概念，不过环境却是小异。但是，又是什么原因将描述语言一分为二呢？

所谓[综合](#)，就是可以描述实际硬件的关键字，所谓[不可综合](#)就是无法描述实际硬件的关键字，首先让我们了解一下 `initial` 这个关键字：

```
reg CLOCK, RESET; // 仿真
initial CLOCK = 0;
initial RESET = 1;

reg [3:0]Reg1 = 4'd0; // 建模
```

代码 4.1.1

`initial` 的作用有如自己命名般——初始化资源。如代码 4.1.1 所示，仿真中，笔者先是声明两个寄存器 `CLOCK` 还有 `RESET`，然后再使用 `initial` 赋予它们初值。换之，建模中，笔者先是声明寄存器 `Reg1` 然后再直接使用赋值操作符 `=` 给 `Reg1` 赋予初值 `4'd0`。好奇的同学可能会问：“初始化还有复位化有什么区别？”，

真是一个好问题 ... [初始化是编译器的活动](#)，或者编译器给予的值，也称为初值；然而[复位化则是硬件的实际活动](#)，或者说复位结果给予的值，也称为复位值。

```
(一) module abc_simulation();
(二) .....
(三) reg Reg1;           // 声明 Reg1
(四) initial Reg1 = 0;   // 初始化 Reg1 为 0
(五) .....
(六) always @ ( ... negedge RESET )
(七)     if( !RESET ) Reg1 <= 1'b1; // 复位化 Reg 为 1
```

```
(八)  ....
(九)  endmodule
```

代码 4.1.2

如代码 4.1.2 所示，在一个名为 abc 的仿真环境当中，笔者先是声明 Reg1 然后在给予初值 1'b0，紧接着给予 Reg1 复位值 1'b1。上述一系列的简单动作告诉我们一个事实，初始化是编译器活动，然而复位化本身是硬件活动，实际上不属于编译器的范畴。然而，仿真环境 abc 却模拟（在线）硬件的复位活动，似的 Reg1 给予复位值 1。结果而言，Reg1 从头到底一共赋值两次，其一是编译器给予的初值 1'b0，其二是经由仿真模拟复位活动给予的复位值 1'b1。

事实上，initial 也使用与建模当中，不过如代码 4.1.1 所示，建模可以使用 = 赋值操作符省略 initial 这个关键字实现初始化。此外，默认下的编译器都会给予所有资源初值 0。

再者，让我们瞧瞧 # 延迟操作 ... 验证语言有一个[延迟操作符](#) #，外表看上去是普通的井号，作用却是延迟仿真时间。

```
1. 'timescale 1 ps/ 1 ps           // 声明时钟刻度
2. module abc_simulation();         // 仿真环境
3. ....
4. reg CLOCK;                       // 声明寄存器
5. initial begin
6.     CLCOK = 0;                   // 初始化寄存器
7.     forever #5 CLOCK= ~CLOCK;    // 每隔 5 隔时钟刻度重复一次 CLOCK 取反动作
8. end
9. ....
10. always @ ( posedge CLOCK )     // 每次 CLOCK 又低变高执行一下
11.     Reg1 <= #5 4' b5;           // Reg1 赋值 4' b5 之前延迟 5 个时钟刻度
12. ....
13. endmodule
```

代码 4.1.3

如代码 4.1.3 所示，笔者先是在仿真环境 abc 的第 4 行声明寄存器 CLOCK，然后在第 6 行给予初值 0。第 7 行的 forever 是验证语言，它的作用虽然相似 always，以及实时执行某种操作，但是两者却是不同的东西。

```
1. always @ ( 铭感区域 ) Reg1 <= ~Reg1;
2. always @ ( * ) Reg1 = ~Reg1;
3. always Reg1 <= ~Reg1;
4. forever Reg1<= ~Reg1;
```

代码 4.1.4

如代码 4.1.4 所示(第 1 行), always 不仅有铭感区域,而且铭感区域也必须满足条件才能实现操作。虽然我们可以为 always 的铭感区域输入 \* (第 2 行), 好让第 2 行实现第 4 行的作用,可是事实却是相反。建模中,第 2 行的代码是用来声明组合逻辑,换做仿真也是如此。第 3~4 行是验证语法,作用是仿真时间每过一个刻度的就取反一下 Reg1。第 3~4 行综合无法实现,因为综合没有“仿真时间”这种东西。

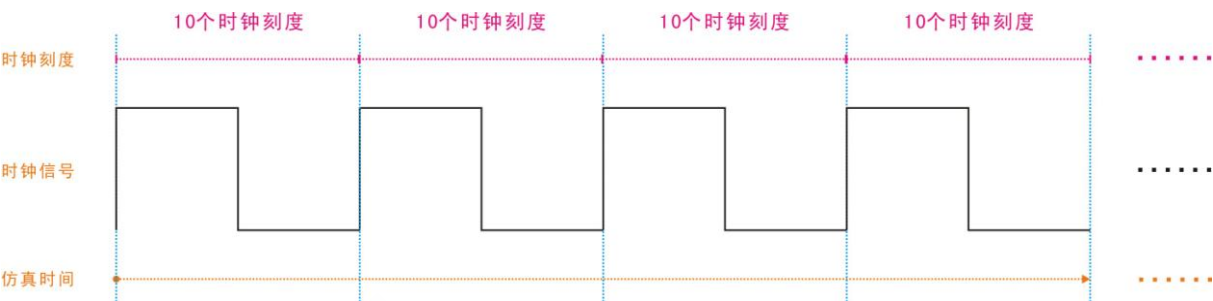


图 4.1.1 forever 产生的理想时钟。

所以 代码 4.1.3 的 forever #5 CLOCK = ~CLOCK; 表示,仿真时间没隔 5 隔刻度,就是取反一下 Reg1 的内容;只要这个动作重复 2 次,我们就会产生一个宽度为 10 个时钟刻度 50% 占空比的理想时钟。换句话说,只要这个动作无限重复下去我们就会产生源源不断的理想时钟,结果如图 4.1.1 所示。

什么是时钟刻度呢?时钟刻度可以理解为最小的仿真时间,如代码 4.1.3 所示,第 1 行出现的 timescale 就是用作声明时钟刻度,然而 1ps / 1ps 就是时钟刻度的单位,语法如下所示:

```
'timescale 实数单位 / 小数单位 // 语法格式
'timescale 1ps / 1ps // 例子 1
'timescale 1ns / 500ps // 例子 2
```

例子 1,实数单位为 ps 级别,小数单位也是 ps 级别,亦即一个时钟刻度拥有 1ps 的实际时间。例子 2,实数单位为 ps 级别,小数单位也是 ps 级别,亦即一个时钟刻度拥有 1ns 又 500ps 的实际时间。时钟刻度是仿真时间的最小单位,如果操作 forever #5 CLOCK = ~CLOCK 一例子 1 执行 20 次,那么:

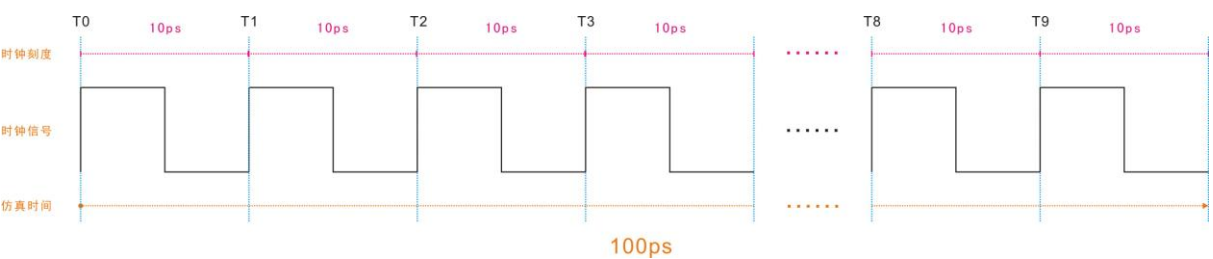


图 4.1.2 时钟刻度为 1ps/1ps, 20 个时钟刻度, 10 个时钟。

20 \* 5 \* 1ps = 100ps, 或者 10 个拥有 10ps 的理想时钟, 结果如图 4.1.2 所示。

如果操作 forever #5 CLOCK = ~CLOCK 一例子 2 执行 20 次, 那么:

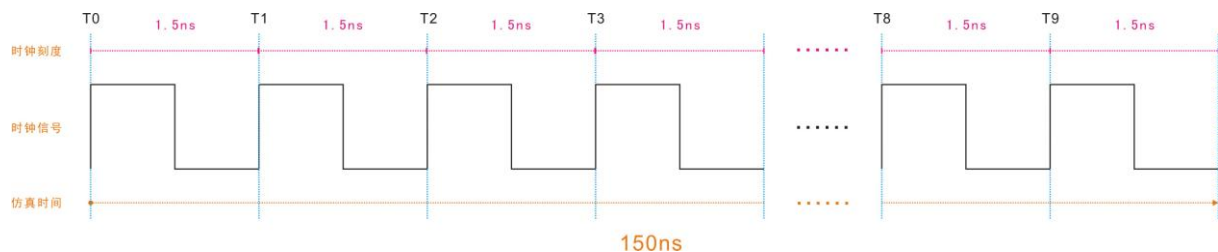


图 4.1.3 时钟刻度为 1ns/500ps，20 个时钟刻度，10 个时钟。

$20 * 5 * 1\text{ns} / 500\text{ps} = 150000\text{ps}$  或者 150ns，又或者 10 个拥有 1.5ns 的理想时钟，结果如图 4.1.3 所示。

当理想时钟产生成功，我们就有**最基本的环境输入**，接着我们就可以使用时钟沿（由低变高）触发 always 的敏感区域，亦即模拟建模的 RTL 级设计。如代码 4.1.3 的第 10 行所示，always @ ( posedge CLOCK )，然而第 11 行则表示，每一个 CLOCK 上升沿，Reg1 都赋值 4'd5，期间赋值的前面是延迟操作符——#。

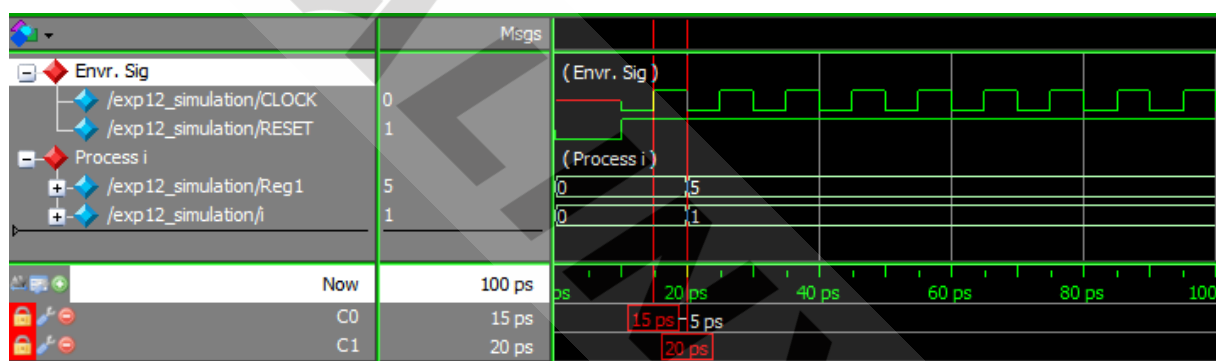


图 4.1.4 代码 4.1.3 仿真结果。

图 4.1.4 是代码 4.1.3 的仿真结果，其中光标 C0~C1 分别指向 T0 还有 T0 的半周期。如图 4.1.4 所示，先是时间点 T0，以及第一个 CLOCK 的上升沿，always 块触发成功，此刻 Reg1 也决定赋值 4'd5，不过意外的是 ... Reg1 的赋值操作遭受 # 延迟操作符的阻碍。#5 表示延迟 5 个时钟刻度，根据代码 4.1.3 的刻度单位，5 个时钟刻度亦即  $5 * 1\text{ps}$ ，也就是 5ps 物理延迟，恰好是半个时钟周期。

结果如图 4.1.4 所示，Reg1 原本应该在 C0 输出未来之星 4'd5，很遗憾的是 ... 经过 # 阻碍以后，未来值 4'd5 出现在 C1 而不是 C0。# 延迟操作符之所以认为验证语言，因为它可以实现物理延迟，然而 # 延迟操作符的使用也仅限于仿真环境（虚拟建模）而已。换之，**# 延迟操作符在建模环境（实际建模）是绝对无法通过**，为何呢？

其一，**建模环境没有仿真时间这种概念。**

其二，建模环境是用来创建理想模块，结果无法描述 5ps 这样小气的物理延迟。

看完整体代码 4.1.3 我们可以这样结论道：

initial, forever, 还有 # 都是验证语。initial 实现资源初始化；forever 每隔一个时钟刻度重复一次操作；# 根据时钟刻度，实现延迟。`timescale 声明时钟刻度，然而实际上`timescale 不是充分的验证语言，它的身上流着一半预处理操作的血（详细情况请自行去浏览语法书）。我们虽然也可以在建模环境使用`timescale，不过会被综合器无视掉，真是遗憾 ...

在此，笔者借助上述几个验证语言的目的是为了告诉读者仿真还有建模之前的微妙关系。正如笔者强调那样，仿真充其量就是在虚拟环境下实现建模（虚拟建模），实际环境有实际时间，结果**仿真环境也必须拥有仿真时间**。仿真时间的刻度（仿真时间的最小单位）必须使用`timescale 声明；实际环境有时钟源，然而仿真可以借助 forever 还有 # 关键字产生虚拟时钟。

最后还有一个重点就是 ... 仿真环境是否实现（模拟）物理参数，那是自由行的选择。如代码 4.1.3 所示，如果执行 Reg1 <= #5 4'd5 操作，Reg1 会经过半个周期的延迟才能输出未来值 4'd5；换之，如果执行 Reg1 <= 4'd5 操作，Reg1 不用经过延迟就能输出未来值 4'd5。

“仿真使用验证语言”——这句话虽然没有错，因为仿真确实需要使用验证语言来模拟实际的环境，**但是仿真不完全等价验证语言** ... 嗯，该怎么说呢？根据笔者的理解，仿真就是在虚拟的环境下实现建模，而且我们必须经由**一些手段来模拟实际环境**，如实际时间，实际时钟信号，实际复位信号。为此，我们必须**借助验证语言的力量，产生虚拟时间（仿真时间），虚拟时钟信号，虚拟复位信号**等。

仿真绝对不是传统流派所言那样，建模（综合）和仿真完全被分割开来，成为两种不同的平台。笔者对此无比愤怒，因为这番不负责任的强言强语，已经害死无数的初学者，笔者自然也是其中一人。为此，读者需要确切明白，**建模还有仿真本是同根，差别只有概念（环境）而已**。以此类推，建模所用的一切方法完全可以照搬到仿真去，这样做不仅可以大大减少学习的劳动，此外我们还可以更加有力掌握建模。

不过非常遗憾的是，权威还有主流不像笔者如此看待仿真，它们始终认为建模还有仿真应该分开。因为如此，描述语言就有综合还有验证之分，举例而言：

```
always #5 CLOCK = ~CLOCK ;
forever #5 CLOCK = ~CLOCK ;
```

上面有两组产生虚拟时钟的方法，always 常被认为它是属于综合语言，然而 forever 不管怎么看都是验证语言的东东。always 关键字可以产生虚拟时钟，forever 关键字也可以产生虚拟时钟，但是参考书不推荐使用 always 产生虚拟时钟 ... 原因也非常荒唐，因为 always 有太多综合的味道，不怎么适合仿真。因此，**forever 就成为验证版本的 always** ...

以此类推 ... **实现同样作用的关键字，就有综合版本还有验证版本**。所以说，学习验证语言实际上是重复学习综合语言一番 ... 此外，好死不死，传统流派却是倾向“调试风

格”仿真手段。笔者也曾经说过，调试是顺序语言的测试手段，为求单向结果。就这样，许多初学者容易产生混乱 ... 它们的脑子好不容易在建模阶段中习惯并行思路，可是“调试风格”的仿真手段，脑子必须 180 度切换至顺序模式 ... 结果，那些来不及切换脑子的同学，脑子就会当机，导致瓶颈。

为了证明笔者不是胡说八道，笔者再举个例子：

fork ... join 语法书说是并行块，begin ... end 则是顺序块，不过这种认知也仅使用在于保守的仿真概念而已。为了吐槽权威主义还有传统流派那些有腐旧有局限的仿真概念，笔者简单举例 2 个不同风格的代码，但是结果都是一样，然后笔者会逐个数落它们。

```
1. 'timescale 1ps/1ps
2. ....
3. reg CLOCK;
4. ....
5. initial CLOCK = 0;
6. forever #5 CLOCK = ~CLOCK;
7. ....
8. begin
9.     #00 Reg1 = 4' d0;    // 等价 Reg1 = 4' d0;
10.    #10 Reg1 = 4' d1;
11.    #10 Reg1 = 4' d2;
12.    #10 Reg1 = 4' d3;
13. end
```

代码 4.15

代码 4.1.5 声明 1ps/1ps 的时钟刻度，也使用 forever 产生虚拟时钟，亦即一个时钟需要 10 个时钟刻度，周期为 10ps。如代码 4.1.5 所示，它使用 begin ... end 块为 Reg1 赋值，根据 begin .. end 的作用，Reg1 先在 0 个时钟刻度的时候赋值 4'd0；经过 10 个时钟刻度以后赋值 4'd1；再及经过 10 个时钟刻度以后赋值 4'd2；再再经过 10 个时钟刻度以后赋值 4'd3。

在此，每 10 个时钟刻度代表一个时钟，因此 Reg1 的赋值操作可以这么认为 ... 第一时钟，Reg1 赋值 4'd0；第二个时钟，Reg1 赋值 4'd1；第三个时钟，Reg1 赋值 4'd2；第四个时钟 Reg1 赋值 4'd3。



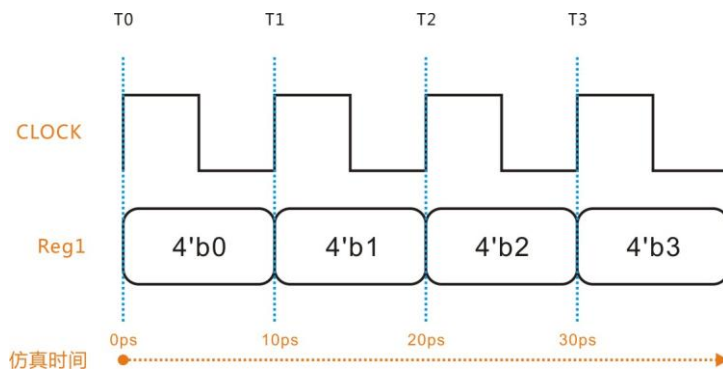


图 4.1.5 代码 4.1.5 产生的时序。

图 4.1.5 是代码 4.1.5 产生的时序，仿真时间 0ps 的时候 Reg1 输出 4'b0；仿真时间 10ps 的时候，Reg1 输出 4'b1；仿真时间 20ps 的时候，Reg1 输出 4'b2；仿真时间 30ps 的时候，Reg1 输出 4'b3。在此 **CLOCK 只是一个转为摆设而用的花瓶而已**，除了美观以外，有没有它 Reg1 还是会照样按照仿真时间输出结果。换句话说，代码 4.1.5 的 CLOCK 信号，还有 Reg1 是保持独立的关系。真是可怜的时钟信号 ...

```

1. 'timescale 1ps/1ps
2. ....
3. reg CLOCK;
4. ....
5. initial CLOCK = 0;
6. forever #5 CLOCK = ~CLOCK;
7. ....
8. fork
9.     #00 Reg1 = 4' d0;
10.    #10 Reg1 = 4' d1;
11.    #20 Reg1 = 4' d2;
12.    #30 Reg1 = 4' d3;
13. join

```

代码 4.1.6

代码 4.1.6 使用 fork ... join 块取代 begin ... end 块。代码 4.1.6 声明 1ps/1ps 的时钟刻度之余，它也在第 6 行使用 forever 产生虚拟时钟，一个时钟为 10 个时钟刻度，周期时间也为 10ps。代码 4.1.6 使用 fork ... join 为 Reg1 执行赋值，然而 fork ... join 相比 begin ... end，它不是顺序作用，而是并行作用。如代码 4.1.6 所示，Reg1 是并行赋值 4'd0，4'd1，4'd2，还有 4'd3。

不过，此刻 # 延迟操作符的作用好比条件判断般，当时钟刻度为 0 的时候 Reg1 就赋值 4'd0；当时钟刻度为 10 的时候，Reg1 就赋值 4'd1；当时钟刻度为 20 的时候，Reg1 就赋值 4'd2；当时钟刻度为 30 的时候，Reg1 就赋值 4'd3。笔者再强调一下，fork ... join 造就 Reg1 同时执行 4 种结果的赋值，不过时钟刻度作为开关条件。用建模来表示的话，代码 4.1.6 会是这种感觉：



```

if( timescale == 0 ) Reg1 = 4' d0;
else if( timescale == 10 ) Reg1 = 4' d1;
else if( timescale == 20 ) Reg1 = 4' d2;
else if( timescale == 30 ) Reg1 = 4' d3;

```

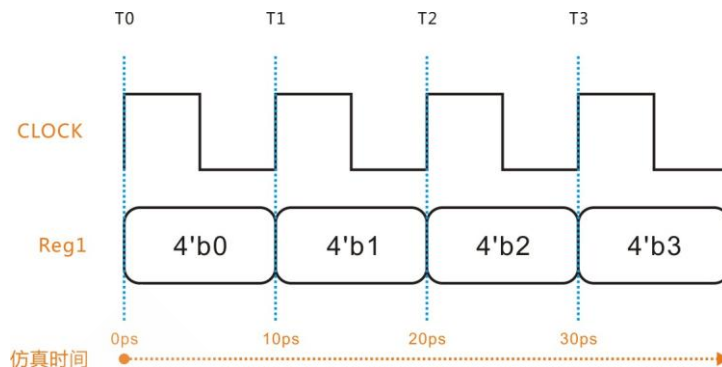


图 4.1.6 代码 4.1.6 产生的时序。

图 4.1.6 是代码 4.1.6 会产生的时序，图 4.1.6 和图 4.1.5 虽为相同结果，但是两者的 Reg1 输出却是不同的操作行为。图 4.1.5 会根据仿真时间接续（顺序）输出结果，图 4.1.6 则将仿真时间作为结果的输出条件。此刻时钟信号也有同样的困境，因为它只是摆设的花瓶而已，除了美观意外就没有其他作用。时钟信号它真是太可怜了 ...

解释完毕代码 4.1.5 还有 代码 4.1.6 以后，笔者接着要开始数落它们了。笔者之所以看它们不顺眼，其一它们既然冷落如此重要的时钟信号，其二仿真手段太接近“调试风格”了。根据代码 4.1.5 还有代码 4.1.6 不管它们使用 begin ... end 还是 fork ... join 为 Reg1 赋值，由始至终的 Reg1 赋值操作始终让人觉得好似 c 语言那样：

```

reg1 = 0;
delayps( 10 );
reg1 = 1;
delayps( 10 );
reg1 = 2;
delayps( 10 );
reg1 = 3;

```

根据 RTL 级设计的基础，我们知道寄存器如果没有时钟源触发是不会发生操作，可见代码 4.1.5 还有代码 4.1.6 已经完全违背这个重点。对此，笔者实在无法接受 ...

```

1. 'timescale 1ps/1ps
2. ....
3. reg CLOCK;
4. ....
5. initial CLOCK = 0;

```

```
6. forever #5 CLOCK = ~CLOCK;
7. ....
8. always @ ( posedge CLOCK )
9.     case ( i )
10.         0: begin Reg1 <= 4' d0; i <= i + 1' b1; end
11.         1: begin Reg1 <= 4' d1; i <= i + 1' b1; end
12.         2: begin Reg1 <= 4' d2; i <= i + 1' b1; end
13.         3: begin Reg1 <= 4' d3; i <= i + 1' b1; end
14.         .....
15.     endcase
```

代码 4.1.7

代码 4.1.7 是大伙熟悉的仿顺序操作的用法模板，i 指向时钟之余，i 又指向步骤，当然这些都不是重点。如代码 4.1.7 所示，在 T0 的时候（步骤 0），Reg1 赋值 4d'0；在 T1 的时候（步骤 1），Reg1 赋值 4'd1；在 T2 的时候（步骤 2），Reg1 赋值 4'd2；在 T3 的时候（步骤 3），Reg1 赋值 4'd3。

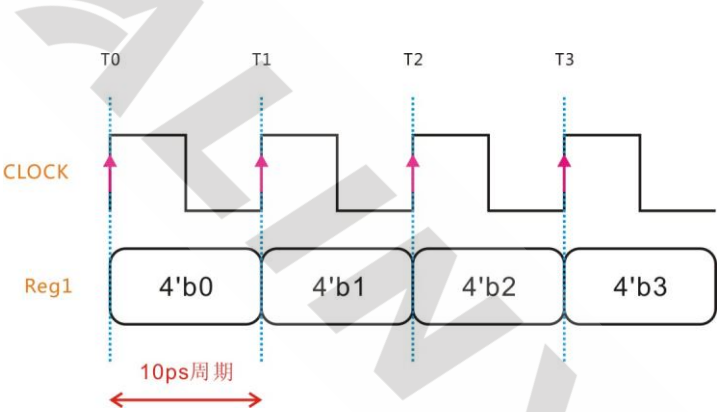


图 4.1.7 代码 4.1.7 产生的时序。

图 4.1.7 是代码 4.1.7 产生的时序结果，虽然代码 4.1.7 与代码 4.1.6 还有代码 4.1.5 都有同样的时序结果，但是代码 4.1.7 相交之下，它有决定性的不同 ... 亦即，代码 4.1.7 不是根据仿真时间为 Reg1 执行赋值，而是根据时钟信号的上升沿为 Reg1 赋值。如图 4.1.7 所示，在 T0 的时候，Reg1 决定赋值 4'b0，结果 Reg1 输出未来值 4'b0；在 T1 的时候，Reg1 决定赋值 4'b1，结果 Reg1 输出未来值 4'b1；在 T2 的时候，Reg1 决定赋值 4'b2，结果 Reg1 输出未来值 4'b2；在 T3 的时候，Reg1 决定赋值 4'b3，结果 Reg1 输出未来值 4'b3。

代码 4.1.5~7 之间的区别如表 4.1.1 所示：

表示 4.1.1 代码 4.1.5~7 之间的区别。

代码 \ 细节	CLOCK 信号	Reg1 赋值控制	时序表现	时钟指向
代码 4.1.5	花瓶	仿真时间	没有	没有
代码 4.1.6	花瓶	仿真时间	没有	没有
代码 4.1.7	不是花瓶	CLOCK 信号	有	有

如表 4.1.1 所示,代码 4.1.5 与代码 4.1.6 因为使用仿真时间控制 Reg1 的赋值操作,结果 CLOCK 信号沦落为花瓶。问题不仅如此,代码 4.1.5 还有代码 4.1.6 虽有时序图,但是时序图却没有时序表现,此外代码页没有指向时钟,这些问题的源头是以为它们忽略 CLOCK 信号的关系。

相反的,代码 4.1.7 不在视 CLOCK 信号为花瓶,而且还充分使用它控制 Reg1 的赋值操作,结果代码 4.1.7 所产生的时序图也包含时序表现。然而,更为重要的是 ... 代码 4.1.7 有用指向工具指向时钟。

代码 4.1.5~7 经过一轮的搏斗以后,代码 4.1.5~6 虽然外表上给人简洁的假象,可是暗地里它是破烂不堪的。换之,代码 4.1.7 给人感觉它比前两者书写较多代码,实际上这是支撑整体的结果。笔者曾经说过结构的重要性,传统流派为了贪图早期的便利,于是无视结构的重要性,最终为后期带来许多麻烦。

但是代码 4.1.5~6 最为糟糕的地方是“调试风格”的仿真习惯,这是最危险也是最可怕的祸种。这种破坏性十足的危险习惯,会为精神产生无数令人绝望的切糕。因为它已经偏离 RTL 级设计的基础,而且代码整体也非常松散,好似摇摇欲坠的高塔,代码的数量越是增加,危险性越是提升。但是笔者所担心的问题是思维上的疲劳,初学者在建模阶段要适应并行模式已经是非常吃力了,如果“调试风格”的仿真习惯导致思维来不起切换,又或者切换过度 ... 最终降低学习的效率。

当然,笔者之所以举例代码 4.1.5~7 是为了揭露权威主义还有传统流派那种破烂不堪的仿真习惯之余。此外,笔者还要强调,传统流派不过是将验证语言作为无意义的重复性学习而已。它们保守既局限的仿真思想,实际上已经扭曲验证语言原本的面貌,还有仿真实际的意义。它们这样作,一切仅仅是为了稳固自己的地位而已,然而却要牺牲千千万万的我们,实在可恶,实在可悲,实在可恨 ....

笔者既然已经插代码 4.1.5~6 已经那么多刀了,笔者也无妨多插一刀。用仿真时间控制赋值操作仅仅适用几个时钟数量的操作而已。如果操作消耗 N 十个时钟,然后又有 N 个资源需要赋值,代码 4.1.5~6 就会立即自爆 ... 因为代码 4.1.5~6 另外一个弱点就是没有结构支撑,所以代码 4.1.5~6 是不适合稍微操作复杂一点,或者信号数量稍微所一点的仿真。

最后让我们来总结一下这个章节的重点:

根据笔者的观念,建模还有仿真之间其实只有“概念”或者“环境”的差别而已,亦即前者为实际建模,放着为虚拟建模。验证语言真正的意义就是模拟实际环境,例如模拟实际时间(仿真时间),产生虚拟时钟信号,或者虚拟复位信号等。仿真不是使用验证语言,而是仿真需要验证语言。但是传统流派却扭曲这个事实。

传统流派还有权威主义为了面子,为了地位 ... 它们死硬将“综合语言”还有“验证语言”分为两个不同平台的东西。因此,常规上,传统流派的仿真方法,实际是重复综合

语言的学习而已。话虽如此，这种倾向“调试风格”的仿真习惯，不仅是违背 RTL 级设计的初衷，而且还 180 度扭曲仿真真实的意义。

代码 4.1.5~7 之间的比较就是最好的例子，代码 4.1.7 不仅可以实现代码 4.1.5~6 的时序结果，而且代码 4.1.7 的风格同时也适用于“建模（实际建模）”还有“仿真（虚拟建模）”之间，可谓是一箭双雕。笔者之所以故意举例代码 4.1.5~7 是为了表示，学习验证语言最为严重的误解意外，笔者还要表达初学者最容易掉入的仿真瓶颈。

传统流派是“调试风格”的仿真习惯，简单说就是一种顺序思维。换之，笔者强调的是仿真应该使用并行思维。笔者这样做是为了“建模（实际建模）”还有“仿真（虚拟建模）”共享同样的思维模式，好让初学者避免因思维切换而引起的脑袋短路问题。脑袋短路时其次的问题，天生节能主义的笔者，当然不愿因浪费精力学习重复性，而且又毫无意义的仿真。

总结完后，好奇的同学可能会问道“笔者，验证语言有很多意义不明的关键字？”，这位同学问得没错，事实却是如此。如果同学翻看官方的语法书，读者一定会看到许多陌生的验证语言，而且数量还非常可观。在这庞大的杨振语言里，其实函数性质的验证语言占据多数，例如常见的 `$readmemb()`；还有 `$display()`；。

好奇的读者可能会有会问道：“笔者，我们是不是学习全部呢？”，作为初学笔者不建议 .... 事实上，只要掌握其中几个就够用了。正如笔者前面所述那样，只有传统流派才会重复无意义的学习而已，事实上“建模”也好，“仿真”也好，我们只要换个思路即可。其实两者拥有许多共同的地方。好奇的读者又可能会不耐烦的问到：“可是，我家的叫兽却说 ...”，不要再可是了！详情会在后面继续 .... 烦死了！

## 4.2 验证语言与时序表现①

某天上午，笔者正在灼热又毒辣的天空下四处游走，支撑不了的笔者被逼躲到树荫下乘凉，忽然耳边有人叫道。

“小哥，天气那么炎热 ... 要不要来杯凉爽的豆浆呢？”，豆浆小贩道。

“嗯嗯，正好口渴 ... 老板来杯特凉特大号的！”，笔者宛如发现绿洲般兴奋道。

“小哥，久等了 ... 这是为小哥特别准备的豆浆！”，豆浆小贩答道。

笔者二话不说，立即将豆浆往口里送去 ... 咕噜咕噜（喝水声），怎么越喝味道越觉得奇怪，**豆浆既然没有豆浆香**，而且甜度未免太过了吧 ... 这真是在豆浆吗，还是糖水？笔者不禁怀疑道。

“老板，怎么豆浆完全没有豆浆的味道？”，笔者向豆浆小贩理论道。

“小哥，别开玩笑 ... 小弟的豆浆无论外表怎么看都是普通豆浆呀。”，豆浆小贩答道。

“老板，你才是别开玩笑，你自己尝尝看，这是豆浆还是糖水？”，笔者强硬道。

忽然，豆浆小贩眯着眼睛，用手将笔者拽到里边，然后用虚无的口调说道。

“小哥，小弟是卖豆浆没错，只是比例有点不同而已 ... ”，豆浆小贩道。

“比例，怎么比例法？”，好奇心忽然驱使笔者反问道。

“5%豆浆，45%糖，50%水 ... ”，豆浆小贩低声道。

“纳尼！老板，这种比例已经不是普通豆浆了。”，笔者惊讶道。

“嘻嘻，**小哥不说，小弟也不说，有谁又会知道呢？**”，豆浆小贩诡异笑道。

笔者一心想要继续理论，可是忽然出现夏天不该出现的阴风，豆浆小贩也随之消失在眼前。此刻，笔者真的吓着了 ... 刚才那是什么？难道是天气太热导致自己看见幻觉吗？不过，嘴巴里那股不是豆浆又似豆浆的味道还残留着，难道!? 疙瘩忽然爬满全身，笔者一刻也不想久留此地，哪怕外边的环境依然是毒辣的酷热。

笔者踩着熟悉的回家路，眺望熟悉的建筑物，不过却埋头思考方才遇见的情况。话说，这种经历又不是第一次，笔者用不着大惊小怪，然而让笔者苦恼的是它们出现的原因。

“小弟的豆浆无论外表怎么看都是普通豆浆呀”，还有“小哥不说，小弟也不说，有谁又会知道呢？”，不知为何 ... 豆浆小贩所言的两句话却一直漂浮在脑海中。想着想着，嘴巴随之吐出一句领悟的“原来如此”！

读者千万别误会，上述内容并不是说灵异故事，这个小节我们还在继续谈论验证语言，但是为了营造学习气氛，笔者才借用灵异故事作为比喻。故事说道，**如果豆浆没有豆浆，不管外表再怎么像极豆浆，豆浆也不是豆浆。如果不是笔者敏感，又或者豆浆小贩如果没有实话实说，笔者真的不知道那杯豆浆既然不是豆浆！**

这个故事告诉我们一个非常重要的信息，如果将豆浆反应在时序的身上，我们可以这样



说道：如果时序没有时序表现，不管外表再怎么像极时序，时序也不是时序。不过不同的是，时序没有豆浆小贩告诉我们这个豆浆（时序）不是豆浆（时序），除非我们细心观察。

好奇的同学可能会问道：“时序不是用来看嘛？怎么时序还在乎有没有时序表现？”，这位同学的好奇绝对有理。正如笔者所言，不是豆浆的豆浆，实际上还有豆浆的外表，喝起来也有豆浆的成分，不过却没有豆浆的味道。时序表现可以比拟是豆浆的味道，正宗的豆浆换做理想时序一定会有浓浓的豆香，然而豆香（时序变现）可以是，启动沿，锁存沿，时间点事件，即时事件，寄存器沟通延迟等。

反之，没有豆香的时序，外表看似时序，不过里边却没有所谓的启动沿，锁存沿，或者其他什么的。事实上，传统流派的仿真习惯可以比喻为不是豆浆的豆浆，没有时序表现的时序。不过，为何时序表现对于时序而言那么重要呢？其实这是见仁见智的问题，但是对于笔者来说追求至高无上的原汁原味是人类的天性。话说，除了傻子就没有人会喜欢不是豆浆的豆浆。

正如比例只有 5%的豆浆已经不再是豆浆那样，不是时序的时序在某种程度上已经偏离时序的本质了。那种豆浆有喝等于没喝，或者喝了是否又会拉肚子呢？仿真虽说是模拟实际环境执行虚拟建模，不管环境是什么，时序应有的时序表现我们还是应该保留下来，不然时序就会失去意义。

一些无良的奸商会使用豆浆精粉充当豆浆的原料，豆浆精粉是可以实现豆浆味道的化学味素，我们知道化学物品吃多一定会对身体造成伤害，最坏还会致癌。同样的道理，使用太多没有时序表现的时序，最终也会扭曲我们对时序的认识。根据笔者的学习经验，时序可以分为两种，以及“理想时序”，还有“物理时序”。理想时序是时序原有的理想状态，物理时序则是时序发生在物理下的状态。不管时序理想或者物理与否，时序表现之间的差别仅是大同小异的。

话了那么多，笔者还是举例例子比较容易搞明白：

```
1. module adder ( input [3:0]D1,D2, output [3:0]Result );
2.     reg [3:0]rData;
3.     always @ ( * ) rData = D1 + D2; // 组合逻辑声明，加法器。
4.     assign Result = rData;
5. endmodule
```

代码 4.2.1

笔者先是建立一个加法器，内如如代码 4.2.1 所示。该加法器有两组输入——D1 和 D2，还有一组输出——Result（第 1 行）；第 2 行声明一个暂存用的寄存器 rData；第 3 行声明一个组合逻辑，rData 则是寄存 D1 与 D2 的综合；第 4 行用 rData 驱动 Result。这是一个单纯的建模，不过根据理想时序还有建模技巧而言，其一该加法器没有时钟供源，所以它是不折不扣的组合逻辑级设计。其二，第 3 行的 rData 用 = 操作符赋值，表示 adder 输出即时结果，话说引发即时事件。

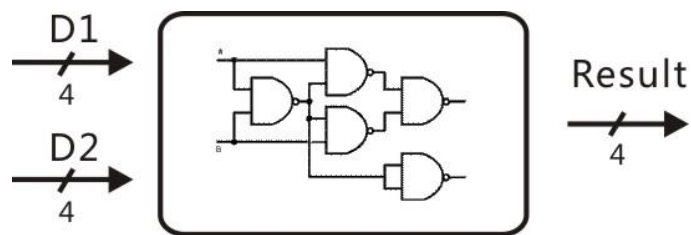


图 4.2.1 代码 4.2.1 的 adder 加法器。

如代码 4.2.1 所示，虽说我们在 2 行声明，不过这是 Verilog 的描述表现而已，因为加法器却没有使用时钟，所以综合结果用不着使用寄存器，而是使用逻辑门资源就能简单组合而成，结果如图 4.2.1 所示。再者假设该加法器是理想状态。

```

1. 'timescale 1ps/1ps
2. module adder_simulation();
3. ....
4. initial
5. begin
6.     #00 D1 = 4' d1; D2 = 4' d1;
7.     #10 D1 = 4' d2; D2 = 4' d2;
8.     #10 D1 = 4' d3; D2 = 4' d3;
9.     #10 D1 = 4' d4; D2 = 4' d4;
10. end

```

代码 4.2.2

代码 4.2.2 是传统流派的仿真风格，第 1 行声明时钟刻度的单位为 1ps/1ps，接着在第 6~9 分别每隔 10ps 经过就为 D1 与 D2 赋予不同的结果。



图 4.2.2 代码 4.2.2 驱动代码 4.2.1 加法器的时序。

图 4.2.2 是代码 4.2.2 驱动该加法器所产生的时序图，但是图 4.2.2 并不存在 CLOCK 信号，换之则有仿真时间。根据代码 4.2.2 的驱动结果，在 0ps 的时候 D1 与 D2 分别输入值 4'd1，由于我们假设加法器是理想状态，所以所有物理延迟都是 0，随之 Result 输出



值 4'd2 也在 0ps 当中；10ps 的时候，D1 与 D2 分别输入 4'd2，结果 Result 输出 4'd4；20ps 的时候，D1 与 D2 分别输入 4'd3，结果 Result 输出 4'd6；30ps 的时候，D1 与 D2 分别输入 4'd4，结果 Result 输出 4'd8。

代码 4.2.1 的加法器，还有代码 4.2.2 仿真的方式是无伤大雅的组合。因为组合逻辑本来就没有时钟的概念，再加上理想状态的组合逻辑，物理延迟都是 0 时间。结果而言，图 4.2.1 有没有时序表现也没有关系，因为实际情况也会产生类似的时序图，所以大体看上去，这个豆浆姑且还是豆浆。不过，事实上果真也是如此吗？

```
1. module adder ( input CLOCK, input [3:0]D1,D2, output [3:0]Result );
2.     reg [3:0]rData;
3.     always @ ( posedge CLOCK ) rData <= D1 + D2; // 加法模块。
4.     assign Result = rData;
5. endmodule
```

代码 4.2.3

笔者稍微将代码 4.2.1 的加法器改动一下，然后在第 1 行为他添加时钟信号之余，第 3 行 rData 的赋值方式也稍微修改一下。

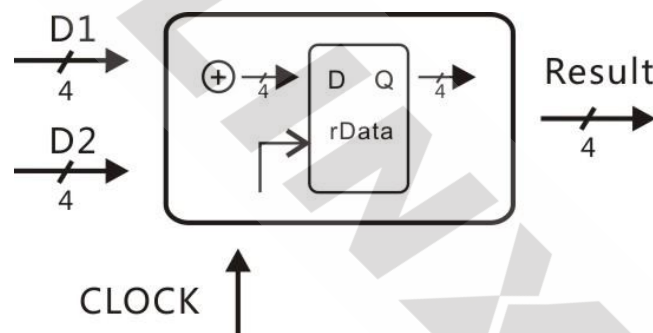


图 4.2.3 代码 4.2.3 的 adder 模块。

图 4.2.3 是代码 4.2.3 生成的 adder 模块，由于该模块搭载了 CLOCK 信号，所以综合器必须使用寄存器暂存还有输出驱动，此刻代码 4.2.3 的第 2 行 rData 会有真正的形体了，不像代码 4.2.1 那样，rData 只是单纯的描述表现而已。图 4.2.3 还显示，模块里边有一个相似异或的符号是加法器，它的真实身份却是代码 4.2.1，不过在此它变成小弟而已。根据代码 4.2.3 表示，D1 与 D1 会先经过那个像足异或符号的加法器，然后再经由时钟沿江加法结果打入寄存器当中。最后再由 rData 的输出 Q 驱动 Result 输出端。

```
1. 'timescale 1ps/1ps
2. module adder_simulation();
3.     .....
4.     reg CLOCK;
5.     intial CLOCK = 0;
6.     forever #5 CLOCK = ~CLOCK;
7.     .....
```

```

8. initial
9. begin
10.     #00 D1 = 4' d1; D2 = 4' d1;
11.     #10 D1 = 4' d2; D2 = 4' d2;
12.     #10 D1 = 4' d3; D2 = 4' d3;
13.     #10 D1 = 4' d4; D2 = 4' d4;
14. end

```

代码 4.2.4

代码 4.2.4 是根据代码 4.2.4 修改以后的结果, 虽然我们在第 6 行产生时钟, 不过第 10~13 行还是老样子, 我行我素般无视时钟, 仅仅借用仿真信号驱动。

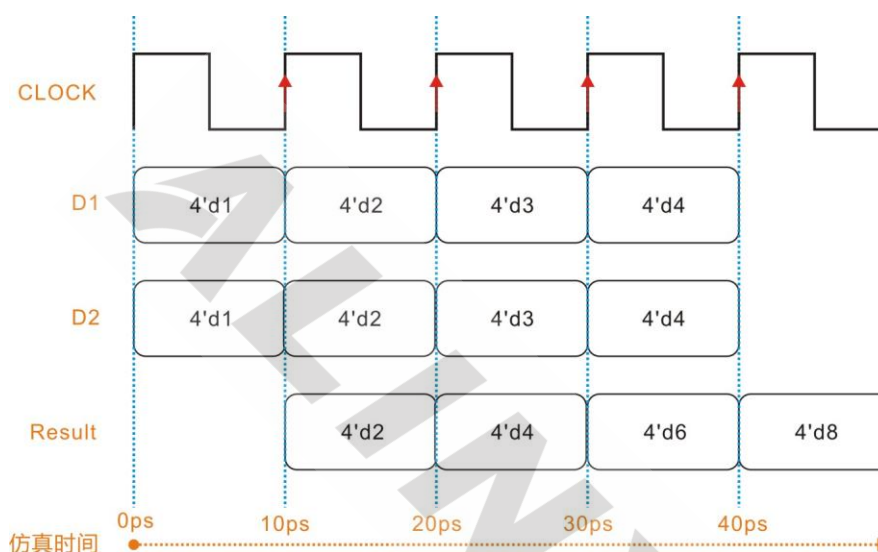


图 4.2.4 代码 4.2.3 用代码 4.2.4 驱动产生的时序图。

图 4.2.4 是代码 4.2.3 的加法模块经由代码 4.2.4 驱动以后所产生的时序图。如图 4.2.4 所示, D1 与 D2 输出时根据仿真时间, 然而 Result 的输出却根据时钟沿。传统流派比较倾向使用仿真时间, 它们认为认为时钟是花瓶, 所以时钟对于它们来说非常抽象又陌生。结果, 它们一定会认为这幅理想时序图那里肯定有问题, 因为它们无法理解, 如果时钟沿不处于数据 D1 与 D2 的正中央, 数据又如何打入寄存器当中? 这时候, 豆浆不是豆浆的问题发生了。

仿真时间就好比豆浆的水成分, 仿真时间使用越多, 水成分的占据比例就会越多, 豆浆的味道因此也会随之变淡, 渐渐豆浆再也不是豆浆, 时序也会失去原本“味道”。理解理想时序的朋友会非常清楚, 实际上图 4.2.4 是没有问题, 但是时钟的概念却是非常模糊, 因为没有使用时钟也没有指向时钟的关系。

再加上 D1 与 D2 又是根据仿真时间驱动, 因而我们根本无法知晓 D1 与 D2 是触发“时间点”事件, 还是“即时事件”?

```

1. 'timescale 1ps/1ps
2. module adder_simulation();

```

```

3. ....
4.     reg CLOCK;
5.     initial CLOCK = 0;
6.     forever #5 CLOCK = ~CLOCK;
7. ....
8. initial
9. begin
10.    #5 D1 = 4' d1; D2 = 4' d1;
11.    #15 D1 = 4' d2; D2 = 4' d2;
12.    #15 D1 = 4' d3; D2 = 4' d3;
13.    #15 D1 = 4' d4; D2 = 4' d4;
14. end

```

代码 4.2.5

修改代码 4.2.4 成为代码 4.2.5 是一种非常赔了夫人又折兵的补救手段 ... 如代码 4.2.5 所示，笔者针对第 10~13 行的延迟参数做出一番修改，以及 D1 与 D2 的输入之前多了 5ps 延迟。

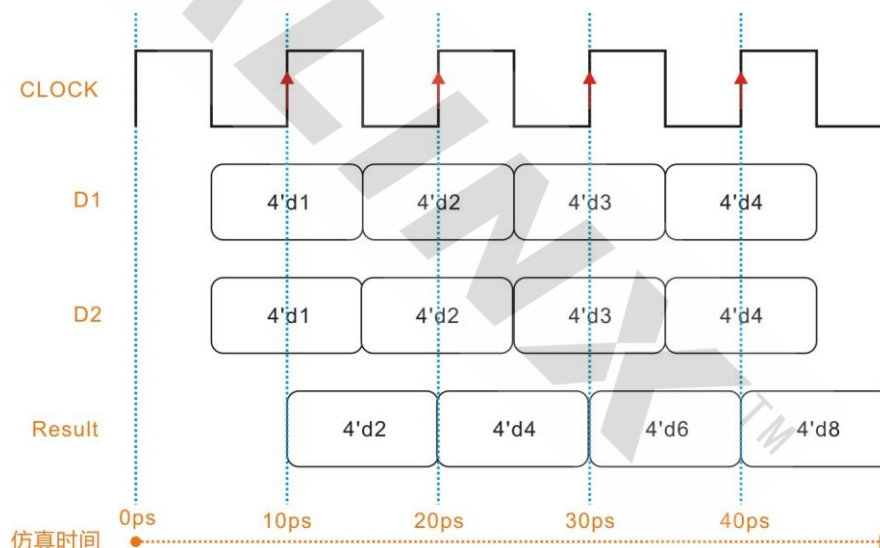


图 4.2.5 代码 4.2.5 驱动 4.2.3 所产生的时序。

图 4.2.5 是代码 5.2.5 驱动代码 4.2.3 模块所产生的时序图。如图 4.2.5 所示，D1 与 D2 的输入相较图 4.2.4，稍微延迟 5ps（半个时钟周期）。根据常规理解，时钟沿因为处于数据 D1 与 D2 的正中央所以，数据才会有效打入寄存器。但是，代码 4.2.5 这种弄巧成拙的补救手段只能勉强挤出几滴时序表现而已，而且还把美丽的时序弄成丑陋的物理时序，因为信号已经失去对齐性。

Verilog 是理想本质，输出的信号理应理想，所以 Verilog 是没有能力描述类似 D1 与 D2 这种物理延迟。坏心眼的同学可能会反驳道：“图 4.2.5 中，D1 与 D2 不过只是延迟半周期而已嘛，如果使用时钟的下降沿触发不就行了嘛，笨蛋笔者！”，啧啧 ... 这位同学想用就用吧，乱葬岗那里处很快就会发现你的尸体。原因很简单，如果 D1 与 D2 不是延迟半周期的话，你又该怎么办呢？

```

1. 'timescale 1ps/1ps
2. module adder_simulation();
3. ....
4.     reg CLOCK;
5.     initial CLOCK = 0;
6.     forever #5 CLOCK = ~CLOCK;
7. ....
8.     reg [3:0]i,D1,D2 ;
9.     always @ ( posedge CLOCK )
10.         case( i )
11.             0:
12.                 begin D1 <= 4' d1; D2 <= 4' d1; i <= i + 1' b1; end
13.             1:
14.                 begin D1 <= 4' d2; D2 <= 4' d2; i <= i + 1' b1; end
15.             2:
16.                 begin D1 <= 4' d3; D2 <= 4' d3; i <= i + 1' b1; end
17.             3:
18.                 begin D1 <= 4' d4; D2 <= 4' d4; i <= i + 1' b1; end
19.             .....
20.         endcase

```

代码 4.2.6

为此，笔者进一步为代码 4.2.5 美化成为代码 4.2.6。如代码 4.2.6 所示，不仅使用用法模板好让 D1 与 D2 的赋值操作是根据时钟变化而不是根据仿真时间。此外，笔者也使用 i 指向步骤还有时钟，而且 D1 也 D2 也有明确的时序表现，亦即 D1 与 D2 都会发生时间点事件。

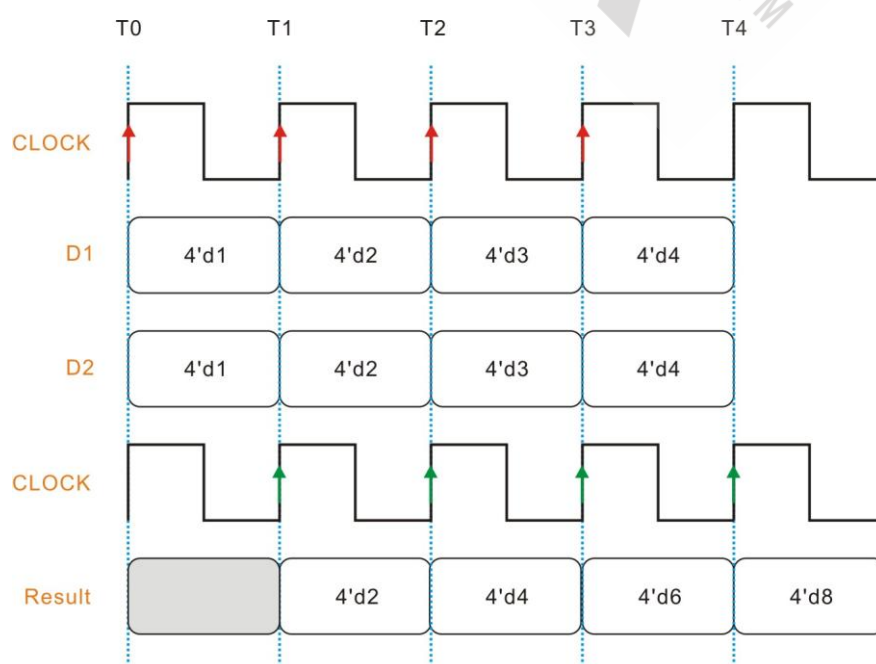


图 4.2.6 代码 4.2.6 驱动代码 4.2.3 所产生的理想时序。

图 4.2.6 是代码 4.2.6 驱动代码 4.2.3 的模块所产生的理想时序。如图 4.2.6 所示，笔者之所以故意画出两行时钟信号时为了强调，**D1 与 D2 不仅是由时钟控制**，而且代码 4.2.3 所描述的**加法器也是由时钟控制**。此外，笔者也故意标示红色箭头表示启动沿，绿色箭头表示锁存沿。D1 与 D2 根据时钟信号的启动沿输出未来值，加法器根据时钟信号的锁存沿读取 D1 与 D2 的未来值，然后输出相关的未来值。

读者是否可以感觉得到 ... 代码 4.2.6 还有图 4.2.6 除了使用仿真时间产生模式时钟信号以外，**所有时序活动都是使用时钟控制**，要时钟沿（启动沿和锁存沿）有时钟沿，要时间点事件有时间点事件，要信号对齐信号就对齐 ... 整体充盈满满的时序香，啊~实在美丽极了 ... 这才是笔者梦寐以求，浓浓时序香的时序图，浓浓豆浆香的豆浆，咕噜噜 ... （饮水声），哈 ... （豪爽的吐气声），实在美味极了！再来一杯！咳咳咳 ... 废话说太多了，结果给豆浆呛到了，真是及时报。

这个章节，笔者想传达的信息组要有两个 ... 其一，就是**豆浆不是豆浆，时序不是时序的细节问题**。时序表现好比豆浆的豆香，时序表现越丰富豆香就越浓，时序更有时序的味道。豆浆浓不浓，时序香不香其实这是非常主观的想法。举例而言，许多著名的演奏家未演奏之前，他们都会把自己关在安静中酝酿音乐情绪。此刻，如果读者不小心插身而过，读者会感觉到浓浓的特殊氛围，那就是音乐香。

再举个例子，艺术家未作画之前都会死死盯着白板，有人说他们是正在将想象具体化，又有人说他们在培养创作的感觉。此刻，如果我们插身而过，我们会感觉独特的艺术气场，艺术家称为艺术香。作为外人，我们是很难理解他们的怪异举止，然而这些怪人却相信，只要“香气”的浓度越高表现就会越好，事实是否如此只有当局者知道 ...

仿真的时候，**如何培养或者酝酿仿真情怀**，笔者认为那也是仿真的课题之一。因为仿真时基于时序，而且时序表现就是时序的“香气”，笔者始终感觉，如果时序表现越是强烈，笔者的仿真欲就会越高，仿真也会越做越起劲。笔者无法理解，那种感觉究竟是不是错觉，香气越弄，时序越有真实感，这里意指的真实感不是破烂不堪的物理时序，而是时序原有的究级理想面貌。笔者一直有一股强烈的冲动想要抓住它，每当触手可及的时候，它就会立即消失不见，原来仿真已在不知不觉的情况下完成了 ...

读者很可能会认为笔者是怪咖，笔者不否认，因为笔者与他们（演奏家或者艺术家）都有相似的本质，亦即追求美丽的原始冲动。笔者一直以来都强调着，心境还有心情是非常重要的 ... 好心情就有好表现，反之亦然。传统流派造就的时序太丑陋了，看着心情也会掉到谷底 ... 更别谈仿真了。

除了上述非常私人的原因以外，笔者也想透过这篇章节像读者表达 ... 传统流派作仿真为何会基于物理时序？原因正如前面所言，**传统流派的仿真习惯非常倾向“调试风格”，后果造就它们过度依旧仿真时间，当时间长了时序就会开始暴走不受控制。为此，它们需要采取补救手段，结果时序就那样变成破烂不堪。**

如果有一杯自高无上的香浓豆浆，还有一杯灌水灌过度的豆浆 ... 请问读者会怎样选择呢？想必没有人会喜欢不是豆浆的豆浆，同样也没有人会喜欢不是时序的时序。早期，那是传统流派横行的年代，笔者因为没有选择，因此笔者才会强迫自己喝下不是豆浆的豆浆。坏东西喝多了胃壁也会穿洞，因此笔者告诫自己要好好珍惜生命。

此外，笔者也想告诉读者 ... 类似传统流派这种仿真手段，[已经停滞 N 年](#)了，说得难听一点，那是[适合门级仿真的仿真](#)手段。笔者当然不是随意口吐妄言，读者随意翻开基本参考书就能验证这点，然而这种仿真手段却不适合门级以外的仿真。期间会出现，问题能容会臃肿，内容凌乱等问题。笔者曾经遇见有 5 级嵌套 if 的激励内容，说实话，笔者的蛋蛋立即就掉在地上然后昏死过去 ...

笔者最后还是再强调一下，味道还有喜好本来就是因人而异，有人认为传统流派是绝对正统；也有人会认为新时代就要新表现；所以不是豆浆的豆浆是否还是豆浆，都是见仁见智的问题。



4.3 验证语言的时序表现②

验证语言拥有无数个关键字，一般都是系统函数占据多。根据官方手册，系统函数开头都有 \$ 美金的符号，紧接着会有函数名还有参数选项 ... 常见的系统函数如表 4.3.1 所示：

表 4.3.1

系统函数	用途
\$display();	打印信息在信息显示界面

看着看着，好奇的同学可能会问道：“笔者，系统函数难道只有 1 个吗？”。是呀！根据笔者的习惯，常用的系统函数的确只有 1 个而已。好奇的同学可能又会问：“笔者，别开玩笑啦！”，笔者没有开玩笑 ... 官方手册的确记录许多系统函数，不过像夏老师那样努力的人勉强也能举例几个。换之，像笔者这种懒人自然也只有这种程度而已。

\$display()函数与 C 语言的 printf() 函数非常相识，不过\$display() 函数稍微细腻一些，如，\$display() 函数常用的格式还有换码如表 4.3.2 所示：

表 4.3.2

格式换码	用途
%h	输出十六进制
%d	输出十进制
%b	输出二进制
%c	输出字符
%f	输出浮点数
\n	换行

有 C 语言功底的同学当然对表 4.3.2 不会感觉陌生，除了 %b 以外。Verilog 语言相比 C 语言，前者拥有更强的为操作能力，对此 \$display() 函数也凸出这点。\$display() 函数的常见用法如代码 4.3.1 所示：

```
1. always @ ( posedge CLOCK )
2.     case( i )
3.         0:
4.             begin reg1 <= 4' d10; i <= i + 1' b1; end
5.             1: // 打印二进制格式
6.             begin $display( "reg1 = 4' b%b " , reg1); i <= i + 1' b1; end
7.             2: // 打印十六进制格式
8.             begin $display( "reg1 = 4' h%h " , reg1); i <= i + 1' b1; end
9.             3: // 打印十进制格式
10.            begin $display( "reg1 = 4' d%d " , reg1); i <= i + 1' b1; end
11.        endcase
```

代码 4.3.1



根据代码 4.3.1 所示，笔者现在步骤 0 为 reg1 赋值 4'd10，然后在步骤 1~3 相序打印不同的输出格式，打印结果如下所示。

```
reg1 = 4' b1010
reg1 = 4' ha
reg1 = 4' d10
```

教授使用 \$display() 函数不是笔者真正的目的，在此读者需要仔细思考 ... 一些系统函数如 \$display() 函数等，**它们就是怎么样的存在**？然而，它们又会拥有**怎样的时序表现**呢？系统函数实际上是一种犯规，或者称为异存在的存有，形象点好比现实世界的超人或者孙悟空般，能力超凡，给人感觉一点也不现实。话虽如此，既然来到仿真的世界，它们再怎么犯规**也要遵守最基本的时序法则**。

```
1. always @ ( posedge CLOCK )
2.     case( i )
3.         0:
4.             begin
5.                 reg1 <= 4' d10; i <= i + 1' b1; end
6.                 $display( "reg1 = 4' b%b " , reg1); // 打印二进制格式
7.                 $display( "reg1 = 4' h%h " , reg1); // 打印十六进制格式
8.                 $display( "reg1 = 4' d%d " , reg1); // 打印十进制格式
9.             end
10. endcase
```

代码 4.3.2

**系统函数如 \$display() 一般引发即时事件**，即时事件也是无视时钟的意思。如代码 4.3.2 所示，是笔者基于代码 4.3.1 的修改。根据修改结果，原本分别需要使用 3 个时钟输出的操作，变成在一个时钟内完成。笔者先是在第 5 行使用 <= 操作符赋值，然后接续在第 6~8 行输出不同格式的结果。读者事先猜猜一下，就是会是怎样的打印信息呢？

```
# reg1 = 4'b0000
# reg1 = 4'h0
# reg1 = 4'd0
```

启动仿真不一会，打印结果也会跟着出现，如上所示，所有打印结果分别都是 0，是不是觉得很奇怪？其实这样打印结果是非常合情合理，正如笔者所言那样，系统函数 \$display() 就算是超人它也必须遵守时序规则，在这里( 时序 )它是触发即时事件的家伙。不过根据理想时序的原理，reg1 是触发时间点事件，结果此刻 reg1 的内容是过去值 0 ( 0 值是初值也是复位值 )，所以 \$display() 函数也相序打印值 0。

```
1. always @ ( posedge CLOCK )
2.     case( i )
3.         0:
```

```

4.      begin
5.          reg1 = 4' d10; i <= i + 1' b1; end
6.          $display( "reg1 = 4' b%b " , reg1); // 打印二进制格式
7.          $display( "reg1 = 4' h%h " , reg1); // 打印十六进制格式
8.          $display( "reg1 = 4' d%d " , reg1); // 打印十进制格式
9.      end
10. endcase

```

代码 4.3.3

代码 4.3.2 相较代码 4.3.3 内容是大同小异,除了第 5 行 reg1 的赋值操作由 = 改为 <=。此刻再重新执行仿真就会得到以下的打印信息。

```

# reg1 = 4'b1010
# reg1 = 4'ha
# reg1 = 4'd10

```

啊拉！？是不是觉得很神呢，此刻由于 reg1 赋予即时值 4'd10，所以 \$display() 函数才得以输出 0 值以外的结果。代码 4.3.2 还有代码 4.3.3 告诉我们一个简单的道理，不管对方是老孙还是玉皇大帝，既然站在时序这块土地上，当然就要乖乖遵守这个世界的法则，不然就要吃不了兜着走。

此外，笔者也想透过代码 4.3.2 与代码 4.3.3 告诉读者一个信息，函数系统的本质其实是顺序语言，所以系统函数的常见用法也非常倾向“调试风格”，例如常见的参考书都会这样使用它 ...

```

1. begin
2.     #00 reg1 = 4' d10 ;
3.     $display( "reg1 = 4' b%b" , reg1 );
4.     #10 reg1 = 4' d12 ;
5.     $display( "reg1 = 4' b%b" , reg1 );
6.     .....
7. end

```

代码 4.3.4

如代码 4.3.4 所示，这是传统流派的爱用方法，亦即使用仿真时间控制整体时序。常规上，我们会这样解读：先在第 2 行将仿真时间延迟 0 个刻度，然后为 reg1 赋值 4'd10；在第 3 行打印 reg1；在第 4 行将仿真时间延迟 10 个刻度，然后为 reg1 赋值 4' d12；在第 5 行打印 reg1。此刻，代码 4.3.4 就会变成不择不扣的“调试风格”，因为代码 4.3.4 解读起来与一般的顺序语言没有两样。

这种半吊子的仿真习惯造[很容易流失时序表现](#)，最终造就豆浆不是豆浆，时序不是时序的窘境。系统函数固然好用，但是我们也要选择最适合的用法，函数始终不是仿真的主角，它的作用宛如[跑龙套](#)般，偶尔在镜头出现几分钟就行了。换个角度而言，系统函数

虽然不是主角,如果喜剧没有路人甲乙丙丁会让人觉得太假了,这就是跑龙套的重要性。同样道理,笔者偶尔也会使用打印函数输出一些信息用于教程,但是更多时候直接观察时序图的信息才是至关重要。

再者一些仿真语言,如 for, while 等循环功能。for 是半阴半阳的奇怪家伙,虽然语法书上解释它是综合语言,但是它的实际行为更接近验证语言。举例而言,在建模的时候 for 是不建议使用的,因为它容易扰乱时钟控制,因此 for 的作用仅限用于初始化 ram 而已,如下代码 4.3.5 所示:

```
1. reg [7:0]i;           // 声明 i
2. reg [7:0] ram [127:0]; // 声明 ram
3.
4. for ( i = 0; i < 128; i = i + 1 ) // 使用 for 初始化 ram
5.     ram[i] = i;
```

代码 4.3.5

现今的集成环境如 Quartus II,我们已经可以使用 mif 文件初始化 ram,为此 for 的作用显得更加没有价值,所以人们将 for 称为建模的鹌鹑,灰心满满的 for 就这样消失在建模的舞台中。不知何时 for 在仿真光芒四射,,笔者曾在前面说过,传统流派的仿真风格是非常倾向“调试”,for 自然而然就成为传统流派的爱将。

for 的常见用法如代码 4.3.6 所示:

```
1. begin
2.     for( i = 0; i < 128; i = i + 1 )
3.         ram[i] = i;
4.     for( i = 0; i < 128; i = i + 1 )
5.         $display( "ram[%d] = %d" , i, ram[i] );
6.     for( i = 0; i < 128; i = i + 1 )
7.         reg1 = #10 ram[i];
8.     .....
9. end
```

代码 4.3.6

如代码 4.3.6 所示,for 的行为会展示顺序语言满满的既视感。对此,笔者不禁会怀疑,这样还是仿真吗?不错,for 的副作用就是冲淡豆浆,好让豆浆不在是豆浆,因为代码 4.3.6 丁点时序香也没有,这种失去时序香的东西,已经不再是笔者梦寐以求的仿真。为此,笔者决定要复仇,誓死驱赶 for,抹杀 for!

建模之所以无法接受 for ... 其一 for 本质是顺序语言之余,其二建模必须有顺序结构支持 for 才得以运作。不管怎么样,既然我们已经知晓 for 的秘密,就算没有 for 我们也是

一样可以实现循环，因此仿顺序操作就诞生了。i 是一件有趣的工具，它除了指向这个，指向那个以外，只要充分使用 i 也能实现各种循环操作。

```
1. always ( posedge CLOCK )
2.     case ( i )
3.
4.         0,1,2,3,4,5,6,7:
5.             begin reg1 <= reg1 + 1' b1; i <= i + 1' b1; end
6.
7.     endcase
```

代码 4.3.7

代码 4.3.7 是非常简单的循环操作，只有将 i 指向多时钟操作，如代码 4.3.7 所示，reg1 一共递增 7 次。在此，代码 4.3.7 与代码 4.3.6 有明确的区别，代码 4.3.6 的 for 是无视时钟实现循环操作，换之代码 4.3.7 的 i 是根据时钟实现循环操作。相比之下，代码 4.3.7 才是香浓的豆浆，然而代码 4.3.6 仅是不是豆浆的豆浆。

好奇的朋友可能会问：“笔者，i 有没有办法实现多次数的循环？”，绝对有！我的朋友 ... 笔者曾经在章节 3 举例过，笔者也无妨重复解释。

```
1. always ( posedge CLOCK )
2.     case ( i )
3.
4.         0:
5.             if ( C1 == 8 ) begin C1 <= 4' d0; i <= i + 1' b1; end
6.             else begin reg1 <= reg1 + 1' b1; C1 <= C1 + 1' b1; end
7.
8.     endcase
9.
10. always ( posedge CLOCK )
11.     case ( i )
12.
13.         0,1,2,3,4,5,6,7:
14.             begin
15.                 reg1 <= reg1 + 1' b1;
16.                 if( C1 == 8 -1 ) begin C1 <= 4' d0; i <= i + 1' b1; end
17.                 else C1 <= C1 + 1' b1;
18.             end
19.
20.     endcase
```

代码 4.3.8

如代码 4.3.8 所示，这是两种不同风格的仿真操作，第 1~8 行是时钟概念比较松散的循

环操作；第 10~20 行则是针对紧密控时的循环操作。两者虽有明确的差异，但是两者都是根据时钟发生循环操作，有浓浓的时序香。反之 for 却是根据步骤或者仿真时间执行的循环操作，时序香非常稀淡。

如果 i 可以取代 for 循环，i 同样可以取代 while 循环，因为两者适用同样的道理。笔者是豆浆的爱好者，一生都在追求浓厚豆香的豆浆，实现循环一定要伴随满满的时序表现，不然这种循环不要也罢。不过，笔者不是恶魔，笔者也不会赶尽杀绝，笔者也为 for 留一条生路。

在前面，笔者说过系统函数如 \$display() 是时序的异存在，其实 for 也是同样的异存在。在笔者的眼中，for 会引起即时事件，举例而言。

```
1. always @ ( posedge CLOCK )
2.     case( i )
3.
4.         0:
5.         begin // 必须使用 = 赋值操作，配合 for 的即时事件
6.             for( x = 0; x < 8; x = x + 1 ) reg1 = reg1 + 1'b1;
7.             $display( "reg1 = 4'b%b",reg1 );
8.             i <= i + 1'b1;
9.         end
10.
11.     endcase
```

代码 4.3.9

代码 4.3.9 是一种比较独特的仿真风格，此刻 for 被认为是即时事件的触发者，不管 for 愿不愿意，for 都被时钟控制得死死。如代码 4.3.9 所示，步骤 0 实际上只是停留一个时钟而已，然而第 6 行的 for 既然执行 8 次循环操作，以致递增 reg1 八次。为了配合 for 所触发的即时事件，reg1 的必须使用 = 赋值操作符。笔者也曾经说过，即时事件是无视时钟的事件，不管第 6 行的 for 是循环执行 8 次，还是 9999 次，它都是在一个时钟内完成。

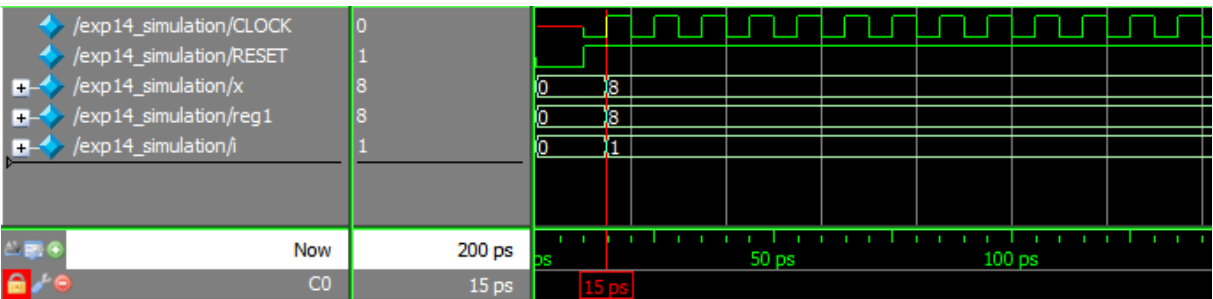


图 4.3.1 代码 4.3.9 的仿真结果 ( exp14\_simulation )。

简单完成 8 次循环操作以后，reg1 也取得即时值 8，然后再由第 7 行的 \$display() 函数

输出，最后第 8 行的 i 递增以示下一个步骤。图 4.3.1 是代码 4.3.9 的仿真结果，相对实现 exp14\_simulation。如图 4.3.1 所示，C0 指向时间点 T0，然而此刻的 x 输出即时值 8，而且 reg1 也输出即时值 8。这种仿真结果暗示，for 循环触发即时事件，受牵连的 reg1 同样也触发即时事件，两起事件都是发生在同样一个时钟内。换句话说，for 循环受限与时钟并且没有暴走。

代码 4.3.9 是笔者为 for 留下的仁慈，即使它脱离传统流派，它也能为美味的豆浆出一份力 ... 此刻 for 可以骄傲说道：“俺是触发即时事件，俺也有时序表现！”，这种共赢局面才是我们期望的结果。作为补充笔者还是要强调一下，for 对时序来说，它是异世界的存在，虽然仿真允许它们为时序出一份力。换做建模（实际建模），不管 for 再怎么努力，它始终无法得到认同，就算凑巧综合成功，我们也必须付出相应的代价。举例而言：

```
1. case(i)
2.
3.     0,1,2,3,4,5,6,7:
4.     begin reg1 <= reg1 + 1'b1; i <= i + 1'b1;
5.
6. endcase
7.
8. case(j)
9.
10.    0:
11.    begin for( x = 0; x < 8; x = x + 1'b1 ) reg2 = reg2 + 1'b1; i <= i + 1'b1; end
12.
13. endcase
```

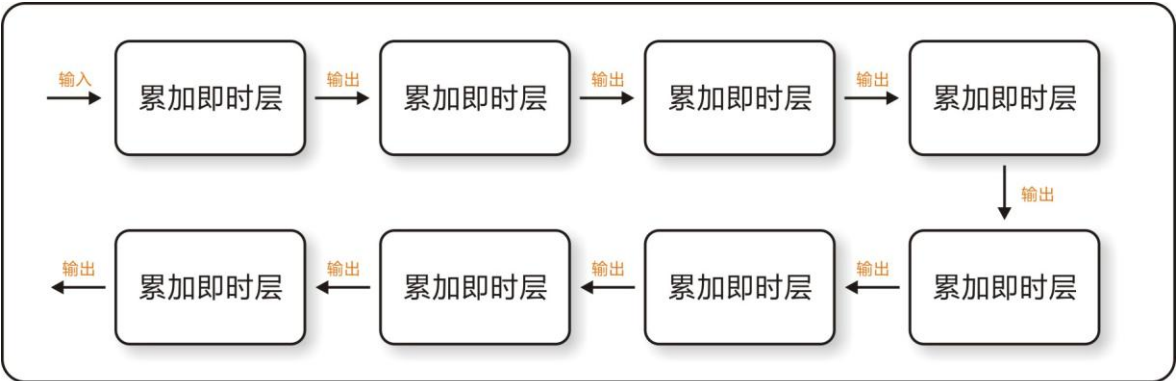
代码 4.3.10

代码 4.3.10 有两个过程，亦即 i（第 1~6 行）与 j（第 8~13 行）。过程 i 利用 8 个时钟递增 reg1 八次 ... 换之，过程 j 则是利用 for 递增 reg2 八次。reg1 与 reg2 虽有相同的操作目的，但是却有不同的操作过程，前者是利用时钟引发时间点事件，后者则是无视时钟引发即时事件。过程 i 无疑会综合成功，换之过程 j 虽然也会综合成功，不过两者是完全不同的综合结果。



图 4.3.2 过程 i 建模示意图。

在建模的角度上，过程 i 的建模十一图如图 4.3.2 所示，实际上它只有一个累加资源，然后利用时钟不断反馈输出，不断重复相同操作，直至满意结果为止。



累加即时模块

图 4.3.3 过程 j 建模示意图。

相比之下，过程 j 使用超过 1 个以上的累加资源，建模示意图如图 4.3.3 所示。因为 for 触发即时事件，而且 for 也重复递增结果八次，所以在人眼无法触及的情况下，8 个称为即时层的累加资源建立而成。输入结果每经过一个累加即时层就执行一次累加操作，直至 8 个即时层游走完毕。为了明确区分过程 i 与过程 j 的区别，笔者建立简单的比较表格（表格 4.3.2）。

表格 4.3.2 过程 i 与过程 j 的区别。

过程 \ 比较参数	时钟消耗	资源消耗	操作模式
过程 i	8	少	循环操作
过程 j	1	很多	即时操作

如表 4.3.2 所示，过程 i 消耗时钟多却消耗资源少；反之，过程 j 消耗时钟少却消耗资源多。反实际上，过程 i 与过程 j 之间的比较已经是“优化与平衡”的范畴，有时候建模必须衡量某种平衡点，如果 FPGA 设备的逻辑资源稀少，我们必须考虑善用时钟作为优化的手段；换之，如果操作速度作为优先，那么善用资源是一种比较适宜的优化手段。

笔者曾经说过，建模还有仿真之间的区别就在于“物理环境”还有“虚拟环境”而已。属于虚拟环境的仿真，FPGA 设备拥有无限的逻辑资源，所以 for 要循环多少次也没有问题，反之亦然。不管怎么样，我们还是把话题却换回来 ...

```
1. 'timescale 1ps/1ps
2. reg CLOCK;
3. initial CLOCK = 0;
4. forever #5 CLOCK = ~CLOCK;
5.
6. begin // 传统流派仿真手段
7.     #00 for ( x = 0 ; x < 8; x = x + 1 ) reg1 = reg1 + 1' b1;
8.     #10 for ( x = 0 ; x < 8; x = x + 1 ) reg2 = reg2 + 1' b1;
```



```

9.      #10 for ( x = 0 ; x < 8 ; x = x + 1 ) reg3 = reg3 + 1' b1;
10. end
11.
12. always @ ( posedge CLOCK ) // 笔者惯用仿真手段
13. case( i )
14.
15.     0:
16.         begin for ( x = 0 ; x < 8 ; x = x + 1 ) reg1 = reg1 + 1' b1; i <= i + 1' b1; end
17.     1:
18.         begin for ( x = 0 ; x < 8 ; x = x + 1 ) reg2 = reg2 + 1' b1; i <= i + 1' b1; end
19.     2:
20.         begin for ( x = 0 ; x < 8 ; x = x + 1 ) reg3 = reg3 + 1' b1; i <= i + 1' b1; end
21.
22. endcase

```

代码 4.3.11

我们足够仿真的真实面貌，其实 for 的使用可以多姿多彩，如代码 4.3.11 所示。第 6~10 行是传统流派的仿真手段，亦即使用仿真时间控制 for 循环执行递增操作。仿真时间为 0 的时候为 reg1 递增 8 次；仿真时间为 10 的时候为 reg2 递增 8 次；仿真时间为 20 的时候为 reg3 递增 8 次。

换之，第 12~22 是笔者惯用的仿真手段，假设每隔 10 个仿真时间恰好是 1 个时钟 ... 如代码 4.3.11 所示，第 10~15 行有 3 个步骤，每一个步骤停留一个时钟，而且每一个步骤也有使用 for 执行递增操作。步骤 0，for 为 reg1 递增 8 次，然后 i 递增以示下一个步骤；步骤 1，for 为 reg2 递增 8 次，然后 i 递增以示下一个步骤；步骤 2，for 为 reg3 递增 8 次，然后 i 递增以示下一个步骤。

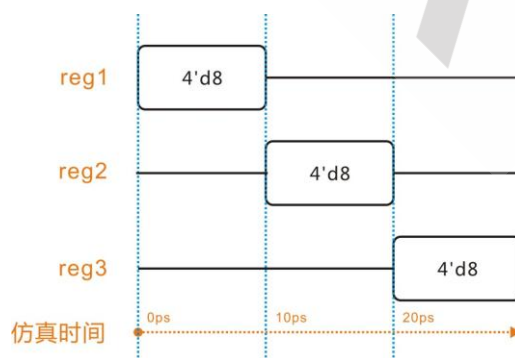


图 4.3.4 代码 4.3.11，第 6~10 行产生的时序。

图 4.3.4 是使用传统流派产生的时序，reg1 在 0ps 输出值 8，reg2 在 10ps 输出值 8，reg3 在 20ps 输出值 8。由于代码 4.3.4 第 6~10 是根据仿真时间输出结果，所以图 4.3.4 有没有 CLOCK 信号也无关紧要。换句话说，这是一个没有时钟概念，也没有任何时序表现的时序图，所以它不是笔者爱喝的豆浆。

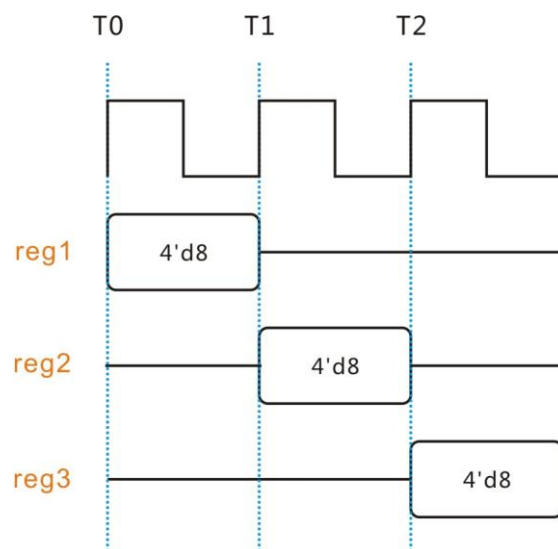


图 4.3.5 代码 4.3.11，第 12~22 行产生的时序。

换之，图 4.3.5 是经过笔者的仿真手段所产生的时序。如图 4.3.5 所示，reg1 在 T0 输出即时值 8，reg2 在 T1 输出即时值 8，reg3 在 T2 输出即时值 8。在此，for 不仅按照时钟工作，for 也有时序表现，所以这是一杯满满豆香的豆浆。当然，重点不仅而已，亮眼的同学一定会发现，如果实际的 FPGA 设备有足够的逻辑资源，其实这种仿真手段也适用建模。笔者是一名贯彻信念活下去的男人，同样手段适用不同环境就是节能本质最美的绝华。

最后总结道，笔者在这个小节引用两个常用的验证语言，亦即 \$display() 函数，还有 for 关键字来表示，验证语言的时序表现。传统流派是一种倾向调试的仿真手法，所以往往会埋没验证语言的时序表现。换之，笔者强烈建议使用时钟而不是仿真时间作为仿真和控制的手段，因为这样做有许多好处。其一，验证语言可以展示时序表现之余，偶尔同样的手段也使用两种环境。

Verilog 有一些比较暧昧的关键字，for 就是好例子，由于它们实在太难控制了，往往会在建模（实际环境）造成巨大的破坏（使用 for 会消耗大量逻辑资源）。因此 for 却被认为是验证语言的一伙，因为虚拟的仿真环境拥有无限的资源任由它消耗。此外，还有一些完全属于验证语言的关键字，如 fork ... join，forever 等，实际上它们是综合语言相对应的兄弟姐妹，例如：forever 与 always，begin ... end 与 fork ... join，至于使不使用它们是见仁见智的问题。

验证语言主要是系统函数占据多，因此许多系统函数有如 \$display() 函数一样触发即时事件。不过系统函数它们完全属于仿真的，所以综合（建模）的时候是绝对不允许系统函数出现。虽然系统函数看似很多很强大，其实系统函数存在一定风险，正如笔者所言，由于系统函数的本质太接近顺序语言了，如果用法不妥当会很容易抹杀时序的时序表现 ... 系统函数的关键字是美金\$。

验证语言除了系统函数占据多以外，其实预处理也有可观的数量，常见的预处理有

``timescale`, ``define` 等, **预处理是综合器 ( 编译器 ) 的工作**, 常常放在建模或者仿真的最开头。大多数的预处理都适用于建模与仿真, 举例而言 ``timescale` 写在仿真开头是声明时钟刻度。换之, 如果 ``timescale` 写在建模开头, 它就会被综合器 ( 编译器 ) 无视掉, 预处理的关键字符是上标点 ```。

不管怎么说, 这个章节只有抛砖引玉的作为而已, 更多更详细的验证语言, 读者必须自行需找其它参考。验证语言的关键字, 系统函数, 还有预处理, 由于它们比较偏向调试风格, 又或者本身就是活生生的顺序语言 ... 不过, 笔者为了养成统一的思维模式 ( 并行思维 ), 所以不怎么喜欢它们。虽然 “欢喜” 还有 “必要” 是不同的话题, 但是过多涉及它们无疑是给仿真带来极大的学习负担, 因此笔者建议[最小程度使用它们就好](#)。

---

又一次, 笔者在炎热的天空下四处跑动, 已经极限的笔者正在使命寻找城市的绿洲, 走着走着笔者来到一处树荫下, 歇气还不及几口, 旁边就有人叫道, 原来是一位卖豆浆的小贩正在向笔者哈拉。

“小哥, 今天的天气还真是热到过分”, 豆浆小贩说道。

“是呀 ... 我快要成为暑糕了”, 笔者回答道。

“小哥, 要不要来一杯清凉又美味的豆浆呢?”, 豆浆小贩示意道。

“哎呀, 真是雪中送炭呀老板! 给我特凉特大杯哪一种!”, 笔者兴奋道。

“好的, 这是为小哥特地准备的豆浆。”, 豆浆小贩笑道。

笔者眼也不眨一下就顺势接过豆浆小贩手中的杯子, 然后立即往口里送去 ...

“嗯唔! ? 老板这是什么豆浆, 实在太美味了 ... ”, 笔者惊讶道。

此刻, 诡异的事情发生了, 树荫下只有笔者一人而已, 不知什么时候那个豆浆小贩已经凭空消失 ... 一股时曾相似的既视感忽然穿过脑髓, 笔者不禁打了一个冷颤。

“见鬼了, 又是哪位豆浆小贩 ... ”, 笔者粗言道。

不过, 可怖的情绪却渐渐被口里那股浓浓的豆香滋润并且消去, 笔者带着满意的心情继续赶路 ...

## 4.4 激励文本的布局

激励是什么？根据笔者的理解，激励这词其实是“刺激”与“反应”的复合体，刺激表示输入，反应表示输出。所谓激励文本，就是所有“刺激与反应”的集中营 ... 简单点说，激励文本就是实现虚拟活动，也就是仿真环境。事实上，仿真环境是笔者的主观观念，笔者认为建模（实际建模）还有仿真（虚拟建模）本是同根，只有概念（环境）不同而已。

常规观念，亦即传统流派，它们认为建模（实际建模）还有仿真（虚拟建模）是不同平台的东西，所以两者之间可以使用不同的思维，模式还有手段。传统流派认识测试文本是一个测试作用的个体，而不是一座测试作用的环境，然而这个个体是偏向“调试”（顺序）的风格还有模式。

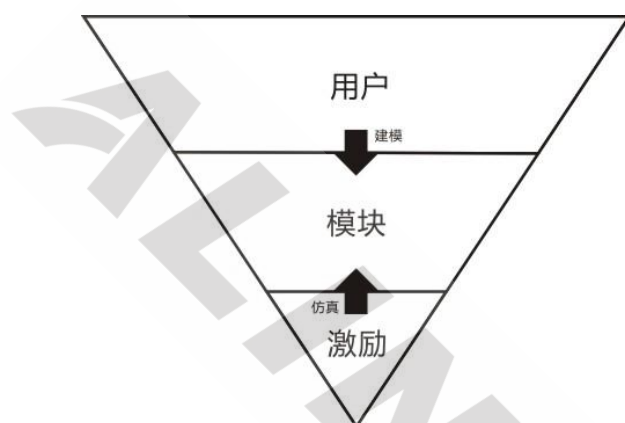


图 4.4.1 传统流派，用户，模块还有激励的关系图。

激励文本由于被传统流派视为个体，所以“用户（设计者）”“模块（仿真对象）”，“激励（激励文本）”，成为一种由上之下，倒金字塔的阶层关系图。如图 4.4.1 所示，设计者占据最高，而且也是分量最大，可见它有多么重要的地位，反之激励文本不仅占据最低而且分量也是最小，可想而知它的地位是多么卑贱。

形象点说，用户可以是主人 A，激励文本可以是奴隶 C，然而模块（仿真对象）可以是爱犬之类的宠物 B。主人 A 除非有命令告知奴隶 C，否则主人 A 与奴隶 B 平时是不相往来，说得难听一天就是主人 A 是宠物 B 的主人，宠物 B 是奴隶 C 的主人，所以主人 A 与奴隶 C 之间是没有任何纽带。

假设，主人 A 用千万黄金从极东买了一只宠物恐龙 B 回来，有一天主人 A 突然心血来潮想知道恐龙 B 的战斗表现，于是主人 A 会用奴隶 C 去测试恐龙 B。无情的主人 A 满脑子只在乎恐龙 B 的战斗表现而已，至于奴隶 C 的死活，主人 A 一律没有兴趣。换句话说，传统流派认识测试文本的是一位不值钱的个体，或者是一只实验性的小白鼠。

市场上，小白鼠是廉价的科学消耗品，价值和用完即丢的抹布一样。所以说，传统流派重视激励文本的程度是非常之低，结果它们不会花费而外的资源还有精力去维护激励文本。因为如此，激励文本内容相比模块内容（仿真对象）会是更加不堪入目的乱，乱到

让人抓狂又尖叫。

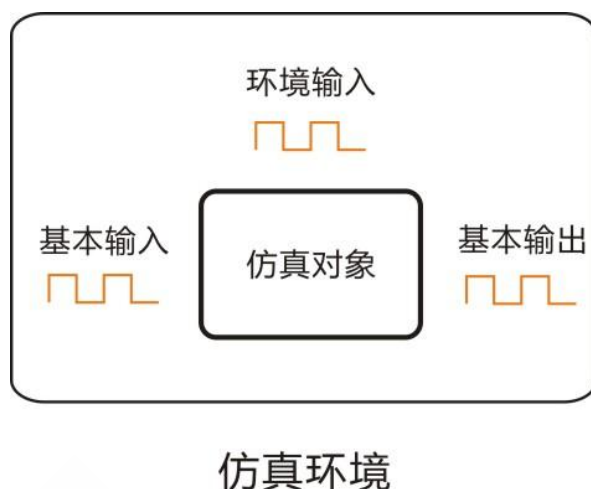


图 4.4.2 笔者观念中的仿真环境。

根据笔者的认识，激励文本不仅不是一个廉价的个体，而是一座非常贵重的仿真环境，然而我们就是创建这个环境的神。此外，仿真还有 3 个必须重视的结构性，其一是仿真对象的结构性，其二是仿真环境的结构性，其三是仿真过程的结构性，三者之间的结构性简单点说：

1. 仿真对象的结构性也是笔者时常挂在嘴边“模块的结构”，如低级建模，用法模板等。
2. 仿真过程是指基本输入，基本输出还有环境输入等内容。仿真过程所谓的结构性是指仿真过程的用法模板。
3. 仿真环境的结构性就是激励文本的布局，也是各种仿真过程在激励文本的位置。

其（一）没有什么好说，这是学习 Verilog 的基础，笔者已经在建模篇讲得很清楚。其（二）是其（一）的缩水，懂得其（一）自然也会晓得其（二）。其（三）是仿真的关键，笔者认为激励文本一个仿真环境，然而仿真过程还有仿真对象都是个体，仿真环境所谓的结构性是指，个体之间如何布局才能使得整体产生最大“表达能力”还有“清晰能力”。

有些同学可能会黯然笑道：“激励文本想怎样写就怎样写嘛，干嘛还要分那么多 ... 真是爱找麻烦的笔者呀。”，这位同学有所不知了，为什么人的屁股不是长在脑袋？如果屁股长在脑袋，大小便既不是要倒立不可？人是如此，激励文本也是如此，结构的重要性不管在什么方面上都有同样的道理。

其实，很久以前笔者就一直在冷嘲热讽传统流派的粗野，传统流派不会在乎结构性这种小细节。失去结构不仅会为前期建模带来困扰，同样，后期仿真也会带来诸多不便，由此可见结构性是多么重要。所以说，笔者实在不知道那些崇拜权威还有传统流派的人们是如何大小便的 ...

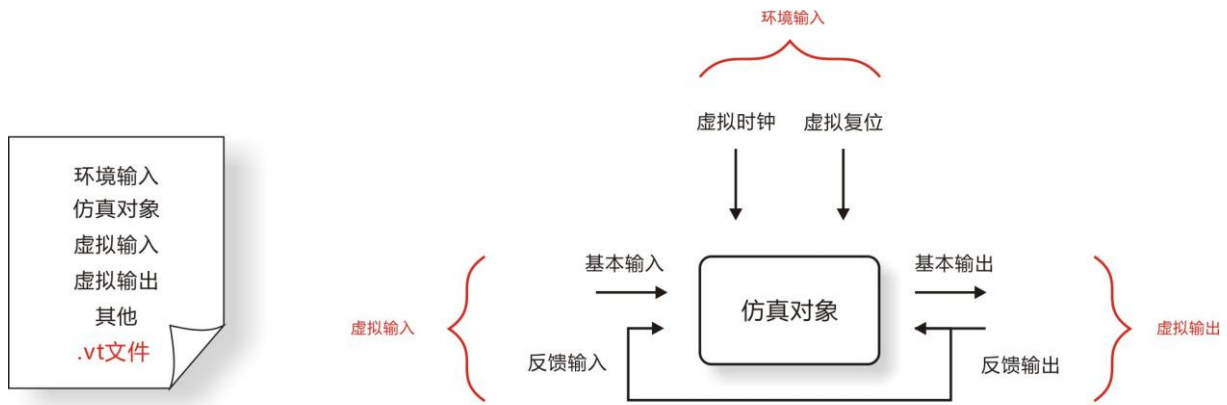


图 4.4.3 仿真环境的布局（结构性）。

图 4.4.3 曾经出现过许多章节，然而这张图却表示笔者认为仿真环境最有效的布局方式。如图 4.4.3 所示，激励文本亦即仿真环境可以分为 5 个个体（部分），亦即环境输入，仿真对象（实例化），虚拟输入，虚拟输出还有其他。环境输入可以是最简单的时钟信号还有复位信号的模拟输出；虚拟输入有基本输入，还有反馈输入；虚拟输出可以是基本输出，还有反馈输出。至于其他则是一些补充作用。

接下来让笔者用简单的实验来讲明一切，打开 exp15 ...

*selfadder\_funcmod.v*

```

1.  module selfadder_funcmod
2.  (
3.      input CLOCK, RESET,
4.      input Start_Sig,
5.      output Done_Sig,
6.      input [7:0]WrData,
7.      output [15:0]RdData
8.  );
9.      /*****/
10.
11.      reg [3:0]i;
12.      reg [15:0]Temp;
13.      reg isDone;
14.
15.      always @ ( posedge CLOCK or negedge RESET )
16.          if( !RESET )
17.              begin
18.                  i <= 4'd0;
19.                  Temp <= 16'd0;    // Sum of pratical product
20.                  isDone <= 1'b0;
21.              end
22.          else if( Start_Sig )

```



```

23.             case( i )
24.
25.                 0:
26.                     begin Temp <= 16'd0; i <= i + 1'b1; end
27.
28.                 1,2,3,4,5,6,7,8:
29.                     begin Temp <= Temp + WrData; i <= i + 1'b1; end
30.
31.                 9:
32.                     begin isDone <= 1'b1; i <= i + 1'b1; end
33.
34.                 10:
35.                     begin isDone <= 1'b0; i <= 4'd0; end
36.
37.             endcase
38.
39.             /***/
40.
41.             assign Done_Sig = isDone;
42.             assign RdData = Temp;
43.
44.             /***/
45.
46. endmodule

```

selfadder\_funcmod 是一个没有意义的功能模块，它的作用就是递增 WrData 八次而已。第 3~7 行是出入端的声明，第 4~5 行的 Start\_sig 与 Done\_Sig 表示它是一个仿顺序性质的模块；第 11~13 行是相关寄存器的声明，i 用于指向时钟还有步骤（当然也包含过程），Temp 用作暂存空间，isDone 用来反馈完成。第 18~20 行是相关的复位操作；第 22~37 是核心功能。步骤 0 是用来初始化相关的寄存器；步骤 1~8 是八次的递增操作；步骤 9~10 是用来反馈完成信号。第 41~42 行是相关的输出驱动。

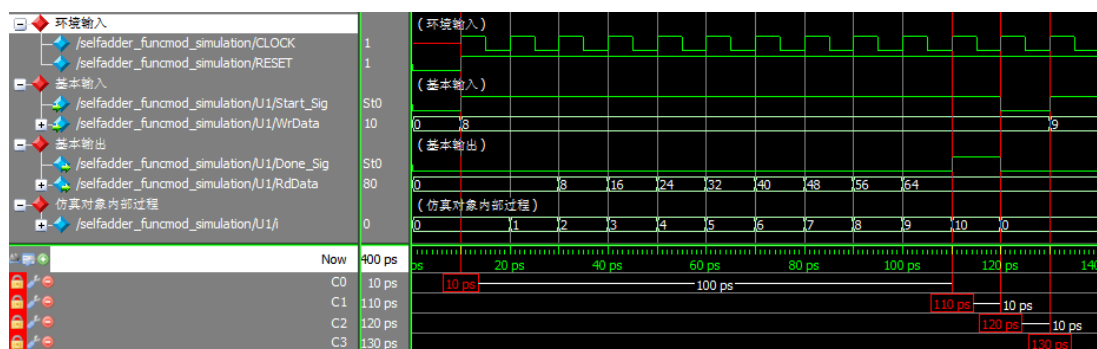


图 4.4.4 selfadder\_funcmod 产生的理想时序。



如图 4.4.4 所示,这是 selfadder\_funcmod 产生的理想时序图,此图也表示该模块的一次性操作过程。C0 指向 T0,此刻如果笔者拉高 Start\_Sig 又为 WrData 输入 8,根据理想时序的理论,数据之间传输至少需要一个时钟,因此在下一个时钟的上升沿该模块就立即运行,时间大约是 20ps 的时候。该模块在步骤 0 ( 20ps ) 初始化 Temp,所以输出没有变化;步骤 1 ( 30ps ) 执行第一次递增,结果输出未来值 8;步骤 2 ( 40ps ) 执行第二次递增,结果输出未来值 16;其它以此类推直到步骤 8 ( 100ps ) 的时候,八次递增已经完成,结果输出未来值 64;

步骤 9~10( 110ps~120ps )是完成信号的产生,结果 Done\_Sig 拉高一个时钟,也是 C1~C2 指向的地方。因此该模块整体经过时间是 10ps~120ps( 步骤 0~步骤 10 ),一共消耗 110ps 时间,再假设 1 个时钟为 10ps 的话,那么一次性的递增操作需要消耗 11 个时钟。如何解读图 4.4.4,理想时序的时间点事件是关键,不相信的读者可以一个时钟一个时钟计算看看是不是 i 是否正确指向时钟呢?

接着让我们来看看这家伙的仿真环境(激励文本)到底是如何编写的!?

*selfadder\_funcmod.vt*

```
1. 'timescale 1 ps/ 1 ps
2. module selfadder_funcmod_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.     reg Start_Sig;
7.     reg [7:0]WrData;
8.     wire Done_Sig;
9.     wire [15:0]RdData;
10.
```

第 1 行是时钟刻度声明最小单位为 1ps/1ps;第 4~9 行是相关的寄存器还有连线声明,它们分别对应仿真对象 selfadder\_funcmod 的出入端,由于仿真环境没有实际的输入引脚还有输出引脚,所以用 reg 作为输入引脚的配置, wire 作为输出引脚的配置。最后稍微注意一下地 2 行的命名方式,笔者习惯为仿真环境的后缀名取为 simulation,前缀名这是仿真对象的名字。

```
11.
12.     /*****/
13.
14.     initial
15.     begin
16.         RESET = 0; #10; RESET = 1;
17.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
18.     end
19.
```

第 14~18 行是环境输入的声明，也是模拟时钟信号还有复位信号。第 14 行的 initial 表示编译的时候，第 16 行的 RESET 为初值 0，然后第 17 行的 CLOCK 为初值 1；接着，第 16 行的 RESET 会延迟拉低 10 个时钟刻度之后又拉高，第 17 行的 CLOCK 则会每隔 5 个时钟刻度取反。第 14~17 行有些读者可能看不明白，内容都是我们之前所学过的东西，不过只是换个形式而已，结果如下：

```
initial begin RESET = 0; CLOCK = 1; end
begin #10 RESET = 1; end
begin forever #5 CLOCK = ~CLOCK;
```

```
20.  /*****/
21.
22.     selfadder_funcmod U1
23.     (
24.         .CLOCK(CLOCK),
25.         .RESET(RESET),
26.         .Start_Sig(Start_Sig),
27.         .Done_Sig(Done_Sig),
28.         .WrData(WrData),
29.         .RdData(RdData)
30.     );
31.
```

第 22~30 行是仿真对象的实例化，内容非常直接，读者自己看着办吧。

```
32.  /*****/
33.
34.     reg [3:0]i;
35.
36.     always @ ( posedge CLOCK or negedge RESET )
37.         if( !RESET )
38.             begin
39.                 i <= 4'd0;
40.                 Start_Sig <= 1'b0;
41.                 WrData <= 8'd0;
42.             end
43.         else
44.             case( i )
45.
46.                 0:
47.                     if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
48.                     else begin WrData <= 8'd8; Start_Sig <= 1'b1; end
```

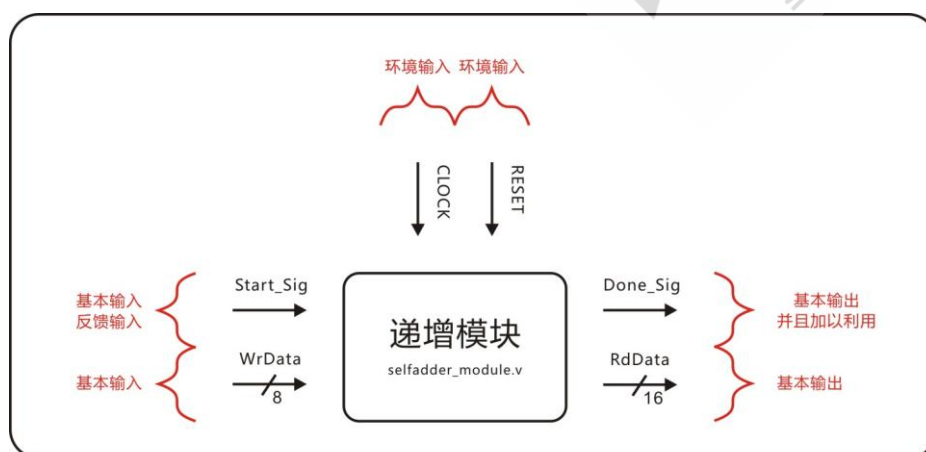
```

49.
50.             1:
51.             if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
52.             else begin WrData <= 8'd9; Start_Sig <= 1'b1; end
53.
54.             2:
55.             if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
56.             else begin WrData <= 8'd10; Start_Sig <= 1'b1; end
57.
58.             3:
59.             begin i <= i; end
60.
61.         endcase
62.
63.         /*****/
64.
65. endmodule

```

第 36~61 行是虚拟输入的激励内容，其中也包括基本输入还有反馈输入。首先第 39~41 行是相关的复位操作，第 44~59 行是虚拟输入的过程（第 36~61 实际上与仿真对象的核心功能拥有相似的结构性，其实这是善用用法模板的结果）。步骤 0 是为仿真对象的 WrData 输入 8 值，步骤 1 则是输入 9 值，步骤 2 则是输入 10 值，步骤 3 则是结束操作（停留）。

为进入解释仿真结果之前，再让笔者好好说明一下仿真环境的结构性。激励文本第 12~18 行是环境输入的部分；第 20~30 行是仿真对象实例化的部分；第 34~61 行则是虚拟输入的部分。在此，好奇的同学会问：“虚拟输出在哪里呢？”



selfadder\_funcmod\_simulation.vt

图 4.4.5 selfadder\_funcmod 仿真环境的布局。

如图 4.4.5 所示，根据该仿真环境的布局方式，CLOCK 信号与 RESET 信号属于环境输

入，Start\_Sig 信号与 WrData 信号属于虚拟输入，其中 Start\_Sig 兼为基本输入还有反馈输入，具体内容往后继续。虚拟输出的 Done\_Sig 还有 RdData 都是基本输出，而且两者都由仿真对象 selfadder\_module 产生以后直接投射在波形图当中。

所以说，我们没有必要在激励文本中多此一举，创建多余的 Done\_Sig 与 RdData 虚拟输出。此外，读者必须注意一下 Start\_Sig 与 Done\_Sig 实际上是一种问答信号，因此 Start\_Sig 需要根据 Done\_Sig 的反馈状态再一次决定拉高又或者拉低 Start\_Sig，所以 Start\_Sig 除了第一次的基本输入之余，还要根据 Done\_Sig 产生另一个虚拟输入。

```
46.          0:
47.          if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
48.          else begin WrData <= 8'd8; Start_Sig <= 1'b1; end
```

接着让我们稍微切换步骤 0 ... 一开始的时候，由于第 47 行的 if 条件为成立（仿真对象还没有一次性的递增操作），结果第 48 行的代码优先执行。此刻，基本输入 WrData 与 Start\_Sig 就开始产生，Start\_Sig 拉高以示仿真对象使能开始工作，WrData 则是递增操作所需的输入。

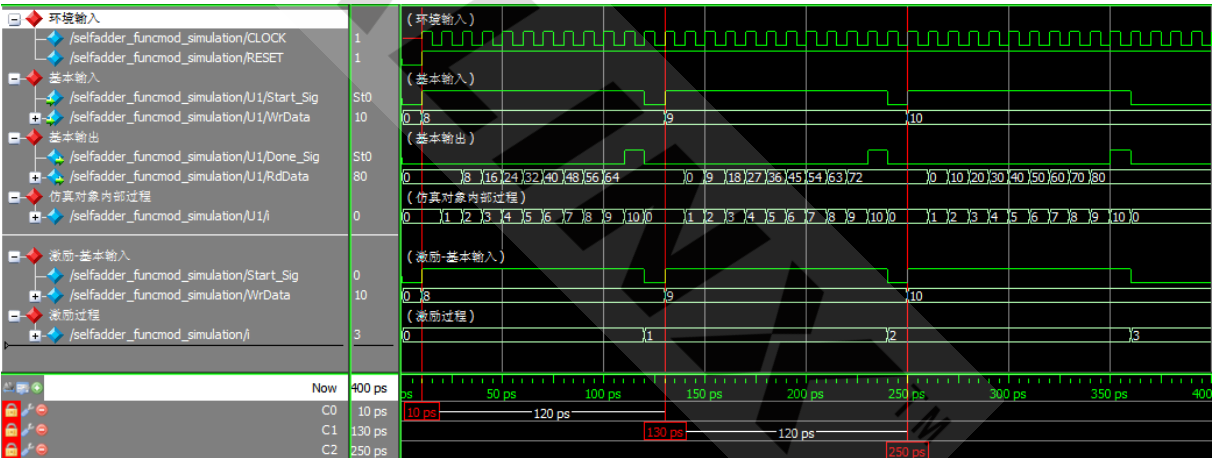


图 4.4.6 selfadder\_funcmod\_simulation 的仿真结果（exp15）。

图 4.4.6 是仿真结果，其中光标 C0 分别指向递增模块三次性的递增操作。如图 4.4.6 所示，笔者为了清晰波形图，稍微分类一下信号。CLOCK 与 RESET 是环境输入；Start\_Sig 与 WrData 是虚拟输入；Done\_Sig 与 RdData 是虚拟输出；至于 i 除了指向仿真对象的内部过程以外还有指向虚拟输入的激励过程。

一开始的时候（C0 指向的地方），步骤 0 为 WrData 输入 8 并且拉高 Start\_Sig，然后仿真对象就会在下一个时钟沿开始工作。仿真对象用了 11 个时钟完成递增操作并且反馈完成信号以示一次性的操作结束。步骤 0 则会根据 Done\_Sig 的结果拉低 Start\_Sig（反馈输入）不使能仿真是对象，然后 i 递增以示下一个虚拟输入的产生。虚拟输入产生 3 次，因此 Temp 会有 3 种基本输出，亦即 64,72 还有 80。

=====

在这里，有些同学可能会抓头喊道，自己无法完全解读图 4.4.6 的仿真结果 ... 这是当然的，此刻笔者只是简单演示一下仿真环境的布局方式而已，我们还没有深入联系仿真结果，激励内容，还有仿真对象等，每个时钟的实际活动 ... 简单点说，**我们还没有准备好将时序活动（仿真结果）与代码（仿真对象与激励文本）之间的联系并且做出解析。**所以读者不能够完全解读图 4.4.6 一点也不奇怪，关于这一点，笔者会在往后讲解。

不过，任性的同学可能会闹脾气道：“不管嘛！我就是要虚拟输出嘛！”，好啦好啦，别闹脾气就是了，笔者服输了，打开 exp16 ...

```
64.     reg [3:0]j;
65.     reg FlagA;
66.
67.     always @ ( posedge CLOCK or negedge RESET )
68.         if( !RESET )
69.             begin
70.                 j <= 4'd0;
71.                 FlagA <= 1'b0;
72.             end
73.         else
74.             case( j )
75.
76.                 0:
77.                     FlagA <= ~FlagA;
78.
79.             endcase
```

exp16 相比 exp16 的激励文本——selfadder\_funcmod\_simulation（仿真环境），笔者在虚拟输入的下面添加虚拟输入。其中，笔者在第 64~65 行声明寄存器 j 用于控制（指向）虚拟输出的产生过程，然而 FlagA 是没有意义的基本输出。如第 76~78 行所示，FlagA 只会毫无意义的翻来翻去而已 ...

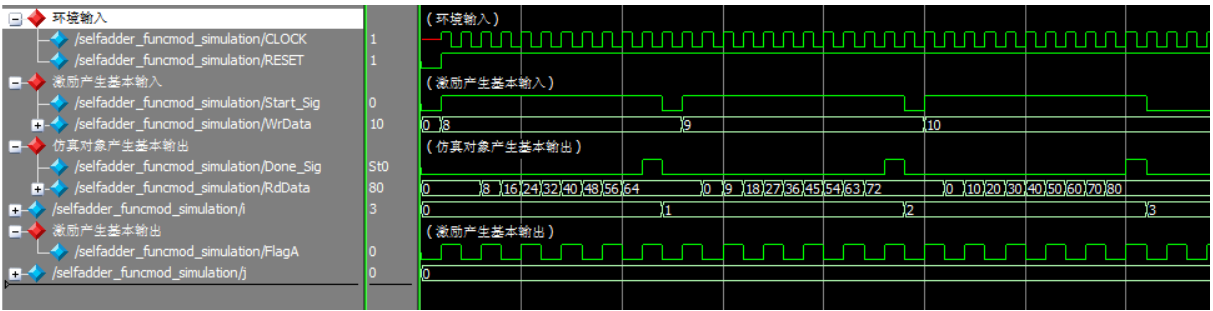
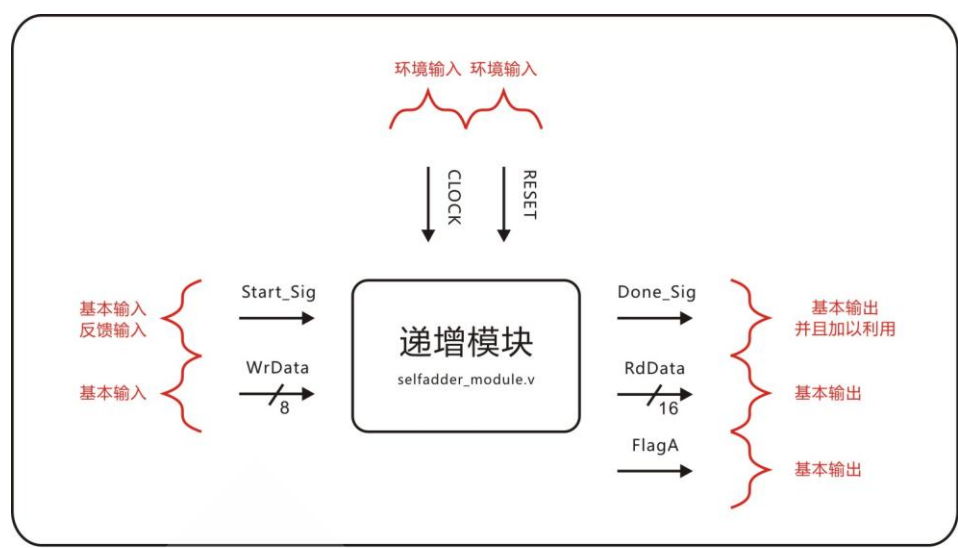


图 4.4.7 exp16 的仿真结果。

图 4.4.7 是添加虚拟输出以后的仿真结果，图 4.4.7 相比图 4.4.6 多了一个 FlagA 信号。

如图 4.4.7 所示，FlagA 在整个时序过程只会翻来覆去而已 ... 多么无聊的家伙。



selfadder\_funcmod\_simulation.vt

图 4.4.8 exp16 仿真环境（激励文本）的布局。

图 4.4.8 相比图 4.4.5，在仿真对象的下面多了一个基本输出 FlagA，不过不同的是 FlagA 信号不是仿真对象产生的输出，而是激励文本产生的输出。

最后笔者可以这样结论道：

到目前为止，我们可能还无法看道，仿真环境经过结构性的布局之，后到底凸显那种优越？不过不管怎么样，有结构性的激励文本域没有结构性的激励文本一定存在明显的差距，因为**没有结构的激励文本比起有结构的激励文本一定会非常破烂不堪（乱）**。此外，这个章节笔者也经由 selfadder\_funcmod 表示，仿真对象的结构性（模块结构），仿真过程的结构性。

仿真对象的结构性就是 selfadder\_funcmod 模块本身的内容，那是基础中的基础。笔者在 selfadder\_funcmod 采用仿顺序操作的用法模板，此而 i 指向时钟之余又指向步骤（当然也包括过程）。此外，仿真过程的结构性则是 ... 笔者用 i 指向虚拟输入的产生过程，用 j 指向虚拟输出的产生过程。事实上，仿真过程的结构形式非常相似模块的结构，或者说是模块结构的简化版，因为**两者都是采用同样的用法模板**。

4.5 仿真模型与反馈输出①



图 4.5.1 仿真环境的布局。

未进入本章之前，我们再来简单复习一下仿真环境的布局方式。如图 4.5.1 所示，激励文本也是仿真环境，位于顶层的当然莫属环境输入，环境输入模拟最基本也是最重要的时钟信号还有复位信号。位于环境输入的下面是仿真对象，仿真对象可以经由模块实例化而成，仿真对象也可以直接在仿真环境中建立。

接着是虚拟输入，虽然环境输入还有虚拟输入都是“输入”，不过不如环境输入有份量。虚拟输入有两个基本分类，亦即基本输入，还有反馈输入。虚拟输入的后面是虚拟输出，虚拟输出也有两个基本分类，亦即基本输出与反馈输出。基本输出一般都是仿真对象产生的反应，很少人为添加在激励文本当中，例如实验 exp15 的 RdData 信号与 Done\_sig 信号。不过，我们也可以自行添加在激励文本当中，例如实验 exp16 的 FlagA 就是如此。

除此之外，虚拟输出还有一种令人厌烦的反馈输出，也是这个章节要探讨的东西。其实，笔者曾经在其它章节稍微讨论过它，不过目的是用来解释 i 为什么要指向过过程，而没有过多深入。那么笔者再一次正式提问“**什么又是反馈输出呢？**”



图 4.5.2 仿真模型①。

如图 4.5.2 所示，这是仿真模型①，也是最基本的仿真模型，亦即激励内容产生虚拟输入刺激仿真模块，仿真模块接受刺激以后就会产生相关的反应，然后直接将结果放映在 wave 界面上，在这种情况下，虚拟输入只有基本输入，虚拟输出也只有基本输出而已。应用仿真模型①的仿真对象一般都是左耳进右耳出的单纯功能，常见的例子有门级逻辑仿真。



仿真模型①是参考书曝光率最强，也是最粗糙的仿真模型，一般上我们**无法在它的身上发现一些如：结构性或者时序表现等细节**。仿真模型①**比较依赖仿真时间**，而且仿真风格也很“调试化”。仿真模型①虽然是最基础也是最常用的仿真模型，不过它的能力却非常有限，单纯的它实际上是无法满足千奇百怪的仿真要求，为了弥补不足，其它仿真模型也相续面世。



图 4.5.3 仿真模型②。

图 4.5.3 是仿真模型②的示意图，仿真模型②相较仿真模型①，它多了更多细节。如图 4.5.3 所示：

- 1. 激励内容产生基本输入刺激仿真对象，仿真对象产生反应以后便将基本输出投射在 wave 界面上。
- 2. 激励内容产生基本基本输入刺激仿真对象，仿真对象产生反应以后将基本输出反馈给激励内容，然后激励内容再产生反馈输入进一步刺激仿真对象。



图 4.5.3 等价的仿真模型②。

图 4.5.3 是等价的仿真模型②，如图 4.5.3 所示：

- 3. 除了仿真对象产生基本输出以外，激励内容也产生基本输出并且投射在 wave 界面。
- 4. 激励内容之间也可以相互刺激。

仿真模型②基于仿真模型①之后**扩展更多细节**，仿真模型②使用的频密性不仅比仿真模型①还高，而且仿真模型②也能兼容仿真模型①，仿真模型②的应用例子有算法模块，问答式模块等**功能稍微多样化的模块**。然而，悲观而言，仿真模型②拥有更多数量的信号，而且方向性也非一处，此外过程（激励内容）也很多。

因此，我们必须采取有结构性的环境布局（仿真环境的结构性），有结构性的激励内容（仿真过程的结构性），有时序表现的理想时序。一般上，仿真模型②只会使用仿真时间模拟环境输入，**绝对不会使用仿真时间控制激励内容** ... 反之，仿真模型②会非常乐意使用时钟控制一切信号。换句话说，门级以外的仿真对象，绝对有理由应有仿真模型②，传统的仿真手段绝对会用到手残。

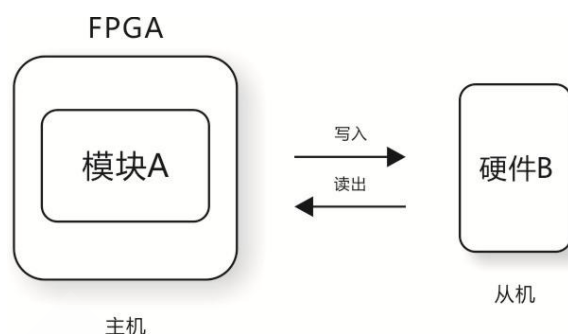


图 4.5.4 FPGA 驱动硬件。

仿真对象很多时候不仅仅是为了观察输出而已，仿真对象还必须与实际硬件发生互动。如图 4.5.4 所示，假设笔者为 FPGA 创建模块 A 用作驱动硬件 B，其中 FPGA 是主机，硬件 B 是从机，而且 FPGA 又是两方访问，换句话说 FPGA 除了向硬件 B 发生写入动作意外，FPGA 也会为硬件 B 执行读出操作。

俗语有云，无穴不来风，事出必有因，由于仿真模型①与仿真模型②无法实现上述内容，因此仿真模型③面世了。虽说仿真模型③的诞生是为了仿真实际硬件，实际上仿真模型③也兼容仿真模型②与仿真模型①。



图 4.5.5 仿真模型③。

图 4.5.5 是仿真模型③的示意图，相较其它仿真模型，仿真模型多了一个虚拟硬件。仿真模型①与②的仿真对象要么就是讲基本输出投射在 wave 界面上，要么就是将基本输出反馈给激励内容。仿真模型③，仿真对象会产生基本输出刺激虚拟硬件，虚拟硬件接受刺激以后除了直接将反应（基本输出）投射在 wave 界面之余，虚拟硬件也会将反应反馈给仿真对象，此刻这种输出方式称为“反馈输出”。



图 4.5.6 仿真模型③，虚拟硬件等价仿真对象。

如果根据等价关系去分析图 4.5.5，虚拟按硬件谓是另一个仿真对象，结果如图 4.5.6 所示。不过，**一般没有人会特意为虚拟硬件建模**，并且在仿真环境当中实例化成为另一个仿真对象 ... 好奇的同学可能会怀疑，笔者是否又在偷懒？别误会，如果虚拟硬件是功能复杂的 IC 的话，假设是串行存储器 AT24CXX 好了，试问同学是否有信息建模而成呢？答案当然是否定的，这种事情世上也只有傻子去干而已。

说来非常惭愧，笔者就是那个傻子 ... 要建模一块虚拟硬件，说实话那种程度的活儿，会耗死笔者的小命，幸好笔者迷途知返，不然笔者早就变成一具尸体。根据节能的角度而言，**我们不会为了测试实际硬件的部分功能，结果特意从轮子开始创建整个虚拟硬件**，就算能力允许，本质也不允许笔者这样做，因为这种做法太没有效率了 ... 为此，**我们要测试那个功能就模拟那个功能即可。**

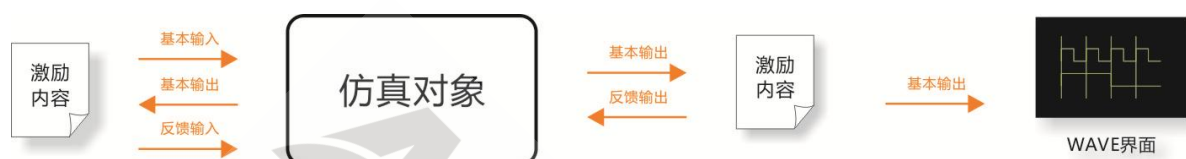


图 4.5.7 仿真模型③，激励内容等价仿真对象（虚拟硬件）。

图 4.5.6 是另外一种等价的仿真模型③，它没有虚拟硬件，虚拟硬件也不是仿真对象，取而代之它是一段激励内容。在此，我们会用“信号”来描述虚拟硬件的部分功能，然而描述过程就写在激励内容当中，至于那个“信号”又是什么信号呢？呵呵，想知道吗？笔者就大发慈悲告诉你们吧！

从第一章至这个章节以来，笔者一直在强调“早期有很好建模，后期就有好仿真” ... 在建模的阶段，笔者建议使用用法模范统一建模风格之余，笔者也建议使用 i 指向模块的内部过程（指向步骤）。虽然这些做法在大体上是为了稳固的结构，增强内容的清晰度，还有提高模块的表达能力 ... 然而这些所作所为却为仿真产生意想不到的正面效果，这种偶然究竟是一场意外的幸运，还是一连串“神的恶作剧”？

正如笔者曾在第三章节 3.5 所言那样，既然 i 有能力指向模块的内部过程，同样也表示 i **有能力标示模块的运行状态**。然而，这些运行状态对仿真而言可是**非常贵重的“描述材料”**，我们可以用这些信息来**描述虚拟硬件的部分功能**。理论上的确如此，不过在此之前我们必须做足一切事先的准备，亦即**规划激励文本的布局，创建可以最大程度支持仿真模型③，并且兼容仿真模型②还有仿真模型①的仿真环境。**

不管怎么样，笔者还是用实验来说话吧，打开 exp17：

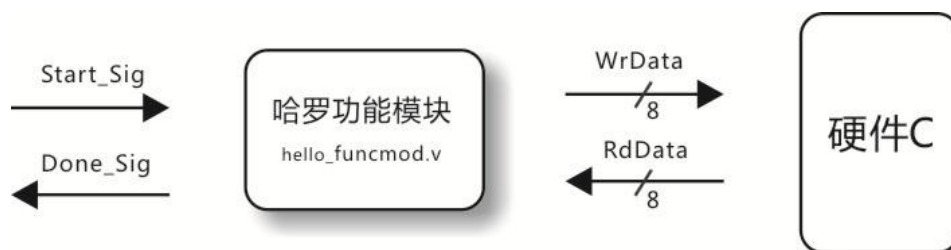


图 4.5.8 哈罗功能模块的建模图。

如图 4.5.8 所示，假设有一个名为哈罗的功能模块，笔者用它驱动硬件 C。哈罗功能模块的左方有 Start\_Sig 信号与 Done\_Sig 信号充当开关，右方的 WrData 信号与 RdData 信号主要是访问硬件 C。

*hello\_funcmod.v*

```

1.  module hello_funcmod
2.  (
3.      input CLOCK, RESET,
4.      input Start_Sig,
5.      output Done_Sig,
6.      output[7:0]WrData,
7.      input [7:0]RdData,
8.      output [3:0]SQ_i
9.  );
10.  /*****/
11.
12.      reg [3:0]i;
13.      reg [7:0]rData;
14.      reg isDone;
15.
16.      always @ ( posedge CLOCK or negedge RESET )
17.          if( !RESET )
18.              begin
19.                  i <= 4'd0;
20.                  rData <= 8'd0;
21.                  isDone <= 1'b0;
22.              end
23.          else if( Start_Sig )
24.              case( i )
25.
26.                  0:
27.                      begin rData <= 8'hAA; i <= i + 1'b1; end
28.

```

```

29.         1:
30.         if( RdData == 8'hBB ) begin i <= i + 1'b1; end
31.
32.         2:
33.         begin rData <= 8'hCC; i <= i + 1'b1; end
34.
35.         3:
36.         if( RdData == 8'hDD ) begin i <= i + 1'b1; end
37.
38.         4:
39.         begin rData <= 8'hEE; i <= i + 1'b1; end
40.
41.         5:
42.         if( RdData == 8'hFF )begin i <= i + 1'b1; end
43.
44.         6:
45.         begin isDone <= 1'b1; i <= i + 1'b1; end
46.
47.         7:
48.         begin isDone <= 1'b0; i <= 4'd0; end
49.
50.     endcase
51.
52.     /*****/
53.
54.     assign Done_Sig = isDone;
55.     assign WrData = rData;
56.
57.     /*****/
58.
59.     assign SQ_i = i;
60.
61.     /*****/
62.
63. endmodule

```

地 3~8 行是出入端的声明，其中 SQ\_i 信号时是将内部过牵引出去；第 12~13 行是相关寄存器的声明，第 19~21 行则是复位操作；第 23~50 行是该模块的核心功能，步骤 0 输出数据 8'hAA 步骤 1 等待数据 8'hBB 步骤 2 发送数据 8'hCC 步骤 3 等待数据 8'hDD；步骤 4 发送数据 8'hEE；步骤 5 等待数据 8'hFF；步骤 6~7 则是反馈完成信号。看着看着，是不是觉得该模块的功能很简单呢？

还未仿真之前，先让我们好好假设一下硬件 C 的基本功能：

- (一) 接收数据 8'hAA , 反馈数据 8'hBB;
- (二) 接收数据 8'hCC , 反馈数据 8'hDD;
- (三) 接收数据 8'hEE , 反馈数据 8'hFF ;

然后我们再给予上述所有信息创建仿真模型③。



图 4.5.9 exp17 的仿真模型③。

图 4.5.9 就是 exp17 的仿真模型③，Start\_Sig 信号由激励内容产生，Done\_Sig 信号还有 WrData 信号则由仿真对象产生，至于 RdData 信号是反馈输出，它由硬件 C (虚拟硬件) 产生。笔者曾在前面说过，不管硬件 C 的功能再怎么简单，我们都不会特意从轮子开始创建整个虚拟硬件，相反我们与采取替代又等价的手段。



图 4.5.10 exp17 等价的仿真模型③。

如图 4.5.10 所示，我们采用另外一种等价的仿真模型③，其中虚拟硬件则有一段激励内容取代，此外仿真对象也将内部过程牵引出来。当我们了解这些信息以后，我们就可以开始创建仿真环境了。

*hello\_funcmod\_simulation.vt*

```
1. timescale 1 ps/ 1 ps
2. module hello_funcmod_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.     reg Start_Sig;
7.     reg [7:0]RdData;
8.     wire [7:0]WrData;
```

```

9.    wire Done_Sig;
10.   wire [3:0]SQ_i;
11.
12.   /*****
13.
14.   initial
15.   begin
16.       RESET = 0; #10; RESET = 1;
17.       CLOCK = 1; forever #5 CLOCK = ~CLOCK;
18.   end
19.
20.   /*****
21.
22.   hello_funcmod U1
23.   (
24.       .CLOCK(CLOCK),
25.       .RESET(RESET),
26.       .Start_Sig(Start_Sig),
27.       .Done_Sig(Done_Sig),
28.       .WrData(WrData),
29.       .RdData(RdData),
30.       .SQ_i( SQ_i )
31.   );
32.
33.   /*****
34.
35.   reg [3:0]i;
36.
37.   always @ ( posedge CLOCK or negedge RESET )
38.       if( !RESET )
39.           begin
40.               i <= 4'd0;
41.               Start_Sig <= 1'b0;
42.           end
43.       else
44.           case( i )
45.
46.               0:
47.                   if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
48.                   else begin Start_Sig <= 1'b1; end
49.
50.               1:
51.                   begin i <= i; end

```



```

52.
53.             endcase
54.
55.     /***/
56.
57.     always @ ( posedge CLOCK or negedge RESET )
58.     if( !RESET )
59.     begin
60.         RdData <= 8'd0;
61.     end
62.     else
63.         case( WrData )
64.
65.             8'hAA: RdData = 8'hBB;
66.             8'hCC: RdData = 8'hDD;
67.             8'hEE: RdData = 8'hFF;
68.
69.         endcase
70.
71.     /***/
72.
73. endmodule

```

如代码 `hello_funcmod_simulation` 所示，第 1 行是时钟刻度（最小仿真时间）的声明，结果为 1ps/1ps；第 2 行则是声明仿真环境——`hello_funcmod_simulation`；第 4~10 行是相关的寄存器与连线声明，由于仿真环境当中没有实际的链接，因此 `reg` 充当输入链接，`wire` 充当输出链接。第 14~18 行是环境输入，`RESET` 信号拉低 10ps，`CLOCK` 信号的周期则是 10ps。

第 22~31 行是仿真对象——哈罗功能模块的实例化；第 37~53 行是虚拟输入的激励内容；第 57~69 行则是虚拟输出的激励内容。首先让我们来瞧瞧虚拟输入的产生过程，根据图 4.5.10 所示，仿真对象的左边只有 `Start_Sig` 信号与 `Done_Sig` 信号而已，然而第 37~53 行的激励内容则是负责这些信号的基本输入还有反馈输入。

我们知道哈罗功能模块有仿顺序建模的外形，因此虚拟输入程仅是在步骤 0（第 48 行），单纯地拉高 `Start_Sig` 信号（基本输入），接着又根据 `Done_Sig` 信号的反馈结果（第 47 行）拉低 `Start_Sig` 信号（反馈输入）。步骤 0 完成后，`i` 递增以示下一个步骤，然后虚拟输入就结束过程（第 47 行）。

第 57~69 行是虚拟输出的激励内容 ... 在此，笔者先举例反馈输出最简单的方法。第 60 行是寄存器 `RdData` 的复位操作，根据图 4.5.9 所示，`RdData` 是虚拟引脚给仿真对象的输入链接，可是仿真环境又没有实际的链接，结果用 `reg` 替代。第 63~69 行是虚拟输出的激励内容，笔者采用最简单的 `case ... endcase` 用作反馈输出。根据硬件 C 的基本功

能：

- (一) 如果 WrData 的输入数据是 8'hAA，RdData 就反馈输出数据 8'hBB；
- (二) 如果 WrData 的输入数据是 8'hCC，RdData 就反馈输出数据 8'hDD；
- (三) 如果 WrData 的输入数据是 8'hEE，RdData 就反馈输出数据 8'hFF；

结果第 63~69 行完全对应上述要求。

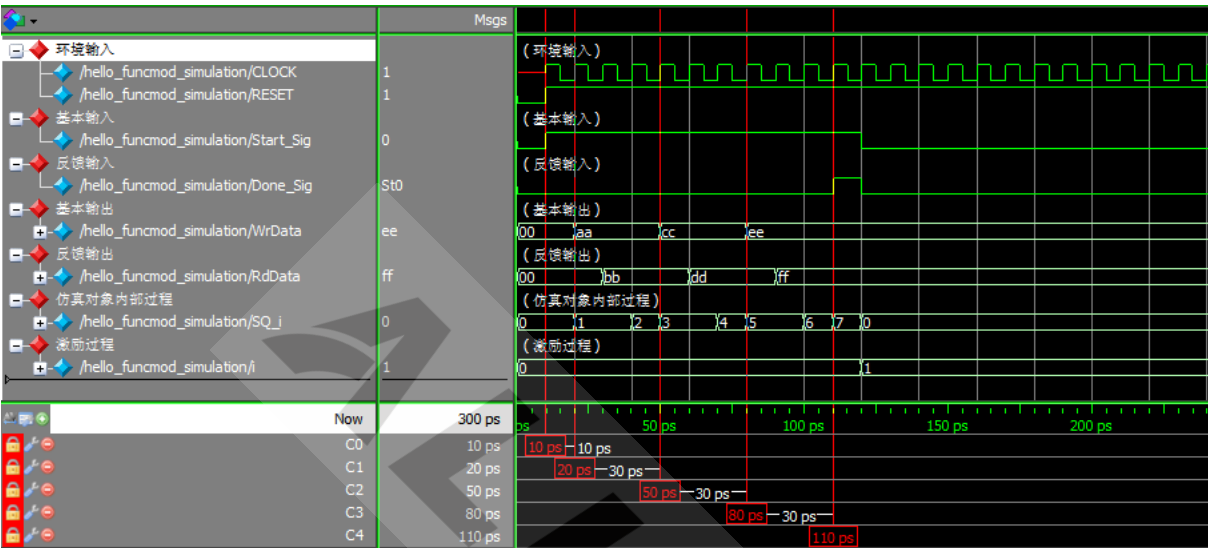


图 4.5.11 exp17 的仿真结果。

图 4.5.11 是 hello\_funcmod\_simulation 的仿真结果，其中就有光标 C0~C4 分别指向各种过程，为了清晰 wave 界面笔者也稍微根据环境布局分类信号，结果如图 4.5.11 所示。C0 指向整个激励过程的开始，首先虚拟输入会拉高 Start\_Sig 信号，然后就停留在原步等待仿真对象反馈完成信号（注意最下面的指向信号 i）。

C1 指向仿真对象开始操作的时间点，此刻仿真对象会经由 WrData 信号输出数据 8'hAA。在下一个时钟，虚拟输出就会根据 WrData 的输出结果，再经由 RdData 信号反馈数据 8'hBB；又在下一个时钟，仿真对象接受反馈数据以后将 i 递增以示下一个步骤（注意仿真对象的指向信号 SQ\_i）。

C2 指向仿真对象经由 WrData 信号输出第二个数据 8'hCC，然而虚拟输出也再下一个时钟接收然后产生经由 RdData 反馈数据 8'hDD。又在下一个时钟，仿真对象接收反馈数据 8'hDD 以后，将 i 递增以示下一个步骤。

C3 指向仿真对象正在发送第三个数据的时候，此刻仿真对象会经由 WrData 信号输出数据 8'hEE，虚拟输出则再下一个时钟接收数据 8'hEE 并且产生反馈数据 8'hFF。过后，仿真对象将在下一个时钟接收反馈数据 8'hFF，然后将 i 递增以示下一个步骤。C4 则是指向仿真对象产生完成信号产生的时候，Done\_Sig 信号是用来通知虚拟输入一次性的操作过程已经圆满结束。

因此，虚拟输入在下一个时钟接收到 Done\_Sig 信号以后，立即拉低 Start\_Sig 信号（反馈输入）表示结束使能仿真对象，并且将 i 递增递增以示下一个步骤，此刻，指向信号 i 的未来值成为 1。

```
65.          8'hAA: RdData = 8'hBB;  
66.          8'hCC: RdData = 8'hDD;  
67.          8'hEE: RdData = 8'hFF;
```

hello\_funcmod\_simulation 的仿真结果大致上就是这种感觉，有些同学可能会非常好奇为什么代码的第 65~67 行是用 = 赋值操作符，而不是 <= 赋值操作符。其实，这是笔者的小习惯，硬件在理想的状态下当然是有多快就多快将输出反馈出去 ... 结果而言，= 产生的即时值相比 <= 产生的未来值，即时值比较快。

这个小节，笔者除了解释 3 种仿真模型以外，笔者也简单举例一下仿真模型③的使用方法，此外笔者也顺便演示一下反馈输出的产生过程。不过很遗憾的是，指向信号 SQ\_i 在此只是用来表示仿真对象的内部过程而已，并没有实际参与虚拟输出的活动。事实上，硬件 C 在 exp17 的功能要求还是非常简单的程度，所以指向信号 SQ\_i 用不着派上用场。往后笔者会继续提高硬件 C 功能要求，以致虐待读者直到需要它 ... 嘻嘻嘻。

4.6 仿真模型与反馈输出②

笔者在上一个章节除了演示仿真模型③的用法以外，笔者也简单举例反馈输出的产生过程。这个章节，我们会继续未完的故事，并且好好认识一下，环境布局（仿真环境的结构性），指向信号，还有仿真模型之间是如何合作无间，以致满足虚拟硬件不断提高的功能要求。



图 4.6.1 exp17 的仿真模型③。



图 4.6.2 exp17 等价的仿真模型③。

前情提要，我们需要创建一座仿真环境用作测试哈罗功能模块与硬件 C 之间的沟通，结果如图 4.6.1 所示。不过，笔者不建议创建完成的虚拟硬件，为此笔者选择等价的替代方法，如图 4.6.2 所示，笔者仅在激励内容模拟硬件 C 的部分功能而已。此外，硬件 C 的第一次功能要求如下：

- （一）接收数据 8'hAA，反馈数据 8'hBB；
- （二）接收数据 8'hCC，反馈数据 8'hDD；
- （三）接收数据 8'hEE，反馈数据 8'hFF；

由于第一次的功能要求过于简单，所以指向信号 SQ\_i 没有出场的机会。结果当天晚上，SQ\_i 跑来向笔者哭诉说它是多么期待当天的演出。无奈之下，笔者必须不断提高硬件 C 的功能要求，以致满足指向信号 SQ\_i 的心愿为止。硬件 C 第二次的功能要求如下：

- （一）第一次接收数据 8'hAA，反馈数据 8'hBB；
- （二）第二次接收数据 8'hAA，反馈数据 8'hDD；
- （三）第三次接收数据 8'hAA，反馈数据 8'hFF；

因此如此，我们需要重新修改一下 exp17 的内容，打开 exp18：

# hello\_funcmod.v

```
1.  module hello_funcmod
2.  (
3.      input CLOCK, RESET,
4.      input Start_Sig,
5.      output Done_Sig,
6.      output[7:0]WrData,
7.      input [7:0]RdData,
8.      output [3:0]SQ_i
9.  );
10.  /***/
11.
12.      reg [3:0]i;
13.      reg [7:0]rData;
14.      reg isDone;
15.
16.      always @ ( posedge CLOCK or negedge RESET )
17.          if( !RESET )
18.              begin
19.                  i <= 4'd0;
20.                  rData <= 8'd0;
21.                  isDone <= 1'b0;
22.              end
23.          else if( Start_Sig )
24.              case( i )
25.
26.                  0:
27.                      begin rData <= 8'hAA; i <= i + 1'b1; end
28.
29.                  1:
30.                      if( RdData == 8'hBB ) begin i <= i + 1'b1; end
31.
32.                  2:
33.                      begin rData <= 8'hAA; i <= i + 1'b1; end
34.
35.                  3:
36.                      if( RdData == 8'hDD ) begin i <= i + 1'b1; end
37.
38.                  4:
39.                      begin rData <= 8'hAA; i <= i + 1'b1; end
40.
41.                  5:
```

```

42.             if( RdData == 8'hFF )begin i <= i + 1'b1; end
43.
44.             6:
45.             begin isDone <= 1'b1; i <= i + 1'b1; end
46.
47.             7:
48.             begin isDone <= 1'b0; i <= 4'd0; end
49.
50.         endcase
51.
52.         /***/
53.
54.         assign Done_Sig = isDone;
55.         assign WrData = rData;
56.
57.         /***/
58.
59.         assign SQ_i = i;
60.
61.         /***/
62.
63.     endmodule

```

上面代码是根据硬件 C 第二次功能要求修改的结果，如代码 `hello_funcmod` 所示，步骤 0，步骤 2 与步骤 4，WrData 的输出结果全为 8'hAA。

*hello\_funcmod\_simulation.vt*

```

1.  `timescale 1 ps/ 1 ps
2.  module hello_funcmod_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg Start_Sig;
7.      reg [7:0]RdData;
8.      wire [7:0]WrData;
9.      wire Done_Sig;
10.     wire [3:0]SQ_i;
11.
12.     /***/
13.
14.     initial
15.     begin

```

```

16.         RESET = 0; #10; RESET = 1;
17.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
18.     end
19.
20.     /*****
21.
22.     hello_funcmod U1
23.     (
24.         .CLOCK(CLOCK),
25.         .RESET(RESET),
26.         .Start_Sig(Start_Sig),
27.         .Done_Sig(Done_Sig),
28.         .WrData(WrData),
29.         .RdData(RdData),
30.         .SQ_i( SQ_i )
31.     );
32.
33.     /*****
34.
35.     reg [3:0]i;
36.
37.     always @ ( posedge CLOCK or negedge RESET )
38.         if( !RESET )
39.             begin
40.                 i <= 4'd0;
41.                 Start_Sig <= 1'b0;
42.             end
43.         else
44.             case( i )
45.
46.                 0:
47.                     if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
48.                     else begin Start_Sig <= 1'b1; end
49.
50.                 1:
51.                     begin i <= i; end
52.
53.             endcase
54.
55.     /*****
56.
57.     reg [3:0]j;
58.

```



```

59.     always @ ( posedge CLOCK or negedge RESET )
60.         if( !RESET )
61.             begin
62.                 RdData <= 8'd0;
63.                 j <= 4'd0;
64.             end
65.         else
66.             case( j )
67.
68.                 0:
69.                     if( WrData == 8'hAA ) begin RdData = 8'hBB; j <= j + 1'b1; end
70.
71.                 1:
72.                     if( WrData == 8'hAA ) begin RdData = 8'hDD; j <= j + 1'b1; end
73.
74.                 2:
75.                     if( WrData == 8'hAA ) begin RdData = 8'hFF; j <= j + 1'b1; end
76.
77.                 3:
78.                     j <= j;
79.
80.             endcase
81.
82.             /*****/
83.
84.     endmodule

```

同样，根据硬件 C 的第二次功能要求，激励文本的虚拟输出也作出相关的修改。首先让我们好好思考一下，硬件 C 的第二次功能要求都是根据接收结果 8'hAA 作出反应，然而比较麻烦的是，数据 8'hAA 有分为第一次，第二次还有第三次。为此，相似 ep17 当中的 case ... endcase 用法实在不妥，因此我们必须采取其它手段。

如代码第 68~80 行所示，笔者应用仿顺序操作的用法模板，然后再根据步骤将反馈数据 RdData 按次序作出 3 次输出。先是步骤 0 检测 WrData 信号是否 8'hAA？是就反馈数据 8'hBB，然后将 j 递增以示下一个步骤，同样行为也发生在步骤 1 与步骤 2 的身上。此外，笔者也事先作好保险措施，将 RdData 改为即时值（注意赋值操作符）。万事虽然已经具备，不过第 68~80 行的虚拟输出是否可以发挥预期的效果，说实在笔者真心不知道 ...

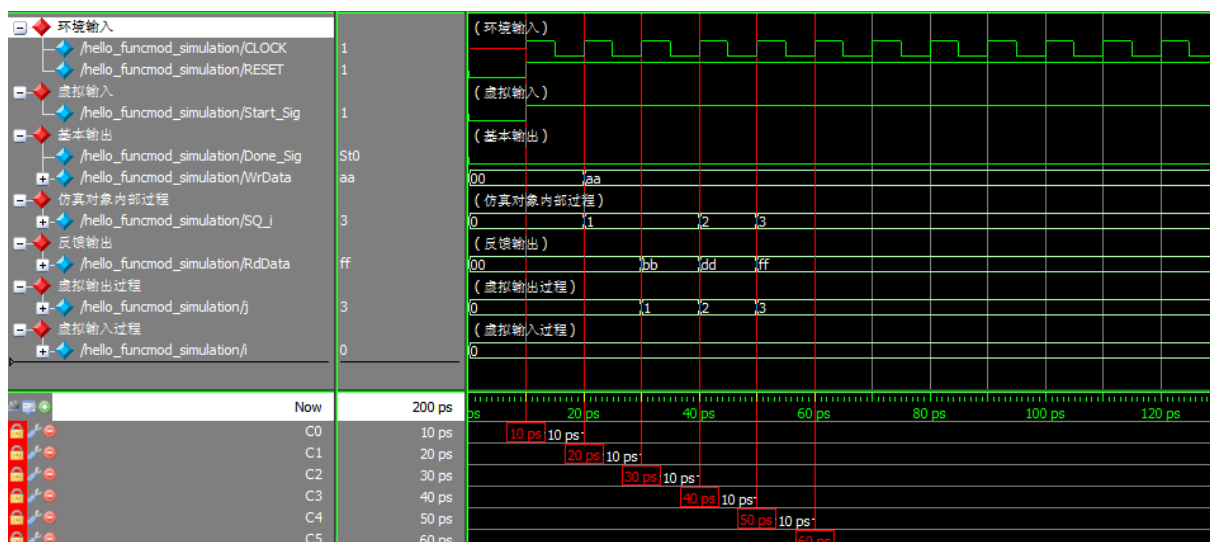


图 4.6.3 exp18 的仿真结果。

图 4.6.3 是 hello\_funcmod\_simulation 的仿真结果。啊！我的天呀 ... 读者看见什么嘛？如图 4.6.3 所示，如果 Done\_Sig 信号没有拉高就表示激励过程失败了，亦即图 4.6.3 不是预期所要的仿真结果，究竟问题是发生在哪里呢？让我们一起瞧瞧吧。

C0 指向整体激励的开始，此刻虚拟输入拉高 Start\_Sig 以示仿真对象开始工作。C1 指向仿真对象开始工作的时候，仿真对象现在步骤 0 经由 WrData 信号输出结果 8'hAA，然后将 i 递增以示下一个步骤。在下一个时钟（C2 指向的地方），虚拟输出接收结果并且经由 RdData 反馈数据 8'hBB，然后将 j 递增以示下一个步骤。

C3 指向的地方，恰好是仿真对象停留在步骤 1 的时候，此刻仿真对象读取 RdData 的过去值为 8'hBB，if 条件成立，然后将 i 递增以示下一个步骤。那么问题发生了！由于 WrData 的输出结果始终是 8'hAA，再同样的时刻（C3 指向的地方）虚拟输出步骤 2 的 if 条件也成立了，因为它读到 WrData 的过去值是 8'hAA，结果它经由 RdData 信号反馈数据 8'hDD，并且将 j 递增以示下一个步骤。

C4 指向的地方是仿真对象停在步骤 2 的时候，此刻仿真对象还在准备经由 WrData 信号更新结果而已，由于 WrData 之前还有之后的结果都一样，所以波形图上的 WrData 没有发生任何变化，不过实际上仿真对象确实已经将它更新。完后，仿真对象将 i 递增以示下一个步骤。糟了糟了，问题像雪球越滚越大了。

在同一个时候（C4 指向的地方），虚拟输出判断 WrData 的过去值为 8'hAA，因此反馈数据 8'hFF，并且将 j 递增以示下一个步骤。C5 指向的地方是也是仿真对象停留在步骤 3 的时候，此刻仿真对象还在傻傻等待接收反馈数据 8'hDD，但它不知道反馈数据 8'hDD 老早已经跑去火星，亦即它已经错过数据 8'hDD。呜呜呜 ... 就这样，仿真对象永远都傻傻般停留在步骤 3 等待反馈数据 8'hDD 的到来 ...

好奇的同学可能会这样问道：“是不是仿真对象还有虚拟输出之间出现不协调的时序沟通？结果数据接收错乱了？”的确如此 ... 好奇的同学可能又会反问道：“这起意外究竟是谁的错？仿真对象，还是虚拟输出呢？”类似问题我们不能随便妄下定论，实际上是

谁都没错，是谁都有责任 ...

这种情况就像发生车祸一般，当务之急不是相互指着是非而是优先送急伤者才是。虚拟输出相比仿真对象，虚拟输出更像是受伤的一方，我们会[优先修改虚拟输出](#)，再者才考虑仿真对象。在此，仿真对象内部的指向信号——SQ\_i 它就派上用场了，前几回由于SQ\_i 没有出演机会，它整天都是以泪洗脸，此刻就是它大发光彩的时候，好让累积许久的能量一次性爆发出来。因此，请打开 exp19：

哈罗功能模块再 exp19 当中没有发生任何修改，所以笔者就不重复了。

*hello\_funcmod\_simulation.vt*

```
1.  `timescale 1 ps/ 1 ps
2.  module hello_funcmod_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg Start_Sig;
7.      reg [7:0]RdData;
8.      wire [7:0]WrData;
9.      wire Done_Sig;
10.     wire [3:0]SQ_i;
11.
12.     /******
13.
14.     initial
15.     begin
16.         RESET = 0; #10; RESET = 1;
17.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
18.     end
19.
20.     /******
21.
22.     hello_funcmod U1
23.     (
24.         .CLOCK(CLOCK),
25.         .RESET(RESET),
26.         .Start_Sig(Start_Sig),
27.         .Done_Sig(Done_Sig),
28.         .WrData(WrData),
29.         .RdData(RdData),
30.         .SQ_i( SQ_i )
31.     );
```

```

32.
33.  /***/
34.
35.  reg [3:0]i;
36.
37.  always @ ( posedge CLOCK or negedge RESET )
38.      if( !RESET )
39.          begin
40.              i <= 4'd0;
41.              Start_Sig <= 1'b0;
42.          end
43.      else
44.          case( i )
45.
46.              0:
47.                  if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
48.                  else begin Start_Sig <= 1'b1; end
49.
50.              1:
51.                  begin i <= i; end
52.
53.          endcase
54.
55.  /***/
56.
57.  always @ ( posedge CLOCK or negedge RESET )
58.      if( !RESET )
59.          begin
60.              RdData <= 8'd0;
61.          end
62.      else
63.          case( SQ_i )
64.
65.              0: RdData = 8'hBB;
66.              2: RdData = 8'hDD;
67.              4: RdData = 8'hFF;
68.
69.          endcase
70.
71.  /***/
72.
73.  endmodule

```

exp19 的激励文本，笔者稍微修改了一下第 57~69 行的虚拟输出。笔者抛弃步骤 j，取而代之就是使用指向仿真对象内部过程的信号 SQ\_i。exp19 的虚拟输出与 exp17 的虚拟输出有点相似，不过 case ... endcase 之间的判断信号是 SQ\_i。我们知道仿真对象在步骤 0 的时候发送第一次数据 8'hAA，结果第 65 行是针对步骤 0 的反馈操作，RdData 赋值为 8'hBB；仿真对象在步骤 2 发送第二次数据 8'hAA，所以虚拟输出再第 66 行为 RdData 赋值 8'hDD；仿真对象在步骤 4 发送第三次数据 8'hAA，因此虚拟输出再第 67 行为 RdData 赋值 8'hFF。

完后再让我们仿真一次看看：

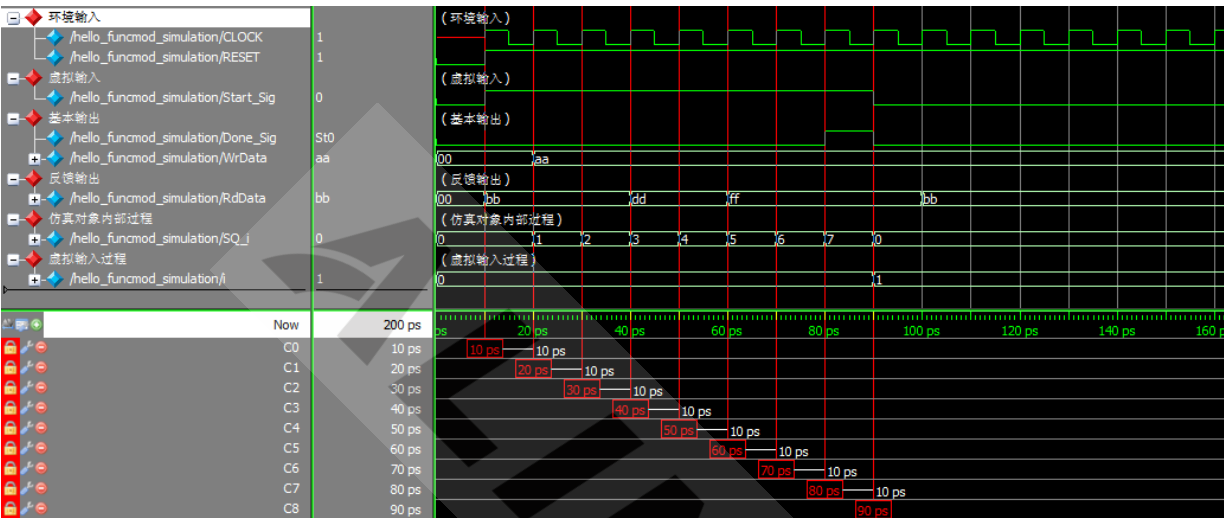


图 4.6.4 exp19 的仿真结果。

图 4.6.4 是 exp19 的仿真结果 ... 如图 4.6.4 所示，Done\_Sig 信号可以完美输出这就表示仿真结果已经符合预期的期待。C0 指向的地方是整个激励过程的开始，此刻虚拟输入开始拉高 Start\_Sig 以示使能仿真对象。在同一时刻，虚拟输出也产生反应，然后立即输出反馈数据 8'hBB。

C1 指向的地方是仿真对象开始操作的时候，此刻步骤对象在步骤经由 WrData 信号输出 8'hAA。由于仿真对象还是停留在步骤 0（注意 SQ\_i 的过去值），因此虚拟输出没有产生任何变化。C2 指向仿真对象停留在步骤 1 的时候，仿真对象检测 RdData 的过去值，结果满足 if 条件，然后将 i 递增以示下一个步骤。

C3 指向的地方是仿真对象停留在步骤 2 的时候，此刻仿真对象决定输出第二次输出 8'hAA，完后将 i 递增以示下一个步骤。再同一个时刻，虚拟输出检测 SQ\_i 的过去值是 2，结果产生反应输出反馈数据 8'hDD。

C4 指向的地方却是仿真对象停留在步骤 3 的时候，此刻它检测 RdData 的过去值为 8'hDD，因此 if 条件成立，i 递增以示下一个步骤。之余虚拟输出则没有什么活动发生。C5 指向的地方是仿真对象停留在步骤 4 的时候，此刻他决定为信号 WrData 输出 8'hAA，然后再将 i 递增以示下一个步骤。再同一个时刻，虚拟输出检测 SQ\_i 的过去值为 4，它产生反应并且输出反馈数据 8'hFF。

C6 指向的地方是仿真对象停留在步骤 4 的时候，此刻它检测到 RdData 的过去值为 8'hFF，事后将 i 递增以示下一个步骤。换之，此刻的虚拟输出正闲着没事干。至于 C7 与 C8 指向的地方则是仿真对象的步骤 6~7，并且也是完成信号产生的步骤（注意 SQ\_i 的过去值）。虚拟输入再 C8 指向的时刻检测 Done\_Sig 的过去值是 1，因此它决定拉低 Start\_Sig（反馈输入）不使能方针对象，然后再将 i 递增以示下一个步骤。

虽然虚拟输出再 C8 之后的时钟里还在根据 SQ\_i 结果产生反应，不过这是不管紧要的小，因为我们已经确定在 C0~C8 之间，仿真对象可以正确的完成一次性的操作，这样就已经足够了。在此，读者是否渐渐发觉指向信号 SQ\_i 是如何巧妙描述硬件 C 的部分功能呢？总结 exp17~19 的仿真结果，我们可以结论道 ...

虽然虚拟输出可以根据仿真对象输出的 WrData 产生相关的反馈输出，不过这是一种非常被动的方式，而且方法也有诸多局限，exp18 就是最好的证明，因为 exp18 无法实现硬件 C 的第二次功能要求。为此，仿真对象内部的指向信号——SQ\_i 就会凸显自己的重要性，SQ\_i 除了指向过程以外，SQ\_i 也表示仿真对象当前的操作状态，结果我们可以根据 [SQ\\_i 指向的内部状态实装虚拟输出，让它成为描述虚拟硬件的原材料](#)。

除了 SQ\_i 以外，仿真对象的使能信号 [Start\\_Sig](#) 也能成为描述虚拟硬件的原材料，就让笔者稍微扩充一下 exp19 的虚拟输出：

```
57.     always @ ( posedge CLOCK or negedge RESET )
58.         if( !RESET )
59.             begin
60.                 RdData <= 8'd0;
61.             end
62.         else if( Start_Sig )
63.             case( SQ_i )
64.
65.                 0: RdData = 8'hBB;
66.                 2: RdData = 8'hDD;
67.                 4: RdData = 8'hFF;
68.
69.             endcase
```

大致的感觉如上述代码所示，笔者在第 62 行用 Start\_Sig 信号为虚拟输出添加一行 if 条件，就这样 ... 虚拟硬件也有使能的功能了。

虽然章节 4.5 与 4.6 我们都是谈论仿真模型还有反馈输出的故事而已，不过眼睛犀利的朋友，隐约可以察觉这一切的一切必须是仿真环境拥有结构性作为前提，此外仿真对象还有仿真过程的结构性也很重要，不然的话我们就不能轻易应用仿真模型还有实现反馈输出了。



## 总结

不知不觉之间笔者已经将第四章搞定了，这个章节算是仿真的第二核心吧。第四章的一开始我们就在讨论验证语言的定义，如果没有深入一切，而且一切又是任由传统流派道听途说，结果我们就会觉得验证语言似乎很难很强大。事实上，验证语言只是占据仿真的小部分而已，而且常用的验证语言不过也是小猫两三只而已。我们真是被传统流派吓倒了。

根据笔者的认识，建模（实际建模）还有仿真（虚拟建模），本是同根，差异就有概念（环境）而已。实际上，两者可以共享同样的手段还有思维，建模虽然拥有实际的输入还有输出，不过建模会受限于实际环境。反之，仿真虽然没有物理的局限性，但是仿真必须模拟实际环境中存在的东西，如时钟信号还有复位信号就是最好的例子。

为此，我们必须使用验证语言模拟时钟信号还有复位信号，从这点上 ... 我们就可以看出验证语言的作用不过是[仿真的辅助性行为](#)而已，绝对没有传统流派所说般那样严重。不过真正让人头疼的问题不是验证语言的使用方法，而是如何展示验证语言的时序表现。传统流派的仿真手段非常倾向调试风格之余，它们也非常依赖仿真时间 ... 实际上，这种仿真手段还是停留在门级模块的测试而已。

这种仿真手段不仅有自身局限性，也不怎么适合功能操作复杂的仿真对象。同样，这种仿真手段其实也是抹杀时序表现最大的凶手，因而产生豆浆不是豆浆，时序不是时序的问题。有人可能会问，时序有没有表现真的那么重要吗？时序失去如果时序表现，时序终究再也不是时序，而是一副没有意义的白纸而已，因为“精华”早已不复存在，这种感觉好似没有味道的豆浆一样。

系统函数，还有预处理占据验证语言的大多数，然而真正对仿真有用的关键字用 5 只手指也能表示出来。一般上笔者不怎么推荐过度学习验证语言，因为许多验证语言不仅没有实际的用处，而且又难学，此外还有验证语言是在重复综合语言的功能而已。常规上，验证语言的本质是非常接近顺序语言，为此过多使用它们很难让我们适应仿真应有的并行模式。

笔者虽然讨厌验证语言，不过我们还是需要最小程度利用它们。学习验证语言的关键就是理解它们的时序表现，换句话说就是如何使用时钟控制它们而且不然它们暴走。验证语言有没有时序表现，其实这是非常主观的问题，笔者认为有是因为笔者有追求美味豆浆的原始冲动，至于他人就见仁见智了。

此外，我们还有谈论激励文本的布局，亦即仿真环境的结构性。布局其实是一种自然艺术，好的环境布局就会产生正面发的效果，反之亦然。理解笔者的读者自然不会觉得奇怪，因为笔者是一位重视结构至死的男人，笔者认为前期有好建模，后期就有好仿真，此外笔者也一样重视仿真对象的结构性还有仿真过程的结构性。为什么笔者会如此强调结构的重要性呢？



仿真环境的结构性也好，仿真过程的结构性也好，这一切一切都是为了支撑仿真模型。根据笔者的理解，仿真模型有 3 种，其一是最基础也是门级模块仿真应用最多的模型。仿真模型②相较仿真模型①拥有更大的功能扩张，期间结构性的支撑也会略显重要。最后一个仿真模型也是仿真模型③。它的诞生是为了仿真虚拟硬件，而且它也能兼容前面两个仿真模型。仿真模型③是非常讲究结构性的仿真模型，如果仿真环境，仿真对象，还有仿真过程，其中一放缺少结构都有可能失撑仿真模型③。



## 第五章 仿真就是人生

### 5.1 单向仿真与多向仿真

4. 有人认为学习仿真就是使用 Modelsim，不过 Modelsim 的功能非常接近电视机；
5. 有人认为学习仿真就是认识时序，时序是可视化的活动记录而已；
6. 有人认为学习仿真就是激励，可是激励只是模块的刺激还有反应过程而已；
7. 有人认为学习仿真就是验证语言，但是验证语言只是用来描述激励内容而已。

这个不是那个也不是，那么仿真又是什么呢？有人曾经告诉过笔者，轮胎，方向盘还有遮罩镜虽然是车子的部件，但是它们都不是车子。仿真也有同样的道理，Modelsim，时序，激励还有验证语言（综合语言），充其量只是仿真的部件而已，而不是仿真本身。曾经何时，笔者也认为仿真就是学习 Modelsim，后来才醒觉这一切都是传统流派的阴谋，不过发现已晚，因为笔者已经陷入其中难以自拔。

仿真好比一处异世界，这个世界虽然用来再现现实世界的种种细节，但是这个世界却有自己法则还有自然。这是一处并行，多变数，还有令人绝望的空间，过往的一切顺序知识已经不再适用，为了在这个危机四处的环境活下去，我们必须做好充分的准备。在此，Modelsim 是一件高科技的显示器，它可以接收周围的信号然后将期显示化。不过，Modelsim 显示的内容好比别人眼睛看见的东西，属于第二次视像。

这个世界存在许多奇怪的对象，它们没有听觉，视觉，触觉与味觉，无论我们怎么比手画脚，信息也无法传达进去。异世界的对象，大多数的行为都是静态，它们仅对信号有所感觉，因此激励成为唯一——一个手段可以产与这个世界互动。如果我们没有给予适当的刺激，它们就无法给予预期的反应 ... 不过很庆幸的是，外来者来到这个异世界都会赋予一股神奇的描述力量，这股力量不仅可以用来描述信号，这股力量也能描述对象。这股力量有两种属性，其一就是综合，其二就是验证。

（十四） 的目的就是要求读者熟悉使用 Modelsim 这件高科技产物；第三章的目的就是要求读者认识异世界的法则；第四章的目的就是要求读者熟悉“力量”... 读者须理解，异世界是自由结构的空间，所以哪里的居民没有结构一点也不奇怪。力量除了描述信号给予对象激励以外，力量也能用来描述最基本的结构。我们只要满足上述一切条件，我们在这个世界才有最起码的 10% 生存率保证。

有些同学可能会疑惑道：“既然异世界如此危险，为什么我们还要冒着生命危险去冒险呢？”，让笔者用故事来回答吧。

很久以前，有一处寸草不生绿洲甚少，被当地人称为禁地的沙漠平原，哪里不曾有人类涉足。有一位叫做蒂沙的当地女性，它从小就相信那片沙漠的彼岸，存在传说中的白骆驼之墓，白骆驼被当地人认为是神兽，崇拜程度就像东方的龙与凤，西方的独角兽与飞马。不过，当地的环境文化非常保守，年幼从家，年轻从夫，年老从孙，但是不过 21

岁的蒂沙终于抛弃故乡，涉足沙漠寻找传说。

蒂沙走了三天三夜的路程，环境恶劣还有前路茫茫 ... 就算如此，她还是继续前进。一个夜晚，她不小心卷入强烈的沙尘暴当中，夜晚的沙尘暴属于极寒，冷漠的沙尘渐渐夺走她身上的体温。蒂沙的精神就算不曾放弃，不过已经接近极限的肉体却背叛她，然后意识逐渐离去，如铅一样重的眼皮也随之下沉。迷糊中，她感觉自己被毛茸茸还有发着白光的躯体包裹着，她忽然意识自己正躺在白骆驼的怀抱中，白骆驼注视着她，她也注视着白骆驼。然后，有一股声音传入她的心里 ...

“孩子，为什么要误闯禁地，这里不是人类该来的地方 ... ” 心声道。

“传说，骆驼之墓可以告知追求者的命运，我想知道未来！ ”，蒂沙回应道。

“.....”，心声沉默。

“人类的孩子，这里就是妳的终点 ... ”，心声道。

从此以后，再也没有人看见那蒂沙的身影 ... 这是一个寓言的故事，故事告诉我们，蒂沙的人生有两个重要的抉择，其一就是找个丈夫嫁去，生孩子做婆婆，最后在亲人的哀悼中离开人世；其二就是踏足禁地寻找传说中的白骆驼之坟。某位诗人曾经描述过，平凡又单调的人生称为单向人生，曲折又多难的人生称为多向人生，然而不管是那一种人生，只要找到生命的意义那就是最好的人生。

很明显的是，蒂沙选择曲折多难的人生，在旁人眼中她可能愚蠢至极，不过旁人始终没有资格对她的人生指指点点，因为旁人不是蒂沙本身，旁人也无法理解蒂沙身为当局者的处境，情绪，还有心境状态。在她曲折又多难的人生中，可能她会死在旅途中，也有可能她已被白骆驼接济，结果究竟是如何只有蒂沙自己知道。

它曾经告诉过笔者，**仿真不过是人生的一场缩影**，仿真对象（蒂沙）就是人生的主人，预想所要的仿真结果就是仿真对象需要寻找的生命意义。我们创建仿真环境的人，责任好比白骆驼一样，我们必须引导仿真对象（蒂沙）流向有意义的人生。同样，**平凡单调的仿真称为单向仿真，曲折多难的仿真称为多向仿真**。

仿真是充满许多可能性的空间，单向仿真的出现概率非常低，单向仿真最常见的例子就有门级仿真，或者一些“超简单模块”的仿真而已。

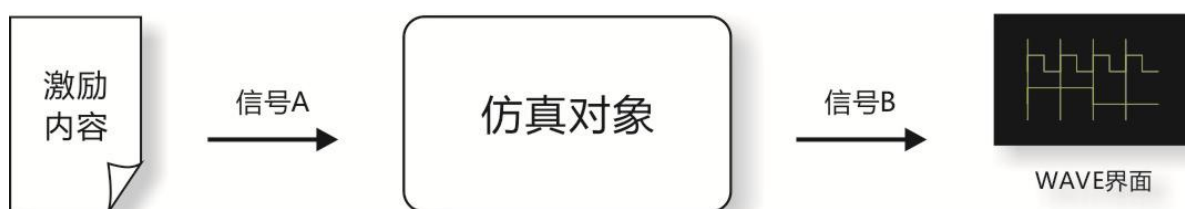


图 5.1.1 单向结果（仿真模型①）。

如图 5.1.1 所示，这是**仿真模型①**，也是**单向仿真最常用的仿真模型**，**信号永远流向一方**。我们可以将图 5.1.1 想象为，激励内容好比蒂沙年幼的时候，仿真对象是蒂沙年轻

的时候，wave 界面是蒂沙年老的时候，信号 A 是蒂沙称为人妻的阶段，信号 B 是蒂沙成为婆婆的阶段。图 5.1.1 的整体感觉好比蒂沙单调又平凡的一生。紧接着，请读者打开 exp20 瞧瞧，单向仿真究竟拥有什么最基本的模样。

*exp20\_simulation.vt*

```
1.  `timescale 1 ps/ 1 ps
2.  module exp20_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg [3:0]rSelect;
7.      reg [7:0]WrData;
8.
9.      /*****/
10.
11.     initial
12.     begin
13.         RESET = 0; #10; RESET = 1;
14.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
15.     end
16.
17.     /*****/
18.
19.     always @ ( * )
20.         case( rSelect )
21.
22.             4'b0001: WrData = 8'hAA;
23.             4'b0010: WrData = 8'hBB;
24.             4'b0100: WrData = 8'hCC;
25.             4'b1000: WrData = 8'hDD;
26.
27.         endcase
28.
29.         /*****/
30.
31.         reg [3:0]i;
32.
33.         always @ ( posedge CLOCK or negedge RESET )
34.             if( !RESET )
35.                 begin
36.                     i <= 4'd0;
37.                     rSelect <= 4'd0;
```

```

38.         end
39.     else
40.         case( i )
41.
42.             0:
43.                 begin rSelect <= 4'b0001; i <= i + 1'b1; end
44.
45.             1:
46.                 begin rSelect <= 4'b0010; i <= i + 1'b1; end
47.
48.             2:
49.                 begin rSelect <= 4'b0100; i <= i + 1'b1; end
50.
51.             3:
52.                 begin rSelect <= 4'b1000; i <= i + 1'b1; end
53.
54.             4:
55.                 i <= i;
56.
57.         endcase
58.
59.     /***/
60.
61. endmodule

```

第 1 行用来声明时钟刻度为 1ps/1ps；第 4~7 是仿真相关的寄存器；第 11~15 行是环境输入；第 19~27 行是仿真对象 ... always @ (\*) 主要是用来建立组合逻辑，其中建模对象是简单的输出选择器，别名又有加码器，根据 rSelect 的输入结果，输出相关的 WrData，大致功能如下：

- （一） rSelect 输入 4'b0001，WrData 就输出 8'hAA；
- （二） rSelect 输入 4'b0010，WrData 就输出 8'hBB；
- （三） rSelect 输入 4'b0100，WrData 就输出 8'hCC；
- （四） rSelect 输入 4'b1000，WrData 就输出 8'hDD。

由于仿真对象的功能太简单了，所以笔者直接在激励文本当中建模。第 33~57 行是虚拟输入 步骤0为 rSelect 写入 4'b0001 然后将 i 递增以示下一个步骤 ... 步骤3为 rSelect 写入 4'b1000，然后将 i 递增以示下一个步骤。

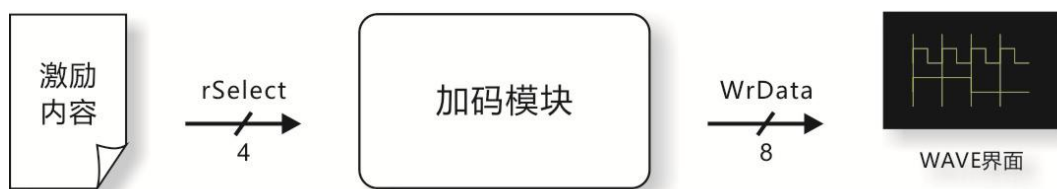


图 5.1.2 exp20 应用仿真模型①。

如图 5.1.2 所示，那是 exp20 应用仿真模型①以后的模样。激励内容（虚拟输入），经过 rSelect 刺激仿真对象（加码模块），然后再由仿真对象产生反应，将 WrData 的输出结果投射到 wave 界面当中。接下来，让我们一起瞧瞧 exp20 的仿真结果：

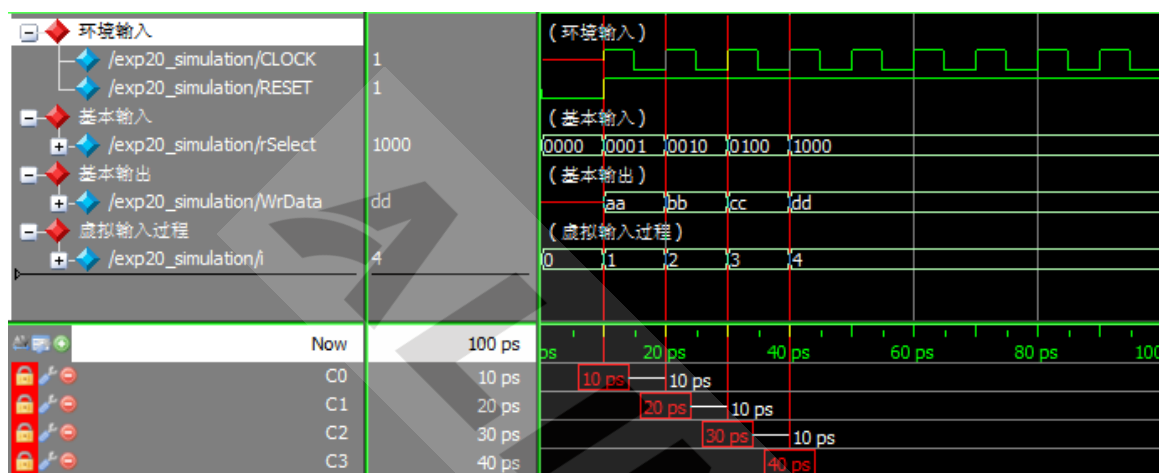


图 5.1.3 exp20 的仿真结果。

图 5.1.3 是 exp20 的仿真结果，其中光标 C0~C3 分别指向 4 个时间点。如图 5.1.3 所示，T0 之前是复位状态，WrData 没有复位值所示呈现红线。T0 的时候（C0 指向的地方），虚拟输入在步骤 0 为 rSelect 输入 4'b0001，结果仿真对象经由 WrData 输出即时值 8'hAA，然后将 i 递增以示下一个步骤。在 T1 的时候（C1 指向的地方），虚拟输入在步骤 1 为 rSelect 输入 4'b0010，结果仿真对象经由 WrData 输出即时值 8'hBB，完后再将 i 递增以示下一个步骤。

在 T2 的时候（C2 指向的地方），虚拟输入在步骤 2 为 rSelect 输入 4'b0100，结果仿真对象经由 WrData 输出即时值 8'hCC，然后再将 i 递增以示下一个步骤。在 T3 的时候（C3 指向的地方），虚拟在步骤 3 为 rSelect 输入 4'b1000，结果仿真对象经由 WrData 输出即时值 8'hDD，最后再将 i 递增以示下一个步骤。

exp20 是门级仿真，同时也是单向仿真最典型的例子。如图 5.1.2 所示，数据永远向一方流动，而且数据的变化状态也非常明了，输入什么就输出什么。此外，输出结果都是在输入给予之后不到一个时钟（或者一个时钟）之内出现 ... 啊！真是多么单调又平凡的仿真呀，仿真对“刷”一声就这样结束了。好了，感叹也十分了 ... 笔者也该换换心情介绍多向仿真。

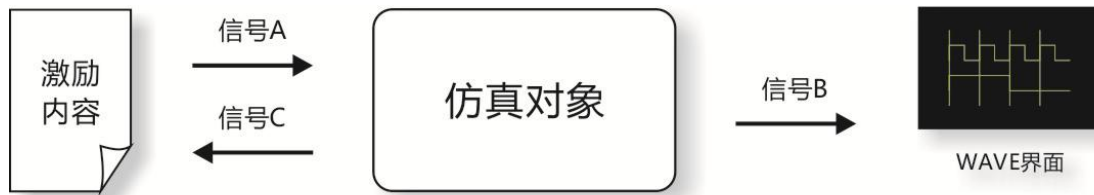


图 5.1.4 仿真模型②。

仿真是一种并行又充满许多可能性的空间，因此多向仿真出现的概率完全是压倒性的，那么甚么又是多向仿真呢？多向仿真好比蒂沙的曲折人生，人生的尽头存在许多未知的可能性，她有可能在旅途中意外死去，她也有可能抵达目的地，不管怎么样，这些可能性不过是其中一种结局的形势而已。如图 5.1.4 所示，这是仿真模型②，仿真模型②相比①，信号再也不是单向而是多向化，因此仿真模型②不可能是单向仿真。

多向仿真的典型应用除了门级仿真以外什么都是。换句话说，曲折多难是仿真对象的默认人生。说着说着，笔者的心情也开始沉重起来。具体内容，笔者还是用实验来讲话吧，打开 exp21：

exp21\_simulation.vt

```

1.  `timescale 1 ps/ 1 ps
2.  module exp21_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg [7:0]rD,rQ;
7.
8.      /*****/
9.
10.     initial
11.     begin
12.         RESET = 0; #10; RESET = 1;
13.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
14.     end
15.
16.     /*****/
17.
18.     reg [7:0]D1,D2,D3;
19.
20.     always @ ( posedge CLOCK or negedge RESET )
21.         if( !RESET )
22.             begin
23.                 { D1,D2,D3 } <= { 8'd0, 8'd0, 8'd0 };

```

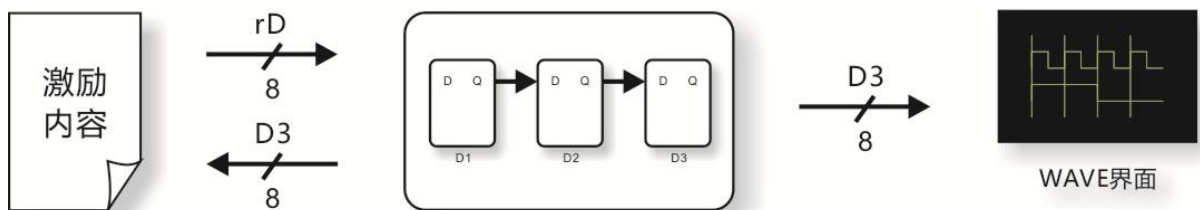


```

24.         end
25.     else
26.         begin
27.             D1 <= rD;
28.             D2 <= D1;
29.             D3 <= D2;
30.         end
31.
32.         /*****
33.
34.     reg [3:0]i;
35.
36.     always @ ( posedge CLOCK or negedge RESET )
37.         if( !RESET )
38.             begin
39.                 i <= 4'd0;
40.                 rD <= 8'd0;
41.             end
42.         else
43.             case( i )
44.
45.                 0:
46.                     begin rD <= 8'hAA; i <= i + 1'b1; end
47.
48.                 1,2,3:
49.                     begin rD <= 8'd0; i <= i + 1'b1; end
50.
51.                 4:
52.                     begin rQ <= D3; i <= i + 1'b1; end
53.
54.                 5:
55.                     i <= i + 1'b1;
56.
57.             endcase
58.
59.         /*****
60.
61.     endmodule

```

第 20~30 行是仿真对象，它是简单的移位寄存器，其中 D1 读入 rD（第 27 行），D2 读入 D1（第 28 行），D3 读入 D2（第 29 行）。第 36~51 行是虚拟输入，虚拟输入首先在步骤 0 为 rD 输入 8'hAA，然后等待 3 个时钟，最后在步骤 4 读入 D3 的结果。



仿真对象

图 5.1.5 exp21 应用仿真模型②。

图 5.1.5 显示 exp21 应用仿真模型 2 的结果，其实 exp21 是一种简单的搬运游戏。首先虚拟输入将数据 8'hAA 送给仿真对象的 D1，然后 D1 又将数据送给 D2，D2 紧接着送给 D3，最后再由 D3 返回虚拟输入的手中。话虽如此，游戏的结果却充满许多可能性，不过在此之前，先让我们先来瞧瞧 exp21 的仿真结果。

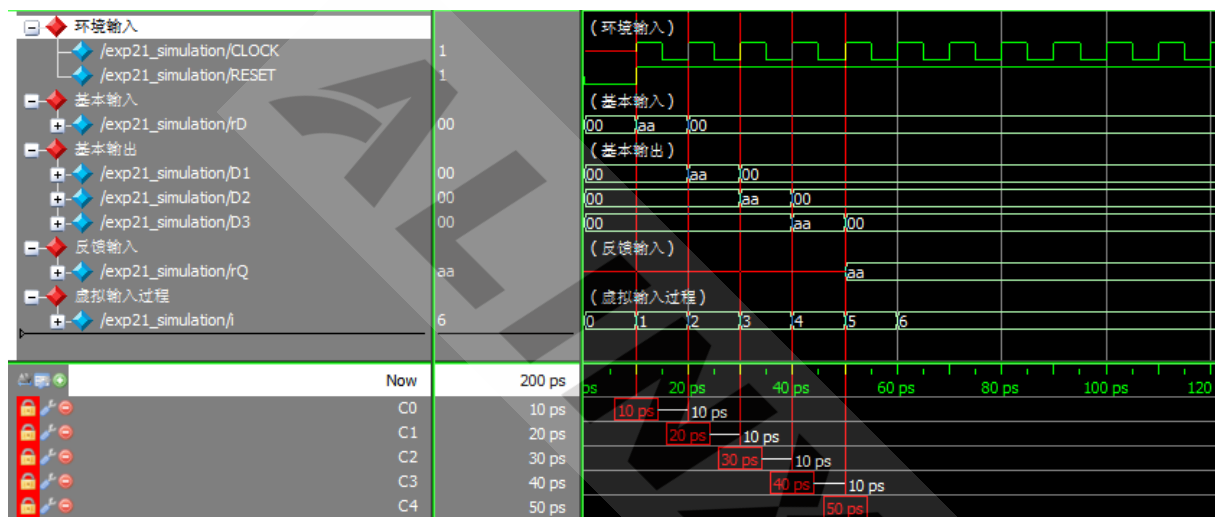


图 5.1.6 exp21 的仿真结果。

图 5.1.6 是 exp21 的仿真结果，其中光标 C0~C4 分别指向 5 个时间点。C0 指向的地方是时间点 T0，此刻虚拟输入开始为 rD 赋值 8'hAA，然后将 i 递增以示下一个步骤（搬运游戏开始）。C1~C3 指向的地方（T1~T3），分别是仿真对象相互传递数据的过程，数据传递至少需要一个时钟是时序不变的表现，因此仿真对象的 D1 在 T1 的时候读取 WrData 的过去值 8'hAA，D2 在 T2 的时候读取 D1 的过去值 8'hAA，D3 在 T3 的时候读取 D2 的过去值 8'hAA。完后，D3 在 T3 的未来输出数据 8'hAA。

C1~C3 也同样指向虚拟输入的步骤 1~步骤 3，不过虚拟输入没有采取任何行动，而是耐心等待数据回递。C4 指向（T4）虚拟输入的步骤 4，此刻虚拟输入读取 D3 的过去值 8'hAA，rQ 输出 8'hAA 的未来值。

exp20 是单向仿真的例子，exp21 则是多向仿真的例子，同学可能会很好奇道：“为何笔者要故意区分仿真成为单向和多向呢？”，这位同学千万别这么认为，一些视以为常的

事情，往往背后却隐藏着许多不可告人的细节，这些细节虽小但足够影响整个仿真的形势。在此之前，同学还是先思考一下，仿真是什么？为甚么我们要仿真？

最简单的问题往往是最复杂的问题 ... 根据笔者的认为，仿真的本意就是联系激励内容，仿真对象还有时序结果，并且做出解析，这一动作笔者也称为“仿真信息解析”。事实上，单向仿真还有多向仿真之间存在巨大差别的信息量，单向仿真的信息量好比一块小饼干，我们有可能一口吃掉。反之，多向仿真的信息量一般都是一块大蛋糕，如果一口吃不掉的话，我们又该怎么办呢？这就是区分单向仿真还有多向仿真的意义之处。

单向仿真非常直接，基本上没什么好谈的，所以解析仿真信息是一件轻松的事情。反之，多向仿真的仿真信息解析工作是一件苦差事，就让笔者用蒂沙的人生作出比喻。蒂沙有着两大人生抉择，亦即单向人生，还有多向人生。如果蒂沙选择单向人生，结果她活着会非常轻松，因为她只要“顺着”传统文化，结婚生子，然后死在亲人哀悼中，整场人生好不过是一只飞往单一处方向的飞矢而已。

多向仿真好比一群胡乱飞翔流失而已，然而只有一只流失会命中目标。因此我们必须无数穿梭在流失之间，寻找那唯一一只命中目标的箭矢。期间我们有可能找错对象，也有可能被流失所伤。

如果蒂沙选择多向人生，那么她的人生尽头就有许多种结局：

- (一) 她有可能旅途中意外死去；
- (二) 她有可能成功抵达沙漠尽头，不过那里没有传说中的白骆驼之墓；
- (三) 她有可能成功抵达沙漠尽头，然后找到的白骆驼之墓，不过是一片古迹而已；
- (四) 她有可能忍受不了孤独自寻短见；
- (五) 她有可能从中折返，放弃旅行；
- (六) 她有可遇见白骆驼。

如果我们根据故事作出推断的话，蒂沙的结局可能是一，二，三与六，不过对蒂沙而言最有意义的结局就是遇见白骆驼，因为白骆驼告诉她命运的答案。然而，蒂沙为了遇见白骆驼，她必须抛弃故乡遇见意外，不然第六种结局是不可能发生的。如果将这些内容反映到仿真当中，多向仿真拥有无数信号，而且信号也是无数流向，因而产生许多仿真结果，但是只有一个流向只能迎来最有意义的结果。

此刻“仿真信息解析”工作是非常头痛的 ... 假设我们追踪信号 A，信号 A 忽然 180 度折返，我们的注意力也要跟着折返，迎面而来的惯性有可能会冲晕脑袋。然后我们继续追踪信号 A，可是信号 A 分支成为信号 B 与信号 C，在此我们就要分半注意力同时追踪两个信号 ... 话说，读者累不累呢？

上述这些内容还不是最严重的问题，真正让笔者感觉害怕的是“可能性”。有些同学可能暂时无法理解这番话，不过不打紧，只要更紧笔者继续解释 exp21，读者就会深感体会。

=====

exp21 有一件小细节，如果笔者不解释它，笔者就会觉得心痒痒直至坐立不安，笔者一直觉得奇怪：“为什么虚拟输入的 rQ 会如此偶然在时间点 T4 成功回收数据 8'hAA？”。

偶然？偶然？偶然？不可能的！绝对不可能的！当中一定存有蹊跷。某位圣人说过，偶然会摧毁世界，所以万物都是必然。rQ 成功回收数据 8'hAA 绝对是必然，但是必然必须触发连环事件，满足所有特定条件以后才能实现。于是，笔者开始思考 ...

必然是将结局导向有意义结果的重要关键 ... 但是，如果结局有太多可能性，寻找必然是一件非常苦难的事情，因此笔者必须事先约束变数。

- (一) 完成一次性搬运游戏至少需要 5 个时钟。
- (二) 虚拟输入必须在 T0 为 WrData 赋值 8'hAA。
- (三) 数据传递必须遵守时序表现。

约束变数以后，rQ 产生的可能性就会缩小到以下 5 种而已：

- (一) rQ 在 T0 接收到 8'h00；
- (二) rQ 在 T1 接收到 8'h00；
- (三) rQ 在 T2 接收到 8'h00；
- (四) rQ 在 T3 接收到 8'h00；
- (五) rQ 在 T4 接收到 8'hAA；

我们知道在 5 个可能性当中，只有第五个可能性才是最有意义的结局 ... 不过又是什么特定条件将过程引导到第五个可能性呢？

- (一) 数据 8'hAA 从 WrData 移向 D1 用了 1 个时钟。
- (二) 数据 8'hAA 从 D1 移向 D3 用了 3 个时钟。
- (三) 数据 8'hAA 从 D3 移向 rQ 用了 1 个时钟。

上述 3 个特定条件就是将结果引导到第五 5 个可能性。换句话说，上述 3 个特定条件就是第 5 个可能性的立旗事件 (Flagging Event)，任少一个必然是不会实现。完后，笔者得出如下结论：“由于笔者数约 3 变数，结果衍生 5 种可能性，为了将结局引导到第 5 个可能性，笔者必须满足 3 种特定条件。”，在此，有些同学可能会开始怀疑笔者 ... 是否开始精神错乱了？又变数，又必然，又特定条件什么的 ... 不，笔者很正常。

联系激励内容，仿真对象还有时序结果并且做出解读，笔者称呼为 “仿真信息解析”。解析仿真信息原本是仿真最大的精髓，不过很遗憾的是 ... 人不是电脑，人不仅脑力有限，人的集中力也有时间限制，因此长时间解释仿真信息，无疑这是一种折寿的任务，最终还有可能演变爆肝的悲剧。

此外，exp21 也告诉我们一个事实，亦即变数还会无限放大可能性，从而海量化仿真信息。假设，解释一条可能性需要耗费 1 千卡路里还有 20 十分钟，如果变数将可能性放大至 10，我们就要消耗 1 万卡路里，还有花费 200 分钟去解释 10 条可能性。又如果变

数不留情将可能性放大 100 倍，我们既不是要消耗 10 万卡路里，花费 2000 分钟去解释 100 条可能性吗？别开玩笑，笔者不想爆肝也不想这样蹉跎青春。

庄子劝告华夏子孙，别用有限的躯体去追逐无限的梦想，这个道理同样也适用于在这种情况下 ... 人类未曾做到“解析海量仿真信息”这种壮举，即使超人出现它也有可能需要耗费一生去解释信息。笔者不是在吓读者，笔者只是在述说事实而已 ... 曾经何时笔者也年轻过，自负过，冲动过，笔者相信自己终有一天可以成为勇者，于是笔者决心完成这一壮举。结果，笔者被人抬着濒死的身體回老家去 ...

在此，笔者用经历告知所有读者，仿真真正可怕的地方，不是学习 Modelsim，验证语言或者激励文本。仿真真正最可怕的地方是，解析让人窒息的海量仿信息。普通人的用脑习惯都是倾向左脑，因此逻辑思维，还有顺序操作就会成为默认的用脑模式。换句话说，如果没有旁人特意提醒，我们就会傻乎乎地一条一条可能性解析下去 ... 其实这是温水煮青蛙的死法，死在不知不觉之间。

最后，笔者可以郑重的说：“这个世界（仿真）是充满杀机的空间，稍有差池小命就会不保！过往一切，那些习以为常的顺序手段，已经一去不返 ... 留念它们只会威胁小命而已！”，因为变数会影响可能性的衍生数量，然而在无数个可能性之中，只有一个生还的可能性（符合预期的仿真结果），其余可能性就是各种死法。

因此，如何约束变数减少可能性，还有如何清晰化发特定条件让它更加容易触发，都是这个世界里（仿真）的生存战略。不过很遗憾的是，常规的仿真手段似乎没有这方面的概念。因此，我们必须重新思考，寻找另一种较为适合的仿真手段，好让我们在这个异世界（仿真）继续游走下去 ...

## 5.2 仿真的变数——时钟用量

笔者在上一个小节说道，仿真最基本的本意是，联系激励内容，仿真对象，还有时序结果，然后作出解析，这种行为称为“仿真信息解析”。如果仿真不是单向仿真，那么仿真结果就会随着变数增大从而海量化仿真信息。此刻，解析仿真信息无疑是一种痛苦的工作。为了解决这种现象，在此之前我们知晓，何为仿真的变数？

算命师常常会如此狡辩道：“人生充满变数，算命结果不过是一种可能性而已，只要努力作人命运是可以改变的。”既然仿真是人生的缩影，仿真存在变数其实一点也不奇怪，不过传统流派却不怎么认为，因为传统流派是属于“爱拼才能赢”的自信份子。爱拼的人会在 100 个可能性当中都逐个尝试，直到成功为止。但是现实却告诉我们，10 人打拼只有 1 个成功，其余 9 人成为人间悲剧。

说实话，笔者实在没有兴趣在 100 个可能性当中蹉跎青春，笔者认为人生短暂而且生命有限，反之如何用最小的力气去寻找最大的成果才是活着的艺术。因此，笔者必须晓得如何压缩可能性 ... 可能性是变数衍生的产物，因此约束可能性就会间接缩小可能性的产生数量，根据笔者的认识，多向仿真存在以下几种：

1. 时钟用量
2. 信号数量
3. 信号方向

时钟不管是仿真（虚拟建模）还是建模（实际建模）都是一个非常重要的概念，不过很遗憾的是 ... 传统流派偏偏喜欢使用仿真时间驱动仿真流逝，而不是时钟本身。这种行为本来就存在许多缺陷，除了流失时序表现，还有违背 RTL 级设计的本质以外，最终还会产生非常不协调的物理时序，说着说着心情也沉重起来了 ...

笔者以前曾用过 i 指向时钟，笔者这样做就是为了清晰化还有具体化时钟，不然的话，时钟是不会轻易被我们捉着的。时钟用量一般是指“一次性活动所需的时钟”，然而这种认识又可以分为全体时钟还有个体时钟。接着，就让笔者继续使用 exp21 简单的举例吧。



图 5.2.1 exp21 的仿真内容。

首先让我们简单回忆一下 exp21 的仿真内容。如图 5.2.1 所示，激励内容先是（虚拟输入）经由 rD 给仿真对象输入数据 8'hAA；仿真对象是一个简单的移位寄存器，数据



8'hAA 在内部从 D1 游向 D2 至 D3，然后数据 8'hAA 又折返激励内容，紧接着 rQ 将数据 8'hAA 暂存并且投射在 wave 界面上。

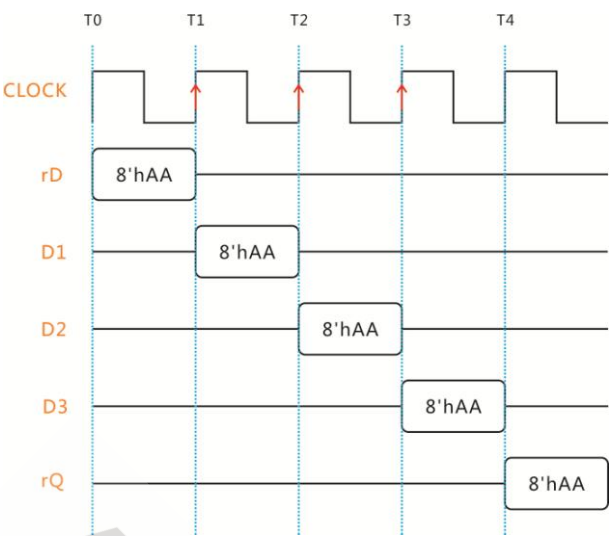


图 5.2.2 exp21 的理想时序结果（个体时钟视角）。

如图 5.2.2 所示，这是 exp21 的理想时序结果，过程如下：

- （一）rD 在 T0 输出未来值 8'hAA；
- （二）D1 在 T1 读入 rD 的过去值，并且输出未来值 8'hAA；
- （三）D2 在 T2 读入 D1 的过去值，并且输出未来值 8'hAA；
- （四）D3 在 T3 读入 D2 的过去值，并且输出未来值 8'hAA；
- （五）rQ 在 T4 读入 D3 的过去值，并且输出未来值 8'hAA；

读者有没有注意到？笔者故意为图 5.22 的 T1~T3 画上时钟沿，时钟沿总共有 3 个亦即仿真对象的一次性操作需要 3 个时钟用量，然而笔者将它们称为“个体时钟”。个体时钟是时间用量非常重要的一个概念，它是一种“当局者（微观）”的时钟视角。

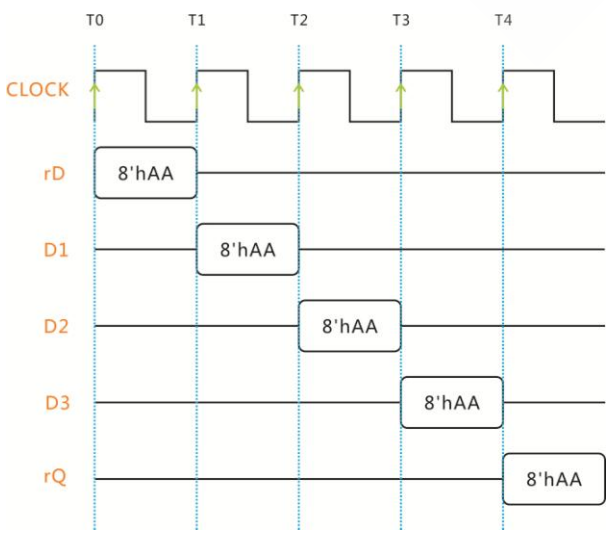


图 5.2.3 exp21 的理想时序结果（整体时钟视角）。



接着再让我们稍微更换一下时钟视角，如图 5.2.3 所示，笔者故意为添加 5 个时钟沿以示整体时钟，整体时钟是一种“旁观者（宏观）”的时钟视角。整体时钟除了 T1~T3 的个体时钟用量以外，又包含 T0 与 T4 这两枚整体沟通所需的时钟用量，因此整体的时钟用量有 5 个时钟。

笔者比较喜欢将模块看成有生气活物，模块沟通是模块遵守时序表现传输数据的一种“自然”现象，如果模块之间按照时间点事件发生沟通，那么模块之间至少需要 1 个时钟用量。换句话说，只要模块遵守时序，那么沟通所需的时钟用量一般都是固定的。相比之下，个体时钟不仅非固定，它还会伴随功能的“复杂程度”而成正比关系。换言之，功能越复杂，个体时钟越多，整体时钟也会相续增加，时间用量因而增大，结果产生更多的可能性。

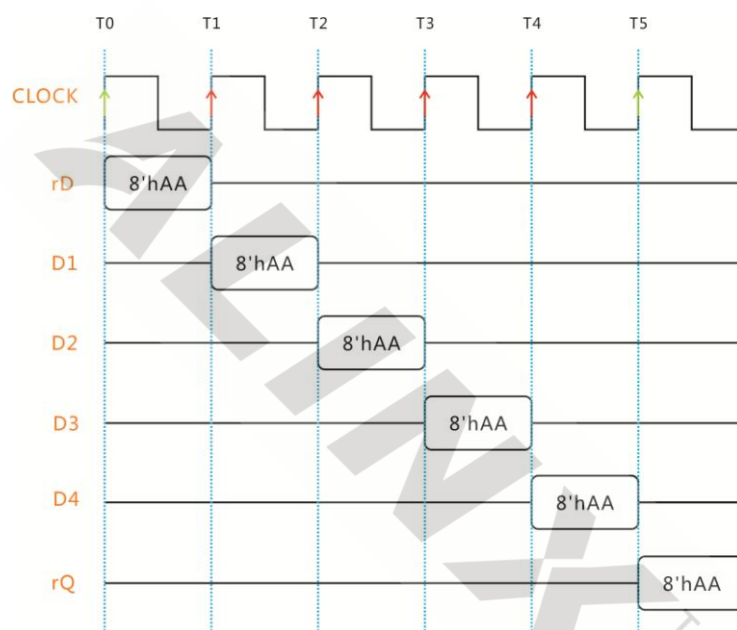


图 5.2.4 exp21 个体时钟增加以后的理想时序结果。

假设笔者增加移位模块的深度从原本的 D1~D3 变成 D1~D4，亦即移位模块的个体时钟从原来的 3 个时钟变成 4 个的话，结果如图 5.2.4 所示，T1~T4 是个体时钟，T0 与 T5 是沟通所需的时钟耗量，那么 exp21 的一次性搬运游戏至少需要 6 个时钟 ... 换句话说，时钟用量因为受到个体时钟的影响关系，从原来的 5 个时钟用量变成 6 个时钟用量。好奇的同学可能会问：“个体时钟还有时钟用量到底有什么变数作用？”，这位同学真是问了一个好问题。

举例而言，笔者曾经在上一节为 exp21 约束变数，如下所示：

- （一）完成一次性搬运游戏至少需要 5 个时钟。
- （二）虚拟输入必须在 T0 为 WrData 赋值 8'hAA。
- （三）数据传递必须遵守时序表现。

其中第一条约束变数就是时钟变量的约束，原本移位模块的个体时钟只要消耗 3 个时钟即可，不过经过笔者修改以后，个体时钟已经变成 4 个时钟。因为如此，rQ 成功接回数据 8'hAA 的可能性就提高了，如下所示：

- (一) rQ 在 T0 接收到 8'h00；
- (二) rQ 在 T1 接收到 8'h00；
- (三) rQ 在 T2 接收到 8'h00；
- (四) rQ 在 T3 接收到 8'h00；
- (五) rQ 在 T4 接收到 8'h00；
- (六) rQ 在 T5 接收到 8'hAA；

也就是说，原本是衍生 5 个可能性的，由于时间用量产生变化，结果就衍生 6 个可能性，其中第 6 个可能性是我们预期所要的结果。在这里，笔者使用简单的例子作为抛砖引玉的效果，笔者希望读者可以重视时钟用量，它是仿真**最初也是影响甚重的变数**，所谓一牵动全山就是这种意思。

根据概率论而言，投递 2 次银币就有 3 种可能性，投递 3 次银币就有 4 种可能性。反过来而言，只要减少投递的次数，可能性也会减少。不过，**时钟用量是前期建模的考虑范畴**，建模的时候要**尽量压缩个体的时间用量**，从效果上来讲，**个体的时间用量当然是越小越好**。为了控制时钟，指向时钟就会凸显其重要性，如果没有使用工具指向时钟，时钟就会变成非常模糊又虚幻。

说到指向工具，笔者不得不说 ... 虽然笔者常常举例用 i 指向个体时钟可是却未曾举例如何指向整体时钟？在此之前，先让我们好好理解一些个体时钟还有整体时钟之间的微妙关系。

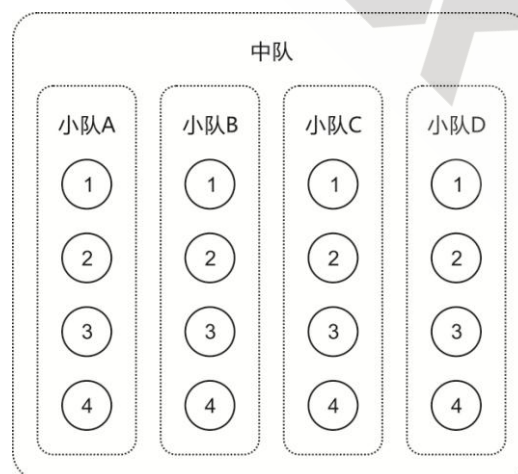


图 5.2.5 整体与个体的微妙关系，示意图。

如图 5.2.5 所示，一个中队里边包含 A，B，C，D 四个小队，然后每个小队又包含队员 1~4。假设小组是整体，那么队员就是个体；假设中队是整体，然后无视队员，那么小队就是个体；假设中队是整体，队员则没有被无视，那么小队是局部整体，队员是个体。一般上，整体都是指最包裹力与组合力最强的哪一圈，所谓的整体时钟也是从这一视角

出发的时钟。接下来请打开 exp22 :

*exp22\_simulation.vt*

```
1.  `timescale 1 ps/ 1 ps
2.  module exp22_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg [7:0]rD,rQ;
7.
8.      /*****/
9.
10.     initial
11.     begin
12.         RESET = 0; #10; RESET = 1;
13.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
14.     end
15.
16.     reg [19:0]G;
17.     always @ ( posedge CLOCK or negedge RESET )
18.         if( !RESET )
19.             G <= 20'd0;
20.         else
21.             G <= G + 1'b1;
22.
23.     /*****/
```

exp22\_simulation 与 exp21\_simulation 相比，除了在第 16~21 行指向整体时钟以外，其它都一样。如代码行第 16~21 所示，第 16 行声明为 G 的整体时钟指向工具，G 大写代表 God 或者 Global 的意思，此外 G 的位宽一般有多大就给多大，因为 G 指向整体时钟，所以它必须有这样的容量。第 17~20 行则是简单的计数器。

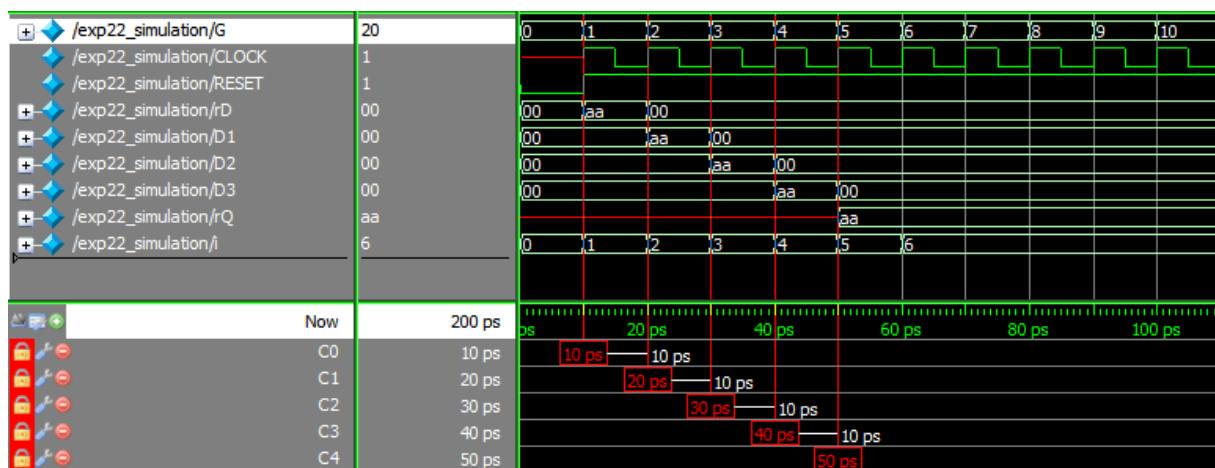


图 5.2.6 exp22 仿真结果。

如图 5.2.6 所示，这是 exp22 的仿真结果，图 5.2.6 相比图 5.1.6 只是信号的最上方多了一个指向信号 G 而已，指向信号 G 是使用过去值标示当前的时钟点。G 指向信号更上一层楼清晰化时序的表达能力 ... 好奇的同学可能会问：“为何笔者在之前的仿真都没有使用 G 指向整体时钟呢？”。

笔者认为，如果指向个体时钟都作不好，指向整体时钟不过是没有意义的补救手段而已。换之，如果我们做好指向个体时钟的工作，指向信号 G 会发挥如虎添翼的效果。不过，老虎到底需不需要翅膀又是另外的话题，老虎有翅膀本来就是一件非常奇怪的事情。如果读者是时钟概念很弱的人，指向信号 G 固然是很好的辅助工具 ... 反过来讲，如果读者有很强的时钟概念，指向信号 G 是可有可无的存在，还不如少一个信号 wave 界面不是更加眼见清净吗？

笔者还有一个不使用指向信号 G 的原因就是，笔者不怎么喜欢用“神的视角”去俯视整场仿真流程。如果我们用神的视角去俯视一切，即我们必须同时“吸收”所有仿真信息，并且消化 ... 说实话，笔者实在没有“神一般的承受能力”。抱歉，笔者又离题了 ... 让我们切回时钟用量这个话题吧。

时钟用量是仿真最直接性的变数 ... 理论上，时钟用量越少，变数所衍生的可能性就会越少。不过要控制好时钟用量，那是前期建模的考量范畴。建模期间，我们不仅要用工具指向时钟，我们还要特别了解时序表现，不然的话，时钟用量不仅不会乖乖就范，反而还会趁机暴走。

时钟用量除了影响可能性的产生以外，时钟用量还能决定最大的仿真时间，于是笔者作出这样的推理：

个体时钟增加 => 整体时钟增加 => 仿真时间增加 => wave 界面更长 => Modelsim 吃更多内存 => 拖慢计算机 => 放缓仿真进度 => 蹉跎青春。

细节决定成败，这句话果然不是盖的 ... 小小一个时钟用量既然可以间接影响我们的青春。

### 5.3 仿真的变数——信号数量与信号方向



图 5.3.1 不同性质的变数，直接性（左），间接性（右）

乘我们还没有进入主题之前，首先让我们先简单了解一下不同本质的变数。如图 5.3.1 所示，变数有直接性（左图），还有间接性（右图），两者之间的差异就在于 ... 直线性变数会经过自己衍生可能性，反道间接性变数，它必须影响对象才能产生可能性。此外直接性变数是一种可以改变的变数 ... 反之，间接性变数是一种不可改变的变数。

时间用量相它是一种直接性的变数，这种情况好比读者呆在在马路上越久，危险的可能性就会越高，然而只要我们立即抽身离开，危险也会立即消失。换之，信号数量则是一种间接性的变数，如马路上的汽车越多，马路就会越危险，但是我们却没有能力减少汽车的数量。信号方向也是一种间接性的变数，我们知道十字路口相较单向通道，车祸的发生的频率越高，那是因为十字路口相比单向通道拥有更加多的方向，同样我们也没有能力改变马路的布局。

笔者也说过间接性变数是一种不可改变的变数，打个比方说，如：RS232 的传输协议是一帧十一位数据就是一帧十一位数据，按照理论，数据位越长，数据损坏的可能性越高，然而我们却不能私自减少数据位，因为一帧十一位数据是 RS232 的传输标准。为此，我们又该如何插手“信号数量”还有“信号方向”这两个变数呢？这个时候，我们必须换个视角看待问题了。

曾经那么一次 ... 师兄命令我们这些新手师弟为功能 A 建模，然后产生时序 B。注视黑板上的功能 A，那似蜘蛛网的状态机，还有臃肿的模块内容 ... 所有人当场都头皮发麻了。再来注视左边的时序 B，看着那副极度不协调的时序图，现场所有人顿时露出比死人更难看的脸色。师兄命令我们在一个时辰内交出功课之后便离去 ... 不一会，一阵喧嚷响彻整个空间。

笔者当然知晓师兄的用意，建立复杂功能还有生成复杂时序都是每个新手必须克服的苦难之一 ... 但是笔者就是非常反感这种“强坑硬塞”的授学方式，在笔者的眼里那简直是另类的暴力，这种态度无疑是用来炫耀“高手”的能力一般，“看吧，蠢货们！这就是实力的差距 ... 不甘心的话，就克服给咱看看！”。根据概率论而言，百人之中可能只有那么一两个人才能成为“高手”而已，其余的蠢货都会沦落为“高手”的食物，这就是现实，残酷即无理。

笔者闭上眼睛切断全身的感觉，好让意识可以从丑陋的空间当中分割开来 ... 不知不觉间，感觉周围的吵杂声已经逝去不再，取而代之是安心的歌声，笔者下意识睁开眼睛眺望四周 ... “这是哪里？”——疑问随之从口中流了出来。笔者发现自己正处在花田之中，五彩缤纷的花儿们覆盖整片大地，这种情景宛如在地面上铺了一层五颜六色的地毯，微风走过，花儿优柔地摇晃身子然后发出协调的旋律。看着看着，笔者不禁入迷起来 ... 忽然，熟悉的声音传入耳中。

“来了吗，孩子？”神秘声道。

笔者下意识追寻声音的主人 ... 没错就是它，不知什么时候它已经站在笔者的身边，笔者依旧看不清楚它模糊的脸庞。

“孩子，又是什么问题将你引导到这里？”它道。

笔者稍微回忆一下，然后将问题向它讲述一番。它沉默了一会便举起有力的右手指向不远处的花田。

“孩子，哪里是什么？”它问道。

“不就是一处花田吗？”笔者答道。

“孩子，让我们走近看看好吗？”它提议道。

“嗯”，笔者点头道。

不一会，笔者便抵达它指向的花田之处。它再一次使用有力的右手指向一只花儿，随着笔者连忙蹲下身子，仔细瞧瞧 ...

“看见了吗，孩子？这里有什么 ... ”它道。

“有虫子！不仅一只，而且还很多”，笔者答道。

“远处看去，我们只能看见花田的整体，从近处看，我们就会看见花田的个体”，它道。

它的话永远都是充满寓意，笔者仔细思考了一会，“原来如此”，领悟的四个字不不经意从口中跑了出来。整体与个体之间的微妙关系，其实只是视角的远近而已，亦即宏观与微观。从宏观的角度看去，花田是一片复杂的整体 ... 换之，如果从微观的角度去干，花田里边其实有许多简单的个体。换句话说，无数简单的个体组成一具复杂的整体。

笔者转过头，用敬佩的眼神仰视它，忽然间有股念头从脑海中蹦出，笔者想继续向它提问 ... 但是，当笔者再度睁开眼睛的时候，周围又恢复原先的喧嚣声。原来是笔者睡糊涂了，笔者用手逝去嘴角的口水之后，立即望向时钟 ... 糟糕，笔者还有剩下半个时辰的时间而已，于是笔者急忙着手完毕功课。

期间，笔者一只在思考 ... “单位”是计算还有分类的基础，例如 1 个人与 3 只动物，其中“1 与 3”就是量化的单位，“个与只”是分类的单位。换言之，如果我们想划分整体模块就必须依靠单位才行 ... 话虽如此，单位的定义又是什么呢？笔者又该去哪里寻找



呢？事实上，低级建模的准则早已经给我们答案，那就是“**功能**”。

建模技巧（低级建模）的作用不可能只是单纯地为模块提供最基础的结构而已，其实建模技巧的绝对准则，亦即“**一个模块一个功能**”在不知不觉间，已经为仿真理下细化的种子。换句话说，一座复杂的功能可以经由建模技巧划分为数个简单的功能，复杂的功能就是整体，简单的功能就是个体。我们虽然不能随便简化整体的复杂性，但是我们可以经过某种手段，将整体有规则地分成许多更小的个体。

形象上来讲，**一块一口气吃不下的蛋糕，我们可以按着比例将蛋糕划分为无数规整的小蛋糕，然后逐个吃完**。其实这是一种很奇怪的心理现象，2kg 的蛋糕不管怎么划分，最终吃下肚子也是 2kg 的分量，但是一份 2kg 的蛋糕所给予的心理负担，比起 10 份 0.2kg 的蛋糕还要沉重。**心境决定行为的成败，好心情就是成功的开始** ... 抱歉，笔者又稍微离题了。

好奇的同学可能会问道：“**划分功能与这些变数（信号数量还有信号方向）究竟有什么关系？**”，真是一个好问题，首先让我们换个角度去思考问题吧。我们知道车祸一般都是人为引起的悲剧，假设有一处地带哪里集中 100 量车子 ... 根据概率论而言，如果车子的数量越是集中，那么车祸越容易发生。反之，如果我们将 100 量车子放在平均安放在 10 处不同的地方，根据概率理论，车祸发生的概率不仅可以分化，而且车祸的概率也可以减小。

同样的道理也适用在仿真当中，**功能的复杂程度还有信号的数量理应是成正比关系，亦即功能越复杂，功能数量越多**。笔者在前面也说过，信号数量是一种间接性的变数，它不会直接影响可能性的衍生，但是它可以影响人为活动，然后经由人为失误产生更多可能性。

举例而言，假设有 100 条信号产生 100 个可能性，当中要我们死盯其中一条信号，当我们长时间盯着同样的东西，眼睛就会开始疲劳然后幻觉就会出现。当我们将 100 条信号平均划分为 10 份以后，按照理论，可能性也会等着平均分化。死盯 10 条信号之中的一条，相较死盯 100 条信号之中的一条，前者对眼睛的伤害更加小。

笔者曾经尝试长时间在 100 条信号之中死盯着的一条信号 ... 不一会儿，笔者只是眨下眼睛，**信号忽然从 100 条变成 101 条** ... 笔者又揉下眼睛，信号又**从 101 条变成 102 条**。现实中 100 条信号仅是产生 100 个可能性而已，但是在幻觉中笔者却觉得 102 条信号产生 102 个可能性。

因此我们可以断定，**信号数量一种影响精神的变数，它会“弄混”我们，让我们产生幻觉，然后衍生出似存在又不该存在的可能性，因此我们需要分化过度集中的信号，从而消除这份“意外”。**

---

除了信号数量意外，信号方向也是间接性的变数之一 ... 读者尝试想象一下，如果我们**驾着车不停拐弯抹角，话说是不是很危险？车祸更加容易发生呢？**只要我们按照这样的



思路继续思考下去，我们便会知道一个事实，连续拐 10 个弯，相比连续拐 2 个弯，前者比后者发生车祸的可能性更高。

假设有 10 场连环弯，普通人不可能驾着藤原豆腐车，一口气，高速飘移在每场拐弯之中。换之，正常人都会分段拐弯，例如笔者这种怕死的家伙，笔者会将 10 场连环弯分段为 10 场小环。有研究显示，十字路口比起单向道路更容易发生车祸，理由很简单，单向道路只有一个方向，换之十字路口至少有 4 处方向同时拐弯。

在此，好奇的同学可能会问道：“笔者我们又不是在讨论头文字 D，车拐不拐弯又与仿真有什么关系？”

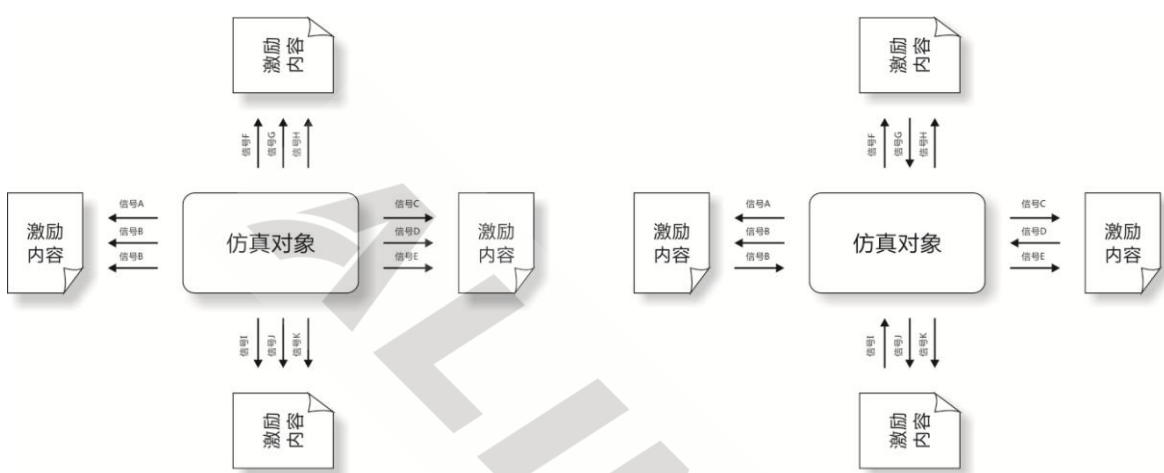


图 5.3.2 信号数量集中但是方位单向（左），信号数量集中但是方位多向（右）。

这位同学有所不知了，信号方向相较起信号数量是个更危险的变数。如图 5.3.2 所示，左图时信号数量集中但是信号的方向非常单一，右图不仅信号数量集中，而且信号的方向错综复杂。我们单是使用肉眼就能简单将左右图之间的差异分辨出来，试问读者那张图看起来比较“不头晕”呢？答案当然是左图。

信号数量好比马路的距离，信号数量越多马路的距离就越长。换之，信号方向还比马路的布局，信号方向越多向马路越是纵横交错。图 5.3.2 的左图可以比拟是一条很长很长的单向马路，然而图 5.3.2 的右图是迷宫般的多向马路。当我们追踪信号从一条到另一条的时候，这种情形好比我们在驾车从一条马路驶去另一条马路。

如果是左图，我们顶多只有长距离旅行的疲劳感而已；相反的，如果是右图，我们不仅感受长途旅行的疲劳，我们宛如嗑药般头不停甩来甩去，试问读者那种情况更容易发生车祸？答案当然是肯定着，右图更加容易发生意外。

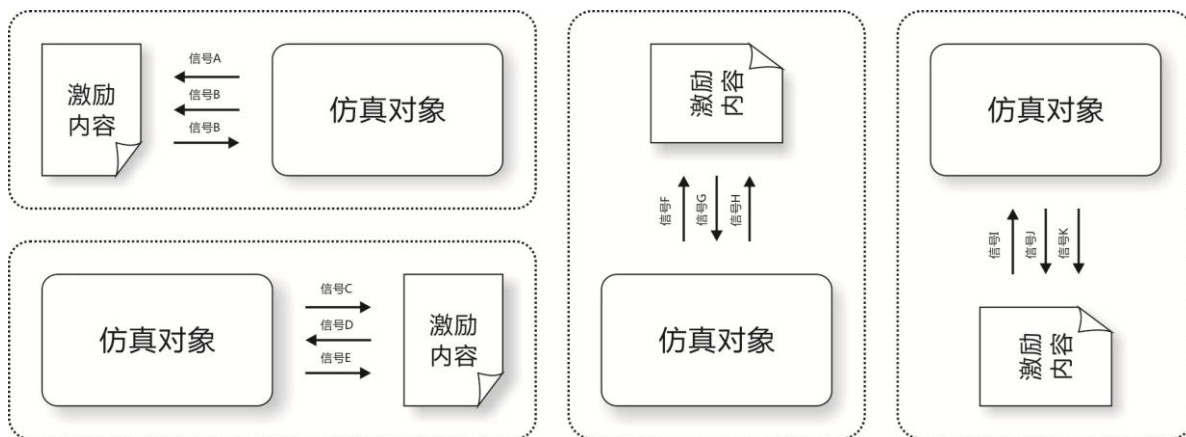


图 5.3.3 信号多向但是不集中。

没错，这就是信号方向的危险性，不过我们可以经过简单的功能划分将这种危险降到最低。如图 5.3.3 所示，笔者尝试将图 5.3.3 的右图划分成为几个等分的成份。当仿真对象的功能经过等分划分以后，虽然信号数量还有信号方向都没有变化 ... 不过不知道是不是心里作用？那种飘移过度所产生的晕眩感却没有之前般那么强烈，实在是太奇怪了。

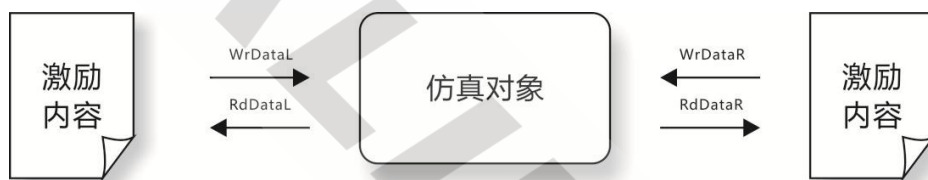


图 5.3.4 信号分化之前。

好奇的同学可能会问道：“信号数量还有信号方向，难道我们将他们分化而已嘛？为什么不能像级时间用量那样直接简化它呢？”，从理论上讲的确是如此，但是那种情况出现的几率非常少，让笔者再举个简单的例子吧。如图 5.3.4 所示，一个仿真对象的左右都有 WrData 与 RdData，对此我们可以这样推断道，仿真对象同时拥有两对非常相似的信号，它应该是可以进一步划分吧？

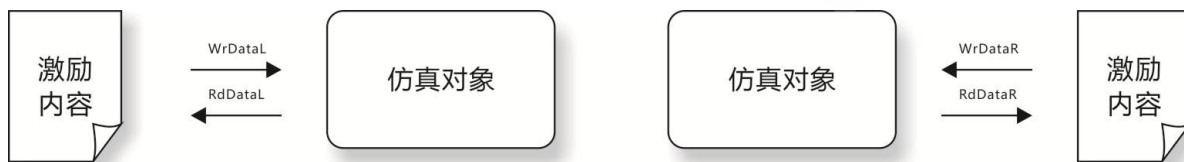


图 5.3.5 信号分化之后。

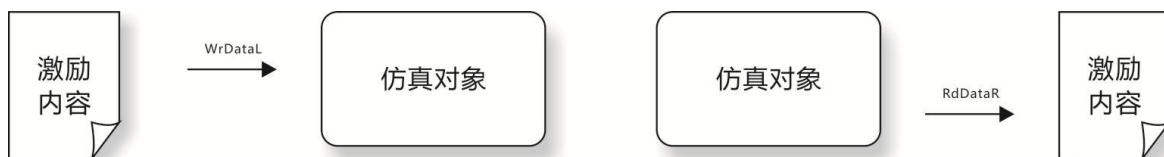


图 5.3.6 信号简化以后，仿真对象的原始模样遭受破坏。

经过一番舞刀弄斧之后，仿真对象的划分结果如图 5.3.5 所示。其中仿真对象一分为二成为两个更小的仿真对象，然后 WrDataL, RdDataL 与 WrDataR, RdDataR 被划分并且隔开出来，但是 WrDataX 与 RdDataX 的数量还有方向依然不变。假设读者饥不择食把其中一份吃掉的话，仿真对象的原始模样就遭受破坏，结果如图 5.3.6 所示。

笔者时常说道：“早期有好的建模，后期就有好的仿真”，其中也包含约束变数这一环。早前建模不仅仅是为了实现某种功能然后实现建模而已 ... 我们应该放长考虑才行，因为越是早期的作业，越是影响后期的表现，俗语不是有一句叫做一牵动全山吗？如此可见，早期作业有多么重要。好了，有关变数的话题就讲到这里为止吧，从下一章节开始我们就要进入仿真的高潮。



## 5.4 理想变数与物理变数

笔者曾经说过，仿真在大多数的情况下都是多向仿真，然而多向仿真的结局（仿真结果）不可能只有一项而已，而是多个可能性。换言之，多向仿真打从一开始就已经存在一定数量的可能性 ... 不过，变数在这基础上有可能再度放大可能性，从而海量仿真信息。茫茫大海当中寻找一种可能性，这份任务无疑是大海捞针，根本接近不可能。

可能性是变数相乘的结果，用典型的投币而言，假设硬币数量一枚然后笔者投币 3 次，然而投币结果会产生以下，如表 5.4.1 所示的 8 种可能性：

表 5.4.1 一枚硬币投币 3 次所产生的可能性。

可能性/结果	第一次结果	第二次结果	第三次结果
可能性 1	公	公	公
可能性 2	公	公	花
可能性 3	公	花	公
可能性 4	公	花	花
可能性 5	花	公	公
可能性 6	花	公	花
可能性 7	花	花	公
可能性 8	花	花	花

假设我们将表 5.4.1 当中，公表示为 0 值，花表示为 1 值，第 N 次投币结果表示为输入，可能性可以表示为输出 ... 看着看着，读者是不是觉得表 5.4.1 有点眼熟？没错，这是换个外皮的二进制表，结果如表 5.4.2 所示：

表 5.4.2 表 5.4.1 等价的二进制表。

输出\输入	[2]	[1]	[0]
Q1	0	0	0
Q2	0	0	1
Q3	0	1	0
Q4	0	1	1
Q5	1	0	0
Q6	1	0	1
Q7	1	1	0
Q8	1	1	1

疑心种的朋友可能会怀疑道：“事到如今，笔者举例这些内容又有什么用？”。朋友，看东西千万别那么死心眼，凡事要从各种角度去思考问题，希望读者可以明白笔者的用心良苦，故事其实是这样的 ...

某天下午，笔者一边思考问题，右手则不停投币解闷，笔者接着将投币结果写在纸张上，因为懒惰的关系，画圈表示公，写 1 表示花，无意间有股灵感袭击笔者的脑袋，心想：

“投币会不会和仿真有关系？”，于是乎笔者绘出表 5.4.1 与表 5.4.2。

乍看下，笔者自己也吓了一跳，两者实在太相似了，还不如说同一个人同时穿上两件不同的外衣而已。然而，表 5.4.2 与表 5.4.1 之间却有根本性的差别，表 5.4.2 是逻辑亦即非黑即白，表 5.4.1 是概率亦即可能性，于是笔者开始思考 ... 想着想着，笔者就发现仿真不仅存在“概率”，而且也存在“变数”。笔者当然知晓这种想法的矛盾性，因为逻辑讲究非黑即白，绝对不会计算什么 percentage，但是不甘愿的笔者又继续思考 ...

笔者曾在小节 5.2~5.3 讲述时钟用量，信号数量还有信号方向，这三种最主要的仿真变数，三者之间却可以这样反映在投币的身上：

- (一) 投币次数好比时钟用量；
- (二) 硬币数量好比信号数量；
- (三) 信号方向还比投币的意外性；

第一点很容易理解，只要仿真对象的时钟用量约多，那么投币次数就会越多，可能性随之也会衍生更多。换之，只要我们减少投币的次数，那么可能性也会减少衍生的数量，同样的道理也适用在时钟用量的身上。第二点也很容易明白，假设投币一枚硬币就会产生公或者花这两种可能性，如果两枚硬币同时投币，那么就会产生公花，公公，花花等三种可能性。同样的道理里也适用在信号数量的身上。

关于第三点，理解起来可能会稍微复杂一点 ... 概率论一般是基于随机性才能成立，所谓随机就是无法预测的因数，典型的投币例子，随机性是指投币的力道，投币的高度等人为因数。由于是人为因数，随机性有时候也可以称为意外性。有人类的地方意外就会发生，因为人类是悲剧的生物，仿真是人类着手的活动，不存在意外才奇怪呢 ... 可是仿真又存在什么意外？

笔者曾在小节 5.3 举例过，如果我们长时间死盯一条信号我们很容易看见幻觉。此外，如果信号的方向性多度多向化，意识就会 360 度来回甩动，直到头昏脑涨看见幻觉。没错，就是幻觉 ... 幻觉衍生更多没有实体的可能性。变数虽然可恶，但是时钟用量，信号数量，还有信号方向还是良性的变数，我们可以透过简化或者分化功能，从而缩小变数衍生的可能性。这些良性变数笔者称为理想变数。

如果变数存在良性，变数理应也存在恶性，没错那就是物理变数。接着，再让笔者使用投币作为举例吧 ... 投币在理想的状态下：

- 4. 硬币都有同等的大小和重量；
- 5. 投币结果为公与花两种；
- 6. 重力为  $9.8\text{m/s}^2$ ；
- 7. 环境为真空状态；
- 8. 投币者健康。

相反的，投币在物理的情况下：

9. 硬币不同等大小还有重量；
10. 投币结果出了公花以外也有直立的可能；
11. 重力为不安定；
12. 环境的空气属性，密度，湿度不等；
13. 投币者不健康。

为什么笔者要说物理变数是恶性变数呢？读者尝试想象一下，如果硬币不等大小的话，我们就有小硬币结果，还有大硬币结果之分。此外，如果投币结果出了公与花以外，还有直立（非公非花）这种不可思议的投币结果 ... 试问读者，可能性是不是会一口气增大许多呢？如果我们还要考虑投币者的健康状态，可能性既不是接近无限？这种情况再也不是大海捞针就可以了事，我们要在无边无际的宇宙当中寻找一粒微尘。

笔者不是在吓唬读者，笔者只是在讲述事实而已 ... 然而这个世界上却有一群傻子想要完成“宇宙寻尘”这种前无古人后无来者的伟大壮举，它们不是别人，正是传统流派。不管笔者翻开那一本参考书，它们都会威吓笔者“仿真要尽量实现物理时序”，任何时候笔者都会怀疑，作者是不是脑子进水了？难道它们看不到物理时序那接近无限的可能性吗？

物理时序主要有 2 个最基本的物理变数：

- （一）物理延迟
- （二）寄存器特性

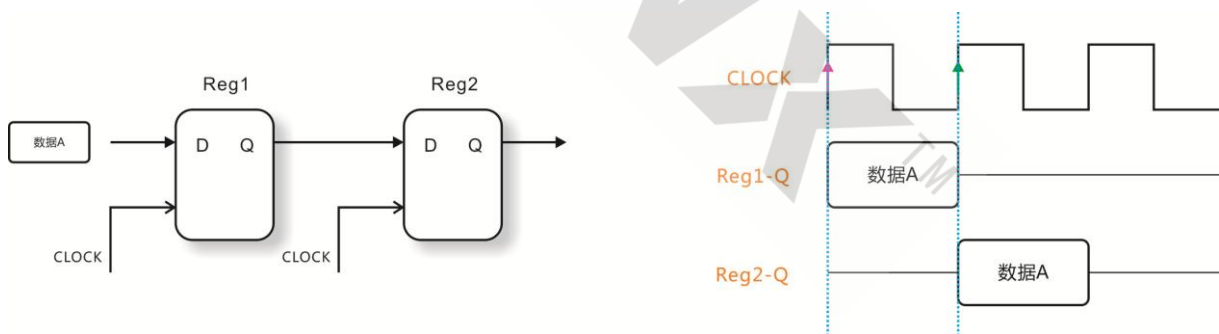


图 5.4.1 没有物理延迟的理想时序。

物理延迟是物理时序最常见的变数，也是最可怕的变数，物理延迟包括数据信号延迟，还有时钟信号延迟。举例而言，假设寄存器 1 将输入 A 读入以后再传递给寄存器 2，如图 5.4.1 所示，这是理想的数据传递，所以时序过程有如右边的理想时序图一样漂亮美丽。此刻，我们知道寄存器 1~2 都是按照理想时序表现传递数据，所以我们不要考虑什么物理延迟这种变数，因此时序结果只有 1 种可能性而已。



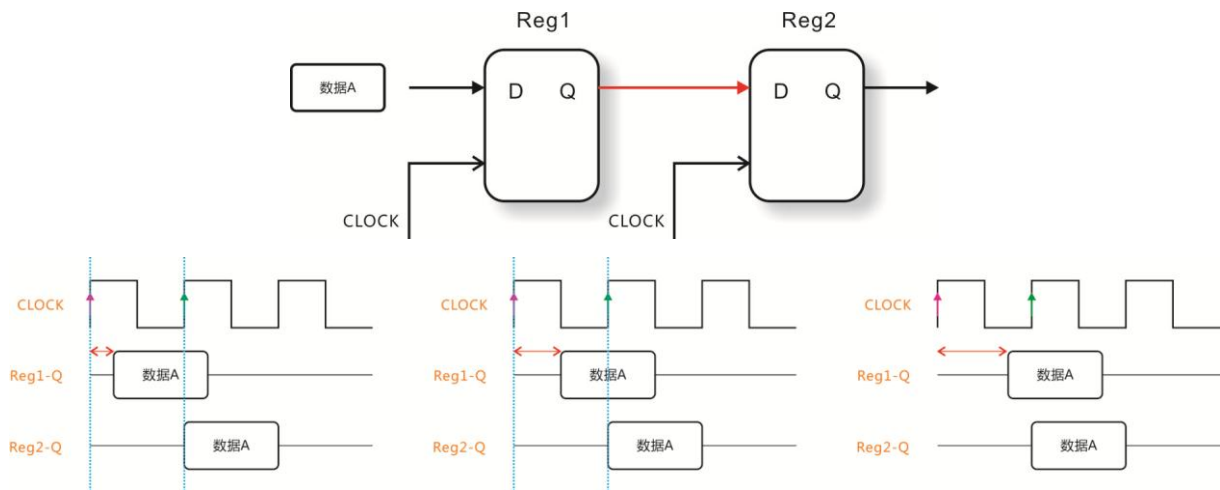


图 5.4.2 数据延迟的物理时序。

如果我们将物理延迟考虑进来，如图 5.4.2 所示，寄存器 1~2 之间的路径存在物理延迟，假设数据时间传递存在 3 种不同程度的数据延迟。在此我们就会开始思考，究竟是第一种时序结果寄存器 2 才能成功锁存数据 A，还是第二种时序结果，或者第三种时序结果呢？在此，我们拥有 3 种可能性的时序结果需要思考 ... 要么第一种？要么第二种？要么是第三种时序结果才是对的？

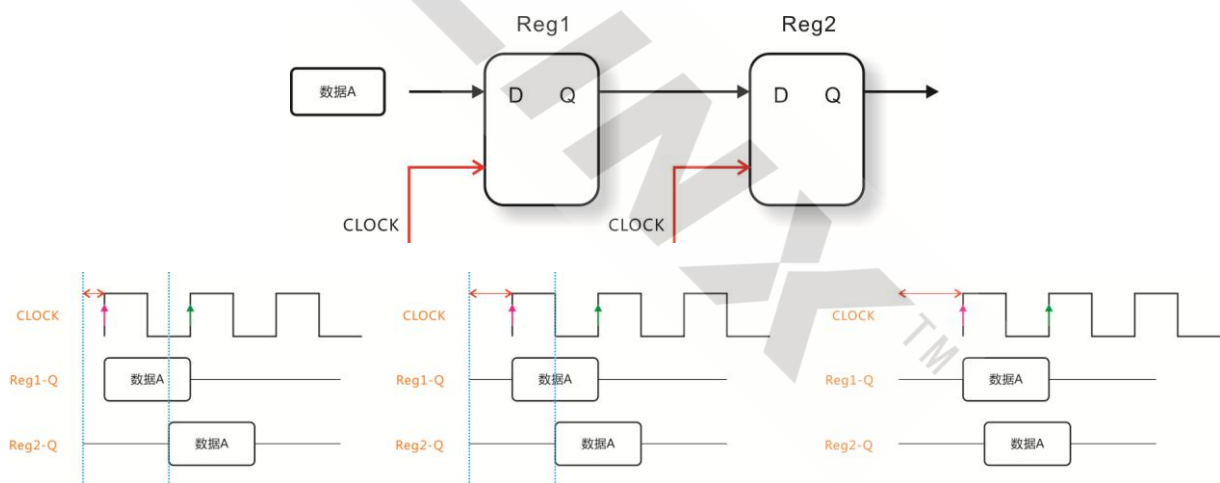


图 5.4.3 时钟延迟的物理时序。

如果物理延迟不是发生在数据信号的身上而是时钟信号的身上，如图 5.4.3 所示，假设寄存器 1~2 共享同样的时钟沿，而且时钟路径也有同样的物理延迟，然后再假设时钟路径有 3 种不同程度的延迟。在此，我们必须思考 3 种时序结果，要么是第一种？还是第二种？或者是第三种时序结果寄存器 2 才能成功锁存数据。读者千万别以为物理延迟的事情这样就完了 ...



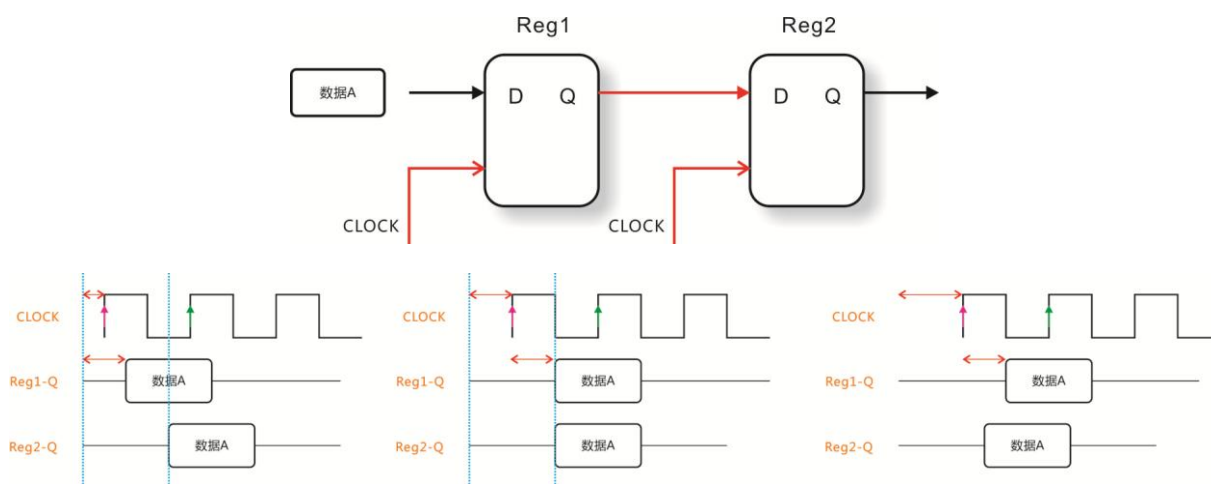


图 5.4.4 数据延迟与时钟延迟的物理时序。

现实残酷即不讲理，如图 5.4.4 所示，实际的物理延迟不可能仅存数据路径或者时钟路径任——者而已，而是两者同时存在。这时候，我们不仅仅是要考虑时钟延迟所产生的 3 种可能性，我们也要考虑数据延迟产生的 3 种可能性，因此我们需要考虑 6 可能性。事实上，图 5.4.4 已经很仁慈了，因为物理时钟不仅仅只有路径的延迟而已，物理时钟还会抖动，从而进一步扩大可能性 ...

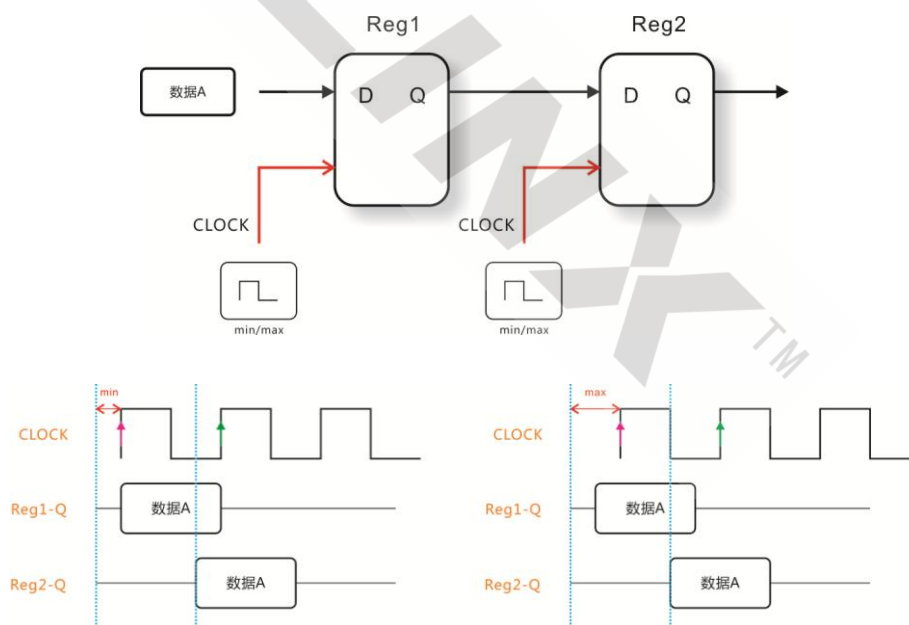


图 5.5.5 物理时钟抖动的物理时序。

如图 5.5.5 所示，假设寄存器 1~2 共享同样时钟源，虽然时钟路径不存在延迟，但是时钟源却产生抖动 min 与 max，就这样两种可能性就这样蹦出来。此刻我们必须同时思考 min 时钟抖动的时序可能性，还有 max 时钟抖动的时序可能性。再假设，如果寄存器 1~2 使用不同时钟源的话，而且两个时钟源都有发生抖动，请问结果会是怎么样的情景呢，读者可以想象吗？不过不用担心的是，时钟抖动所产生的延迟非常小，一般都可以直接忽略掉，在此笔者只是吓唬吓唬读者而已 ... 笑~

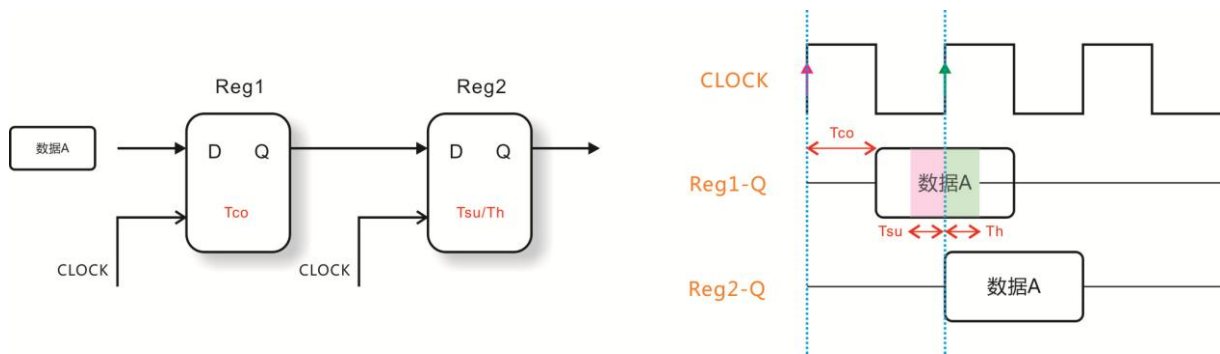


图 5.5.6 寄存器特性的物理时序。

寄存器特性物理变数的其中一种，如图 5.5.6 所示，常见的寄存器特性有  $T_{co}$ ， $T_{su}$  与  $T_h$ 。 $T_{co}$  是数据输出之前所需要的最小热身时间； $T_{su}$  是锁存数据所需的最小建立时间； $T_h$  是锁存数据所需的最小保持时间。虽然它不像物理延迟那样会随着综合的质量而产生改变，一般寄存器特性都是恒定的常值。话虽如此，实际上寄存器也是非常麻烦的物理变数之一。

根据物理时序的解释， $T_{co}$  的作用差不多和数据延迟那样会延迟寄存器的输出，简单看可以是寄存器内部的数据延迟。 $T_{su}$  还有  $T_h$  解释起来稍微麻烦一点，物理时序认为 ... 如果数据要成功锁存，数据必须满足两个条件，亦即  $T_{su}$  与  $T_h$  等最小时间。如图 5.5.6 的右图所示，当寄存器 2 被锁存沿（绿色的时钟沿）触发以后， $T_{su}$  与  $T_h$  的判断工作就开始了。

最佳的情况下，锁存对象——数据 A 不偏不移卡在锁存沿的中间，然后  $T_{su}$  覆盖数据 A 的左边， $T_h$  则覆盖数据 B 的右边，如果  $T_{su}$  与  $T_h$  无法完全覆盖数据 A，结果可以断定数据 A 成功锁存在寄存器 2 的身上，反之亦然。上述内容告诉我们，寄存器特性需要考虑 3 种可能性，亦即  $T_{co}$ ， $T_{su}$  还有  $T_h$ ，任何一种可能性都会影响锁存结果，其中  $T_{su}$  与  $T_h$  更加显得重要。

读者尝试想象一下，假设设计 A 消耗 100 个寄存器，那么可能性衍生的数量会是  $100 * 3 = 300$ 。普通规模的接口模块动不动就会需要消耗 200 以上的寄存器，如果这是玩笑笔者真是一点也笑不出来。

曾经有同学这样问道：“俺的仿真结果正确，可是模块不会发挥实际的效果，俺是不是忘记考虑物理因数呢？”，后来那位同学才发现自己写错模块了，但是那句疑心话——俺是不是忘记考虑物理因数呢？，却让笔者察觉到，常规仿真手段存在的漏洞。经过无数实验以后笔者终于发现，漏洞的地方其实是传统流派误认 HDL 的本质，它们打从一开始就认为 HDL 是物理即破烂的工具，所以产生物理时序是应该的。

但是根据笔者的理解，HDL 是理想的工具，物理因数是综合以后才添上的灰尘。于是，笔者继续思考 ... 物理因数实际上是仿真可有可无的东西，然而它的存在不仅照成仿真而外的负担，而且它还会令人疑神疑鬼，产生空洞的担忧。形象点说，就像那位同学一

样,原本它只要考虑模块的正确性即可,但是物理因数却让它考虑而外即不存在的烦恼,俗称杞人忧天。

理想时序除了违背 HDL 的本质以外,美观也是重要问题,但是主要原因是物理变数会无限放大可能性。物理变数属于间接性变数,它们虽然不会直接影响实际的结果,但是它们却会影响我们产生错觉,让我们在错觉种看见无限的可能性。认真想一想的话,这是一件非常不得了的事实,而且这种程度再也不能使用海里捞针来形容,实在是杀人于无形 ... 因此,笔者才会如此反感“物理”出现在仿真当中。

因此,仿真放弃物理因数是一件明智之举,这种感觉好比房间整整齐齐有一尘不染般,看着心情也爽快许多。不过,最重要的是可能性的衍生数量会大大降低,从而减弱一定的仿真信息。好奇的同学可能会问道:“如果仿真物理因数,那么无理因数又该如何解决?”,回答这个问题之前,先让笔者讲明一下 Modelsim 的用意。

仿真的精髓就是联系激内容,仿真对象,还有时序结果做出解析,简称为仿真信息解析。其中时序结果会播放在 wave 界面上 ... 再度强调!Modelsim 只会播放时序却不会为我们解决时序,就算 Modelsim 有能力播放物理时序,Modelsim 也没有能力解决物理时序,因为 Modelsim 没有这方面的机能。此外,Modelsim 播放的物理时序与实际的物理时序有天壤之别,实际的物理时序会有更多细节。说得难听一点,电视上面的物理时序我们只能看看自寻烦恼而已,实际上却什么也干不了,与其自寻烦恼还不如不看为好,读者说是不是,有没有道理呢?

只要明白这个道理以后,那么笔者就可以继续回答问题 ... 读者是否有听说过静态时序分析呢?它是专门用来处理物理时序的工具,读者可以想象为类似 Modelsim,然而它是用针对物理时序的加强版本,它不仅可以显示物理时序的各种细节,它也能计算并且解决物理时序的问题(时序违规)。说白一点,Modelsim 是用来针对理想时序的工具,然而静态时序分析是用来针对物理时序的工具,理解吗?

最后,让笔者这样总结吧:

时序有理想还有物理之分,变数也有理想还有物理之分 ... 其中,物理变数是可以排除在外,因为它只是会捣乱的没有家伙而已,反之理想时序是真正必须考量的东西。此外,笔者也用概率论最典型的投币来表示,仿真存在变数不是笔者空妄想的东西,实际上那是隐藏在仿真的背后,不为人知的重要细节。

变数会衍生可能性是人之常情,但是读者又否知道,Modelsim 每播放一次时序图,其实是显示多种可能性当中的其中一种而已。如果仿真有 100 个可能性,我们不可能重复 100 次仿真作业,因此简化变数可以降低可能性衍生的数量,分化变数会降低变数过度集中的危险(错觉的可能性)。

## 5.5 仿真的必然——特定条件

圣人说过偶然会摧毁世界，因此万物都是必然。笔者也说过仿真不过是人生的缩影，如果人生存在必然将人生的引向最有意义的结局，那么仿真也存在必然将结果引向预想所要。为了明确解释必然也存在必然，笔者同样也适用典型的投币来举例，假设笔者将一枚硬币 A 投币八次，然后产生如表 5.5.1 所示的结果：

表 5.5.1 硬币 A 投币 8 次的结果。

硬币 A	结果 1	结果 2	结果 3	结果 4	结果 5	结果 6	结果 7	结果 8
可能性 N	公	公	公	公	花	花	花	花

如表 5.5.1 所示，硬币 A 投币 8 次以后，产生前 4 公 4 花的结果，然后笔者姑且称为可能性 N。假设笔者再将一枚硬币 B 投币 8 次，取得可能性 N 的概率是：

$$1/2^8 * 100\% = 0.78125\%$$

计算结果是 0.78125%，可以说是一件令人绝望的数字，比踩狗屎更难发生。根据计算，硬币 B 出现上述结果的概率是 0.78125%，换句话说，硬币 B 重复 128 次才有偶然才出现那么一次 ... 不过笔者也说过，偶然并不存在这个世界，万物都是必然，硬币 B 为了实现可能性 N，硬币 B 必须完成一系列的特定条件：

5. 第 1 次投币的结果是公；
6. 第 2 次投币的结果是公；
7. 第 3 次投币的结果是公；
8. 第 4 次投币的结果是公；
9. 第 5 次投币的结果是花；
10. 第 6 次投币的结果是花；
11. 第 7 次投币的结果是花；
12. 第 8 次投币的结果是花；

现实中，人类的力量不能左右投币的结果，亦即不能控制必然 ... 不过，如果我们将硬币看做信号，必然就能被控制。假设硬币 B 为信号 B，然后投币次数是时钟用量，那么信号 B 的结果会是如表 5.5.2 所示：

表 5.5.2 等价的表 5.5.1。

信号 B	T0	T1	T2	T3	T4	T5	T6	T7
Q	0	0	0	0	1	1	1	1

完后，我们可以用 Verilog 这样表示，如代码 5.5.1 所示。

```
1. always @(posedge CLOCK)
2. case( 0 )
3.
```

```

4.    0,1,2,3:
5.    begin Q <= 1' b0; i <= i + 1' b1; end
6.
7.    4,5,6,7:
8.    begin Q <= 1' b1; i <= i + 1' b1; end
9.
10. endcase

```

代码 5.5.1

代码 5.5.1 的第 3~4 行表示将 Q 拉低 4 个步骤 ( 4 个时钟 ), 然后再第 7~8 行将 Q 拉高 4 个步骤 ( 4 个时钟 )。

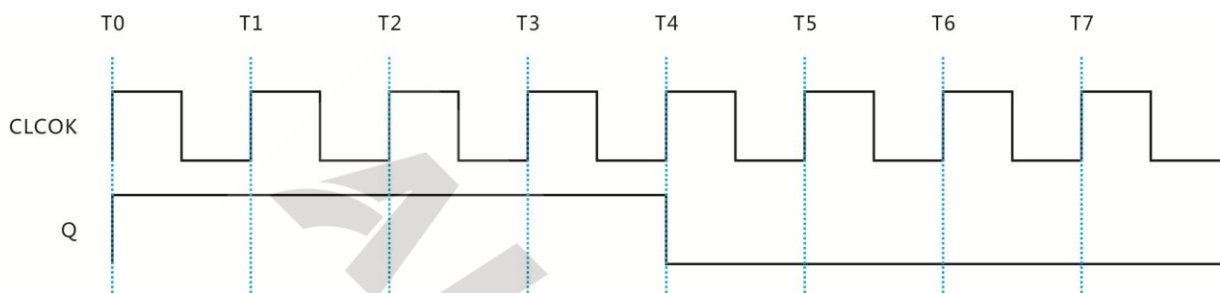


图 5.5.1 代码 5.5.1 产生的时序图。

完后，代码 5.5.1 会产生图 5.5.1 的时序结果。许多同学可能会好奇笔者所做所为，为什么笔者先是举例投币，然后对比信号还有投币之间的等价关系？笔者接着又用 Verilog 描述，甚至绘出时序图 ... 笔者这样做除了为了证明，[仿真存在必然性](#)以外，笔者也想接续蒂沙与白骆驼的故事。

所谓的必然性是指，[什么东西在什么时候必须发生什么结果](#)？在此，有些同学可能会将必然性与第三章的协调产生联系，然后问道：“必然性与协调是不是同样的东西？”。协调与必然性的确在标示同样的东西，这种感觉好比英文的 “apple”，还有日文的 “りんご”，两者分别标示苹果，但是英文还有日文却是不同性质的语言。

[协调纯粹是站在时序的角度](#)上去理解 “什么东西在什么时候发生什么结果”。反之，[必然性则是站在多向仿真的角度](#)上去理解 “什么东西在什么时候发生什么结果”。协调只考虑一种可能性而已，必然性则是考虑多种可能性。读者理解了吗？理解以后，我们就可以继续话题了。

骆驼之所以可以引导蒂沙，是因为白骆驼可以[非常清晰](#)的看见蒂沙的命运，它知道蒂沙[什么时候，发生什么，迎接什么结果](#)，如果白骆驼没有[引导](#)蒂沙，蒂沙就会[死在沙尘暴](#)当中。白骆驼为了引导蒂沙，它借助沙城暴这个契机出现在她眼前，聆听蒂沙的愿望之后，白骆驼便告诉她旅行已经结束并且实现她的愿望。这个故意隐隐约约也透露白骆驼是神的真面目。

我们身为设计者，创建仿真环境，在某种程度上算是这个空间的神，蒂沙好比仿真对象，

我们好比白骆驼，我们必须观察什么信号，在什么时候，得到什么结果。但是问题是，仿真对象的内容，是否足够清晰呢？不然的话，我们想引导也引导不了，结果蒂沙惨死在沙尘暴当中。

当我们将代码 5.51 还有图 5.5.1 联系起来，我们之所以认为代码 5.5.1 还有图 5.5.1 之间没有任何违和感，因为代码 5.5.1 都清清楚楚指向什么信号，在什么时候，输出什么结果。这个事实也告诉我们，**维护必然性，清晰必然性**，都是非常重要的任务，然而这些任务却涉及早期建模，还有激励文本的编辑。

如何维护必然性，清晰必然性其实是有窍门的，低级建模的用法模板就是为了这种目的才诞生。笔者曾在第 4 章解释过，仿真对象还有激励内容最好都使用相同的用法模板，因为如此，必然性都有同样模样的维护性，而且指向工具 i 也帮助清晰必然性。上述的内容又再一次证明“前期有好建模，后期有好仿真”这句话。

首先，让我们来瞧瞧失去维护并且没有清晰必然性的仿真对象，究竟会是什么样子的？请读者打开 exp23.

*exp23\_simulation.vt*

```
1. `timescale 1 ps/ 1 ps
2. module exp23_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.     reg [7:0]rD,rQ;
7.
8.     /******
9.
10.    initial
11.    begin
12.        RESET = 0; #10; RESET = 1;
13.        CLOCK = 1; forever #5 CLOCK = ~CLOCK;
14.    end
15.
16.    /******
17.
18.    reg [7:0]C1,rTemp;
19.
20.    always @ ( posedge CLOCK or negedge RESET )
21.        if( !RESET )
22.            begin
23.                C1 <= 8'd0;
24.                rTemp <= 8'd0;
```



```

25.         rQ <= 8'd0;
26.     end
27.     else if( C1 == rD )
28.         begin
29.             C1 <= 8'd0;
30.             rQ <= rTemp;
31.             rTemp <= 8'd0;
32.         end
33.     else
34.         begin
35.             C1 <= C1 + 1'b1;
36.             rTemp <= rTemp + rD;
37.         end
38.
39.     /******
40.
41. reg [3:0]i;
42.
43.     always @ ( posedge CLOCK or negedge RESET )
44.         if( !RESET )
45.             begin
46.                 i <= 4'd0;
47.                 rD <= 8'd0;
48.             end
49.         else
50.             case( i )
51.
52.                 0:
53.                     begin rD <= 8'd4; end
54.
55.             endcase
56.
57.     /******
58.
59. endmodule

```

exp23\_simulation 中的第 18~37 行是仿真对象，看着它读者是否觉得很乱却又觉得怀念呢？没错，在早期的时候，任何初学者都会采用的用法习惯，正确来说这是一种自由用法的编辑风格。我们不用考虑什么，我们只是尽量将所有操作往 always 块里边塞，这种感觉就好比将垃圾塞进垃圾袋里面。

这种自由用法最大的好处就是自由还有随意，好比叛逆的少年般想怎样搞就怎样搞。反观之下，自由用法也有缺点，首先采用自由用法的模块（仿真对象）必然性是非常模糊



的 ... 如代码第 18~37 所示, C1 等价 rD 以后便清零 C1 还有 rQ 被赋予 rTemp, 然后 rTemp 清零 (第 27~32 行); 第 33~37 行表示, 每个时钟沿便递增 C1 还有累加 rTemp。代码 18~37 行虽然有明显的操作, 但是代码 18~37 行却没有清晰的必然性。

第 41~55 行是虚拟输入, 笔者在步骤 0 只是为 rD 赋值 4 却没有读取 rQ 的内容, 好奇的同学可能会问为什么? 原因很单纯, 由于仿真对象没有清晰的必然性, 所以笔者也不知道仿真对象在什么时候输出 rQ, 因此 exp23\_simulation 至少必须运行一次, 虚拟输入的编辑工作才能继续下去 ...

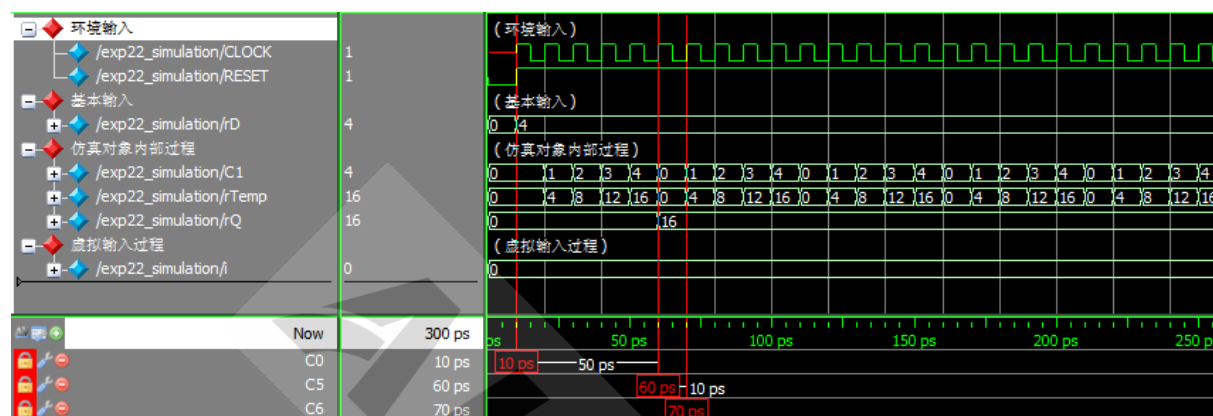


图 5.5.2 exp23\_simulation 的仿真结果。

图 5.5.2 是 exp23\_simulation 的仿真结果, 光标 C0, C5 与 C6 分别指向时钟 T0, T5 还有 T6。如图 5.5.2 所示, 虚拟输入在 T0 为 rD 赋值 4, 仿真对象在下一个时钟接收 rD 的过去值并且开始工作。仿真对象每累加一次 rD 的过去值 4 到 rTemp 里边, C1 就递增一点。当仿真对象执行操作直到 T5 的时候, if( C1 == rD )条件成立, rQ 被赋值与 rTemp 的累加结果 16, 然后 C1 与 rTemp 紧接着被清零。如果虚拟输入要读取 rQ 的值 16, 有效时钟是 T6。

exp24\_simulation.vt

```
1. `timescale 1 ps/ 1 ps
2. module exp24_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.     reg [7:0]rD,rQ;
7.
8.     /*******/
9.
10.    initial
11.    begin
12.        RESET = 0; #10; RESET = 1;
13.        CLOCK = 1; forever #5 CLOCK = ~CLOCK;
```

```

14.     end
15.
16.     /*****
17.
18.     reg [7:0]C1,rTemp;
19.
20.     always @ ( posedge CLOCK or negedge RESET )
21.         if( !RESET )
22.             begin
23.                 C1 <= 8'd0;
24.                 rTemp <= 8'd0;
25.                 rQ <= 8'd0;
26.             end
27.         else if( C1 == rD )
28.             begin
29.                 C1 <= 8'd0;
30.                 rQ <= rTemp;
31.                 rTemp <= 8'd0;
32.             end
33.         else
34.             begin
35.                 C1 <= C1 + 1'b1;
36.                 rTemp <= rTemp + rD;
37.             end
38.
39.     *****/
40.
41.     reg [3:0]i;
42.     reg [7:0]rRead;
43.
44.     always @ ( posedge CLOCK or negedge RESET )
45.         if( !RESET )
46.             begin
47.                 i <= 4'd0;
48.                 rD <= 8'd0;
49.                 rRead <= 8'd0;
50.             end
51.         else
52.             case( i )
53.
54.                 0:
55.                     begin rD <= 8'd4; i <= i + 1'b1; end
56.

```

```
57.             1,2,3,4,5:
58.             begin i <= i + 1'b1; end
59.
60.             6:
61.             rRead <= rQ;
62.
63.         endcase
64.
65.         /*****/
66.
67.     endmodule
```

exp24\_simulation 相较 exp23\_simulation 改变的地方就是第 41~63 行的虚拟输入。其中笔者在 42 行声明 rRead 寄存器用作暂存 rTemp 的结果 第 54 行是虚拟输入为 rD 赋值 4；第 57~58 行是等待 5 个空时钟；第 60 行是读取 rQ 的结果。

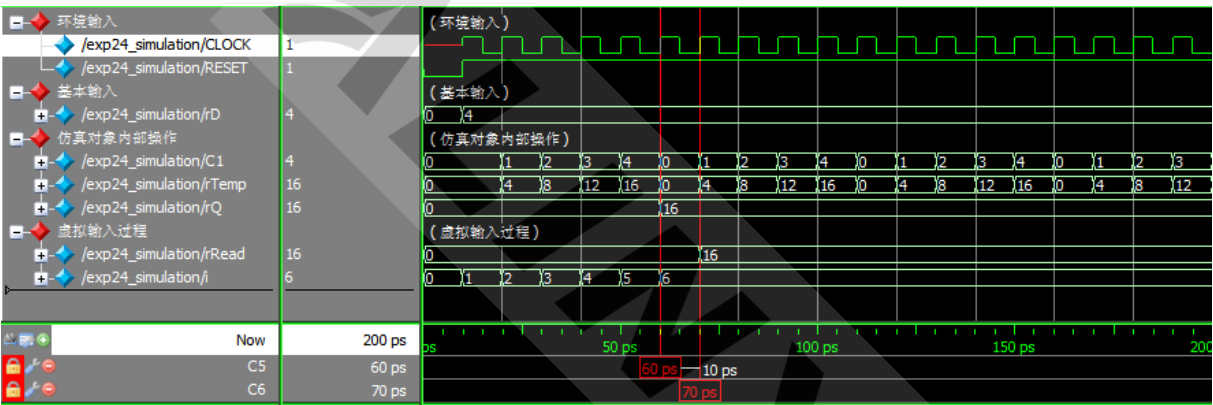


图 5.5.3 exp24\_simulation 的仿真结果。

如图 5.5.3 所示，光标 C5 指向的地方（T5）正好是仿真对象完成一次性操作的时候，然后虚拟输入再 C6 指向的地方（T6）读取 rQ 的过去值 16，因此 rRead 输出未来值 16。

仔细思考 exp23\_simulation 还有 exp24\_simulation，我们知道由于仿真对象（exp23）没有清晰的必然性，因此我们至少需要仿真一次（exp24）才能预测并且正确读取 rQ 的内容。由于仿真对象的必然性很模糊的关系，我们不得不消耗而外的气力，事实上这是非常被动的行为 ... 消耗气力倒不相干，蹉跎岁月才是致命的问题，人生最长不过是几个十年而已。

此外，笔者相信眼睛犀利的朋友已经发觉到，仿真对象的第 27 行 else if( C1 == rD ) 其实是一件非常有问题的写法。简单而言，仿真对象一次性所需的操时间会伴随 rD 的内容而产生改变。换句话说，每当 rD 的输入内容不同，rRead 读取 rD 的时间也会不同，为了让 rRead 有效读取 rD 的结果，我们至少都要预先仿真一次才能预测得到 rRead 读取 rD 的正确时钟 ... 这不是要耗死我们的精力吗！？

exp23~24 的仿真对象有以下两个问题：

6. 模糊的必然性
7. 时钟用量不固定

仿真对象虽然有明显的操作内容，不过它却不能有效告诉我们“什么时候，什么东西，发生什么事情”，因此被认为模糊的必然性；仿真对象也会随着 rD 输入内容的不同，时钟用量也会跟着不同。不管哪一点问题都是非常致命的问题 ...

读者尝试想象一下，身为白骆驼的我们就算窥视蒂沙的命运宏图，然而蒂沙的命运不仅模糊而且还有很强的随机性，这些因数会无限放大可能性，即时白骆驼的能力再怎么强大，白骆驼也无法知晓一切。结果而言，如果白骆驼无法完全掌握蒂沙她的命运流向，就算白骆驼想引导她，它也是心有余而力不足。

为了美丽的蒂沙，可怜的蒂沙，白骆驼豁出去了 ... 必然性之所以模糊，那是因为没有指向工具指向蒂沙的命运（仿真模块的过程），不管指向什么，哪怕一丝一毫也好，指向工具都必须指向某个东西。其中我们知道指向时钟那是无法实现的事实，因为仿真对象没有固定的时钟用量，既然指向时钟不行，指向步骤又如何？

```
1.  always @ ( posedge CLOCK or negedge RESET)
2.  if( !RESET )
3.      begin
4.          C1 <= 8'd0;
5.          rTemp <= 8'd0;
6.          rQ <= 8'd0;
7.      end
8.  else
9.      case ( j )
10.
11.          0 :
12.              if( C1 == rD ) begin C1 <= 8' d0; rQ <= rTemp; rTemp <= 8' d0; end
13.              else begin C1 <= C1 + 1' b1; rTemp <= rTemp + rD; end
14.
15.          endcase
```

代码 5.5.2

如代码 5.5.2 所示，仿真对象已经套用用法模板，其中 j 是指向工具，其中步骤 0 表示仿真对象原本的功能 ... 第 13 行的 C1 会根据每个时钟递增，而 rTemp 也会累加，当第 12 行的 if 条件成立以后，rQ 赋予 rTemp 的内容，然而 C1 还有 rTemp 都会被清零。当仿真对象套用用法模板以后，必然性随之也清晰起来。

不过不管我们怎么看，代码 5.5.2 始终都觉得少点什么？这种感觉好比用餐少了餐具 ...

然而代码 5.5.2 究竟少了什么关键的东西？

人生在冥冥之中，往往都会因为遇见某种契机，生命因而开始发生改变。契机是必然的一种，契机有时候也称为邂逅，但是对象不一定局限于人，它也可以是事物或者某种因缘。契机是命运的恶作剧，契机也是命运的黏糊剂，为了引导生命流向最有意义的结果，人的一生当中很有可能会存在许多重要的契机。

从另一个角度来讲，在生命的宏图当中，其实存在许多命运的碎片（命运的拼图），然而一条完整的命运线，都由无数的契机将无数的命运碎片串联起来。仿真就是人生的缩影，所以仿真当然也存在契机，但是问题是如何将“契机”描述出来呢？

```
1.  always @ ( posedge CLOCK or negedge RESET)
2.  if( !RESET )
3.      begin
4.          C1 <= 8'd0;
5.          rTemp <= 8'd0;
6.          rQ <= 8'd0;
7.          j <= 4' d0;
8.      end
9.  else if( isStart )
10.     case ( j )
11.
12.         0 :
13.             if( C1 == rD ) begin C1 <= 8' d0; rQ <= rTemp; rTemp <= 8' d0; j <= j + 1' b1;end
14.             else begin C1 <= C1 + 1' b1; rTemp <= rTemp + rD; end
15.
16.         1:
17.             begin isDone <= 1' b1; j <= i + 1' b1; end
18.
19.         2:
20.             begin isDone <= 1' b0; j <= 4' d0; end
21.
22.     endcase
```

代码 5.5.3

如代码 5.5.3 所示，仿真对象会以 isStart 信号作为契机而开始工作（第 9 行）。仿真对象完也会以 isDone 信号作为结束操作的契机告诉他人（第 16~20 行）。读者没有看错，代码 5.5.3 确实是应用了仿顺序操作 ... 从建模的角度而言，仿顺序操作仅是模仿顺序操作而已。换之，如果从仿真的角度去看的话，仿真对象会因某种信号作为契机开始工作，仿真对象也会因某种信号作为契机结束操作，类似契机作用的信号也称为沟通。

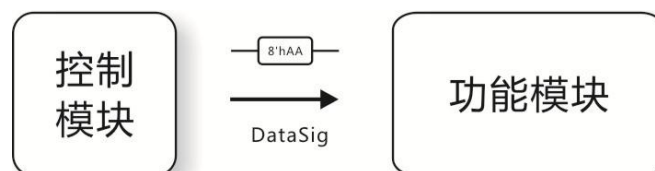


图 5.5.4 默认的数据传输。

沟通”原意是指模块之间相互传输数据的时候，由于时序表现过度协调（一开一关都非常有默契），结果让人误以为模块宛如活物般正在执行沟通。默认下模块会按照时序相互传输数据，如图 5.5.4 所示，控制模块会按照时序表现发送数据，然后功能模块也会按照时序表现锁存数据。然而“沟通”会是基于默认下的数据传输，换句话说“沟通”一般都会用到 DataSig 以外的契机信号才对。

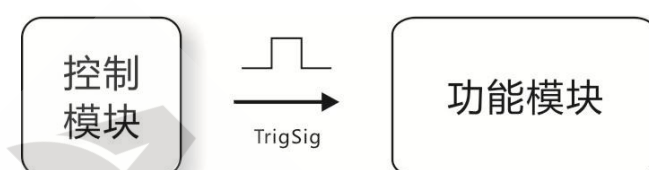


图 5.5.5 触发式沟通。

如图 5.5.5 所示，触发式沟通时最简单的沟通方式，其中控制模块会经由 TrigSig 信号发送一个时钟周期高脉冲作为契机触发功能模块开始工作。我们可以用 Verilog 这样描述，结果如代码 5.5.4 所示。

```

1.  else
2.      case(i)
3.
4.          0:
5.              if( TrigSig ) i <= i + 1' b1; // 等待触发信号的高脉冲
6.
7.          1:
8.              ..... //操作开始执行；
9.
10.         2:
11.             i <= 4' d0; // 操作完成返回步骤 0
12.
13.     endcase

```

代码 5.5.4

如代码 5.5.4 所示，操作一开始的时候会停留在步骤 0 一直等待 TrigSig，直到 TrigSig 引来高脉冲，那么 i 就会递增以示下一个步骤。当 i 指向步骤 1 的时候，操作就会开始执行，然而操作结束以后，步骤会指向 2，其中指向工具会被清零以示一次性的操作已经结束，紧接着返回步骤 0 等待下一个 TrigSig 的高脉冲，以示执行下一次性的操作。触发式沟通的应用范围很广不过都是小功用的沟通而已，例如电平状态变化。

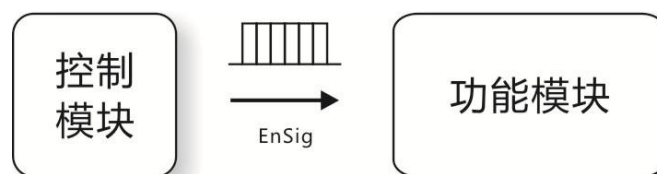


图 5.5.6 使能式沟通。

如图 5.5.6 所示，控制模块经过 EnSig 信号使能功能模块以示执行工作，其中 EnSig 拉高状态作为契机，功能模块才会一直工作。我们则可以用 Verilog 这样描述，结果如代码 5.5.5 所示：

```

1. else if( EnSig ) // EnSig 高电平，开始执行操作
2.     case(i)
3.
4.         0,1,2,3:
5.             ..... // 操作内容
6.
7.     endcase

```

代码 5.5.5

如代码 5.5.5 所示，如果 EnSig 不拉高操作就不会执行。使能式沟通算是一种比较细腻的沟通方式，其中我们必须知晓功能模块的时钟用量，假设功能模块需要 4 个时钟执行一次性的操作，那么控制模块必须拉高 EnSig 四个时钟。反之，如果功能模块有非固定耗时的操作，那么使能式沟通时无能为力的。

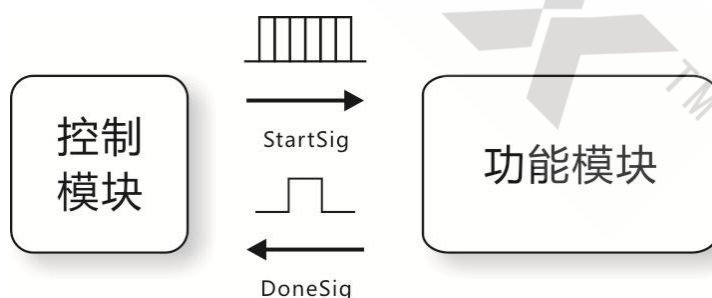


图 5.5.7 问答式沟通。

如图 5.5.7 所示，控制模块持续拉高 StartSig 作为契机功能便开始工作，当功能模块完成工作以后，它便会经由 DoneSig 发送一个高脉冲作为收工的契机，好使控制模块拉低 StartSig 以示结束一次性的操作。读者没有看错，问答式沟通其实是触发式沟通还有使能式沟通的结合体，其中使能式沟通的 StartSig 作为开工契机，触发式沟通的 DoneSig 作为收工的契机。Verilog 的描述方法如代码 5.5.6 所示：

```

1. else if( StartSig ) // StartSig 拉高，开始执行操作
2.     case(i)
3.

```



```

4.      0,1,2,3:
5.      begin .....; i <= i + 1' b1; end // 操作内容
6.
7.      4:
8.      begin DoneSig <= 1' b1; i <= i + 1' b1; end // 产生完成信号
9.
10.     5:
11.     begin DoneSig <= 1' b0; i <= 4' d0; end
12.
13.     endcase

```

代码 5.5.6

如代码 5.5.6 所示，当 StartSig 拉高的时候，功能模块开始执行操作，直到操作结束，功能会经由 DoneSig 产生高脉冲以示一次性的操作结束。当控制模块接收到高脉冲的 DoneSig 以后，它也会拉低 StartSig 以示一次性的操作已经结束。问答式沟通时应用范围最广也是最好用的沟通方式，它除了模仿顺序操作意外，问答式沟通不像使能式沟通那样必须了解功能模块的详细耗时数量，所以不管功能模块有没有固定的耗时，它也能霸王硬上弓。

经过各种各样的沟通方式讨论以后，无疑我们知道问答式沟通是可以解决 exp23~24——仿真模块的问题。在此之前读者必须好好理解，如果仿真对象失去用法模板，基本上是无法实现问答式沟通 ... 在此，好奇的同学可能会问：“笔者，为什么那么重视用法模板还有沟通方式呢？”

假设模块 A 与 B 发生沟通，模块 C 与 D 发生沟通，为了维护还有清晰所有模块的必然性，所有模块笔者都会采用问答式沟通。这样一来不管什么模块都用相同的“框架”，操作过程也好，沟通模式也罢，结构上都是一模一样的。如此一来，解析仿真信息的时候，我们可以省下许的精力还有时间。

*exp25\_simulation.vt*

```

1.  `timescale 1 ps/ 1 ps
2.  module exp25_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg [7:0]rD,rQ;
7.
8.      /*****
9.
10.     initial
11.     begin

```

```

12.     RESET = 0; #10; RESET = 1;
13.     CLOCK = 1; forever #5 CLOCK = ~CLOCK;
14.     end
15.
16.     /*****
17.
18.     reg isStart,isDone;
19.     reg [7:0]C1,rTemp;
20.     reg [3:0]j;
21.
22.     always @ ( posedge CLOCK or negedge RESET )
23.         if( !RESET )
24.             begin
25.                 C1 <= 8'd0;
26.                 rTemp <= 8'd0;
27.                 rQ <= 8'd0;
28.                 j <= 4'd0;
29.                 isDone <= 1'b0;
30.             end
31.         else if( isStart )
32.             case( j )
33.
34.                 0:
35.                     if( C1 == rD ) begin    C1 <= 8'd0; rQ <= rTemp; rTemp <= 8'd0; j <= j + 1'b1; end
36.                     else begin C1 <= C1 + 1'b1; rTemp <= rTemp + rD; end
37.
38.                 1:
39.                     begin isDone <= 1'b1; j <= j + 1'b1; end
40.
41.                 2:
42.                     begin isDone <= 1'b0; j <= 4'd0; end
43.
44.             endcase
45.
46.         /****
47.
48.         reg [3:0]i;
49.         reg [7:0]rRead;
50.
51.         always @ ( posedge CLOCK or negedge RESET )
52.             if( !RESET )
53.                 begin
54.                     i <= 4'd0;

```

```

55.             rD <= 8'd0;
56.             rRead <= 8'd0;
57.             isStart <= 1'b0;
58.         end
59.     else
60.         case( i )
61.
62.             0:
63.                 if( isDone ) begin isStart <= 1'b0; rRead <= rQ; i <= i + 1'b1; end
64.                 else begin isStart <= 1'b1; rD <= 8'd4; end
65.
66.             1:
67.                 if( isDone ) begin isStart <= 1'b0; rRead <= rQ; i <= i + 1'b1; end
68.                 else begin isStart <= 1'b1; rD <= 8'd5; end
69.
70.         endcase
71.
72.         /*****/
73.
74.     endmodule

```

exp25 是 exp23~24 的改良版，其中仿真对象应用用法模板还有问答式沟通以外，虚拟输入也使用相同的用法模板。如代码 exp25\_simulation 所示，第 22~44 行是仿真对象，内容与代码 5.5.3 相似；第 51~70 行是虚拟输入，其中步骤 0 为仿真对象输入 4，步骤 1 则是为仿真对象输入 5。

exp25 与 exp23~24 相比，最大的改变除了仿真对象以外，虚拟输入也产生很大的改变，我们知道仿真对象的功能是不固定的，亦即时钟用量随着输入内容而产生改变 ... 由于仿真对象已经采用问答式沟通，那么虚拟输入再也不用预测时钟读取 rQ 的结果（不用预先运行一次仿真），换之虚拟输入只要等待 isDone 的反馈，然后再读取 rQ 的结果即可。

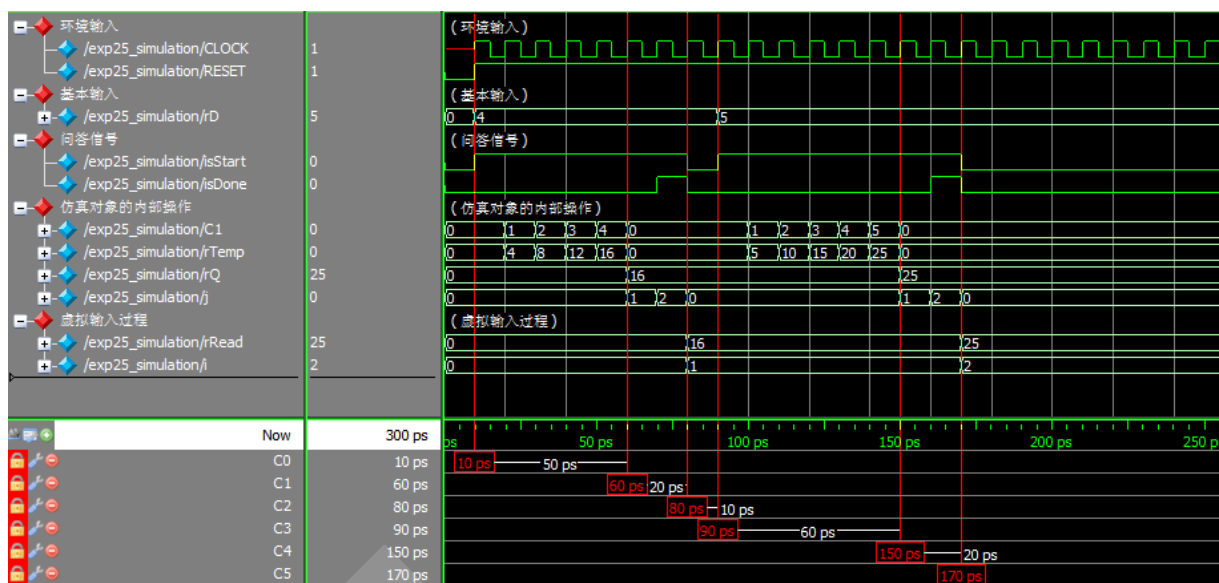


图 5.5.8 exp25 的仿真结果。

图 5.5.8 是 exp25 的仿真结果 ... 如图 5.5.8 所示, C0 指向的地方是仿真对象的第一次操作, 虚拟输入拉高 isStart 之际, 又为 rD 赋值 4。仿真对象在下一个时钟检测到 isStart 的过去值是 1, 然后开始执行操作。C1 指向的地方是仿真对象完成操作的时候, rQ 已经更新未来值 16。C2 指向的地方是仿真对象拉低 isDone 的时刻, 同时也是虚拟输入接收完成反馈 (注意 isDone 的过去值), 虚拟输入除了读取 rQ 的过去值以外, 它也着手拉低 isStart (注意 isStart 的未来值) 以示一次性的操作已经结束。

C3 指向的地方是仿真对象的第二次操作, 虚拟输入拉高 isStart 之际, 它也为 rD 赋值 5。仿真对象在下一个时钟检测到 isStart 的过去值是 1, 然后开始执行操作。C4 指向的地方是仿真对象完成操作的时候, rQ 已经更新未来值 25。C5 指向的地方是仿真对象拉低 isDone 的时刻, 同时也是虚拟输入接收完成反馈 (注意 isDone 的过去值), 虚拟输入除了读取 rQ 的过去值以外, 它也着手拉低 isStart (注意 isStart 的未来值) 以示一次性的操作已经结束。

除此之外, 我们也可以从图 5.5.8 当中知道, 仿真对象的两一次性操作都有不同的时钟用量, 如 C0~C2 的周期是 70ns (第一次操作), C3~C5 的周期是 80ns (第二次操作)。而且, 我们也知道 i 指向虚拟输入的过程, j 则指向仿真对象的过程。这一切的一切已经证明, 我们已经巧妙的使用用法模板, 还有沟通方式解决 exp23~24 的两大难点。虽然遗憾的事情是我们无法指向时钟 (无论是仿真对象还是虚拟输入), 不过作为补偿, 使用指向工具指向步骤已经我们是最大的努力了。

我们一般在建模的时候, 我们都必须**尽最大的努力使操作内容有固定的时钟用量**, 但是世事不是都能如人所愿, 有些操作必须是非固定时钟用量。操作有固定时钟用量的好处是可以仔细指向时钟, 好让必然性更加清晰。如果操作是非固定时钟用量, 指向时钟就很难实现, 换做补偿, 指向对象从时钟移向步骤。

好了, 这个小节差不多要结束了, 结束之前让我们好好回忆小节 5.5 的重点内容:

笔者使用投币的例子示意仿真存在必然性，必然性的意思是指“什么东西，在什么时候，有什么结果”，必然性之所以那么重要，因为必然性引导仿真结果走向预想所要的关键。建模阶段，我们有两件事情必须考量：

- （一）维护必然性
- （二）清晰必然性

维护必然性一般是指我们使用用法模板“固定”操作内容；清晰必然性则是指用指向工具指向操作内容的细节。此外，契机也是必然性的一种，人会某种契机命运则会发生某种改变，模块也会某种契机发生某些动作。沟通是模块作为数据信号以外的信号作为契机而发生互动，常见的沟通方式有三种：

- （一）触发式沟通
- （二）使能式沟通
- （三）问答式沟通

触发式沟通是使用 TrigSig 的高脉冲最为契机；使能式沟通时使用 EnSig 的持续拉高状态作为契机；问答式沟通则是前两者的复合体，EnSig 作为开工的契机，TrigSig 则是作为收工的契机。

最后的重点，读者如果要充分理解“必然”，读者必须站在白骆驼的角度去思考，白骆驼的为了引导蒂沙，作为前提条件，蒂沙的命运宏图必须是清晰直观，，然而白骆驼不仅要观察每个命运碎片的内容，它也要观察碎片与碎片之间的契机

## 5.6 仿真迷宫①

今天一大早，笔者与同辈们都聚集在一座圆形广场当中，广场被 5 米高的围墙团团围绕，围墙一角有一座小台，台上却站着一排神情严肃的师兄们。面向眼前的古怪情况，同辈们开始窃窃私语，声音无意间也流入笔者的耳朵里。

“今天怎么搞的？一大早就把所有人召集在这里”，同辈 A 道。

“俺怎么知道啊？”，同辈 B 道。

“嘘 ... 小声点，师兄望向这里了！”，同辈 C 道。

很显然，大伙也不知道师兄们的用意 ... 不过，笔者隐约间察觉到，隐藏在气氛之中的不详之意，直觉告诉笔者，接下来很可能会举行一起大规模的危险活动。不一会儿，一位资深的师兄开场白道。

“首先恭喜聚集在这里的诸位，因为诸位已经完成基础考验”，资深师兄道。

“今天把诸位聚齐是为了举行一项传统活动”，资深师兄道。

“诸位只要通过这项传统活动，诸位就会成为独当一面的男人”，资深师兄道。

话后，那位资深的师兄似乎有意停顿演讲，于是人群中又开始窃窃私语，笔者偷偷望向前方的同辈，有人露出兴奋的神情，有人露出担忧的神情 ... 不用说大伙也知道，师兄所谓的传统活动其实是最后考试，活动本意不复杂，亦即成者健在败者淘汰的能力测试。不过真正让笔者担心的是活动的内容，根据传统流派一贯的强硬作风，笔者相信活动本身一定存在危险性。

随之，笔者把注意力放回台上，一瞬笔者好似看见资深师兄在阴笑，眨眼几下，阴笑即消失，于是笔者便踌躇自己是不是疑心过度了，一瞬的阴笑难道是幻觉吗？就在笔者胡思乱想之际，暂停的演讲又继续道。

“诸位，请看左边”，师兄道。

不知什么时候，一座高 4 米的巨大笼子逐渐在广场的一角浮升上来，笼子的内容让在场每一位师弟都睁目结舌 ... 此刻，包括笔者还有在场的所有人，生平第一次体验到，真正的恐惧。“那是什么？”吃惊的话语不经意从笔者的口中流了出来，笼子的里边站立一只，不 ... 确切来说是一块，既不是死物也不是生物的东西，纤小的四肢支撑不等均衡的四方躯体，身上布满密密麻麻的逻辑，然而那双似目不是目，宛如无底洞般的漆黑小孔，仿佛可以吞噬一切希望，让人不颤而栗。

“那 ... 那是 ... 什么！？”，同辈 A 颤抖道。

“诸位，这是称为切糕的怪物”，师兄道。

“活动很简单 ... 要么就是把它啃掉，要么就是被它啃掉”，师兄冷漠道。

“那么，祝各位好运！”，师兄用吊丧的口气道。

师兄话音刚落，碰一声！入口的大门便紧紧锁死，人群开始喧嚣起来 ...

“可恶！这是怎么回事，拜托给咱说明清楚！”同辈 B 喊道。

“喂 ... 喂！看 ... 看哪里！”同辈 C 颤抖道。

一位同辈用颤抖不停的手指向广场一角，敲击笼子的“嘡嘡声”不停从那边传来，其它同辈们也随之望向声音的出处，人们的脸上立即染上青白两色，一瞬时间仿佛也失去流动。方才还是一副睡脸的巨大怪物，宛如饿狼禁不住食物的诱惑开始暴动起来。“嘡嘡声”一下，“嘡嘡声”又一下，众人的心脏也跟着敲击声一起在剧烈跳动。随着响起一股沉闷的金属落地声，僵住的时间再度恢复流动。

“咕咕咕 ....”，怪物声。

众人意识到，不妙的事情即将发生了，但是恐惧的情绪让它们忘了入口早已封死，然而众人却你退我挤地冲到哪里。

“开门！师兄拜托了！快开门啦！”，同辈 D 叫喊道。

“别开玩笑！那种怪物根本不是同一个次元的对手！”，同辈 E 愤怒道。

“啊 ....！”，惨叫声。

笔者向惨叫声望去，有几名同辈瞬间倒在切糕的旁边，它们两眼翻白口吐白沫。切糕发出呵呵两声以后，继续寻找下一个猎物。云之间，笔者的视线和切糕对上了，喷涌而上的恐惧麻痹全部神经，不停使唤的双脚让笔者想逃也逃不了，眼见切糕正一步步逼近，脑海却忽然浮现家乡的小花，笔者还不想死，因为笔者还没有向她告白 ... 可是笔者什么也做不了，笔者只能绝望地等待切糕的到来。

---

上述一段故事是笔者第一次执行仿真的时候所体验的经历，那时候笔者还年轻，没有任何准备就开始仿真，结果就是一次又一次的惨死。仿真既有单向也有多向，但是在默认下都是多向仿真，不管读者愿不愿意，多向仿真老早就存在一定程度的变数，而且这些变数会衍生许多可能性，结果让仿真成为一座迷宫。

有玩过迷宫探索游戏的同学一定知道，迷宫只有一个出口，找寻这座出口也是唯一——一个通关条件，除此之外迷宫也存在许多陷阱还有假出口，踏入这些陷阱或者假出的游戏任务往往都是用 Game Over 换来结局。当仿真开始执行的时候，蒂沙就会掉进迷宫当中，我们的身份好比白骆驼般，帮助蒂沙避开生命的陷阱，引导她走出人生的迷宫。

身在迷宫当中有危险的不仅仅是蒂沙本人，就连我们这些引导者也存在危险。笔者年轻的时候曾经玩过一款古董 RPG，笔者操纵勇者进入一个高难度的迷宫，老旧 RPG 基本上都没有地图辅助系统，记录迷宫除了使用脑力以外笔者也会用笔和纸张，但是不管笔者怎么努力，笔者花了 3 天的时间也走不出迷宫，其中勇者不是掉进落穴成为蜜蜂窝，就是被天花板压成纸片，仅是一些惨不忍睹的结局。



心灰意冷的笔者发誓再也不玩迷宫探索游戏,但是笔者早就中毒已深。身为玩家的我们,虽然不像勇者那样一会变成蜜蜂窝一会又变成纸片。但是,勇者每次经历死亡笔者的心情也跟着沉了几下,勇者来回死亡无数次以后,笔者的心情沉到谷底。心情接连几天都是灰灰的,笔者不仅没心情听课也没有力气作功课,感叹上学超麻烦,活着没意义 ... 真心希望有一粒陨石,忽然从天而降引起侏罗纪悲剧多好。后来被母亲教训一顿以后才重新振作。

笔者后来反思道,原来这是一种失去希望的心境,也是俗称的绝望。绝望的时候,我们什么都觉得无所谓 ... 然而,绝望给人最大的冲击就是自身的无能,不管笔者再怎么积极也好,燃烧的火焰不可能永远继续下去,挑战的心情也有消失殆尽的一天。笔者自责无能,无法帮助勇者创出迷宫,最后放弃游戏,放弃挑战。

天意弄人,十年后的今天,笔者还要面对迷宫,不过这次不是游戏迷宫而是仿真迷宫。迷宫中有一种怪物,它高 3 米,重量 200kg,长方形的身躯,最为特色是那双黑洞般的漆黑小孔,人称恐怖大王——绝望切糕。**切糕是仿真信息的集合体,压倒性的信息量是切糕最大的武器**,它时常将自己隐藏在黑暗的角落,窥视猎物捕食它们的希望。

猎物起初会不知所措,胡乱逃跑,然后颤栗会麻痹神经,目光开始浑浊渐渐失去生气,最后放弃希望。读者会认为笔者又在吹牛了 ... 什么恐怖大王,别笑死人了,哈哈! 嘘 ... 读者要取笑笔者没有问题,但是不要笑太大声,切糕最喜欢啃食充满希望的年轻人,而且自负会让人大意,最糟的结局就是成为切糕的食物。

切糕一般是指**压倒性的仿真信息**,远远超过一般人可以承受的分量。然而,产生切糕的原因就是“不正当”的仿真手段,文中的不正当不是字面上的意思,而是指没有考量前期建模,也没有准备适当的仿真手段。事实上,**传统流派就是切糕的父母**,那是因为传统流派没有适当的仿真手段,如:**不会考量前期建模,没有遵守时序表现,激励文本没有布局,激励内容没有用法模板,没有指向工具等** ... 都是切糕诞生的原因。

切糕是异世界(仿真)的怪物,它喜欢藏身在两个地方,亦即纪激励文本(仿真之前),还有解析仿真信息(仿真之间)。

处在编辑激励文本期间:

激励文本好比 c 语言唯一的一个 main 函数,读者尝试想象一下,如果什么操作都集中在 main 函数会是什么样的情况?是不是看下就觉得头疼?其实 C 语言的情况还好,因为 C 语言自身除了拥有顺序结构以外,C 语言也有许多隐性操作(编译器代劳)。相比之下,激励文本就是一场仿真环境,我们不仅需要声明出入端,我们也要使用寄存器(reg)还有连线(wire)模拟出入端。结果如代码 5.6.1 所示:

(注:reg 模拟输入端,wire 模拟输出端):

```
1. reg CLOCK;           // 寄存器 CLOCK 模拟 输入端 CLOCK
2. reg [7:0]reg_WrData;  // 寄存器 WrData 模拟 输入端 WrData
```

```

3. wire [7:0]wire_RdData; // 连线 RdData 模拟 输出端 RdData
4.
5. abc_module U1
6. (
7.     .CLOCK( CLOCK ),           // 寄存器 CLOCK 驱动 仿真对象 的 CLOCK 输入端
8.     .WrData( reg_WrData ),     // 寄存器 WrData 驱动 仿真对象 的 WrData 输入端
9.     .RdData( wire_RdData ),    // 连线 RdData 牵引 仿真对象 的 RdData 输出端
10.);

```

代码 5.6.1

一般，仿真对象在激励文本实例化以后，激励内容都会引用这些模拟出入端的寄存器还有连线，结果如代码 5.6.2 所示：

```

1. always @ ( posedge CLOCK ) // 引用 寄存器 CLOCK
2.     ....
3.     case( i )
4.
5.         0:
6.             begin reg_WrData <= 8' hAA; i <= i + 1' b1; end // 引用 寄存器 CLOCK
7.
8.             .....
9.
10.        5:
11.            begin rRead <= wire_RdData; i <= i + 1' b1; end // 引用 连线 RdData
12.
13.    endcase

```

代码 5.6.2

不过，如果仿真对象过度庞大，出入端的引用状况好比蜘蛛网上的一群蚂蚁，一群蚂蚁是指无数的寄存器（输入端模拟），蜘蛛网是指纵横交错的连线（输出端模拟）。如果激励文本没有一定的结构性在背后支持，激励文本编辑起来会变得非常麻烦，耗时又费力。此外，激励文本也会变成非常混乱不堪，然而这些混乱到极点的代码信息，解读起来会是一场非常辛苦的战斗。切糕就是隐身在其中。

总结而言 ... 仿真前，如果仿真对象有太多模块或者功能过度集中，仿真对象必定存在无数的出入端。出入端太多，不仅会影响激励文本的美观，而且激励文本也会臃肿起来。此外，激励文本的编辑工作也会变得非常麻烦，更加耗时，更加费力，最糟糕就是激励内容的清晰度会大打折扣。

期间，我们先是越来越焦急，因为太多输入端让我们一时之间喘不过气来；

- 接着，我们会越来越失意，因为不管我们怎么编辑激励内容，都无法如愿以偿；
- 后来，我们会越来越颓废并且开始自暴自弃，一边咒骂屏幕一边狂敲键盘；
- 最后，我们会萌生放弃的念头，感觉自己什么也作不好，还不如死掉算了。此刻，

我们可以断定，切糕正一步步捕食我们的希望。

切糕除了存在于激励文本（仿真之前）以外，切糕也存在于解析仿真信息（仿真之间）。笔者曾经说过变数衍生可能性，如果没有适当仿真手段控制变数，一旦变数暴走起来，变数便会无限放大可能性，从而海量仿真信息。每当我们启动 Modelsim 开始仿真，wave 界面所播放的时序结果，只是无限可能性当中的一种可能性而已，不过不管哪一个可能性，切糕都隐身其中。

解析仿真信息期间：

- 宛如洪水一般的时序结果，滔滔不绝即绵绵不断冲击我们的大脑，我们越看越焦急，因为我们无法承受压倒性的信息量；
- 接着，我们会开始寻找的切入口，然而这是海里捞针的作业，我们越找心越灰；
- 虽然找到切入口会换来短暂的兴奋，但是真正的恶梦才开始，解析作业必须同时联系仿真对象，激励内容还有时序结果，我们越是追踪流向，脑袋越是感觉要爆开；
- 越是期待流向的结局越是大失所望，因为流向大多数会进入死胡同；
- 上述作业不停轮回，我们也会越来越绝望，放弃的念头越来越旺盛，因为不管我们重复多少次都是遇见死胡同；
- 最后，燃尽希望火焰的我们会觉得仿真必死更难受。此刻，我们可以断定，切糕正在一步一步吸食我们的希望。

读者理解了吗？切糕究竟是如何潜伏黑暗，吸食我们的希望。虽然仿真之前，还有仿真之间都存在切糕，但是后者的切糕不仅数量更多，而且杀伤力也更强。形象点说，仿真之前好比勇者在草原上遇见几只小切糕，只要我们做好一定程度的准备，如：早期建模有规划，模块有用法模板，激励文本有结构性 ... 我们多多少少，也能干掉这些小切糕。

仿真之间，好比勇者走入迷宫，迷宫里不仅存在数之不尽的小切糕，迷宫也存老大级别的大切糕。勇者就算准备再怎么充足，能力再怎么高强，勇者一旦踏入迷宫，入口立即就会封死，然后就会成为无止境的消耗战。不管勇者死亡多少次，勇者都会重生然后返回起始之处，勇者的肉体虽然可以无限恢复，但并不表示勇者的精神也一样，精神状态反而会伴随轮回的次数表现越来越差。

---

仿真就是这样一个恶性质的迷宫，它不允许入侵者离开，同样入侵者的灵魂也休想离开，入侵者只要一天都无法找到出口入侵者都无法离开迷宫。入侵者经过无数死亡还有重生以后，人心跟着坏掉，直至变成一具没有希望的行尸走肉，永远徘徊在迷宫之中。虽说进入迷宫的人不是我们而是勇者，或者说蒂沙进入迷宫，然后我们要引导她走出迷宫。但是，绝望切糕不管对象是人还是神，它都会无差别攻击。

传统流派是自负的流派，也是爱拼才会赢一族。它们不会在意事先准备，它们也不在乎迷宫有多复杂（不管仿真有多复杂），它们都相信自己可以克服切糕（解析海量信息），找到出口并且征服迷宫（得到预想所要的仿真结果）。它们始终相信这个世界（仿真）不用弱者，淘汰弱者。

同样，一般参考书也有类似的问题，参考书本身不会为仿真解释过多，如果读者无法理解，无法承受，这一切只怪读者自己太弱了。笔者因此认为，参考书是切糕的帮凶，它间接伤害我们的信心还有仿真的希望。笔者就是其中一位受害者，每当气候风吹起，记忆的伤口都会隐隐作疼。

好了，这个小节也差不多是总结的时间了 ... 虽然笔者在这个章节里边，说了许多无关紧要的故事，读者会认为这是废话也不奇怪。在此，笔者是有意用故事比喻仿真的种种细节，还有仿真当下的心境情况。常规的参考书除了赠送一副简单的时序图，还有简短的解释以外，余下就是一段混乱不堪的激励文本。参考书是不会为仿真作出各种角度的解释。

仿真的本意(精髓)就是将仿真对象，激励文本，还有时序结果联系起来并且做出解析，但是仿真不是调试，它没有指向工具，也没有 C 语言般的隐性处理。此外，解析对象可能是众多可能性之中的一种可能性而已。仿真的之前，我们都要好好必须考虑自己的能耐，试问自己究竟是不是已经做好充分的准备？不然的话，仿真会是一件极为耗命的工作。

笔者曾经仿真仿到头昏脑胀，失去食欲甚至呕吐起来，笔者本来就是身体衰落的老人，健康不能拿来开玩笑，经历那么一次体验以后，笔者开始领悟，**仿真需要用健康来交保**。于是，笔者开始寻找一种不伤害精神和肉体仿真手段，想着想着，笔者开始回忆往事，那段让人极度厌恶的迷宫探索游戏。

笔者将，仿真形容为恶性质的迷宫，海量的仿真信息形容为切糕，虽然自认有点幼气，不过却肝肺非常恰当。我们是白骆驼(玩家)，仿真对象是蒂沙(勇者)，然而我们的任务是引导蒂沙走出迷宫，不过隐藏在迷宫之中存在无数切糕怪物，它们不仅会攻击蒂沙，它们也会攻击我们 ... 它让我们心灰意冷，渐渐失去希望(其实这是一种精神伤害)。

切糕除了发动精神攻击以外，切糕也会发动物理攻击。仿真原本就是并行操作，解读时序之际，我们需要追踪数个信号，这种行为无疑我们会耗费更多脑力还有集中力 ... 此外，我们还要同时联系仿真对象，还有激励内容，因此我们不得不全神关注。当我们**长时间盯着屏幕看时序**，切糕就会暗地里攻击我们，我们会感到**眼睛不适，脑袋发疼**。如果我们稍微分散集中力，情绪会逐渐焦虑，心情也不能平静，而且还会感觉疲惫。

切糕不是笔者妄想的怪物，它确实存在 ... 试问读者，是否曾经对仿真产生厌恶感，或者对仿真失去信心，又或者一作仿真就头脑发疼？有的话，那就证明读者曾经被切糕攻击过。我们受到切糕攻击以后，就会渐渐失去希望，心情会非常低落，心理还有生理都会觉得疲惫，如果长期处在这种心境，人会作出一些伤己的行为，其实一点也不奇怪。仿真除了存在切糕吸食我们的希望以外，其实仿真也是一种毒瘾，它会侵蚀我们的心。

## 5.7 仿真迷宫②

不 ... 不要呀！当笔者再一次醒来的时候，笔者发觉自己躺在病床上，笔者不禁在心里询问自己究竟是如何来到这里。笔者开始寻找记忆的蛛丝马迹，脑袋好疼 ... 脑袋仿佛对笔者发出抗议一般拒绝运动，笔者立即按住太阳穴缓和头疼。渐渐地，笔者开始意识到周围充满无尽的呻吟声，笔者想起身探个究竟，可是一名身穿白大褂的中年即时阻止笔者。

“小哥 ... 现在还不能起来！”，中年道。

“先 ... 先生，发生什么事了？”，笔者问道。

“...”，中年沉默道。

“这里！究竟是怎么回事！？”，笔者大喊道。

笔者是一位懒惰发脾气的男人，难得今天笔者就是压抑不了冲动，中年摸了几下下巴，稍微仰头思考一会后，然后开始道出事情的由来。中年是大夫，它说笔者已经在病坊昏迷了许多天，笔者的确感觉浑身充斥疲态感。

“先生，别管小弟 ... 那些人怎么办呢？”，笔者急忙问道。

“已经没有希望了 ... ”，中年遗憾道。

“没有希望？先生没听见呻吟声吗？”，笔者反驳道。

“小哥，仔细听听 ... ”，中年指示道。

话后，笔者立即竖起耳朵聆听呻吟声 ...

“俺不要切糕了！俺不要仿真了！绕绕俺吧！”，同辈 A 诉求道。

“嘻嘻嘻，仿真仿真仿真仿真！”，同辈 B 兴奋道。

笔者用疑惑的眼神望向大夫，大夫摇头表示它们已经没有希望了。笔者从中了解到，这里所有人都是那场活动的遇难者，在场所有人都会被切糕吞噬希望，醒来以后就成为那副模样。

“除了小哥以外，其它人都坏掉了 ... ”，中年遗憾道。

“坏掉了？什么坏掉了？”，笔者问道。

“抱歉 ... 正确来说，它们都是仿真癌病患，无药可救。”，中年冷漠道。

“仿真癌！？”，笔者重复道。

大夫说道，笔者也是其中一名病患，不过很庆幸的是笔者还能保持自我。所谓仿真癌病是一种心理癌症，“强迫仿真，恐惧仿真”的矛盾压力会不断在心理扩大，直至压垮病患的精神。强迫仿真一种上瘾心态，每当我们建模的时候，未知的建模内容会让我们心存不安，余下也只有仿真能消除这份不安 ... 但是 Verilog 不是顺序语言，仿真也不是调试，所以我们无法一边建模一边仿真。



这份无法消除的不安会愈来愈糟糕，仿真的冲动也会愈来愈强烈，心理的失衡也会愈演愈烈，这种情形会持续到建模结束，仿真完毕为止，然而这份解放感会依存在心理深处。上述现象来回重复一定程度以后，人们就会养成对于仿真的依赖，也是俗称的仿真上瘾。这种感觉好比我们憋尿许久，解放以后我们会产生莫名的快感，为了再度体验这份快感，我们会养成憋尿的坏习惯。

恐惧仿真是一种恐怖心态，每当我们执行仿真之际，庞大的信息量越让我们愈加焦急，再加上屡试屡败的挫折感，心理深处就会对仿真产生剧烈的抗拒感。上述现象重复一定程度以后，我们会越来越畏惧仿真，然后我们会妄想逃到没有仿真的角落。这种感觉好似我们被蛇咬，从此以后都会莫名畏惧草绳，走路再也不会靠近草丛。

强迫仿真还有恐惧仿真，原本是一种相斥的极端心态。不过很遗憾的事，如果没有恰当的仿真手段，同时患上两种心态是绝无法对避免的事实。此外，这两极端又矛盾的心态会不停左右我们的心理，折磨我们的心理 ... 这种感觉好比灵魂的左天使与右恶魔同时耳语道，“这样做！”，“不要这样做！”，我们最终会承受不了幻听把头撞墙以求一份安静。但是，矛盾的窃窃私语不仅不会停止，反而会愈来愈强烈，直至心理崩溃。

或许有人一生也都无法发现，自己处在强迫与恐惧心态之间执行仿真，然而他们却会觉得仿真非常疲惫 ... 这种疲惫感却是仿真癌正逐渐侵蚀内心的最好证据。有些同学可能会好奇道，如果内心过度被侵蚀，结果究竟会怎样？这些一人一旦接近仿真就会开始抓狂，情绪不稳定，出现自毁倾向 ... 是不是觉得很可怕呢？

同学可能又会觉得笔者在瞎扯了 ... 非也！笔者只是述说过来人的经历而已，绝无瞎扯成分。那么究竟是什么原因让人患上仿真癌（强迫 vs 恐惧）？我们又该怎么避免？然后又该怎么治疗？

---

试问读者是否有过这样的感觉 ... 我们在建模的时候，心理都会出现莫名的忐忑感，仿佛每写一行代码，呼吸也跟着急促起来，感觉自己快要窒息。这种感觉好似读者返家途中，夜路不仅散发上一丝丝的凉意，而且还安静得出奇可怕，故障的路灯一闪一闪地更加渲染诡异的气氛，一路上都觉得有东西在背后尾行 ...

这是不安的生理反应，我们之所以会感觉不安，那是因为我们**过度意识未知的建模内容**。很多时候，我们虽然理解代码的具体意义，但是我们却**无法预测代码的时序表现** ... 就是这种看不到，听不到的情况，让心里深处萌生不安。然而又竟是什么原因导致我们，“仅理解代码目的却无法预见时序表现呢？”，答案是，亦建模内容不清晰，模块的表达能力不强，结构散乱等一系列原因。

笔者与许多同学一样，都是经由传统流派认识建模，认识仿真。传统流派是自负的流派，自信的它们相信自己不用准备就可以克服建模，克服仿真，然后成为勇者，然而成为勇者仅是一小部分强人而已 ... 弱者多数会被淘汰。笔者当初也不知道，**原来前期建模对仿真影响至深**，笔者是一位弱者也经受不了折磨，它甚至伤害笔者的小小心灵，结果萌

生退意。

笔者曾经有过放弃学习的冲动，原来那是笔者的潜意识在害怕不安，潜意识选择逃避来自我保护，逃避不是什么可耻的事情，那是生物的求生反应，还不如说回避才是人之常情。但是笔者就是不甘，经过长时间的思考以后，笔者才发觉**不安的源头，就是没有做好前期建模的准备。**

**好的模块都有直接的表达能力，清晰的内容，望眼一看就知道模块的功用**，稍微浏览也能知道大概的意思。此外，清晰的内容不仅让人理解代码目的，也能让人预测代码的时序表现。这里所谓的预测，是指“**脑海的时序结果**”，而不是仿真的时序结果 ... 我们虽然不能一边建模一边仿真，但我们却能一边建模，一边用脑仿真代码。如此一来，我们就能消除未知内容所导致的不安心理。不安得以消除，仿真强迫症也会自然无药而愈。

---

“原来如此 ... ”，笔者点头道。

中年大夫，滔滔不绝的解说完毕以后，笔者立即陷入思考。用脑仿真模块，人脑的记忆容量不仅少而且储存时间也非常短暂，而且人脑的记忆还是非常暧昧，要用脑袋实现 Modelsim 一样的工作，说实在这是一种大胆的尝试 ... 想着想着，头脑又开始发疼了。中年大夫见状，便急忙劝说放松 ... 可恶！还不是那家伙的错，说了那么多道理，好奇心旺盛的笔者不去思考才怪！

头疼稍微缓和后，笔者继续话题。

“那么，如何解决恐惧仿真？”，笔者问道。

“这 ... 这个，那个 ... ”，大夫嘟囔道。

中年大夫显然有难言之隐，它是担心笔者的身体？还是考虑其它顾虑呢？中年大夫犹豫一会儿后，以咳嗽声示意话题继续，笔者过眼一视，中年大夫却露出方才没有的坚毅神情，它似乎在心理深处做好什么觉悟似的 ...

仿真是一座恶性质的迷宫，笔者曾经这么形容过 ... 此外，还有一种名为切糕的恐怖大王，它们都是恐惧仿真的源头。一般上，我们都是建模结束以后才执行仿真，**仿真不是调试，不是一键 <F5> 或者 <F6> 就了事，我们必须事先预设，然后准备激励文本，最后才能启动仿真。**

由于仿真是建模以后的工作，仿真对象也是建模完毕的模块，结果体积都会非常庞大。这里所谓的体积庞大是指，仿真对象有过多模块集中在一次，笔者曾经说过，如果过多模块集于一身，仿真对象就会有很多出入端。如果出入端太多，实例化仿真对象会是一件非常辛苦的劳动，而且激励文本的编辑工作也是非常猥琐，直至让人抓狂。此刻，切糕在仿真前出现了。

猥琐或者辛苦的劳动，笔者相信只要紧咬牙关就能挺过去 ... 然而，读者也许不知道，



切糕给予我们的第一次打击，事实上我们已经流失过半的体力与精神。形象点说，勇者为进入迷宫之前，它就得先消耗过半的 HP 与 MP 在路途的小切糕身上。勇者流失 HP 只要喝喝几口回复药就能立即回复，反之流失 MP 却使用回复药补充确是有限的手段。为了维护游戏的平衡性，一般的 MP 回复药都比 HP 回复药还要贵上 10 倍，无奈之下笔者必须靠休息来回复 MP。（千万别吐槽现实世界没卖回复药）当勇者进入迷宫的时候，真正的噩梦才总算开始 ...

勇者进入迷宫之际，也是仿真开始的时候 ... 迷宫是仿真信息的形象形容，一个可能性的仿真结果其实仅是总体仿真信息的一部分而已。[面向错综复杂的仿真信息](#)，我们的**第一反应就是胆怯**，亦即切糕给予我们的第一次攻击。常规的仿真手段都是没有指向工具，仿真信息宛如一片茫茫的大海，然而要在密密麻麻的信号之间寻找一个入口，就是大海捞针的作业。此刻切糕给给予我们第二击攻击。

“找到入口了！”，当我们兴奋喊着喜讯，所有切糕就会被希望所吸引，然后虎视眈眈的切糕便会开始行动。

- 20. 没有指向工具，容易迷失方向；
- 21. 没有遵守时序表现，迷宫会非常错综复杂，而且崎岖难行（信号没有对齐性）；
- 22. 没有布局的激励文本，迷宫会上下颠倒（联系时序结果与激励内容很难）；
- 23. 没有控制变数，迷宫会进一步扩展（仿真信息海量）

上述四点是常规仿真手段所犯下的错误，读者可能无法留意，如果我们无视上述四点，使劲追着一一条一条信号在跑，此刻我们的体力和精神也会大把大把在流失。此刻，切糕藏身在黑暗之中，无限攻击我们。假设，我们很庆幸拖着半死的状态来到迷宫的终点，心理正要欢喜的时候，谁知这是一个错误的仿真结果 ... 我们的心会立即跌到谷底，然后咒骂自己那么辛苦，既然才得到一个渣渣而已。此刻，切糕 BOSS 一击便 KO 我们。

上述情况经过无数次以后，还是一直在仿真之间不停绕圈圈。身为白骆驼（玩家）的我们可能会也越来越焦急，很想骂脏话。其实，这些都是正常的生理反应，因为我们在无意识之下，不断累积对于仿真的恐惧 ... 最后，直至我们用尽一切希望，仿真的恐怖就会深深烙印在我们的心底深处。

为了改善上述的问题，为了引导蒂沙走出迷宫，笔者做了以下几点补救手段：

- 24. 应用指向工具，维护迷宫的方向；指向工具好比路牌，它很大程度为仿真树立大概的方向，读者千万表小看路牌，失去它们交通系统会变成非常混乱。
- 25. 应用时钟，遵守时序表现，好使迷宫整齐；使用时钟驱动仿真，信号就能遵守时序表现，为此信号不仅对齐，时序表现会散发浓厚的时序香。
- 26. 激励文本有布局，保证迷宫有一定的结构性；激励文本有结构性，激励内容会更加直观。

27. 控制变数，收缩迷宫；变数会衍生可能性，可能性就是仿真信息的容量，控制变数也能间接缩小仿真信息。

当我们完成上述 4 点的补救手段，基本上仿真已经从恶性质的迷宫变成良性质的迷宫，切糕的数量也会大大缩小。

---

话到这里，中年大夫的然后脸上露出一丝忧愁 ...

“不行 ... 这样还不行！”，大夫喃喃道。

“怎么不行呢？这些补救手段不是大大改善仿真了吗？”，笔者好奇道。

“这种程度的补救手段只是治标不治本而已”，大夫道。

“什么意思？”，笔者问道。

“仿真的确已经得到改善，但是却不能改善人们对仿真的恐惧 ... ”，大夫摇头道。

笔者开始思大夫话中的含义。我们虽然可以透过补救手段让仿真这座迷宫变得美丽清晰，不过不管我们再怎么改善迷宫，迷宫始终还是迷宫 ... 仿真真正让人们感到恐惧的是它压倒性的信息量。换言之就是，人们的信息吞吐量有极限，简单点说，过多的仿真信息容易造成，脑袋的消化不良 ... 为此，人们无意识畏惧起来。

读者仔细思考一会，笔者为何将海量化的信息量形容为切糕呢？切糕原本是新疆的甜点，但是那夸张的分量，不仅让 4 个大男人抬得动，即时 10 个人吃尽 10 天 10 夜也不一定吃完。不管是什么山珍海味，我们只要进食过度也会反胃，再看下去就想吐 ... 脑袋和胃袋也有同样的道理，用脑超过极限就会开始反脑（烦恼），然后觉得头昏脑涨。反胃也好，反脑也好，那是身体为了自保，使命释放“恐怖信息”刺激脑袋，刺激胃袋 ...

正常的家伙遇见恐惧会适可而止，不过传统流派那些失常家伙会使劲强迫自己，干下去！do 下去！直至“恐怖信息”累积越来越多，成为一生都消除不了的“心理烙印”。

“对不起，老夫有点激动了 ... ”，大夫抱歉道。

中年大夫尴尬的摸着后脑勺，不过它脸上的一阵泛红却变现它的愤怒 ... 为此笔者也产生共鸣，传统流派每年都有数万的新手入门，想必多少幼苗曾经被它们摧残过，现场不停呻吟的病患就是最好的铁证。天职为救人的大夫却也只能白白看它们送死而已，可见中年大夫已经在心理深处累积许多悔恨。如今它却向笔者坦白一切，笔者不禁开始担心起来 ...

“大夫，透露这些事情出来，真的没有关系吗？”，笔者问道。

“怎么没有关系 ... 老夫今晚就要跑路，离开这处鬼地方！”，大夫决意道。

“跑路！？为什么？”，笔者反问道。

“它们不会放过叛徒！”，大夫道。

“既然如此，为何又将秘密告诉小弟呢？”，笔者好奇道。

“老夫再也无法瞒着良心做人 ... ”，大夫感叹道。

“小哥是这场劫难的幸存者，可能这是某种缘分吧 ...”，大夫继续道。

笔者如今可以却侥幸活了过来，可见补救手段确在遇难之际确实已经发挥功效。根据笔者的理解，**脑袋的吞吐量因人而异**，切糕杀人不仅会吸食我们的希望，它还会将自身的海量信息**强制灌输进入猎物的脑袋**，直至脑袋当机。笔者自问脑袋不是怎么好的人，常理上遇见切糕理应必死无疑，可是补救手段却为笔者的脑袋腾出一定程度的消化空间 ... 简言之，补救手段**过滤掉大量无用的仿真信息**，因而笔者才可以勉强承受切糕的攻击，保住小命。

遇见切糕是绝对的恶梦，也是笔者生平第一次品尝恐惧 ... 如今，切糕的恐怖已经深深烙印在笔者的心理深处。笔者是脆弱的男人，绝对没有自信能诺然无事再度面对切糕，面对仿真 ... 在此，笔者完全理解大夫的忧虑，**补救手段充其量只是治标不治本，就算我们把大量的无用信息过滤掉，余下的海量信息，也是普通人无法承受的级别**。除非大脑经过强化升级，否则休想面对面啃下所有仿真信息。

在此，笔者可以断定，恐惧仿真的源头就是人类的弱小，这种感觉好比经历洪水，地震等大自然灾害 ... 面向绝对力量的面前，人类也只能一边恐惧一边打颤而已。海量的仿真信息有如天灾一样，让人畏缩，让人逃避，让人不敢直面，唯一例外就是弱小的人类变成超人，那么人类就可以正面单挑仿真，不过这也是非现实的妄想而已。

真是一起沉重的话题，笔者不经意产生疑惑 ... 这种摆明的事实，难道传统流派不曾留意？不曾想尽办法改善吗？中年大夫见状，也是无力地摇摇头 ...

“它们即使知道，也没有理由这么作 ... ”，大夫道。

“为什么！？” ，笔者怒喝道。

“小哥你知道古帝国是如何支配人民吗？” ，大夫提问道。

“很抱歉，小弟的历史不好 ... ”，笔者答道。

“恐惧，还有愚昧就是最好的支配工具 ... ”，大夫冷漠的回答道。

这一番话突然让笔者豁然大悟 ... 这个世界上有数以万计的人从事仿真的活动，仿真会有百花齐放，百家争鸣的流派才是自然现象，然而现实却是传统流派“一派独大”。这种结果，不管怎么想处处都有违和感，难道是传统流派垄断仿真！？事实的背后可能隐藏巨大的阴谋，究竟是莫大的利益，还是绝对的权利 ... 为了维护这份力量，究竟要牺牲他人到什么程度？想到这里，笔者不禁后背发凉。

“如今，老夫已经将真相泄露出去，不得不立即跑路，不然会糟灭口！”，大夫认真道。

“但是，同辈们该怎么办？” ，笔者问道。

“它们两眼无光是心智失律的表现，基本是没有希望了 ... ”，大夫遗憾道。

“可是 ... 可是 ... ”，笔者焦急道。

“小哥，它们活着只会更痛苦 ... 小哥帮它们解脱吧！”，大夫无情道。

话完，中年大夫向笔者递过匕首，笔者不禁双手颤抖起来 ... 小明是第一位向笔者搭话的同辈；小王很喜欢讲冷笑话；大牛最讲义气；阿福为人正直；还有 ... 还有，双眼忽

然湿润起来，笔者强忍眼泪逐个逐个为最好的同辈解脱，它们每个人都是笑着离开 ....

---

入夜的天空黯淡无光，笔者的心情必也是一样 ... 中年大夫为同辈们善后以后就连夜跑路去了，而且它还特别告诫道，切糕袭击他人以后，心里会产生一段时间的空洞，期间千万不要胡思乱想，绝望之意会乘虚而入，让人作出傻事 ...

笔者一个人漫无目的来到附近的山崖边，山崖对面是一望无际的彼岸，哪里没有烦恼，没有恐惧，只有拥抱一切的安静。如今，双手还残留夺走生命的触感，不是是否夜风寒冷，笔者的肩膀一直在颤抖不停 ... 大夫没有说错，恐惧还有绝望早已深深嵌入，笔者已经失去面对仿真的勇气还有信心，笔者不经意自嘲，那个半死不活的自己还有什么作为呢？

除了自身的问题以外，熟知真相以后，传统流派已经成为巨大的噩梦，想必以后一定不得安眠 ... 与其庸人自扰，还不如抛开烦恼，眼下只要笔者再踏出几步就会得到解脱。啊 ... 笔者看见同辈正向这里招手道，来吧！来吧！

第一步，笔者毫无犹豫；  
第二步，笔者期待着；  
第三步，笔者露出微笑；  
第四步，笔者的愿望终于实现了。

忽然，一只强而有力的右手捉住笔者，熟悉的声音映入耳膜。

“孩子，千万别做傻事，明天还有希望 ...”，它道。

## 5.8 主动思想

当笔者再度睁开眼睛的时候，笔者发现自己正躺在花田之中，笔者不仅怀疑自己是不是去了天国？这里充溢怀念的味道，还有似曾相识的景色，一瞬笔者立即回忆曾经有过的梦境！笔者立即起身，四处寻找它 ... 然后笔者在不远之处，看见熟悉的身影。它在等候笔者 ...

面对它，笔者有数之不尽的问题，不过它仅是点头示意理解，然后它用右手向下指。笔者自然地，顺着那个的地方望去。

“远处看去，这里是什么？”，它道。

“一处小花田 ...”，笔者答道。

“近处看去，这里是什么？”，它道。

“有小虫，有花儿 ...”，笔者答案。

笔者之前也说过个体与整体的视角问题，它的用意正是如此。自古以来都是**个体造就整体，而不是整体造就个体** ... 如果按照这个角度去思考，仿真是一个整体，个体包括仿真对象，激励文本，还有时序结果。**解析仿真信息就是站在整体的视角去了解仿真情况**，解读仿真情况，不过仿真信息是在是太庞大，远远超过常人可以承受的程度。为此，笔者才会烦恼，然后烦恼把笔者带来这里。

“远处看去，哪里是什么？”，它道。

“还是一处小花田 ...”，笔者答道。

“近处看去，哪里是什么？”，它道。

“还是有小虫和花儿 ...”，笔者答道。

“没错，就是如此。”，它道。

话后，笔者开始思考其中的含义 ... 整体花田就是因为无法一眼望去，才会看成基础小花田。反过来说，如果一座整体无法直接消化，我们必须设想**分开好几个个体逐步消耗**。笔者当然明白整座仿真可以分化成为无数个体仿真，在此笔者必须用到低级建模的准则，因为低级建模的准则是以“功能”为单位分化模块。

话虽如此，但是笔者还欠缺关键的东西 ... 简言之，当整体被分化以后，**整体就会失去集中性，还有连接性** ... 这种感觉好比一个大家庭，因为利益纷争，最后被迫分为数个小家庭，从此不相往来。然而那份关键的东西即是 ... 即时大家庭分化为数个小家庭，它们之间**依然也能保持集中性与连接性**。笔者相信，那个“关键的东西”是解决仿真的良药。

“孩子，这是什么小虫？它有什么特征？”，它道。

“这是蜜蜂，它有剧毒的尾刺。”，笔者答道。

“孩子，那是什么小虫？它有什么特征？”，它道。

“那是蝴蝶，它有华丽的翅膀”，笔者答道。



瞬间，一股念头让笔者不经意反问自己 ... 蜜蜂还有蝴蝶都是昆虫，然而笔者的认识却是：

昆虫 + 剧毒的尾刺 = 蜜蜂；

昆虫 + 华丽的翅膀 = 蝴蝶；

奇怪，实在太奇怪了 ... 这究竟是怎么一回事？为什么笔者会认为昆虫加剧毒的尾刺就是蜜蜂？昆虫加华丽的翅膀就是蝴蝶？如果思考换做计算机，不管是那个昆虫组合，计算机绝对不可能得出蜜蜂还有蝴蝶的结论。想着想着，领悟的“原来如此”不经意从嘴巴流了出来 ... 剧毒的尾刺是蜜蜂最大的特征，华丽的翅膀是蝴蝶的特征，原来笔者是以特征去认识，还有分辨昆虫。

话虽如此，究竟又是什么原因导致笔者用特征去认识昆虫？好奇，还有求知的冲动，不停驱使思考高速运动，不过还是找不到原因 ... 无奈之下，笔者用诚恳的眼神哀求它，酷到极点的它还是一贯我行我素的作风，无视笔者的恳求 ...

“孩子，这个小花田有什么？”，它道。

“一只蝴蝶，两朵茉莉花。”，笔者回答道。

“那个小花田又有什么？”，它道。

“一只蜜蜂，两朵向日葵。”，笔者答道。

开窍的啊一声，肩膀也跟着弹跳起来 .... 假设这里有小花田 A，与小花田 B，然而：

小花田 A = 1 蝴蝶 + 2 茉莉花

小花田 B = 1 蜜蜂 + 2 向日葵

认识小花田与认识昆虫其实是使用同样的道理，简单点说 ... 小花田 A 最大的特征就是 1 只蝴蝶，还有 2 朵茉莉花；小花田 B 的最大特征则是 1 只蜜蜂，2 朵向日葵。假设小花田是一个整体，那么昆虫还有花朵都是个体，然而笔者却以“昆虫的特征”，还有“花朵的特征”等两个个体特征，分辨整体 A 与整体 B。

换句话说，1 只蝴蝶还有 2 朵茉莉花都是笔者认识小花田 A 的深刻个体印象；换之，1 只蜜蜂还有 2 朵向日葵都是笔者认识小花田 B 的深刻个体印象；打从骨子里，笔者从来就没有认真去记忆小花田（整体），反之小花田的昆虫与花朵（个体），让笔者更加在意，更加注视。

这种情况，笔者好似在哪里看过报告 ... 打铁趁热，笔者立即闭上双眼全面搜索记忆的资料库。笔者曾经读过一份有关人脑记忆的科学报告，人脑有两种记忆能力，其一是清晰记忆，其二是模糊记忆。清晰记忆属于左脑的记忆类型，它好比计算机的内存，非黑即白的逻辑内容。模糊记忆属于右脑的记忆类型，保存似乎，还像等暧昧内容。笔者非常好奇，为什么人的脑袋会有两种不同能力的记忆力？

普遍上，我们会认为清晰记忆比模糊记忆更加有用，但是事实却是相反。假设我们用清晰记忆去认识小花田 A 与 B 的话，那么我们**必须记忆小花田 A 还有小花田 B 的所有细节**，然而常人是**无法办到这起壮举**，估计还没有记忆到一半我们的脑袋就会当机了。根据笔者的猜想，清晰记忆必须耗费大量脑力集中精神，然后刻意记录每个细节，为此记录工作还没有结束，大脑就会超载。

相反的，假设我们使用模糊记忆去认识小花田 A 与 B 的话，我们**用不着记忆所有细节**，换之我们记忆**印象深刻的个体特征**即可，例如小花田 A 有蝴蝶与茉莉花，小花田 B 有蜜蜂与向日葵。然而，花田的其它细节又该怎么办？其**它印象不深刻的细节，模糊记忆会自己 YY 脑补**。

清晰记忆也有显记忆，短期记忆，左脑记忆等别名；模糊记忆则有，长期记忆，右脑记忆等别名。报告也说过，左脑擅长顺序，处理逻辑的处理工作，右脑则是用来妄想。现代人很明显是倾向左脑使用，左脑的要求效率就是快狠准三字，这种现象自然可以从日常生活中观察得到，顺序语言就是最好的例子。

想着想着，笔者发觉神向花田吹气，不一会儿，一层蒙蒙的云雾就覆盖整片花田，这种感觉好比在照片上面打上马赛克似的，景色开始模糊起来 ... 接着，它用右手指向花田某处。

“孩子，那处小花田有什么？”，它道。

“雾蒙蒙的 ... 看不清楚 ... ”，笔者回答道。

过后，神又向同处的花田吹气，云雾部分即消失，模糊的景色也有好几个部分清晰起来。

“孩子，那处小花田有什么？”，它道。

“一只蝴蝶，两朵茉莉花 ... ”，笔者回答道。

“回答得很好。”，它道。

一会给花田打上马赛克，一会又消除花田的马赛克，笔者不禁迷惑起来 ... 花田被云雾遮盖的时候，**所有景色就会模糊起来，昆虫还有花朵等个体特征也会跟着模糊**；但是，接着，神也仅只是消除部分云雾，**好让花田的个体特征清晰出来**，然而笔者立即就能辨认出，那处是小花田 A。模糊？清晰？笔者理解了，**是清晰！是个体的强烈印象**！让笔者用例子来解释吧 ...

假设有 CONTROL 七个英文字母，如果我们用清晰记忆少记录一个 T 字母，余下 CONTROL 只是没有意义的断片记忆。换之，如果 CTRL 给予模糊记忆有强烈的印象，那么只要我们**好好保持 CTRL 的清晰**，就算我们没有完整的 CONTROL 七个英文之母，我们都有办法分辨 CTRL 就是 CONTROL。

原来如此！原来如此！领悟的四个字就像口水般不停从口角流出来。那些困死笔者的难题，它只是轻轻为笔者点一下，结果就把笔者点通了，神就是神，笔者真是佩服到五体



投地。

现代人都有倾向左脑的用脑习惯，左脑的缺点就是小容量的清晰记忆。然而，这种用脑习惯也不知不觉之间流入仿真之中 ... 我们之所以觉得仿真信息快要塞爆脑袋，那是我们在无意识之间，使劲将仿真信息送往左脑。笔者说过，清晰记忆必须记录每个细节才能发挥功效，为了这个目的我们需要耗费大量的脑力用于集中。

结果而言，左脑不仅要一边使劲集中精神，它也要一边使劲记忆信息，最后它还要使劲处理信息，左脑就这样超载工作然后当掉。左脑为了自保，无间断发送“恐怖信息”阻止我们不要再折磨它了！换句话说，我们之所以恐惧仿真，原来是左脑为了自保才导致的问题。简言之，恐惧仿真其实是用脑不当（用脑不平衡）的病因 ... 为此，我们必须寻找一个用脑平衡的仿真手段。

平衡用脑简单而言就是讲一半以上的“记忆信息”送往右脑以致减轻左脑的负担。有些同学可能会觉得疑惑，右脑究竟要如何分担左脑的压力 ... 即是，运用我们的想象力。想象力？没错，就是想象力 ... 想象力也是模糊记忆的能力之一，例如笔者认识车子会有以下几个特征：

汽车 = 四个轮子 + 一家车身 + 前后两面遮罩镜

每当笔者识别“汽车”，笔者只要回忆上面 3 个特征，其余细节都由想象力自行填充。

名牌汽车 = 四个轮子 + 一家车身 + 前后两面遮罩镜 + 宾士

再者，笔者认识名车也只是所追加一个车标特征而已，其它细节任由想象力填充。接下来是时候切入正题了，让笔者用模块来举例：

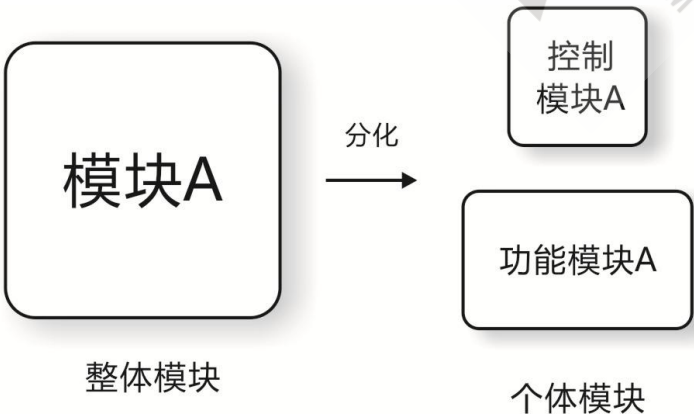


图 5.8.1 按照建模技巧分化的模块。

如图 5.8.1 所示，假设我们有一个整体模块 A，首先它必须经由低级建模的准则“一个模块，一个功能”，按照功能这个单位划分数个个体模块，如控制模块 A，功能模块 A 等。低级建模是一种有规划的多模块建模，个体模块等均匀划分；换之常规的多模块建

模是随意，并且不规则将整体模块划分为数个不等均匀的个体模块。形象点说，低级建模会将大切糕切分为**数个个相同大小分量的小切糕**，大面积的小切糕就少重量，小面积的小切糕就多重量；反之，常规多模块建模会将大切糕切成稀巴烂，好像被大炮轰炸过一般。

按照“功能”划分整体模块是，清晰模块，控制变数的前期工作。根据低级建模，模块会是 xx 控制模块，yy 功能模块等，这是其一提高模块表达能力的方法，在旁随便一望，我们就会知晓个体模块的大概功用。除此之外，笔者也在前面讲述过，**分化模块也是将过度集中的变数，平均分化开来 ... 以致降低变数所引起的意外（错觉可能性）**。

```
1. always @ ( posedge CLOCK )
2.     ....
3.     else
4.         case( i )
5.             0:
6.                 begin ...; i <= i + 1' b1; en
7.             1:
8.                 .....
9.         endcase( i )
```

代码 5.8.1

此外，个体模块的内容都会应用相同的用法模板来维护结构性，效果如图 5.8.1.所示。某位同学曾经这样询问过笔者，笔者设计的模块，框架怎么都一模一样呢？与其说是框架，笔者更希望那位同学认为为“结构”还要好。类似 5.8.1 的用法模板是基于仿顺序操作，它除了提供最基本的顺序支持以外，它也替代那传统的状态机。

此外，如果模块内容有相同的结构，想象工作也会更加轻松，例如笔者记忆名车 A，与名车 B：

名牌汽车 A = 四个轮子 + 一家车身 + 前后两面遮罩镜 + 宾士  
名牌汽车 B = 四个轮子 + 一家车身 + 前后两面遮罩镜 + 本田

汽车基本上都有相同的结构性，然而名车 A 与名车 B 的差别，笔者只是识别“宾士”还有“本田”两个车标特征而已。用法模板除了可以帮助想象以外，用法模板有最根本的功能，那就是指向功能。常见的用法模范都有 i, j 等指向工具，笔者也多次强调指向工具的重要性 ... 指向时钟，指向步骤，还是指向过程都是“**清晰**”模块内容的重要的工作。

那么，重点来了 ... 清晰？我们究竟是为了清晰什么才作清晰？答案非常明显，**清晰个体的特征，强化个体的印象**。强化个体的印象？没错 ... 对于想象力（模糊记忆）来说，**个体的强烈印象是不可或缺的拼图，强烈的个体印象会带动周边的模糊细节，个体印象越强烈，整体的面貌越加清晰。**

举例而言，人类在识别它人脸部的时候，由于面部有太多细节信息，那样的工作左脑很难胜任 ... 在此，右脑就大展拳脚了。右脑不会记录所有面部细节，换之右脑会记录人脸部的局部特征，例如左眼下的泪痣，性感的樱桃小嘴等 ... 然后某天，类似面部特征的女人出现在笔者的面前，笔者就会产生好像，似乎的感觉。换句话说，局部印象越是强烈，想象力（模糊记忆）越能发挥功效。

所以说，笔者在建模的时候会使劲提高模块的表达能力，使劲清晰模块的内容，为了就是为个体模块留下强烈的印象。

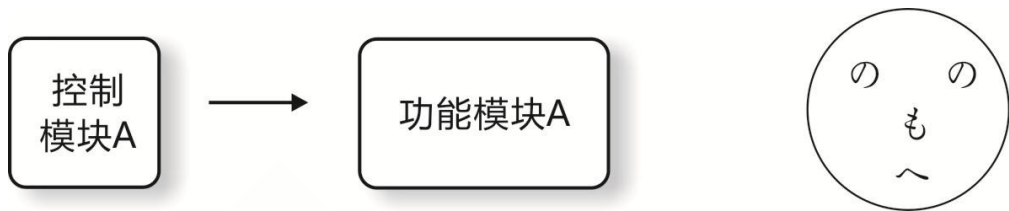


图 5.8.2 建模图的布局。

除此之外，建模图，布局也是重要的关键之一。如图 5.8.2 所示，建模图是站在整体视角所看到的景色，然而个体的布局位置，多多少少会影响模糊记忆。这种情况好比我们在注视某人的脸般，眼睛位于最上方，鼻子位于中间，嘴巴位于下方。如果将其反映在建模图的话 ... 现代人都习惯按照从左至右的顺序浏览事物，但是建模图无法像级人脸那样有固定的布局位置，为此建模图都会代用箭头建立模块之前的关系。

如图 5.8.2 所示，笔者一般习惯将控制模块放在建模图的左边，然后功能模块放在右边，接着设置箭头为两个个体模块生成关系。这种感觉眼睛的下方就是鼻子，鼻子的下方就是嘴巴。

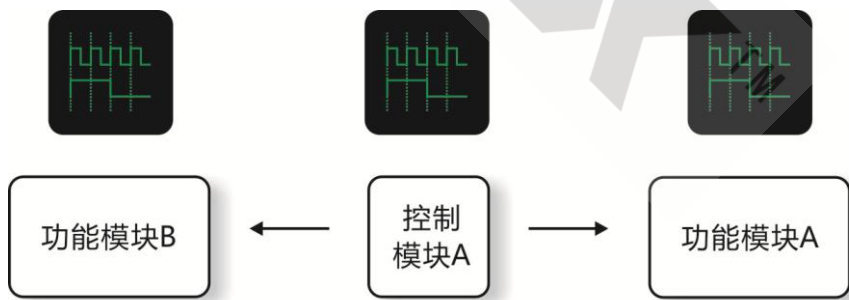


图 5.8.3 局部仿真信息。

一块的整体模块经过平局划分以后，仿真信息理应也会划分为数个，结果如 5.8.3 所示。换言之，一块一口吃不完的切糕，我们换做多口吃掉 ... 不过，在此有些同学可能会担忧，经过划分以后的仿真信息是不是会失去集中性与链接性呢？同学别当心，此刻我们的想象力就会发挥功效了 ...

体模表达能力高，模块内容很清晰，基本上个体模块就会成为强烈印象的局部特征，然后再由建模图建立大概的布局，一个模糊的宏图就会在我们的想象力之中建立完成。紧接着，个体印象会发挥拼图的作用，局部填充这个模糊宏图 ... 个体印象愈是强烈，

宏图的局部特征愈是清晰，周围细节的连带填充效果愈是越好。就这样，任其想象力 yy 一阵子后，我们就能在脑海当中看见整体的仿真信息。

笔者一边妄想，嘴角一边流着口水 ... 好兴奋，笔者不曾那么兴奋过，原来想象力不仅可以用来 YY，想象力也作仿真的超强利器。先将整体分化为无数个体，再强化个体印象，创建大概布局，然后刻意在模糊记忆之中，最后任由想象力填充其它细节。如此一来，左脑不仅更加轻松，左脑也能更多集中力去处理自己擅长的东西，真是可惜可贺！

“孩子，昆虫和花朵有什么关系”，它道

“啊 ... 花朵生产花蜜，昆虫采花蜜 ...”，笔者回答道。

“孩子，回答的很好”，它难得笑道。

神怎么一笑，让笔者更加觉得事有蹊跷了 ... 可恶，它就是那样一副德性，重来不把话说清楚，让笔者有种热窝蚂蚁的感觉。仔细一想，昆虫飞来飞去采花蜜是花田常见的场景 ... 花田常见的场景！？笔者之所以这么认为，原来是笔者站在整体的视角观察事物，既然如此，个体视角的花田会是甚么样的情况呢？

于是，笔者开始妄想 ... 如果笔者是一朵花的话，生产花蜜是笔者的任务，笔者只要使劲生产花蜜就行了，笔者才懒得管它什么昆虫；反之，如果笔者是一只蜜蜂，采花蜜是笔者的任务，笔者只要使劲采花蜜就行了，笔者才懒得管它是甚么花朵。原理如此，笔者开始了解它的笑意了 ...

当我们站在整体（花田）的视角长观察事物，花朵还有昆虫好似无私的好朋友般，花朵给予昆虫食物，昆虫帮忙花朵授粉，两者合作无间，彼此都是互惠关系。换之，如果我们站在个体（花朵与昆虫）的视角上观察事物，花朵和昆虫都是自私的陌生人，花朵为了生殖才大利用昆虫帮忙授粉 ... 然而，昆虫为了一份温饱才去采花蜜。实际上，两者是各怀鬼胎，互相利用，彼此都是独立关系

呵呵呵！想到这里，笔者不经意笑了出来 ... 原本看似矛盾的事情只要换个视角观事情又见合情合理，大自然实在太有趣了。那么，花朵与昆虫的故事又是传达什么呢？两个字独立。同样的事情如果反映在模块的身上，个体模块应该越独立（自私）越好。所谓独立并不表示孤独，首先个体必须确保它人无法侵犯自己，但是个体又不可以过度自封，因此个体必须打开外交管道，亦即沟通方式。

好奇的同学可能会觉得疑惑，个体模块之间为甚么要保持独立关系然后又故意打开外交管道呢？让笔者举例现实的实例，21 世纪的今天，C 国不希望 A 国插手自己的家事，但是 C 国也不能完全无视 A 国，换之 A 国也有同样的情况，因此 C 国与 A 国便打开外交的窗口，一起互勉，一起互喷。

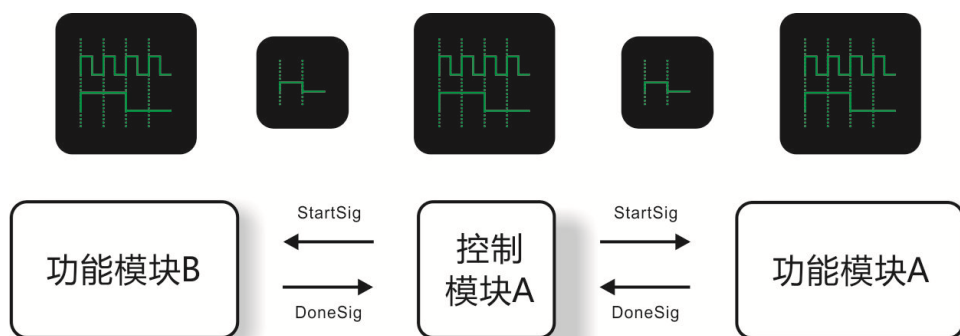


图 5.8.4 个体保持独立关系却建立沟通保持整体的连接性。

相比之下，模块才没有现实那么丑陋，**个体模块之所以保持独立关系是为了局部仿真能够顺利执行**；个体模块之间之所以建立沟通方式，是为了**不失整体的连接性**。如图 5.8.4 所示，模块之间基本都是保持独立的关系，因此可以执行局部的仿真。此外个体模块之间也建立外交 ... 啊！不，是沟通方式，以致保持整体的连接性。

如图 5.8.4 所示，个体模块都采用问答式沟通，换句话说 ... 它们之间都有同样的沟通模式。从仿真的角度看，个体模块之间的沟通时序基本上都是一样的，这种情况好比 C 国与 A 国都用 E 语言执行外交。如此一来，个体模块没事的时候就自作自的，有事的时候再经由相同的沟通互勉互喷，真是可喜可贺。

最后，笔者可以这样总结道：

由于我们不小心用左脑去解析仿真信息，但是海量的仿真信息让左脑喘不过气来，左脑为了自保便使劲发出“危险信息” ... 因此，我们会在无意识下恐惧仿真。一块打切糕如果不能一口吃下，就要有规划品均分化为无数个小切糕。笔者则是按照低级建模的准则将整体模块划分为无数个个体模块。

个体模块仿真之前必须先做好一些准备，例如想尽办法提高个体模块的表达能力，清晰化个体模块的内容，这样做不仅是为了方便仿真，也为了强化个体的印象。此外，为了进一步提高个体模块的独立关系又不失整体的连接性，模块之间必须建立沟通窗口。当我们完成局部仿真以后（个体模块逐个仿真），右脑就会建立模糊的时序宏图。

此刻，强烈的个体印象开始发挥想效果了，模糊记忆会特别记录印象强烈的局部特征，个体越是强烈，局部特征越是清晰，周围细节的填充效果会越好。如此一来，整体时序（仿真信息）会似拼图般在脑海里逐渐成型与清晰起来。

总体来说，大模块变成小模块以后，整体仿真信息随之也会变成无数个体仿真信息。左脑遇见整体仿真信息会汗救命 ... 换之，左脑遇见个体仿真信息会解析得游刃有余。个体仿真信息经过左脑解析以便交由右脑储存，不过前提是个体结果必须有强烈的印象。就这样，左脑解析一块个体仿真信息，右脑就记录一块局部信息，直至所有模块处理完毕。完后，想象力就会发挥作用，整体的仿真信息就会在想象中重现。



“孩子，还在绝望吗？”它道。  
“不了，现在恰好相反...”笔者答道。  
“很好的回答...孩子千万别忘了，白骆驼何时何地都在身边”，它道。  
“嗯！”，笔者感触道。  
“那么，再见”，它道。  
“感谢指导！！！”笔者大力鞠躬道。

=====

当笔者再度睁开眼睛的时候，天空已经是接近黎明。笔者用力掐了一下自己的脸颊，好疼！好疼就表示还活着...笔者察觉自己正躺在山崖一角，原来笔者在那里不知不觉睡着了。主动思想吗？呵呵！笔者笑笑两声以示那场不可思议的梦境，还有那个有趣的思想。啊...不知为何，此刻的心情既然如此轻松。

稍微清理一下灰尘后，笔者便站起身子，然后望向山崖另一面的彼岸，此刻朝阳正逐渐从地平线上升起。笔者一边欣赏风景，一边自言自语道...一个对象，如果整体视角看得太大，容易产生恐惧，此刻我们就必须改换个体视角重新看待，事实上是个体造就了整体，而不是整体造就了个体。

笔者也深刻反省至今为止的用脑习惯...上天为人类创建的左右脑，然而人类却失衡用脑，结果产生许多问题。右脑虽然性格古怪，但是比起左脑更有不可思议的能力，笔者可以站此YY妄想都是托右脑的福。说着说着，朝阳已经上升到某种程度，温柔的阳关一视同仁地为万物注入生命力，世界开始美丽起来...传统流派即使丑陋，阳关也会不惜吝啬分它一份羹。

大自然的无私拥抱让笔者对明天充满了希望。笔者使出决意的转身，然后踏出勇敢的一步，继续前进...随之便消失在晨光的照耀下。

总结：

第五章总与写完了，第五章相比前面几章是比较特别的一章，其中笔者引用蒂沙还有白骆驼的故事来比喻仿真——仿真就是人生的缩影。蒂沙在人生的转折点上，有两个未来，其一选择单调即平凡的人生；其二选择冒险 ... 前者是单向人生，后者是多向人生，因为在旅行的路上蒂沙呼引来许多可能性的结局。

反观仿真，仿真也有单向与多向。一般上下也只有门级实验应用仿真模型①才有单向仿真的可能 ... 换之，多向仿真是默认仿真模式。多向仿真讲述变数衍生可能性，然而常见的变数有时钟变量，信号数量，还有信号方向 ... 它们都是理想变数，此外也存在物理变数，不过物理变数是自寻烦恼的东西，不建议考虑在仿真之中。

必然性也是多向仿真的其中一种概念，必然性会将仿真流逝引导至最有意义，也是我们最期待的结果。为了保证必然性，维护必然性与清晰必然性就成为了重要的工作。除此之外，必然性还有一个叫做契机的有趣东西，契机就是命运拼图的黏糊剂，然而契机被笔者用作沟通。不管是约束（控制）变数也好，或者保证必然性也好，这些都是前期建模应该考量的范畴。

“我是谁？我是甚么？”，看似简单的问题，其实是这个世界上最难的问题。笔者也曾经多次自问“仿真究竟是甚么？”，根据笔者的理解，仿真是时序结果，仿真对象还有激励内容的复合体，联系他们并且做出解析就是仿真的本意，笔者也称之为解析仿真信息。

多向仿真会有变数衍生许多可能性，许多可能性结合就成为海量般的仿真信息。为了方便理解，笔者将多向仿真形容为迷宫，然而压倒性的仿真信息形容为切糕怪物。切糕物如其名，是分量大得夸张的小食，如果仿真手段不但我们就会被“切糕”淹没。除了切糕以外，仿真也存在仿真癌这样的心理病。

仿真癌是由强迫仿真，还有恐惧仿真两种互斥的极端心态组成。前者让人不由自主去仿真，后者让人不由自主回避仿真，久而久之便会产生某种奇怪的心理病。强迫仿真还好解决，只要前期建模做好准备即可；反之，恐惧仿真就稍微麻烦一点 ... 根据理解，现代人都有倾向左脑的用脑习惯，然而要用左脑处理仿真的海量信息实际上是拿健康来开玩笑的事情。

为此，笔者尝试使用想象力来分担左脑的记忆工作。我们首先必须按“功能”将整体模块分化数个更小的个体模块，然后想尽办法强化个体模块的存在，好似留下强烈的个体印象。虽说模块分散以后会失去连接性，在此我们可以使用“沟通”来问题，并且进一步独立化个体模块，事后再逐个仿真个体模块即可。完后，整体信息自然而然会建立在脑海当中。

这个章节，许多内容看似不相关，不过只要读者仔细一想，上述内容其实都是一般参考书故意忽略的细节。当然，读者也可以认为笔者是杞人忧天，胡思乱想太多了，但是结



果终究怎样，唯有见仁见智。笔者在此也是将自己的所想所言用文字表达出来而已。



## 第六章 结束就是开始

### 6.1 不可仿真对象

仿真好比吸烟一样，两者都要使用健康来交保 ... 爱惜生命也好，节能主义也罢，我们都应该尽量减低仿真使用频率。仿真除了存在恐怖的绝望切糕以外，仿真也存在不可仿真对象，那么什么又是不可仿真对象呢？不可仿真对象不是没有仿真的意义，而是[仿真起来会不断蹉跎青春，浪费无谓的精力](#)。

我们知道仿真的本意是将仿真对象，激励内容，还有时序结果联系以后并且做出解析 ... 然而，不可仿真对象则是[妨碍联系过程，让仿真信息解析工作无法有效执行](#)。那么，到底有存在多少不可仿真对象呢？根据笔者的理解，不可仿真对象有以下 3 个：

- (一) 超傻模块
- (二) 超烦模块
- (三) 超乱模块

其一，[超傻模块是指功能过度简单](#)，例如小功能模块或者小组合逻辑。小功能模块虽然也是功能模块，但它不仅[用时简短](#)（大概有 1~3 个时钟），而且[功能也非常单纯](#)，典型的例子有，加码器，移位寄存器等。如果超傻模块作为个体建模，模块内容一般都是清晰直观，稍看一眼就立即知道大意。

```
1. module encode_module( input CLOCK, input [3:0]NumSig, output[7:0]SMGCode );
2.
3.     reg [7:0]rSMG;
4.     always @ ( posedge CLOCK )
5.         case( NumSig )
6.             4'd0 : rSMG <= 8'b1100_0000;
7.             4'd1 : rSMG <= 8'b1111_1001;
8.             4'd2 : rSMG <= 8'b1010_0100;
9.             4'd3 : rSMG <= 8'b1011_0000;
10.            4'd4 : rSMG <= 8'b1001_1001;
11.            4'd5 : rSMG <= 8'b1001_0010;
12.            4'd6 : rSMG <= 8'b1000_0010;
13.            4'd7 : rSMG <= 8'b1111_1000;
14.            4'd8 : rSMG <= 8'b1000_0000;
15.            4'd9 : rSMG <= 8'b1001_0000;
16.        endcase
17.
18.     assign SMGCode = rSMG;
19.
20. endmodule
```

### 代码 6.1.1

代码 6.1.1 是加码器的典型例子 ... 如代码 6.1 所示，这是一个数码管的加码模块，第 1 行的 NumSig 是输入信息，SMGCode 则是加码信息；第 5 的 case ... endcase 会根据 NumSig 的输入，为 rSMG 赋予不同的加码信息。如第 6 行所示，NumSig 输入为 4'd0，操作将加码信息 8'b1100\_0000 赋予 rSMG 然后再由 rSMG 驱动 SMGCode (第 18 行)。

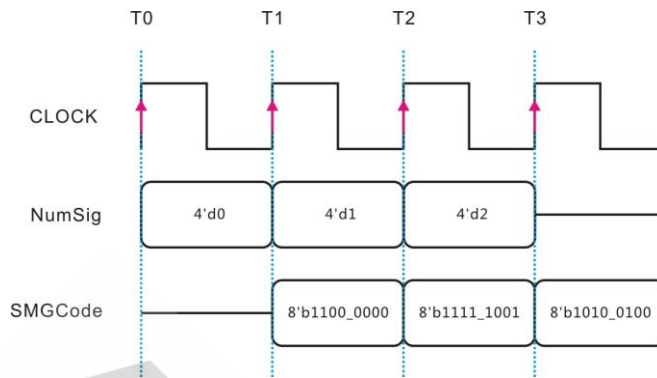


图 6.1.1 想象的理想时序图。

至于代码 6.1，我们只要善用想象力稍微脑补一下，时序图就会出现在脑海当中。如图 6.1.1 所示，那是笔者脑补以后的时序结果 ... 假设 NumSig 在 T0 的未来值是 4'd0，那么加码模块会在 T1 读取该过去值，并且经由 SMGCode 输出为未来值 8'b1100\_0000。其它 NumSig 也以此类推。

小组合逻辑，一般是指小功能的组合逻辑，小组合逻辑与小功能模块虽然大伙的都是超傻的同伴，但是前者没有时钟源。笔者曾经说过，组合逻辑的时序表现是即时事件，而且即时事件也无视时钟，结果我们用不着脑补时序图。小组合逻辑的典型例子有 Verilog 自带的运算符，如 \*，/，% 等之余，还有一些常用的选择器也是。

```
1. module devide_module( input [7:0]A,B, output [15:0]Result );
2.
3.     reg [15:0]rResult;
4.     always @ (*) rResult = A * B;
5.
6.     assign Result = rResult;
7.
8. endmodule
```

### 代码 6.1.2

代码 6.1.2 乘法器的简单例子，第 4 行除了使用 always @ (\*) 声明组合逻辑以外，第 4 行业使用 Verilog 自带的相乘运算符执行 A\*B 操作，然后将结果赋予 rResult 寄存器。最后再由该寄存器驱动 Result 出入端（第 6 行）。代码 6.1.2 基本上已经傻到不能再傻了，因为内容不管怎么看都是太直接了，所以用不着脑补时序图。

```

1. module xx_basemod ( output [7:0]WrData );
2.
3.     wire [7:0]Data_U1;
4.     a_module U1
5.     (
6.         ...
7.         .StartSig( StartSig_U1 ),
8.         .Data( Data_U1 )
9.     )
10.
11.    wire [7:0]Data_U2;
12.    a_module U2
13.    (
14.        ...
15.        .StartSig( StartSig_U2 ),
16.        .Data( Data_U2 )
17.    )
18.
19.    reg [7:0]rData;
20.    always @ ( * )
21.        if( StartSig_U1 ) rData = Data_U1;
22.        else if( StartSig_U2 ) rData = Data_U2;
23.        else rData = 8' dx;
24.
25.    assign WrData = rData;
26.
27. endmodule

```

代码 6.1.3

代码 6.1.3 则是选择器的典型例子，笔者先是在第 4~9 行，还有第 12~17 行实例化同样的 a\_module 为 U1 与 U2，然而第 8 行，还有第 16 行的 Data 输出都争用同一个 WrData 输出端（第 1 行），此刻两仪性的问题发生了。为此，笔者在 20~23 行建立一个选择器来协调 U1~U2 公用同一个输出端。

如代码 6.1.3 所示，第 20 行使用 always @ (\*) 声明并且创建组合逻辑，而且组合逻辑也是触发即时事件，因此第 21~23 行的操作同样也是无法脑补时序图。不过代码 20~23 的内容早已非常清晰的告诉我们，如果 U1 使能，输出端 WrData 就交由 U1 使用（第 21 行），反之亦然。

---

超傻模块作为个体而言，模块内容足以清晰直至刺激我们的想象力，任我们想忘也忘不

了。在此，有些同学可能会疑惑道，超傻模块实际上便没有阻碍仿真的联系工作，换之，仿真超傻模块反更简单 ... 为何笔者要将它们列为不可仿真对象呢？嗯 ... 这种感觉还比老师询问读者一加一的结果，读者是将结果心算出来呢？还是用纸加铅笔计算出来呢？又或者使用计算器计算出来呢？不用说，答案当然是心算出来。

不管仿真对象再怎么简单，仿真本来就是麻烦的工作，尤其是是仿真的准备工作更加猥琐耗时 ... 我们不仅要预设仿真，我们也要建立仿真项目，再者就是创建激励文本。试问我们有多少青春这样去蹉跎呢？超傻模块最大的罪状就是消耗我们的青春，结果它必须死！必须成为不可仿真对象！



## 6.2 超烦模块与不等比例缩放

“小哥，吃饱了吗？”

“小哥，身体还好吗？”

“小哥，天气转凉了 ... ”

隔壁的小花三五四次总是重复一样的问候，然而这些关心话，笔者曾经认为烦心又吵死人 ... 如今离家许久，笔者渐渐地开始怀念小花的唠叨了。人会啰嗦，模块同样也会啰嗦，这些啰嗦的模块笔者称为超烦模块。好奇的同学可能会疑惑，模块又不会说话，那来唠叨呢？这位同学有所不知了，超烦模块并不指意模块会滔滔不绝说废话，而是模块让人觉得“烦心”才是重点。

烦心？没错，就是烦心。试问读者，除了唠叨以外还有什么东西会让人觉得烦心呢？笔者认为，等候女人是最烦心的事情 ... 笔者始终不理解，为何女人都会无止境照镜子？换装会犹豫不决？一个好好的约会，笔者每次都要用上小时去等待她们 ... 期间，笔者会不断思考，女人这种生物究竟用有什么结构创建的？

除了等候女人以外，等候模块也会让笔者觉得十分烦心 ... 一般上，超烦模块都有一处共同点，亦即内容夹带计数器（定时器）。虽说等候计数没有等候女人那么可怕，然而等候计数期间，我们分分秒秒都在掉血，时间还有精力无意间就这样浪费掉了。为了更好理解超烦模块的破坏力，请打开 exp26 ...

*exp26\_simulation.vt*

```
1.  `timescale 1 ns/ 1 ns
2.  module exp26_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.
7.      /*******/
8.
9.      initial
10.     begin
11.         RESET = 0; #10; RESET = 1;
12.         CLOCK = 1; forever #10 CLOCK = ~CLOCK;
13.     end
14.
15.     /*******/
16.
17.     parameter T100MS = 24'd500_000;
18.
19.     reg [3:0]i;
```



```

20.     reg [23:0]C1;
21.     reg [3:0]LED;
22.
23.     always @ ( posedge CLOCK or negedge RESET )
24.         if( !RESET )
25.             begin
26.                 i <= 4'd0;
27.                 C1 <= 24'd0;
28.                 LED <= 4'd0;
29.             end
30.         else
31.             case( i )
32.
33.                 0:
34.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
35.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0001; end
36.
37.                 1:
38.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
39.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0010; end
40.
41.                 2:
42.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
43.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0100; end
44.
45.                 3:
46.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= 4'd0; end
47.                     else begin C1 <= C1 + 1'b1; LED <= 4'b1000; end
48.
49.             endcase
50.
51.     /***/
52.
53. endmodule

```

exp26 是无伤大雅的流水灯实验，假设笔者想要模拟 50Mhz 的实际时钟 ... 笔者先在第 1 行将时钟刻度设置为 1ns/1ns。50Mhz 有 20ns 的时钟周期，因此第 12 行的 CLOCK 寄存器每隔 10 个时钟刻度翻转一次。假设间隔是 100ms，于是笔者在第 17 行声明 100ms 的常量为 24'd500\_000。

第 19~21 行是相关的寄存器，i 为指向步骤，C1 为计数器，LED 寄存器为模拟 LED 效果。第 31~49 是流水灯的操作，步骤 0 先将 LED 设置为 4'b0001，然后延迟 100ms；步骤 1 设置 LED 为 4'b0010 然后延迟 100ms 步骤 2 设置 LED 为 4'b0100 然后延迟 100ms

；步骤 3 设置 LED 为 4'b1000，然后延迟 100ms；最后再重新返回步骤 0。

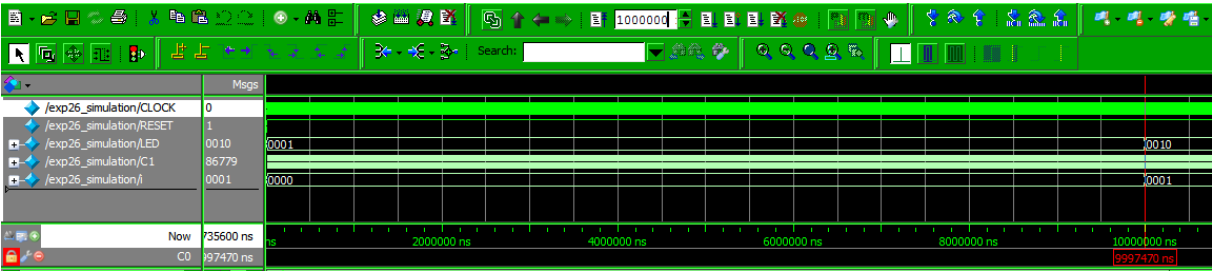


图 6.2.1 exp26 的仿真结果。

读者没有看错，图 6.2.1 是 exp26 的仿真结果，不过笔者中途放弃等候了 ... 因为实在是太烦心了。如图 6.2.1 所示，我们可以看见无间断的 CLOCK 信号与 C1 计数，然而我们却看不见 C1 的实际计数内容，真是哦买狗！

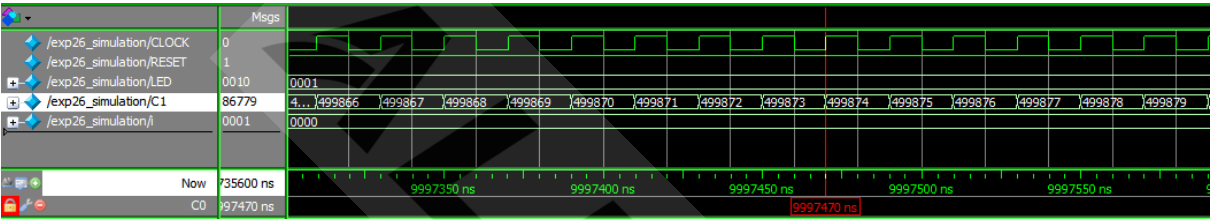


图 6.2.2 将 wave 界面放大几十倍。

为了看见 C1 的计数内容的话，笔者估计将 wave 界面放大，话虽如此 ... 如图 6.2.2 所示，那是笔者将 wave 界面放大几十倍以后的结果，细心的同学一定会发现，光标 C0 并没有准确指向时钟。为了调整 C0 指向的地方，笔者必须不停拖动滚动轴，或者不停缩小再放大 ... 做着做着：

“气死人了！受够了，实在烦心死了！不玩了！不玩了！exp26 不仿真了！”

结果笔者开始自暴自弃了 ... 在此，想必读者也是身同感受。没错，这就是超烦模块惊人的破坏力，它会让人们觉得烦心，然后自暴自弃，感受绝望。笔者曾经说过，时钟用量是一种直接性的变数，亦即时钟用量越长，意外的可能性就会越高，而且可能性也会指向上身，甚至海量化仿真信息。换句话说，超烦模块是切糕的兄弟，它助长切糕肆虐。

除此之外，模块愈烦时钟用量愈多，仿真愈加吃尽内存，直至拖慢整架计算机，甚至当机。根据笔者的经历，笔者的老机器最的极限是 1ms 的最大仿真时间，如果最大仿真时间超过 1ms，计算器就会立即罢工。机器罢工还是小事，真正让笔者忧心的是过长的 wave 界面 ...

如图 6.2.1 所示，过长的 wave 界面会大大降低仿真效率 ... 期间，指向工作不仅会变得困难起来，而且还要不停滑动鼠标中间将 wave 界面拉来拉去，缩小又放大。一个简单的流失灯实验，笔者至少花费了 1 个小时在瞎搞，最终得到一个烦心的结果 ... 说实在，滑鼠，手指，还有青春都在发疼。试问我们有多少青春可以这样去蹉跎？超烦模块最大

的罪状就是[贱踏我们的青春](#)，结果它必须死！必须成为不可仿真对象！

读者千万别小看可爱漂亮的女人，她们外表越好看，打扮起来越花时间，实际等候会发挥无与伦比的破坏力，摧毁我们的青春。话虽如此，作为男人有时候也不得不等待，但也不能过度牺牲小我。为此，我们必须拿出[绅士的风范](#)，告诉她们“[我最爱真实的妳](#)”。如此一来，问题就能巧妙解决。超烦模块就是如此，我们不能无视它，也又不能直视它，为此我们必须拿出绅士的手段。

超烦模块一般都是[计数器（定时器）在里边不停灌水](#)才会导致的悲剧。反过来问，我们究竟是为了什么去仿真计数器（定时器），答案就是为了实现精密控时。精密控时究竟是什么，相比有些同学可能会觉得困惑？简单而言，好比 exp26 我们都必须为每个步骤计数 5000000 个时钟以致实现 100ms 的间隔时间。

精密控时一般有两种手段实现，其一是笔者爱用的整合技巧，整合技巧所拥有的独特代码风格，实际上是针对控时所设计 ... 不过很遗憾的是，笔者并不打算在这里解释整合技巧，取而代之笔者会举例另一种更加直接的方法，那就是[不等例缩放](#)。

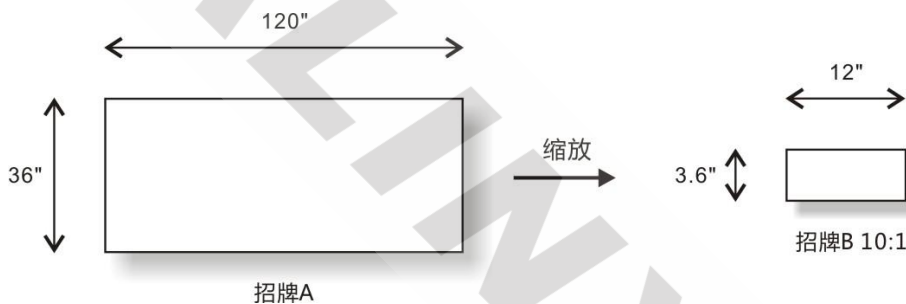


图 6.2.3 等例缩放。

假设笔者是一名黑铁技工，然而有一位客人要求读者为它建立一副高为 36 英寸（3 英尺），长为 120 英寸（10 英尺）的招牌 A。笔者首先必须为客人提供招牌 A 的软设计，由于 36 " × 120" 的面积实在太太，笔者的老机器无法承受。因此，笔者创建 10:1 等例缩放的招牌 B 供客人参考，结果如图 6.2.3 所示。

等例缩放是一种日常的解决手段，目的是为了节能 ... 看见“节能”两个字，笔者仿佛触电般两眼又发光了。没错！等例缩放是一种节能的解决手段，而且仿真也能应用 ... 但是仿真软件不及其它设计软件，简单点击几下鼠标就能等例缩放。结果，[仿真软件只能人为实现不等例缩放](#)。不等例缩放也是一种破坏性的手段，所以执行之前，我们必须准备好副本才行。

代开 exp27，它是 exp26 的副本 ... 接下来，我们便要用它实现不等例缩放。

*exp27\_simulation.vt*

1. `timescale 1 ps/ 1 ps

```

2.  module exp27_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.
7.      /******
8.
9.      initial
10.     begin
11.         RESET = 0; #10; RESET = 1;
12.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
13.     end
14.
15.     /******
16.
17.     parameter T100MS = 24'd5;
18.
19.     reg [3:0]i;
20.     reg [23:0]C1;
21.     reg [3:0]LED;
22.
23.     always @ ( posedge CLOCK or negedge RESET )
24.         if( !RESET )
25.             begin
26.                 i <= 4'd0;
27.                 C1 <= 24'd0;
28.                 LED <= 4'd0;
29.             end
30.         else
31.             case( i )
32.
33.                 0:
34.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
35.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0001; end
36.
37.                 1:
38.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
39.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0010; end
40.
41.                 2:
42.                     if( C1 == T100MS ) begin C1 <= 24'd0; i <= i + 1'b1; end
43.                     else begin C1 <= C1 + 1'b1; LED <= 4'b0100; end
44.

```



```

43.           else begin C1 <= C1 + 1'b1; LED <= 4'b0100; end
44.
45.           3:
46.           if( C1 == T100MS -1) begin C1 <= 24'd0; i <= 4'd0; end
47.           else begin C1 <= C1 + 1'b1; LED <= 4'b1000; end
48.
49.       endcase

```

exp28 是纠错以后的结果 ... 代码 31~39 所示，第 34，42，42，还有 46 行，笔者都添加 -1 就少步骤逗留的时间。

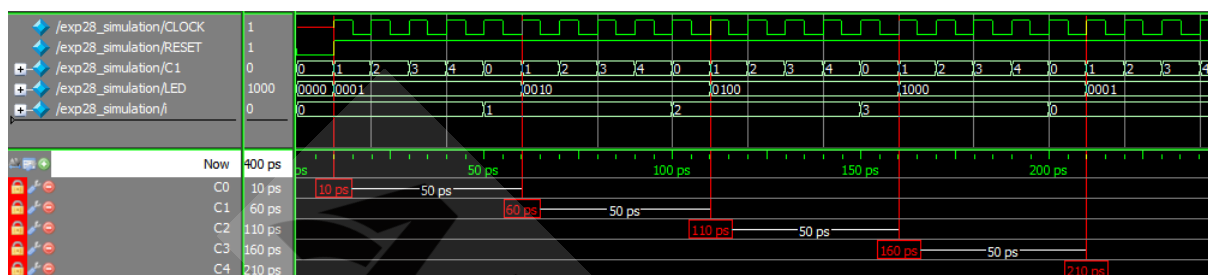


图 6.2.5 exp28 的仿真结果。

图 6.2.5 是 exp28 的仿真结果。如图 6.2.5 所示，光标 C0~C4 分别指向步骤 0~3 的停留时间，每个光标也有 50ps 的间隔时间，亦即每个步骤都停留 5 个时钟，因此我们可以断定 exp28 是预想所要的仿真结果。为此，我可以将 exp28 的纠错手段往 exp26 照搬即可。这样一来，我们就用不着使劲滚动滑鼠中间，无尽缩放 wave 界面又拖来拖去，然后带着轻松的心境完成仿真。

曾经何时笔者也犯过同样的错误 ... 计数器灌水过多导致仿真又长又臭，可是笔者当时也没有其它解决的手段，唯一的选择就是成为勇者硬闯而已，结果每次都成为破烂回来。如果当时有白骆驼引导笔者，想必笔者也不会选择勇者这条道路。为了避免后人重蹈笔者的后尘，笔者希望读者好好知晓 ... 超烦模块其一点也不可爱，还不如说超可怕！但是小花是例外，她虽然喜欢重复同样的问候，但她绝不会让人等候多时，也没有自恋的怪癖。

## 6.3 不可仿真对象——超乱模块①

“臭小鬼，又把房间弄乱了！赶紧给老娘收拾干净，不然晚饭吃空气！”

笔者很懒，房间自然也很乱，所以才会常常惹怒母亲大人。模块内容好比房间一样，如果不努力去维护它，内容会像滚雪球那样越写越乱，直至不敢入目，类似的模块笔者称为超乱模块。超乱模块是不可仿真对象的一种，我们知道仿真的本意就将仿真对象，激励内容，还有时序结果联系起来并且做出解析。如果仿真对象（模块内容）无法解读，那么解析仿真信息也无从谈起。

超乱模块的例子有：无结构模块，还有官方插件模块。无结构模块就是自由结构的模块，Verilog 是一门自由的语言，模块自由到没有结构其实一点也不奇怪。笔者年轻的时候建模很 Free Style，结果模块像果冻般软绵绵地 ... 越往上建模，感觉越是给人一种“要倒了！要倒了！”的危机感。完后，笔者回头一看，哦买狗！乱糟糟的内容，让笔者吃不尽热狗。

“内容很乱！内容超乱”，笔者震撼道。

模块内容凌乱是后期工作的定时炸弹，它任何时候会炸飞一切，而且一瞬足以让我们的努力灰飞烟灭。为了避免悲剧发生，笔者开始 Verilog 的旅程，直至遇见它 ... 是它告诉笔者结构的重要性，它也说过没有结构，模块就是一只史莱姆而已，又弱又惨。为此，如何为模块注入结构——这门学问，成为了今天的建模技巧。

一般上，超乱模块的内容，除了设计者以外，其他人是没有办法读懂的 ... 作为契机，超乱模块就成为了保护商业秘密最好的手段，官方插件模块就是如此。笔者曾经认为官方老大是一位不私指教的好人，所以官方插件模块理应也是平易近人。但笔者打开内容来看的那刻瞬间，蛋蛋就立即掉到地上 .... 那是什么东西！？比笔者的房间还乱！？

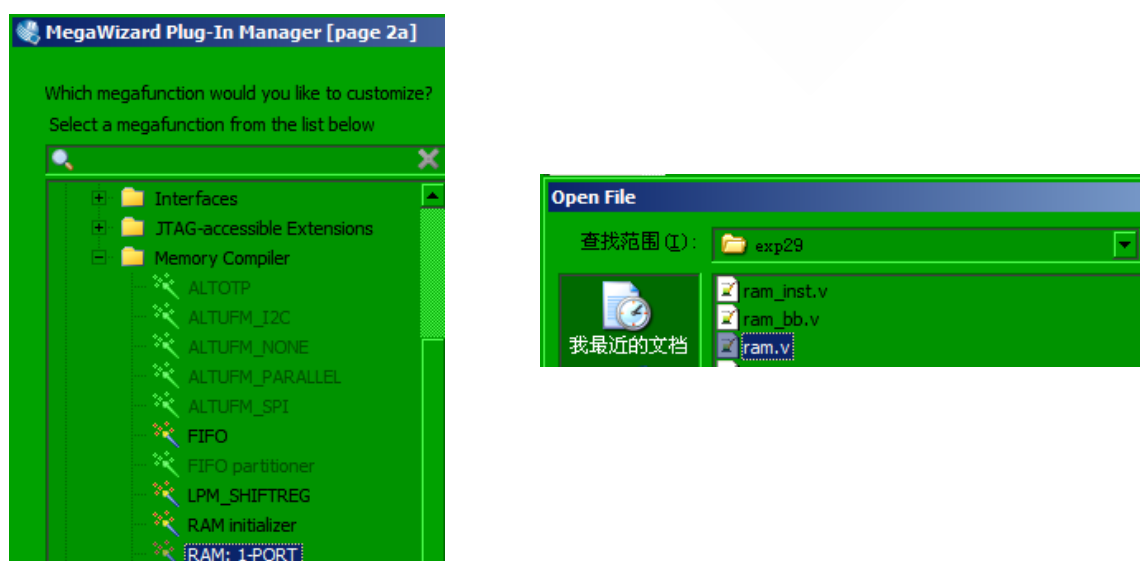


图 6.3.1 创建官方插件模块 1Port RAM



如图 6.3.1 所示，假设笔者经由集成环境（Quartus II）的 Mega Wizard 创建官方插件模块 1 Port RAM（作图）。模块创建完毕以后，相关的 ram.v 文件就会出现在指定的目录下（右图）... 然后再将 ram.v 打开浏览。

*ram.v*

```
1. 'timescale 1 ps / 1 ps
2. // synopsys translate_on
3. module ram ( address, clock, data, wren, q);
4.
5.     input  [7:0] address;
6.     input   clock;
7.     input  [7:0] data;
8.     input   wren;
9.     output [7:0] q;
10.
11. 'ifndef ALTERA_RESERVED_QIS
12. // synopsys translate_off
13. 'endif
14.     tri1    clock;
15. 'ifndef ALTERA_RESERVED_QIS
16. // synopsys translate_on
17. 'endif
18.
19.     wire [7:0] sub_wire0;
20.     wire [7:0] q = sub_wire0[7:0];
21.
22.     altsyncram altsyncram_component (
23.         .address_a (address),
24.         .clock0 (clock),
25.         .data_a (data),
26.         .wren_a (wren),
27.         .q_a (sub_wire0),
28.         .aclr0 (1'b0),
29.         .aclr1 (1'b0),
30.         .address_b (1'b1),
31.         .addressstall_a (1'b0),
32.         .addressstall_b (1'b0),
33.         .byteena_a (1'b1),
34.         .byteena_b (1'b1),
35.         .clock1 (1'b1),
36.         .cloccken0 (1'b1),
```

```

37.         .clocken1 (1'b1),
38.         .clocken2 (1'b1),
39.         .clocken3 (1'b1),
40.         .data_b (1'b1),
41.         .eccstatus (),
42.         .q_b (),
43.         .rden_a (1'b1),
44.         .rden_b (1'b1),
45.         .wren_b (1'b0));
46. defparam
47.     altsyncram_component.clock_enable_input_a = "BYPASS",
48.     altsyncram_component.clock_enable_output_a = "BYPASS",
49.     altsyncram_component.intended_device_family = "Cyclone IV GX",
50.     altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
51.     altsyncram_component.lpm_type = "altsyncram",
52.     altsyncram_component.numwords_a = 256,
53.     altsyncram_component.operation_mode = "SINGLE_PORT",
54.     altsyncram_component.outdata_aclr_a = "NONE",
55.     altsyncram_component.outdata_reg_a = "UNREGISTERED",
56.     altsyncram_component.power_up_uninitialized = "FALSE",
57.     altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
58.     altsyncram_component.widthad_a = 8,
59.     altsyncram_component.width_a = 8,
60.     altsyncram_component.width_byteena_a = 1;
61.
62. endmodule

```

读者看见什么吗？蛋蛋是否在发疼呢？没错，我们会看见一堆意义不明的关键字，具体意义千万别问笔者，笔者早已放弃思考了。除此之外，官方插件模块也不能随意仿真，想要仿真家伙就要给老大交钱！交钱？这句话是什么意思？就是字面上的意思 ...

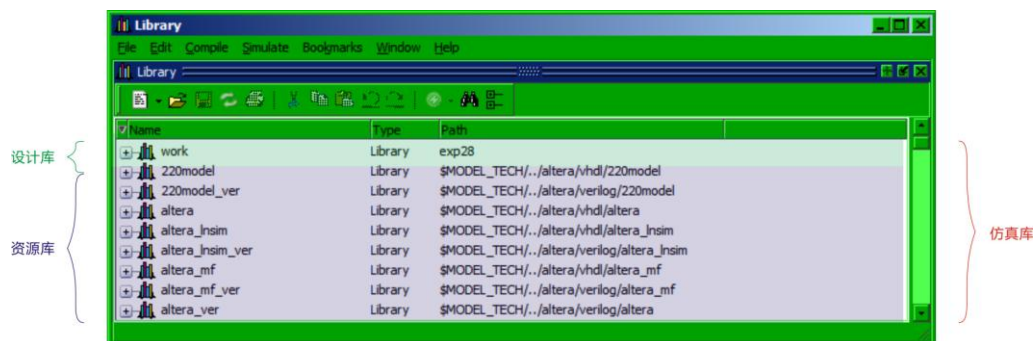


图 6.3.1 仿真库界面，设计库与资源库。

我们曾在第二章学过，Modelsim 除了 wave 界面以外还有许多形形色色的界面，仿真库

界面就是其中一个。如图 6.3.1 所示，仿真库又可以分为设计库还有资源库，设计库是仿真的临时空间（也称为工作库）；反之，资源库则是存储官方插件模块所需的仿真资源。默认下，Modelsim 只为我们提供标准库（Standard Library），标准库可以支持的程度也只有 HDL 官方指定的范畴而已。

然而，官方插件模块的内容却远远超过 HDL 官方所指定的范畴，用 ram.v 为例的话，如：什么 NEW\_DATA\_NO\_NBE\_READ？什么 altsyncram？这些关键字的背后都是不可告人的商业秘密。结果而言，Modelsim 为了支持 Altera 官方的插件模块，Modelsim 必须拥有 Altera 官方自定义的仿真库不可。

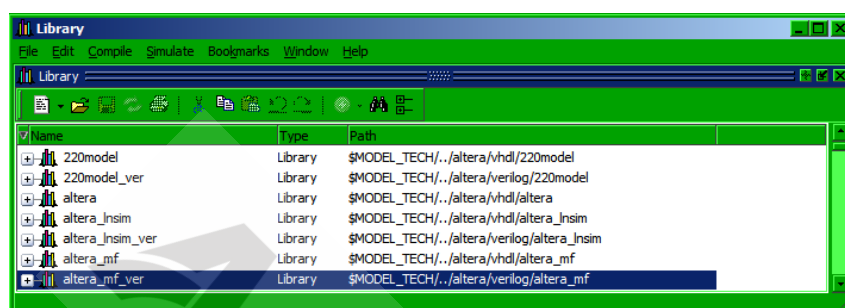


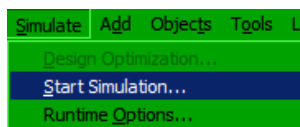
图.6.3.2 支持 RAM 的资源库。

如图 6.3.2 所示，支持 RAM 所属的资源库是 altera\_mf，因为 ram.v 用 Verilog 所创建，所以资源库选为 altera\_mf\_ver。笔者曾经疑问过，Altera-modelsim AE 与 Altera-modelsim SE 之间究竟有什么区别？后来发现 SE 除了免费以外，SE 也有行数限制，仿真库（资源库）也不如 AE 丰富。世界真是现实呀 ... 笔者不尽感叹道，不过，作为初学 SE 已经够用了。

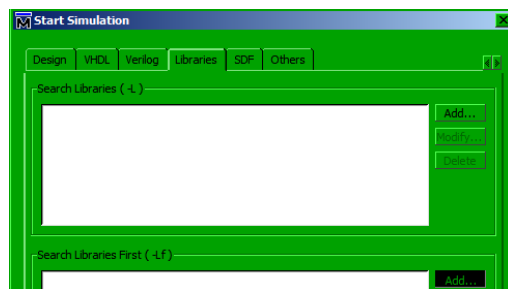
除此之外，还有一些琐碎的细节读者需要稍微注意，即手动编译官方插件模块相比自动编译多了一项操作。exp29 是自动编译的仿真项目，基本上都是一键搞定。相比之下，exp30 是手动编译的仿真项目，如果按照往常的步骤，Modelsim 会狠狠警告我们“altsyncram 是什么！？”，然后返回一个 error loading design 错误。可恶！身为软件竟敢给老子嚣张！

```
* Error: (vsim-3033) .../Experiment06/exp30/ram.v(85): Instantiation of 'altsyncram' failed. The design unit was not found.
# Error loading design
```

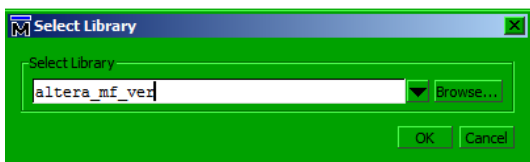
为了教训嚣张的 Modelsim，我们必须告诉它什么叫做恐惧！



①Start Simulation 界面。



② 选择 Library , 为 Search Libraries First 添加资源库。



③选择 altera\_mf\_ver 资源库。



④资源库添加完毕。

图 6.3.3 手动添加资源库。

如图 6.3.3 所示，这是手动将资源库添加至设计库的过程。首先按照步骤① 打开 Start Simulation 窗口；接着按照② 选择 Libraries，然后为点击 Search Libraries First 的 Add 按钮添加资源库；随之会弹出如 3 所示 Select Library 的小窗口，按下搜索 altera\_mf\_ver 资源库然后点击 OK 生效；完后，altera\_mf\_ver 资源库会添加在 Search Libraries First 的列表下。

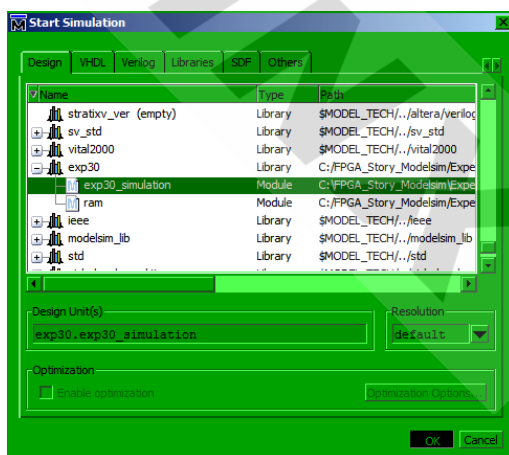


图 6.3.4 手动添加资源库。

再者也可利用 Design，选择 exp30\_simulation，点击 OK 便启动仿真，接续的操作和往常一样。接下来，让我们悄悄 exp30 究竟拥有什么激励内容？

*exp30\_simulation.vt*

1. `timescale 1 ps/ 1 ps
2. module exp30\_simulation();
- 3.

```

4.    reg CLOCK;
5.    reg RESET;
6.    reg RAM_WrEn;
7.    reg [7:0] RAM_Addr,RAM_WrData;
8.    wire [7:0]RAM_RdData;
9.
10.   /******
11.
12.   initial
13.   begin
14.       RESET = 0; #10; RESET = 1;
15.       CLOCK = 1; forever #5 CLOCK = ~CLOCK;
16.   end
17.
18.   /******
19.
20.   ramU1
21.   (
22.       .address ( RAM_Addr ),
23.       .clock ( CLOCK ),
24.       .data ( RAM_WrData ),
25.       .wren ( RAM_WrEn ),
26.       .q ( RAM_RdData )
27.   );
28.
29.   reg [3:0]i;
30.
31.   always @ ( posedge CLOCK or negedge RESET )
32.       if( !RESET )
33.           begin
34.               i <= 4'd0;
35.               RAM_WrEn <= 1'b0;
36.               RAM_Addr <= 8'd0;
37.               RAM_WrData <= 8'd0;
38.           end
39.       else
40.           case( i )
41.
42.               0:
43.                   begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd0; RAM_WrData <= 8'hAA; i <= i + 1'b1; end
44.
45.               1:
46.                   begin RAM_WrEn <= 1'b0; i <= i + 1'b1; end

```

```

47.
48.             2:
49.             i <= i;
50.
51.         endcase
52.
53.     /***/
54.
55. endmodule

```

exp30 代码行第 20~27 行是仿真对象的实例化，其中第 31~51 行是虚拟输入。虚拟输入在步骤 0 为 RAM\_WrEn 赋值 1，RAM\_Addr 赋值 8'd0，RAM\_WrData 赋值 8'hAA，然后 i 递增以示下一个步骤。步骤 1 仅为 RAM\_WrEn 清零，接着将 i 递增以示下一个步骤。

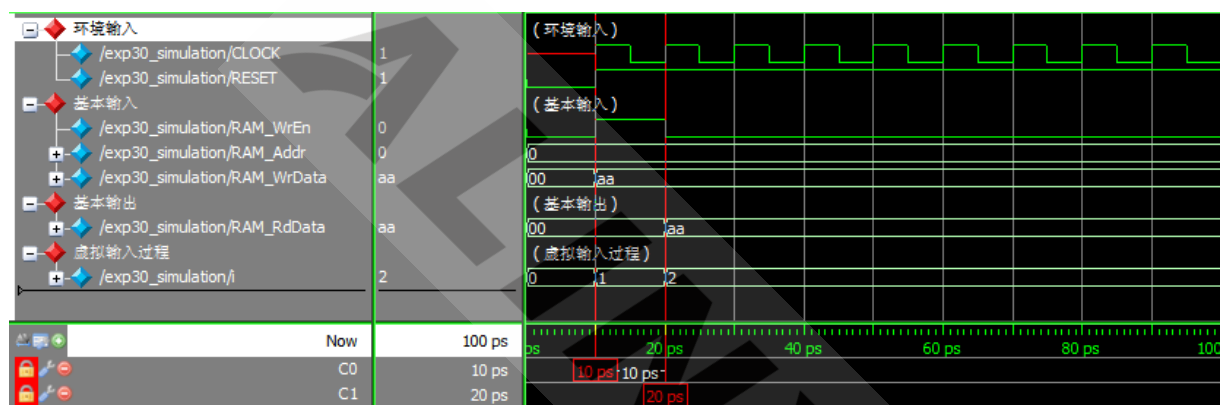


图 6.3.5 exp30 仿真结果。

如图 6.3.5 所示，C0 指向虚拟输入的步骤 0，C1 指向虚拟输入的步骤 1 ... 期间，虚拟输入再 T0 的时候，RAM\_Addr 输出未来值 8'd0，RAM\_WrData 则输出未来值 8'hAA，RAM\_WrEn 输出未来值 1'b1。下一刻，亦即 T1 的时候，仿真对象检测 RAM\_WrEn 的过去值为 1，并且决定读取 RAM\_WrData 的过去值 8'hAA 至地址 8'd0，然后再经由 RAM\_RdData 输出未来值 8'hAA。

根据仿真结果显示，**RAM\_WrEn 充当写入数据的锁匙，拉高有效**。那么问题来了！为什么仿真对象好死不死，**写入数据必须拉高 RAM\_WrEn 才行呢**？我们知道仿真对象是官方插件模块，内容必须隐藏不可。如此一来，解析仿真信息工作会因为仿真对象无法解读而发生中断，因为我们无法理解，为何写入数据不得不拉高 RAM\_WrEn？最终仿真也无法继续下去。这种感觉好比大号上到一半，忽然有人按铃，实在令人讨厌无比！

我们知道官方插件模块是官方的产品，也是商业秘密 ... 官方没有理由公开内容，但是官方又不得不公开，官方别扭许久后，最后想到一个好办法！作为**弥补手段，官方为用户创建厚厚的鸟文手册供参考**。即使内容看不懂，我们也不能追究官方的责任 ... 说实话，这招**四两拨千斤**实在太厉害了！

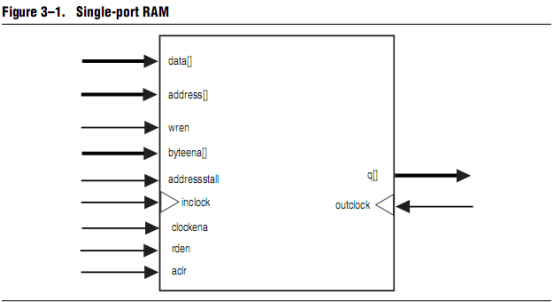


图 6.3.6 官方提供的 Single port RAM 图形。

图 6.3.6 是官方为我们提供的 Single port RAM 图形，话虽如此 ... 实际上除了 data，address，wren，inclock，outclock，还有 q 信号以外，基本上笔者都不认识，想必同学也一定会看到抓狂。

Port Name	Type	Required	Description
data_a	Input	Optional	Data input to port A of the memory. The data_a port is required if the operation_mode is set to any of the following values: <ul style="list-style-type: none"><li>■ SINGLE_PORT</li><li>■ DUAL_PORT</li><li>■ BIDIR_DUAL_PORT</li></ul>
address_a	Input	Yes	Address input to port A of the memory. The address_a port is required for all operation modes.
byteena_a	Input	Optional	Byte enable input to mask the data_a port so that only specific bytes, nibbles, or bits of the data are written. The byteena_a port is not supported in the following conditions: <ul style="list-style-type: none"><li>■ If implement_in_les parameter is set to ON</li><li>■ If operation_mode parameter is set to ROM</li></ul> For more information about byte enable feature and the criterion that you must follow to use the feature correctly, refer to "Byte Enable" on page 3-13.

图 6.3.7 官方提供的出入端说明（部分截图）。

当然，用心的官方也为我们提供详细的出入端说明，结果如图 6.3.7 所示。由于内容实在太多，笔者只是部分截图而已，如 data 信号，它是输入端，选择性，送入数据的信号；address 信号，它是输入端，固定性，送入地址的型号；byteena 信号，输入端，选择性，用来遮盖部分写数据位。官方所提供的输入端说明虽然详细，但是详细不代表清晰，也不一定具体，这种感觉好比走马看花似的，内容都是模模糊糊，尽是让人焦急又觉得心烦。

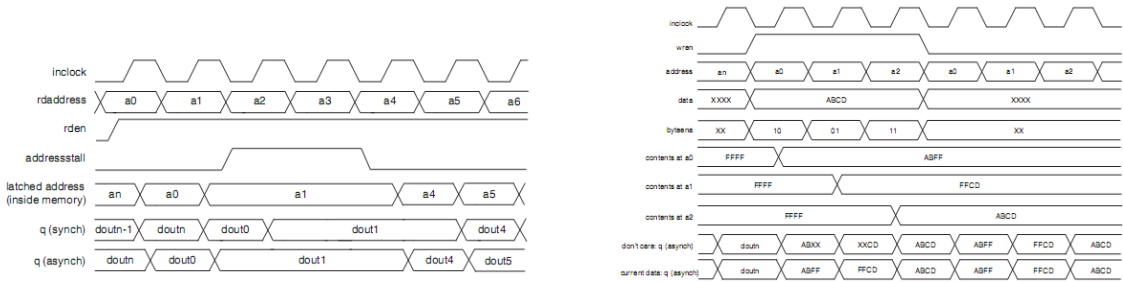


图 6.3.8 官方提供的时序图（部分截图）。

除此之外，用心的官方担心我们都不懂应用，也会为我们提供时序结果。如图 6.3.8 所



示，那是部分时序结果（由于内容实在太多笔者也懒惰——载图了），大致的意思是指某某信号有什么用途又该怎么使用 ... 想必有些同学一定会看到满头雾水，不知道它在讲什么，结果愈看脑袋愈要爆炸般发疼。

在某种程度上来说，官方的确已经尽心尽力为用户提供这个，提供那个，然而最关键的内容始终不见踪影。这种感觉好比给钱要狗狗自己去买食物般实在是太趣味了！聪明的狗狗说不定会自行拿钱买好料吃 ... 换之，如果是刚出生不久的狗仔，基本上都不知道什么是钱？结果又怎样晓得自行拿钱买吃呢？搞不定会错当把钱当成食物吃掉 ...

笔者曾是狗仔，现在也是狗仔！不管怎么样，这个问题确实困扰笔者许久，**笔者认为仿真对象左右了仿真效果，如果仿真对象无法解读效果也会大打折扣！**不过，官方竟也不是狼心狗肺，铁石心肠，经过多次别扭以后，虽然它们不能透露确切的内容，取而代之它们却公开另外的**等价内容**。

```
1. module single_port_ram
2. (
3.     input [7:0] data,
4.     input [7:0] addr,
5.     input we, clk,
6.     output [7:0] q
7. );
8.     reg [7:0] ram[255:0];
9.     reg [7:0] addr_reg;
10.
11.     always @ (posedge clk)
12.     begin
13.         if (we)
14.             ram[addr] <= data;
15.         addr_reg <= addr;
16.     end
17.
18.     assign q = ram[addr_reg];
19.
20. endmodule
```

代码 6.3.1

如代码 6.3.1 所示，一眼望去就知道这是官方所提供的等价内容，而且还是精简版。事实上，这是官方最大的让步，我们做人不应该得寸进尺，应该学会见好就收。第 3~6 行是相关的出入端声明；第 8 行声明位宽为 8 深度为 256 的 RAM，第 9 行则是用来寄存地址的寄存器 `addr_reg`；其中第 13~14 行才是关键，`if(we) ram[addr] <= data` 这段代码表示，将数据写入 ram 必须拉高 `we` 不可。

图 6.3.5 的仿真对象写入数据之所以不得不拉高 `RAM_WrEn`，**原来是这行代码在作怪**，

原来如此，原来这样，这样一来所有谜题都豁然开来，真相大白！但是仔细一想，我们为了仿真这样一个简单的 RAM 模块，必须大费周章解读手册，又分析时序，又寻找等价内容 ... 经过一番波折以后，结果却换来一只小小的成果而已。一心爱着节能的笔者，实在是无法接受这份事实！因为官方插件模块妨碍我们来不及讴歌青春 ... 结果它必须死，必须成为不可仿真对象。



## 6.4 不可仿真对象——超乱模块②

不管太阳再怎么毒辣，即使资料再怎么难消化，只要一杯特凉的豆浆送入口中，笔者也能取回冷静的心境。喝完一杯豆浆以后，笔者望着天花板开始思考，是谁规定模块非仿真不可？然而模块为什么又要仿真呢？思考之际，笔者随手翻看参考书，然后遇见这样一张流程图（Concept Flow Chart）。

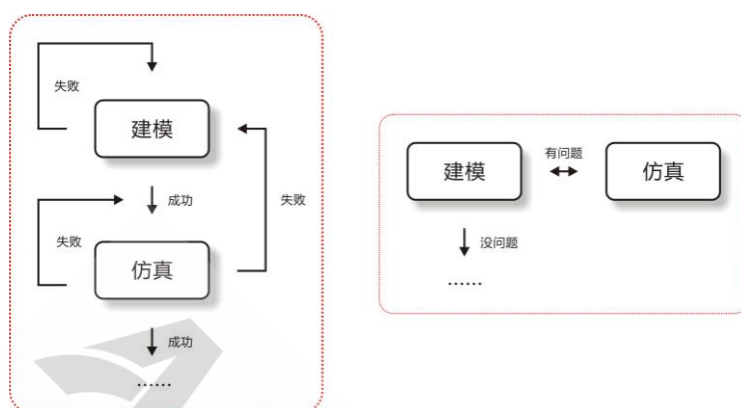


图 6.4.1 仿真流程图。

如图 6.4.1 所示，左图是参考书的流程图，那是非常正规的仿真流程，即建模失败继续返回建模；建模成功推向仿真；仿真失败就返回建模，或者再次仿真；仿真成功则往下继续推。左图表面上总是给人一种绝对有理的感觉，我们很难找到任何反驳的理由，结果不得不俯首称臣，屈服与它。但是反过的话，左图好似命令我们这样干那样干而已，却从来不说理由，其中成功的定义是什么？失败的定义又是什么？

小小的气愤之后，笔者随笔画了图 6.4.1 的右图。首先，建模还有仿真必须是同一个等级，示意建模还有仿真都有相同的根却有不同概念。紧接着，有问题的双箭头示意仿真是用来测试有问题的模块，**反之没有问题的模块则不用仿真** ... 想到这里，笔者开始停顿了一下。

“有问题的模块 ... 没有问题的模块 ...”，笔者嘟囔道。

“**官方插件模块有问题吗？**”，笔者继续嘟囔道。

没错，官方不可能发布有问题的插件模块，官方插件模块既然没有问题，我们为什么又要仿真呢？难道嫌青春太寂寞了吗？但是，**官方插件模块最大的问题就是很难驾驭**，虽然手册有详细的图形，出入端说明，还有时序结果，但是对于初学者而言，那些内容有如天书般实在很难理解。除非给钱，不然官方根本没有义务——为我们解释清楚。

除此之外，从自动思想的角度而言，没有问题的模块就是一只完整的个体，但问题是官方插件模块是**没有明显的特征**，因为内容被隐藏起来，结果我们无法留下深刻的印象。在此我们有几个解决手段：

（一）解读手册；

## （二）寻找替代；

虽说，驾驭官方插件模块最好的方法就是解读手册，然而关键是如何看懂时序图。为了实现这一点，除了提升基础能力以外，就是不停实验还有不停测试而已，笔者实在想不到其它好办法。官方插件模块虽然好用，只要我们习得驾驭的窍门，它们便会成为便利的家伙，对于那些喜欢快餐的朋友更是臭味相投。

这个世界上有两种懒人，一种是越懒越进步的人，另一种是越懒越退步的人。笔者作为一个懒人，虽然喜欢节能，但却不不怎么喜欢便利，因为贪图便利会让人退步。我们知道快餐虽然便利，但是吃久了就会伤害身体，同样的道理也适用建模。有些人会因为过度依赖官方插件模块，结果不仅失去建模能力，逐渐也会放弃思考，最后沦落为一只笨重又痴肥的懒家伙。因此，相较解读手册这个解决手段，笔者比较倾向寻找替代这个解决手段。

```
1. module single_port_ram
2. (
3.     input [7:0] data,
4.     input [7:0] addr,
5.     input we, clk,
6.     output [7:0] q
7. );
8.     reg [7:0] ram[255:0];
9.     reg [7:0] addr_reg;
10.
11.     always @ (posedge clk)
12.     begin
13.         if (we)
14.             ram[addr] <= data;
15.         addr_reg <= addr;
16.     end
17.
18.     assign q = ram[addr_reg];
19.
20. endmodule
```

代码 6.4.1

代码 6.4.1 是官方为我们提供的等价内容，我们可以基于代码 6.4.1 进一步修改让它，最终会成为更加直观更实用的模块。

```
1. module ram
2. (
3.     input CLOCK , RESET ,
4.     input RAM_WrEn,
```

```

5.     input [7:0] RAM_Addr,
6.     input [7:0] RAM_WrData,
7.     output [7:0] RAM_RdData
8. );
9.     reg [7:0] ram[255:0];
10.    reg [7:0] rData;
11.
12.    always @ (posedge CLOCK or negedge RESET)
13.        if( !RESET )
14.            rData <= 8' d0;
15.        else if( RAM_WrEn )
16.            ram[ RAM_Addr ] <= RAM_WrData;
17.        else
18.            rData <= ram[ RAM_Addr ];
19.
20.    assign RAM_RdData = rData;
21.
22. endmodule

```

代码 6.4.2

代码 6.4.2 是笔者简单修改以后的 RAM 模块，第 3~7 行是出入段的声明，基本上和代码 6.4.1 差不多，只是命名方法比较强调个体存在，为了进一步加深个体印象。第 10 行是 rData 寄存器是用来暂存读取结果。至于第 12~20 行却有很大的改变，代码 6.4.1 原本使用 rAddr 寄存器暂存地址数据，然后再用 ram 驱动 q 输出端。

相较之下，代码 6.4.2 不但没有暂存地址数据，而是直接应用地址数据选择 ram 的储存范围（第 16 行）。此外，代码 6.4.2 则使用 rData 寄存器暂存 ram 的读出结果（第 18 行），再用 rData 寄存器驱动 RAM\_RdData 输出端（第 20 行）。在此，有些同学可能不明白代码 6.4.2 的修改意义，其实代码 6.4.2 这样做是为了避免“写时读”的问题。有关写时读的详细内容，读者自己参考手册吧，关键字是 Read During Write。

接着，请打开 exp31 ... exp31 相比 exp29~30，最大的差别就是 exp31 使用了自建的 ram，然而 exp29~30 则是使用官方插件模块。

*exp31\_simulation.vt*

```

1.  `timescale 1 ps/ 1 ps
2.  module exp31_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg RAM_WrEn;
7.      reg [7:0] RAM_Addr,RAM_WrData;
8.      wire [7:0]RAM_RdData;

```

```

9.
10.  /*****/
11.
12.  initial
13.  begin
14.      RESET = 0; #10; RESET = 1;
15.      CLOCK = 1; forever #5 CLOCK = ~CLOCK;
16.  end
17.
18.  /*****/
19.
20.  ram U1
21.  (
22.      .RAM_Addr ( RAM_Addr ),
23.      .CLOCK ( CLOCK ),
24.      .RESET( RESET ),
25.      .RAM_WrData( RAM_WrData ),
26.      .RAM_WrEn ( RAM_WrEn ),
27.      .RAM_RdData ( RAM_RdData )
28.  );
29.
30.  /*****/
31.
32.  reg [3:0]i;
33.
34.  always @ ( posedge CLOCK or negedge RESET )
35.  if( !RESET )
36.  begin
37.      i <= 4'd0;
38.      RAM_WrEn <= 1'b0;
39.      RAM_Addr <= 8'd0;
40.      RAM_WrData <= 8'd0;
41.  end
42.  else
43.  case( i )
44.
45.      0:
46.          begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd0; RAM_WrData <=
8'hAA; i <= i + 1'b1; end
47.
48.      1:
49.          begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd1; RAM_WrData <=
8'hBB; i <= i + 1'b1; end

```

```

50.
51.          2:
52.          begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd2; RAM_WrData <=
      8'hCC; i <= i + 1'b1; end
53.
54.          3:
55.          begin RAM_WrEn <= 1'b0; RAM_Addr <= 8'd0; i <= i + 1'b1; end
56.
57.          4:
58.          begin RAM_WrEn <= 1'b0; RAM_Addr <= 8'd1; i <= i + 1'b1; end
59.
60.          5:
61.          begin RAM_WrEn <= 1'b0; RAM_Addr <= 8'd2; i <= i + 1'b1; end
62.
63.          6:
64.          i <= i;
65.
66.      endcase
67.
68.      /*****/
69.
70. endmodule

```

如代码 exp31\_simulation 所示，第 4~8 行是仿真所需的模拟寄存器与连线；第 12~16 行则是产生环境输入；第 20~28 行是自建 ram 的仿真对象；第 34~66 行则是虚拟输入。步骤 0~2 分别为 ram 写入：地址 0 数据 8'hAA，地址 1 数据 8'hBB，地址 2 数据 8'hCC。步骤 3~5 分别则是从 ram 哪里读取地址 0~2 的数据。

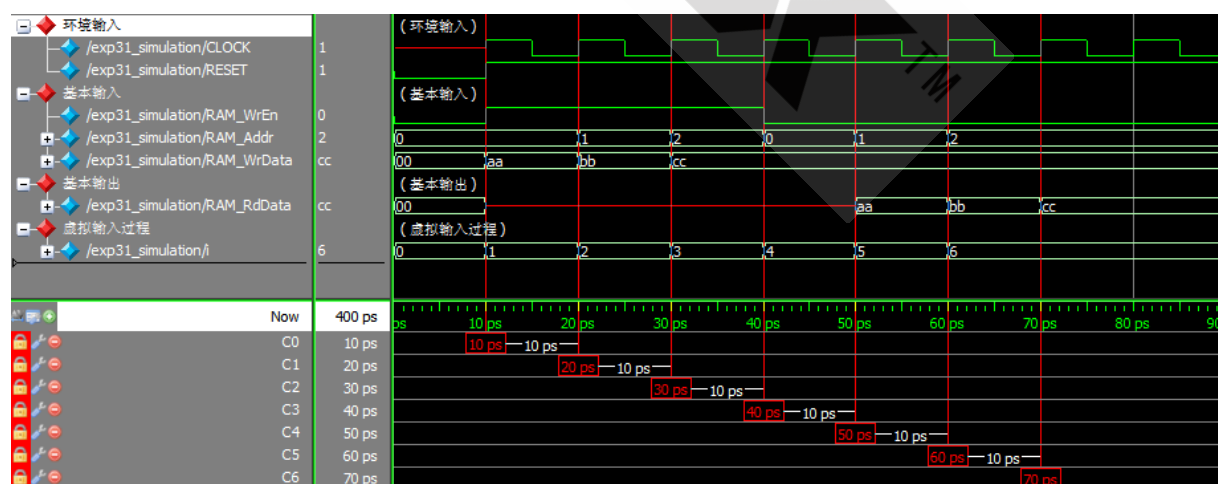


图 6.4.2 exp31 的仿真结果。

图 6.4.2 是 exp31 的仿真结果 其中光标 C0~C6 分别指向虚拟输入的步骤 0~6 也是 T0~T6。虚拟输入在 T0~T6 之间的操作：

T0，为仿真对象发送相关的未来值，WrEn 为 1，Addr 为 0，WrData 为 8'hAA；



T1, 为仿真对象发送相关的未来值, WrEn 为 1, Addr 为 1, WrData 为 8'hBB ;  
T2, 为仿真对象发送相关的未来值, WrEn 为 1, Addr 为 2, WrData 为 8'hCC ;  
T3, 为仿真对象发送相关的未来值, WrEn 为 0, Addr 为 0 ;  
T4, 为仿真对象发送相关的未来值, WrEn 为 0, Addr 为 1 ;  
T5, 为仿真对象发送相关的未来值, WrEn 为 0, Addr 为 2 ;  
T6, 无所事事。

仿真对象在 T0~T6 之间的操作 :

T0, 无所事事。

T1, 读取相关的过去值, WeEn 为 1 以示写操作, 将数据 8'hAA, 存入地址 0 ;  
T2, 读取相关的过去值, WeEn 为 1 以示写操作, 将数据 8'hBB, 存入地址 1 ;  
T3, 读取相关的过去值, WeEn 为 1 以示写操作, 将数据 8'hCC, 存入地址 2 ;  
T4, WrEn 过去值为 0 以示读操作, 将地址过去值 0 指向的数据 8'hAA, 输出为未来值 ;  
T5, WrEn 过去值为 0 以示读操作, 将地址过去值 1 指向的数据 8'hBB, 输出为未来值 ;  
T6, WrEn 过去值为 0 以示读操作, 将地址过去值 2 指向的数据 8'hCC, 输出为未来值 ;

上述的内容表示了 :

exp31 自建 ram 的功能, 相较官方插件模块 ram, 两者之间没有什么差别。但是自建 ram 有明显的优势, 亦即内容不仅公开而且也清晰, 解析仿真信息没有中断的问题, 因为仿真对象如何按照刺激产生反应都一一表明。这种感觉好比顺利上完大号般实在爽快无比, 期间也忍不住想唱快乐天堂: “告诉你一个神秘的地方~♪”。

站在主动思想的角度而言, 公开内容表示个体炫耀特征, 再加上清晰的内容, 个体无疑强化了自己的存在感 ... 强调自生对个体模块来说, 没有什么事情比这个更重要的。此外, 我们也用不着考虑 ( 仿真 ) 资源库的问题。

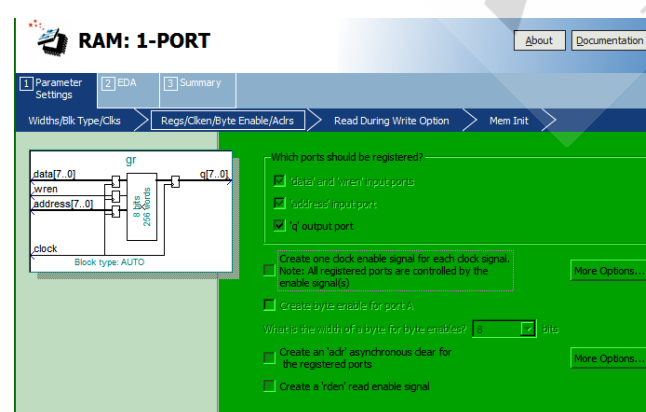


图 6.4.3 自定义官方插件模块。

除了这些琐碎的事情以外, 官方插件模块还有一个问题, 亦即自定义的自由度不大。如图 6.4.3 所示, 我可以透过集成环境自定义官方插件模块, 如输出端 q 启用寄存器, 或者启用 rden 读使能等 ... 然而, 官方插件模块愈是启用更多复选功能, 官方插件模块愈是越难驾驭, 特征还有存在感也会愈来愈模糊。官方插件模块虽然可以自定义, 但是

自定义的自由度也是有限的，**不能超过官方指定的范围。**

举例笔者是完全的新手，笔者觉得 ram 模块的时序很难控制，为此笔者需要自定义 ram，为它加入问答信号，让它变得更加容易控制。

```
1.  module ram
2.  (
3.      input CLOCK,RESET,
4.      input RAM_WrEn,
5.      input [7:0] RAM_Addr,
6.      input [7:0] RAM_WrData,
7.      input RAM_RdEn,
8.      output [7:0] RAM_RdData,
9.      output RAM_DnSig
10. );
11.     reg [7:0] ram[255:0];
12.     reg [7:0] rData;
13.     reg [3:0] i;
14.     reg isDone;
15.
16.     always @ (posedge CLOCK or negedge RESET)
17.         if( !RESET )
18.             begin
19.                 rData <= 8'd0;
20.                 i <= 2'd0;
21.                 isDone <= 1'b0;
22.             end
23.         else if( RAM_WrEn )
24.             case( i )
25.
26.                 0:
27.                     begin ram[ RAM_Addr ] <= RAM_WrData; i <= i + 1'b1; end
28.
29.                 1:
30.                     begin isDone <= 1'b1; i <= i + 1'b1; end
31.
32.                 2:
33.                     begin isDone <= 1'b0; i <= 2'd0; end
34.
35.             endcase
36.         else if( RAM_RdEn )
37.             case( i )
38.
```

```

39.             0:
40.             begin rData <= ram[ RAM_Addr ]; i <= i + 1'b1; end
41.
42.             1:
43.             begin isDone <= 1'b1; i <= i + 1'b1; end
44.
45.             2:
46.             begin isDone <= 1'b0; i <= 2'd0; end
47.
48.         endcase
49.
50.     assign RAM_RdData = rData;
51.     assign RAM_DnSig = isDone;
52.
53. endmodule

```

代码 6.4.3

代码 6.4.3 是加入问答信号的 ram 模块，第 3~9 行是相关的出入端声明，其中多添加了 RAM\_RdEn 与 RAM\_DnSig；第 11~14 行是相关的寄存器声明，i 用来指向步骤，isDone 用来反馈完成信号；第 23~35 行是 ram 的写操作，RAM\_WrEn 作为使能信号；第 36~48 行则是 ram 的读操作，RAM\_RdEn 作为使能信号。第 51 的 RAM\_DnSig 输出端则用 isDone 寄存器来驱动。

#### *exp32\_simulation.vt*

```

1.  `timescale 1 ps/ 1 ps
2.  module exp32_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg RAM_WrEn;
7.      reg [7:0] RAM_Addr,RAM_WrData;
8.      reg RAM_RdEn;
9.      wire [7:0]RAM_RdData;
10.     wire RAM_DnSig;
11.
12.     /*******/
13.
14.     initial
15.     begin
16.         RESET = 0; #10; RESET = 1;
17.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
18.     end
19.

```

```

20.  /*****/
21.
22.  ram U1
23.  (
24.      .CLOCK ( CLOCK ),
25.      .RESET( RESET ),
26.      .RAM_WrEn ( RAM_WrEn ),
27.      .RAM_Addr ( RAM_Addr ),
28.      .RAM_WrData( RAM_WrData ),
29.      .RAM_RdEn ( RAM_RdEn ),
30.      .RAM_RdData ( RAM_RdData ),
31.      .RAM_DnSig( RAM_DnSig )
32.  );
33.
34.  /*****/
35.
36.  reg [3:0]i;
37.
38.  always @ ( posedge CLOCK or negedge RESET )
39.      if( !RESET )
40.          begin
41.              i <= 4'd0;
42.              RAM_WrEn <= 1'b0;
43.              RAM_RdEn <= 1'b0;
44.              RAM_Addr <= 8'd0;
45.              RAM_WrData <= 8'd0;
46.          end
47.      else
48.          case( i )
49.
50.              0:
51.                  if( RAM_DnSig ) begin RAM_WrEn <= 1'b0; i <= i + 1'b1; end
52.                  else begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd0; RAM_WrData <= 8'hAA; end
53.
54.              1:
55.                  if( RAM_DnSig ) begin RAM_WrEn <= 1'b0; i <= i + 1'b1; end
56.                  else begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd1; RAM_WrData <= 8'hBB; end
57.
58.              2:
59.                  if( RAM_DnSig ) begin RAM_WrEn <= 1'b0; i <= i + 1'b1; end
60.                  else begin RAM_WrEn <= 1'b1; RAM_Addr <= 8'd2; RAM_WrData <= 8'hCC; end
61.
62.              3:

```

```

63.         if( RAM_DnSig ) begin RAM_RdEn <= 1'b0; i <= i + 1'b1; end
64.         else begin RAM_RdEn <= 1'b1; RAM_Addr <= 8'd0; end
65.
66.         4:
67.         if( RAM_DnSig ) begin RAM_RdEn <= 1'b0; i <= i + 1'b1; end
68.         else begin RAM_RdEn <= 1'b1; RAM_Addr <= 8'd1; end
69.
70.         5:
71.         if( RAM_DnSig ) begin RAM_RdEn <= 1'b0; i <= i + 1'b1; end
72.         else begin RAM_RdEn <= 1'b1; RAM_Addr <= 8'd2; end
73.
74.         6:
75.         i <= i;
76.
77.     endcase
78.
79.     /***/
80.
81. endmodule

```

exp32 基本上与 exp31 差不了多少，除了第 8 行 RAM\_RdEn 寄存器，还有第 10 行 RAM\_DnSig 连线。第 38~77 行是虚拟输入，由于仿真对象应用了问答信号，所以控制仿真对象也会变得非常简单。步骤 0~2 接续为仿真对象写入数据 8'hAA ,8'hBB ,8'hCC 在地址 0~2。步骤 3~5 分别接续从仿真对象的地址 0~2 读出数据 8'hAA ,8'hBB ,8'hCC。

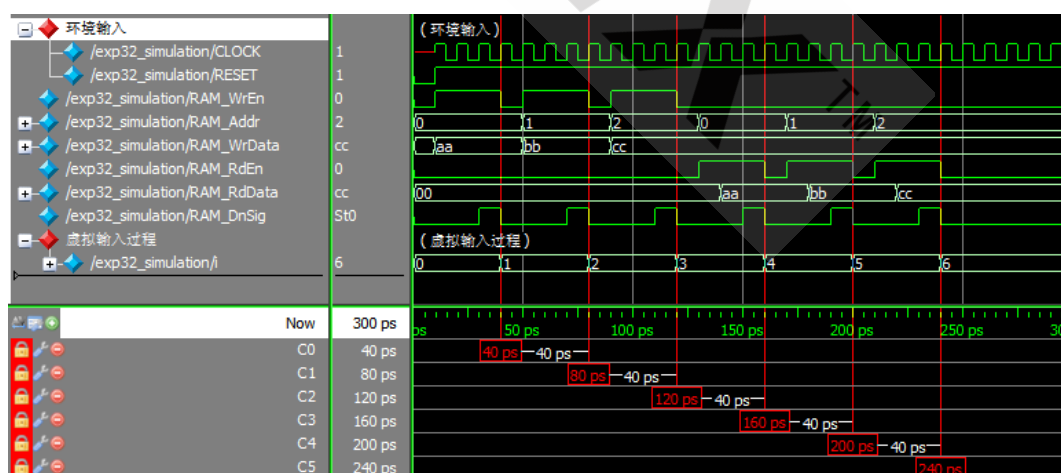


图 6.4.4 exp32 的仿真结果。

图 6.4.4 是 exp32 的仿真结果，光标 C0~C5 分别指向 3 个写操作 3 个读操作的完成信号。

期间：

C0 指向的地方是虚拟输入为仿真对象写入数据 8'hAA 在地址 0；

C1 指向的地方是虚拟输入为仿真对象写入数据 8'hBB 在地址 1 ;  
C2 指向的地方是虚拟输入为仿真对象写入数据 8'hCC 在地址 2 ;  
C3 指向的地方是虚拟输入从仿真对象的地址 0 读出数据 8'hAA ;  
C4 指向的地方是虚拟输入从仿真对象的地址 1 读出数据 8'hBB ;  
C5 指向的地方是虚拟输入从仿真对象的地址 2 读出数据 8'hCC ;

无可否认，[仿真对象应用问答信号以后变得更加容易控制](#)，期间我们用不着考虑时序的是否精密，不过[作为代价，exp32 用了更长的时间进行读写操作](#)。同学可能会好奇道，官方插件模块是否也能这样，应用问答信号执行呢？答案虽然是可以，但是过程比较猥琐。我们知道官方插件信号不能直接修改内容，除非经过集成环境 ... 为此，首先我们需要在顶层模块的身上将官方插件模块实例化，然后才能自定义。

```
1.  module ram_top
2.  (
3.      input CLOCK,RESET,
4.      input RAM_WrEn,
5.      input [7:0] RAM_Addr,
6.      input [7:0] RAM_WrData,
7.      input RAM_RdEn,
8.      output [7:0] RAM_RdData,
9.      output RAM_DnSig
10. );
11.     wire [7:0] wRAM_RdData;
12.     ramU1
13.     (
14.         .address ( RAM_Addr ),
15.         .clock ( CLOCK ),
16.         .data ( RAM_WrData ),
17.         .wren ( RAM_WrEn ),
18.         .q ( RAM_RdData )
19.     );
20.
21.     reg [3:0]i;
22.     reg isDone;
23.
24.     always @ (posedge CLOCK or negedge RESET)
25.         if( !RESET )
26.             begin
27.                 i <= 2'd0;
28.                 isDone <= 1'b0;
29.             end
30.         else if( RAM_WrEn )
31.             case( i )
```

```

32.
33.             0:
34.             begin isDone <= 1'b1; i <= i + 1'b1; end
35.
36.             1:
37.             begin isDone <= 1'b0; i <= 2'd0; end
38.
39.         endcase
40.     else if( RAM_RdEn )
41.         case( i )
42.
43.             0:
44.             begin isDone <= 1'b1; i <= i + 1'b1; end
45.
46.             1:
47.             begin isDone <= 1'b0; i <= 2'd0; end
48.
49.         endcase
50.
51.     assign RAM_DnSig = isDone;
52.
53. endmodule

```

代码 6.4.4

大致的感觉入代码 6.4.4 所示。笔者现在顶层模块实例化 ram 模块 ( 第 11~19 行 ), 然后再加入问答机制 ( 第 30~49 行 )。代码 6.4.4 相比代码 6.4.3 , 读写操作都少了一个步骤 , 但却发挥一样的功能。仔细比较一下代码 6.4.3 还有代码 6.4.4 , 我们会发现代码 6.4.3 的自定义会比较直接也比较省力。

---

最后 , 笔者可以这样总结道 :

超乱模块是不可仿真对象之一 , 因为仿真对象解读不可 , 结果阻碍了仿真对象 , 激励内容 , 还有时序结果之间的联系 , 仿真信息的解析工作随之也糟中断。超乱模块的典型例子有 : 失去结构的模块 , 还有官方插件模块 ... 前者可以注入结构解决问题 , 后者则是比较麻烦一点。由于官方插件模块是隐藏实际内容 , 而且模块内容也有许多意义不明的关键字 , 为了执行仿真 , 我们必须做好以下准备 :

4. 加入资源库 ;
5. 解读手册 , 寻找等价内容 , 然后再摸索大概的模块内容 ;

如果能找到等价内容 , 那当然是最好的事情 ... 我们可以基于等价内容再自行创建同样功能的模块用来替代官方插件模块 , 否则我们必须硬啃手册 , 进一步摸索隐藏内容才行。



## 6.5 主动设计①

玩过 RPG 游戏的同学一定知晓，勇者基本上都不能可能一次通过 BOSS 战役，除非那是臭游戏，没有做好平衡微调。勇者必须用死亡换取 BOSS 的信息，经过数次累积以后才能完全攻略 BOSS。游戏中，勇者可以无限重生，但是反比**玩家，游戏时间越长精神就会越差** ...

同样的事情也发生在仿真的身上，BOSS 战役好比仿真，我们好比玩家，期间仿真会经历无数次失败，然后有用的信息就会累积越多，直至仿真成功。仿真虽然可以无数次重复，但是我们的精力，时间，还有忍耐力是有限的，仿真时间越是长久，我们越是疲倦，仿真效率越是低下 ... 如果这种轮回不停重复，最终会成为噩梦般的恶性循环。**这种依靠死亡换来的成功，笔者称为被动设计。**

**“被动”意指，我们终被仿真牵着鼻子走，而不是我们引导仿真前进。**传统流派也好，常规也罢，基本上都是采用被动设计的仿真手段 ... 从小到大，笔者都是一幅弱身子，笔者实在无法忍受仿真的折腾，笔者每经一场战役，笔者越是感觉生命仿佛短了几年。某天，笔者忽然察觉不妙，如果总是被仿真拖着鼻子走下去，笔者总有一天一定会成为人干。为此，笔者开始思考如何夺取仿真的主导权。

期间，笔者反问自己，为何总是被仿真牵着鼻子走呢？首先，我们知道 Modelsim 是用来播放可视化时序信息 ... 换句话说，**失去仿真我们好比失去一对窥视时序的眼睛**，变成时序盲子。不管如何，**仿真始终都是他人的眼睛**，我们必须借用别人的眼睛走路，状况极为被动。笔者随之又思考道，我们又不是没长眼睛，只是眼睛看不清楚时序而已？

**“为什么，我们的眼睛看不清楚时序？无论是肉眼还是心眼 ... ”**笔者感叹道。

事实上，我们的眼睛**不是看不清楚时序，而是时序本身被一层模糊烟雾遮盖罢了**，为了看清时序，我们必须将其去除以致明朗化。为此，笔者发现“提高模块的表达能力”还有“清晰模块内容”是取回主导权的关键。不过，笔者感觉只有这些还是**物所不足**，笔者需要一个思想，**一个可以连贯一切的思想**，那就是主动思想。

主动思想的副作用是**个体拥有清晰的内容**，我们用不着仿真也能脑补个体的时序图。然而，主动思想的主作用是，**用清晰的个体内容留下强烈的局部印象**，然后深深映在想象当中。无数局部印象不停填充以后，最终整体的时序图会脑补完成。结果而言，**想象力便成为最便捷的仿真工具**，我们可以一边建模一边在脑补时序。

这种想象的仿真工具虽然没有形体但它至少**是我们的眼睛，让我们摆脱被动** ... 此外，想象力也是自由的，只要脑补能力足够强大（想象力丰富）这个工具基本上是无所不能，因此它才是最强的仿真工具。不过作为前提条件，我们必须运用技巧将信息，深刻地映入脑海当中。这种应用**主动思想的设计技巧，笔者称为主动设计。**

接下来，让笔者用简单的例子来演示主动设计的优劣点。此外，读者也可以当成是最后

综合复习。

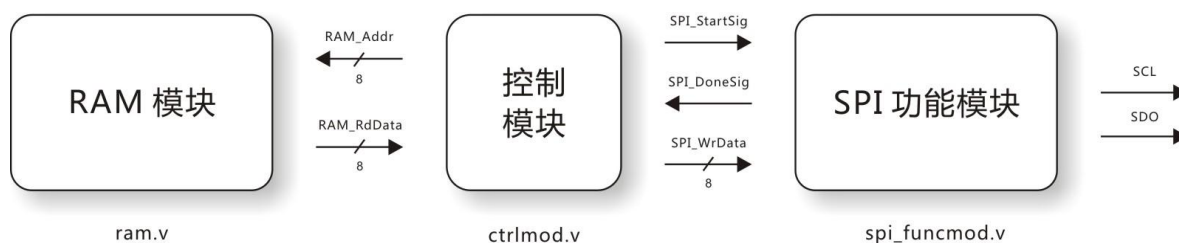


图 6.5.1 exp33 建模图。

图 6.5.1 是 exp33 的建模图，其中包裹 ram 模块，控制模块，还有 spi 功能模块。ram 模块用来储存信息，spi 功能模块用来发送串行数据，控制模块则是两者之间的协调。exp33 用意很简单，控制模块先从 ram 模块读取数据，然后再经由 spi 功能模块发送出去。接下来，让我们来瞧瞧它们的具体内容。

*ram.v*

```
1. module ram
2. (
3.     input CLOCK,RESET,
4.     input RAM_WrEn,
5.     input [7:0] RAM_Addr,
6.     input [7:0] RAM_WrData,
7.     output [7:0] RAM_RdData
8. );
9.     reg [7:0] ram[255:0];
10.    reg [7:0] rData;
11.
12.    initial begin
13.        ram[1] = 8'b1100_0011;
14.        ram[2] = 8'b1000_0001;
15.    end
16.
17.    always @ (posedge CLOCK or negedge RESET)
18.        if( !RESET )
19.            rData <= 8'd0;
20.        else if( RAM_WrEn )
21.            ram[ RAM_Addr ] <= RAM_WrData;
22.        else
23.            rData <= ram[ RAM_Addr ];
24.
25.    assign RAM_RdData = rData;
26.
```

上述代码是 ram 模块，笔者直接向 exp31 借用，然而第 12~14 则是初始化 ram 的写法之一，为了后续的仿真作准备。除此之外我们也可以用 for 来初始化 ram。

```
reg [9:0]x;
for( x = 0; x < 256; x = x + 1' b1 )
    ram[x] <= x;
```

*spi\_funcmod.v*

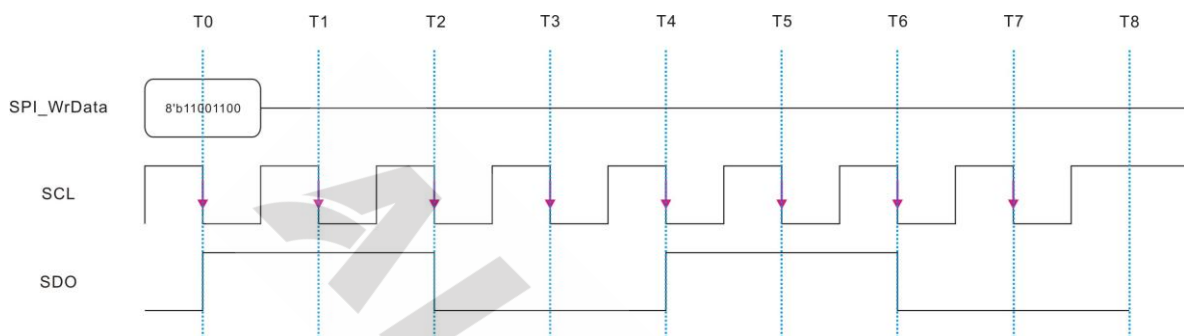


图 6.5.2 SPI 的写操作时序（主机）。

SPI 传输协议是应用较为普遍的传输协议之一，SPI 传输协议一般有分主机与从机，主机是控制 SCL 的一方，反之亦然。SPI 都是 SCL 下降设置数据（发送数据），SCL 上升沿锁存数据（读取数据），不管写操作还是读操作都是从数据的最高位开始。如图 6.5.2 所示，这是从主机视角看去的写操作，其中发送写数据是 8'b11001100，因此 SDO 在 T0~T1 拉高，T2~T3 拉低，T4~T5 拉高，T6~T7 拉低。

在这之前笔者先假定 spi\_funcmod 使用 50Mhz 的时钟源，然而 SCL 的频率是 1Mhz，因此：

$$(1/1\text{Mhz}) / (1/50\text{Mhz}) = 50$$

50Mhz 时钟源计数 50 次等于 1 个 SCL 时钟，半周期是 25。不过，笔者认为计数 25 下的水分太多，为此笔者将它不等例缩放为 5。接下来，让我们悄悄 spi\_funcmod 具体的内容：

```
1. module spi_funcmod
2. (
3.     input CLOCK,RESET,
4.     input SPI_StartSig,
5.     output SPI_DoneSig,
6.     input [7:0] SPI_WrData,
7.     output SCL,SDO
```

```

8. );
9.     parameter T1US = 8'd5; // (1/1Mhz) / (1/50Mhz) = 50;
10.
11.     reg [4:0]i;
12.     reg [7:0]C1;
13.     reg rSDO,rSCL;
14.     reg isDone;
15.
16.     always @ (posedge CLOCK or negedge RESET)
17.         if( !RESET )
18.             begin
19.                 i <= 5'd0;
20.                 C1 <= 8'd0;
21.                 {rSDO, rSCL} <= 2'b01;
22.                 isDone <= 1'b0;
23.             end
24.         else if( SPI_StartSig )
25.             case( i )
26.
27.                 0,2,4,6,8,10,12,14:
28.                     if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
29.                     else begin C1 <= C1 + 1'b1; rSCL <= 1'b0; rSDO <= SPI_WrData[7-(i>1)]; end
30.
31.                 1,3,5,7,9,11,13,15:
32.                     if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
33.                     else begin C1 <= C1 + 1'b1; rSCL <= 1'b1; end
34.
35.                 16:
36.                     begin isDone <= 1'b1; i <= i + 1'b1; end
37.
38.                 17:
39.                     begin isDone <= 1'b0; i <= 5'd0; end
40.
41.             endcase
42.
43.     assign SCL = rSCL;
44.     assign SDO = rSDO;
45.     assign SPI_DoneSig = isDone;
46.
47. endmodule

```

第 3~7 行是相关的出入端声明；第 9 行是 T1US 的常量定义，但是笔者将它不等例缩放为 5；第 11~14 是相关的寄存器声明；第 16~41 行是核心操作，SPI 传输协议默认下 SCL

时钟都是拉高，因此第 21 的复位操作 rSCL 为 1。第 24 行表示该操作必须使能 SPI\_StartSig 不可。第 27~29 是用来拉低 SCL 和设置数据（输出数据）的步骤，占有 8 个步骤。

第 29 行的 SPI\_WrData[7-(i>>1)]表示输出数据从最高位开始。第 31~33 行是用来拉高 SCL 而已，也是占有 8 个步骤。步骤 16~17 则是用来反馈完成信号。之余第 43~45 行则是相关的输出驱动。笔者之前也说过，本实验是采用主动设计，主动设计是非常强调个体，而且内容的清晰程度足以让我们脑补时序，甚至留下深刻印象。一般上，笔者为了节能是不做仿真的，不过为了讲明主动设计，我们还是稍作仿真吧。

### *spi\_funcmod.vt*

```
1.  `timescale 1 ps/ 1 ps
2.  module spi_funcmod_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.      reg SPI_StartSig;
7.      reg [7:0]SPI_WrData;
8.      wire SPI_DoneSig;
9.      wire SCL,SDO;
10.
11.      /*****/
12.
13.      initial
14.      begin
15.          RESET = 0; #10; RESET = 1;
16.          CLOCK = 1; forever #5 CLOCK = ~CLOCK;
17.      end
18.
19.      /*****/
20.
21.      spi_funcmod U1
22.      (
23.          .CLOCK ( CLOCK ),
24.          .RESET( RESET ),
25.          .SPI_StartSig( SPI_StartSig ),
26.          .SPI_DoneSig( SPI_DoneSig ),
27.          .SPI_WrData( SPI_WrData ),
28.          .SCL ( SCL ),
29.          .SDO ( SDO )
30.      );
31.
```

```

32.      /***/
33.
34.      reg [3:0]i;
35.
36.      always @ ( posedge CLOCK or negedge RESET )
37.      if( !RESET )
38.      begin
39.          i <= 4'd0;
40.          SPI_StartSig <= 1'b0;
41.          SPI_WrData <= 8'd0;
42.      end
43.      else
44.      case( i )
45.
46.          0:
47.              if( SPI_DoneSig ) begin SPI_StartSig <= 1'b0; i <= i + 1'b1; end
48.              else begin SPI_StartSig <= 1'b1; SPI_WrData <= 8'b1001_1001; end
49.
50.          1:
51.              i <= i;
52.
53.      endcase
54.
55.      /***/
56.
57.  endmodule

```

spi\_funcmod.vt 是 spi\_funcmod.v 的个体仿真环境，也是所谓的局部仿真。因为是不等例缩放缘故，笔者用不着再现真实的时钟周期。第 21~30 行是仿真对象的实例化，第 36~53 行是虚拟输入的激励内容，其中步骤 0 是为仿真对象写入数据 8'b1001\_1001。

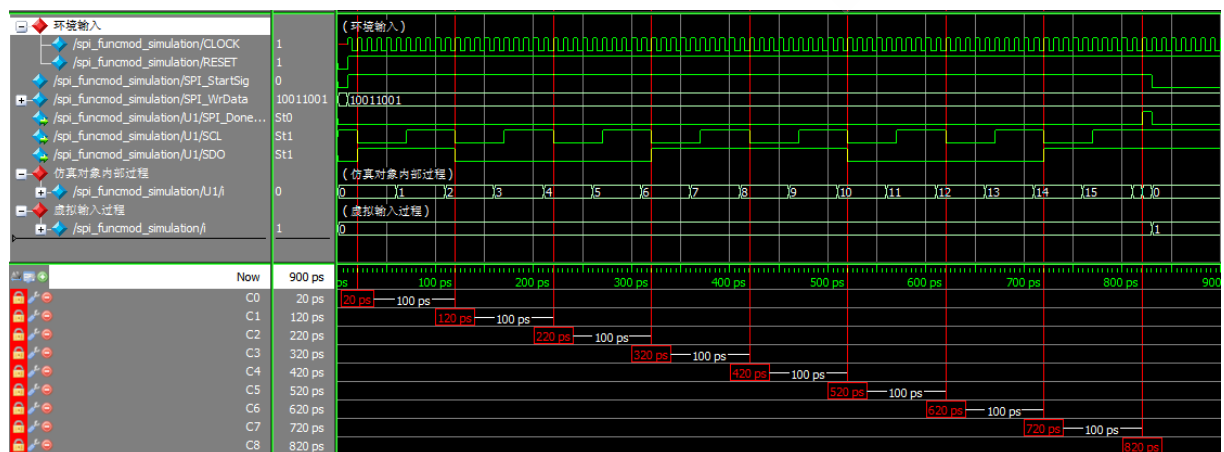


图 6.5.3 spi\_funcmod 的仿真结果。

图 6.5.3 是 spi\_funcmod 的仿真结果，其中光标 C0~C7 分别指向 SCL 的 8 次下降沿。

C0 之际，SD0 输出未来值 1；  
C1 之际，SD0 输出未来值 0；  
C2 之际，SD0 输出未来值 0；  
C3 之际，SD0 输出未来值 1；  
C4 之际，SD0 输出未来值 1；  
C5 之际，SD0 输出未来值 0；  
C6 之际，SD0 输出未来值 0；  
C7 之际，SD0 输出未来值 1；

由于仿真对象应用了问答沟通，结果仿真对象变得容易控制了。C8 指向的地方是仿真对象完成一次性的操作，并且返回完成信号以示一次性的写操作已经完成。

*ctrlmod.v*

```
1.  module ctrlmod
2.  (
3.      input CLOCK,RESET,
4.
5.      output [7:0]RAM_Addr,
6.      input [7:0]RAM_RdData,
7.
8.      output SPI_StartSig,
9.      input SPI_DoneSig,
10.     output [7:0] SPI_WrData
11. );
12.     reg [3:0]i;
13.     reg [7:0]rAddr,rData;
14.     reg isSPI;
15.
16.     always @ (posedge CLOCK or negedge RESET)
17.         if( !RESET )
18.             begin
19.                 i <= 4'd0;
20.                 { rAddr, rData } <= { 8'd0,8'd0 };
21.                 isSPI <= 1'b0;
22.             end
23.     else
24.         case( i )
25.
26.             0,1:
27.                 begin rAddr <= 8'd1; i <= i + 1'b1; end
```



```

28.
29.         2:
30.         begin rData <= RAM_RdData; i <= i + 1'b1; end
31.
32.         3:
33.         if( SPI_DoneSig ) begin isSPI <= 1'b0; i <= i + 1'b1; end
34.         else begin isSPI <= 1'b1; end
35.
36.         4,5:
37.         begin rAddr <= 8'd2; i <= i + 1'b1; end
38.
39.         6:
40.         begin rData <= RAM_RdData; i <= i + 1'b1; end
41.
42.         7:
43.         if( SPI_DoneSig ) begin isSPI <= 1'b0; i <= i + 1'b1; end
44.         else begin isSPI <= 1'b1; end
45.
46.         8:
47.         i <= i;
48.
49.     endcase
50.
51.     assign RAM_Addr = rAddr;
52.     assign SPI_StartSig = isSPI;
53.     assign SPI_WrData = rData;
54.
55. endmodule

```

控制模块是低级建模当中比较重要的模块，控制模块一般都不用仿真，因为它的责任只是协调和发号而已。另一方面，**控制模块也是仿真的切入口**。第 3~10 行是相关的出入口声明；第 12~14 行是相关的寄存器声明；第 16~49 行是控制模块的核心操作。根据 exp33 的实验要求，控制模块的功能是从 ram 读取数据，然后再将数据送往 spi 功能模块发送出去。

它首先在步骤 0 为 RAM\_rAddr 赋值为 1，亦即从 ram 的地址 1 读取数据；步骤 1 是给足 ram 有充分的时间读出数据；步骤 2 则将读出的数据赋给 rData 寄存器；步骤 3 是启动 spi 功能模块将方才读到的数据发送出去。步骤 4~7 则是重复步骤 0~3 的操作不过是发送数据不同而已。

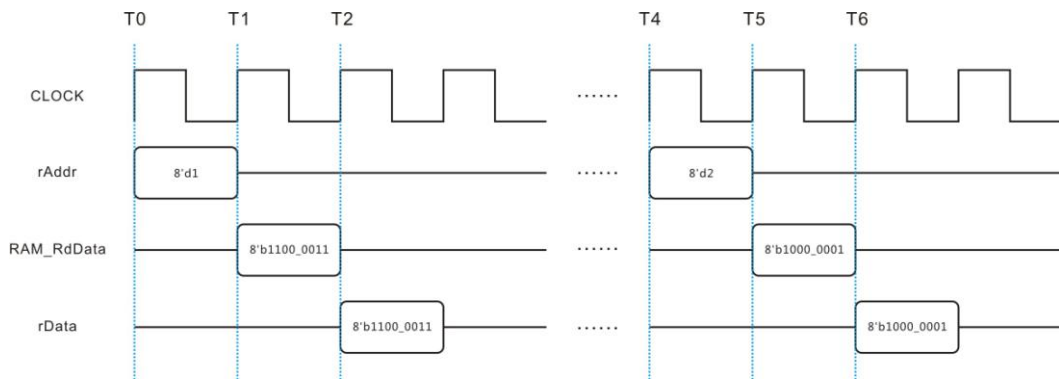


图 6.5.4 笔者脑补的时序图。

步骤 0~2，还有步骤 4~7 假设它们分别指向时钟时钟 T0~T2，T3~T5。i 成功指向时钟以后，内容会无比清晰，此刻笔者就能自己能脑补时序，如图 6.5.4 所示，那是笔者脑补的时序图，过程如下：

- T0 之际，它为 ram 发送地址 8'd1；
- T1 之际，此刻它在发呆。同一时刻，ram 接收地址数据，接着发送数据 8'b1100\_0011；
- T2 之际，它将数据 8'b1100\_0011 在 rData 寄存器里。
- T4 之际，它为 ram 发送地址 8'd2；
- T5 之际，此刻它在发呆。同一时刻，ram 接收地址数据，接着发送数据 8'b1000\_0001；
- T6 之际，它将数据数据 8'b1000\_0001 在 rData 寄存器里。

就这样，我们可以省下许多不必要的仿真时间。

*exp33\_simulation.vt*

```

1. `timescale 1 ps/ 1 ps
2. module exp33_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.
7.     /*****/
8.
9.     initial
10.    begin
11.        RESET = 0; #10; RESET = 1;
12.        CLOCK = 1; forever #5 CLOCK = ~CLOCK;
13.    end
14.
15.    /*****/
16.
17.    wire [7:0]RAM_Addr, RAM_RdData;
```

```

18.
19.     ram U1
20.     (
21.         .RAM_Addr ( RAM_Addr ),
22.         .CLOCK ( CLOCK ),
23.         .RESET( RESET ),
24.         .RAM_WrData( ),
25.         .RAM_WrEn ( ),
26.         .RAM_RdData ( RAM_RdData )
27.     );
28.
29.     wire SPI_StartSig, SPI_DoneSig;
30.     wire [7:0]SPI_WrData;
31.
32.     ctrlmod U2
33.     (
34.         .CLOCK( CLOCK ),
35.         .RESET( RESET ),
36.         .RAM_Addr( RAM_Addr ),
37.         .RAM_RdData( RAM_RdData ),
38.         .SPI_StartSig( SPI_StartSig ),
39.         .SPI_DoneSig( SPI_DoneSig ),
40.         .SPI_WrData( SPI_WrData )
41.     );
42.
43.     spi_funcmod U3
44.     (
45.         .CLOCK ( CLOCK ),
46.         .RESET( RESET ),
47.         .SPI_StartSig( SPI_StartSig ),
48.         .SPI_DoneSig( SPI_DoneSig ),
49.         .SPI_WrData( SPI_WrData ),
50.         .SCL ( SCL ),
51.         .SDO ( SDO )
52.     );
53.
54.     /***/
55.
56. endmodule

```

exp33 是该实验的仿真环境，如果按照平常的建模习惯，笔者一定事先建立组合模块将 ram 模块，控制模块，还有 spi 功能模块组合起来再仿真。但是，根据主动设计而言，上述 3 个模块基本上已有清晰的内容，强烈的印象，还有脑补的时序，已经足够完事了，

如果读者有足够的信心，读者可以无视再仿真。不过世事也有万一的时候，小心能驶万年船，为求安心多作一下仿真也好。

如 exp33 所示，笔者按照 exp33 的建模图将 3 个模块组合起来。其中，第 24~25 行的 RAM\_WrEn 与 RAM\_WrData 信号，笔者有意将它们放空，因为 exp33 没有它们的出场机会。

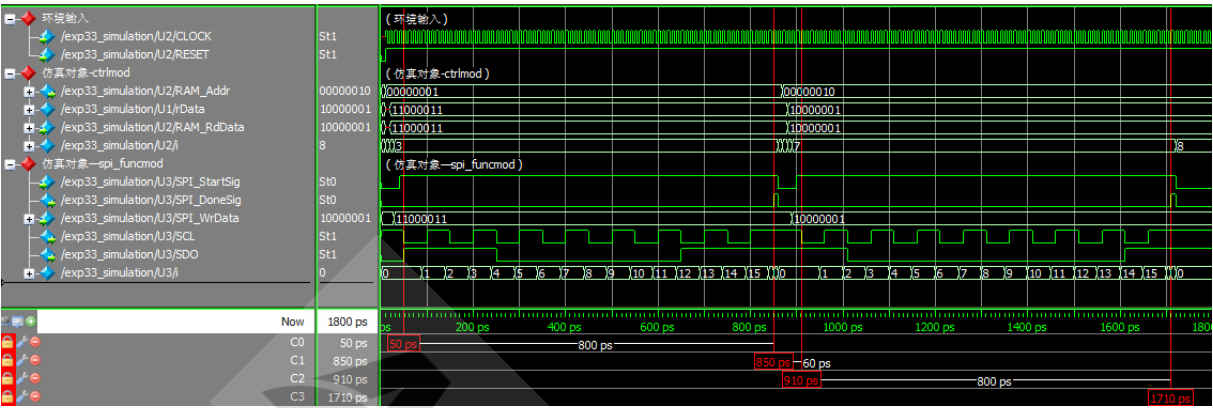


图 6.5.5 exp33 的仿真结果。

图 6.5.5 是 exp33 的仿真结果，其中光标 C0~C1 指向 SPI 的第一次写操作，C2~C3 则指向 SPI 的第二次写操作。根据 exp33 的实验要求，控制模块经由 RAM\_Addr 信号为 ram 模块写入地址，接着 ram 模块便会吐出相对应的数据，然后再由控制模块将数据送入 spi 功能模块，最终以 SPI 的传输方式发送出去。

如图 6.5.5 所示，第一次操作是读出地址 1 的数据 8'b1100\_0011，并且按照 SPI 的传输方式发送出去。第二次操作则是读出地址 2 的数据 8'b1000\_0001，再由 SPI 方式发送出去。图 6.5.5 的时序结果相较笔者脑补的时序图，两者极为接近，虽然仿真工具显示的时序结果非常具体，不过这是借用它人眼睛看到的景象，颇为被动 ... 反之，想象力建立的时序结果虽然抽象，不过那是心眼看到的景象，较为主动。

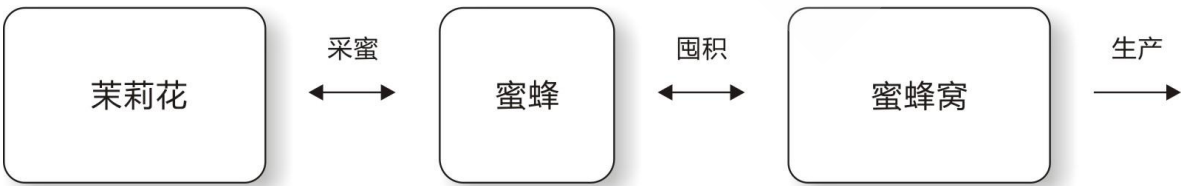


图 6.5.6 exp33 的比喻图。

笔者为了强调主动设计，笔者故意瞎搞一下 exp33。如图 6.5.6 所示，ram 模块比喻为茉莉花，控制模块比喻为蜜蜂，SPI 功能模块比喻为蜜蜂窝。茉莉花的功能是生产花蜜，蜜蜂的工作是从茉莉花哪里采集花蜜以后，再囤积在蜜蜂窝里，至于蜜蜂窝是用来生产蜂蜜的工厂。

如果我们站在整体视角去观察事物，我们自然会认为蜂蜜生产必须依靠茉莉花，蜜蜂，还有蜜蜂窝三者之间的合作才能顺利完成，期间三者保持亲密的关系。整体视角一般也

称为神的视角，神为了掌握所有，神必须接受所有个体情况之余，神还要了解个体之间的交流状况。不管我们的脑袋能力再怎么好，我们始终没有神般的吞吐量，一口吃完整块切糕根本就是痴心妄想。

除此之外，大口大口进食原本就是一种坏习惯，我们分分钟都会消化不良照成反胃。如果读者坚持要用神的视角去观察一切，笔者建议最好挑食一点，只选自己在意的部分。偏食虽说也是一种不好的进食习惯，营养容易失衡，仿真也是理解一块一块，不过那也是没有主动设计的情况下而已。主动设计本来就是个体视角东西，整体视角只是用来看爽 ... 啊不！是辅助性质。

如果我们站在个体视角去观察事物，我们会发现茉莉花尽管生产花蜜，蜜蜂尽管采蜜和囤积，蜜蜂窝尽管生产蜂蜜而已，期间三者都保持独立又陌生的关系。个体视角是一种，先将大切糕分为无数小切糕，然后再逐步消化的方法。因为我们没有神一般的能耐掌握一切，余下我们只能掌握局部信息，然后再运用想象力将所有局部信息结合起来，成为一副完成的巨大的信息。个体视角不仅可以减轻左脑的负担，个体视角也让我们更加集中处理事情。

个体视角好比人类当下的社会现象，人与人之间只有陌生与自私，不过仔细一想，那是个体尽全力活下去的证据。二十一世纪的今天，忙碌的生活早已让人们流失许多闲情，为了造就更好的生活条件，个体必须使劲思考自己，个体必须使劲做好自己，不然这个社会也不会正常运作。嗨 ... 说着说着，叹气声也不经意流了出来，话题太沉重了，如果这个世界真有恶魔的话，麻烦落下一颗陨石来纠正一切吧 ... 拜托了。

## 6.6 主动设计②

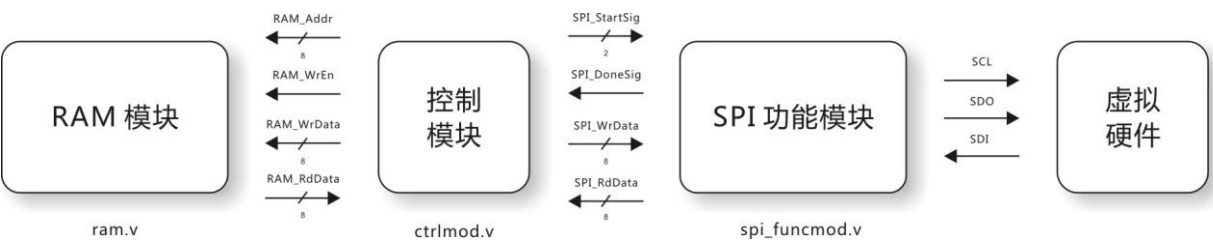


图 6.6.1 exp34 的建模图。

笔者在上一个小节为主动设计举例一个简单的例子，然而例子实在太简单了，为了继续刺激读者成长，这一回笔者稍微提高难度。如图 6.6.1 所示，那是 exp34 的建模图，模块之间除了更多信号以外，exp34 也多了一块虚拟硬件，读者是不是觉得很刺激呢？控制模块与 RAM 模块之间多了 RAM\_WrEn 信号还有 RAM\_WrData 信号，即表示控制模块不仅读取 RAM 的数据，它也为 RAM 模块写数据。

控制模块与 SPI 功能模块之间的变化，除了 SPI\_StartSig 信号多了 1 位之余，两者之间也增加了 SPI\_RdData 信号。至于 SPI 功能模块不再单单发送数据，它也要接收来至虚拟硬件反馈的数据。为此我们开始思考，读写 RAM 之前已经做过实验，所以没有什么难题。此外，控制模块与 SPI 功能模块之间也应用了问答信号，所以控制的问题也不大，不过 SPI 功能模块需要多添加一项读功能了。最后就是 exp34 多了虚拟硬件，亦即多了反馈输出，结果必须应用仿真模型③。

嗯，做好一切思考准备以后，我们就可以开工了。

*ram.v*

```
1. module ram
2. (
3.     input CLOCK,RESET,
4.     input RAM_WrEn,
5.     input [7:0] RAM_Addr,
6.     input [7:0] RAM_WrData,
7.     output [7:0] RAM_RdData
8. );
9.     reg [7:0] ram[255:0];
10.    reg [7:0] rData;
11.
12.    initial begin
13.        ram[1] = 8'b1100_0011;
14.        ram[2] = 8'b0000_0000;
15.    end
```

```

16.
17.     always @ (posedge CLOCK or negedge RESET)
18.         if( !RESET )
19.             rData <= 8'd0;
20.         else if( RAM_WrEn )
21.             ram[ RAM_Addr ] <= RAM_WrData;
22.         else
23.             rData <= ram[ RAM_Addr ];
24.
25.     assign RAM_RdData = rData;
26.
27. endmodule

```

ram 模块修改的地方不多，除了第 14 行的初始化以外，笔者将 ram[2]的空间清零，目的是用来储存虚拟硬件反馈回来的数据。

*spi\_funcmod.v*

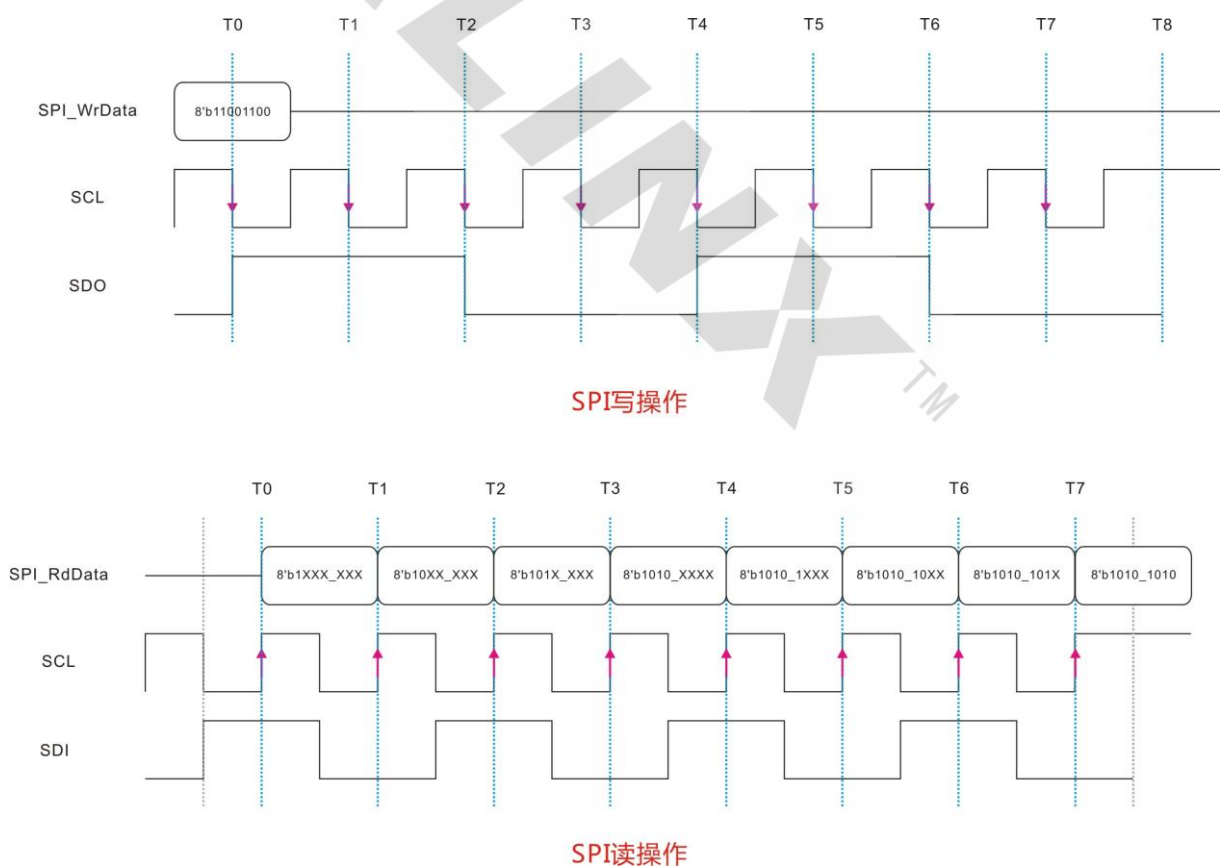


图 6.6.2 SPI 的读写时序。

笔者曾经说过，SPI 传输协议有主机（掌握 SCL 一方）与从机之分，而且 SPI 传输协议在 SCL 的下降沿设置数据，SCL 的上升沿锁存数据。如图 6.6.2 所示，上图是 SPI 写操



作，下图是 SPI 的读操作，期间 SDI 的输入数据是 8'b1010\_1010，因此每次 SCL 上升沿读取数据，SPI\_RdData 最高位道最低分持续发生变化，直至 T7 为止，SPI\_RdData 最终输出数据 8'b1010\_1010。

```

1.  module spi_funcmod
2.  (
3.      input CLOCK,RESET,
4.      input [1:0]SPI_StartSig,
5.      output SPI_DoneSig,
6.      input [7:0] SPI_WrData,
7.      output [7:0] SPI_RdData,
8.      output SCL,SDO,
9.      input SDI,
10.
11.      output [4:0]SQ_i
12. );
13.  parameter T1US = 8'd5; // (1/1Mhz) / (1/50Mhz) = 50; 50/25/5
14.
15.  reg [4:0]i;
16.  reg [7:0]C1;
17.  reg [7:0]rData;
18.  reg rSDO,rSCL;
19.  reg isDone;
20.
21.  always @ (posedge CLOCK or negedge RESET)
22.      if( !RESET )
23.          begin
24.              i <= 5'd0;
25.              C1 <= 8'd0;
26.              rData <= 8'd0;
27.              {rSDO, rSCL} <= 2'b01;
28.              isDone <= 1'b0;
29.          end
30.      else if( SPI_StartSig[1] )
31.          case( i )
32.
33.              0,2,4,6,8,10,12,14:
34.                  if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
35.                  else begin C1 <= C1 + 1'b1; rSCL <= 1'b0; rSDO <= SPI_WrData[7-(i>>1)]; end
36.
37.              1,3,5,7,9,11,13,15:
38.                  if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
39.                  else begin C1 <= C1 + 1'b1; rSCL <= 1'b1; end

```

```

40.
41.             16:
42.             begin isDone <= 1'b1; i <= i + 1'b1; end
43.
44.             17:
45.             begin isDone <= 1'b0; i <= 5'd0; end
46.
47.         endcase
48.     else if( SPI_StartSig[0] )
49.         case( i )
50.
51.             0,2,4,6,8,10,12,14:
52.             if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
53.             else begin C1 <= C1 + 1'b1; rSCL <= 1'b0; end
54.
55.             1,3,5,7,9,11,13,15:
56.             if( C1 == T1US -1) begin C1 <= 8'd0; i <= i + 1'b1; end
57.             else begin C1 <= C1 + 1'b1; rSCL <= 1'b1; rData[7-(i>>1)] <= SDI; end
58.
59.             16:
60.             begin isDone <= 1'b1; i <= i + 1'b1; end
61.
62.             17:
63.             begin isDone <= 1'b0; i <= 5'd0; end
64.
65.         endcase
66.
67.     assign SCL = rSCL;
68.     assign SDO = rSDO;
69.     assign SPI_RdData = rData;
70.     assign SPI_DoneSig = isDone;
71.     assign SQ_i = i;
72.
73. endmodule

```

exp34 的 spi 功能模块相较 exp33 的 spi 功能模块，它多了一个读操作，如代码行 48~65 所示。SPI\_StartSig[1]使能表示启动写操作（第 30 行），SPI\_StartSig[0]使能则表示启动读操作（第 48 行）。SPI 输出数据与读取数据最大的差别就是 SCL 的时钟沿而已，由于下降沿对读操作没有什么作用，因此步骤 0~14（偶数）只是单纯拉低 SCL 而已。

反之，上升沿对读操作来说却非常重要，步骤 1~15（奇数）除了拉高时钟以外，第 57 行的读取操作 rData[7-(i>>1)] <= SDI 表示从数据最高位开始。此外，exp34 拥有虚拟硬件，因此笔者将内部的指向信号 i 引出（第 71 行），以用来描述虚拟硬件的部分功能。

*spi\_funcmod.vt*

```
1.  `timescale 1 ps/ 1 ps
2.  module spi_funcmod_simulation();
3.
4.      reg CLOCK;
5.      reg RESET;
6.
7.      reg [1:0]SPI_StartSig;
8.      reg [7:0]SPI_WrData;
9.      wire SPI_DoneSig;
10.     wire [7:0]SPI_RdData;
11.     wire [4:0]SQ_i;
12.
13.     reg SDI;
14.     wire SCL,SDO;
15.
16.     /***/
17.
18.     initial
19.     begin
20.         RESET = 0; #10; RESET = 1;
21.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
22.     end
23.
24.     /***/
25.
26.     spi_funcmod U1
27.     (
28.         .CLOCK ( CLOCK ),
29.         .RESET( RESET ),
30.         .SPI_StartSig( SPI_StartSig ),
31.         .SPI_DoneSig( SPI_DoneSig ),
32.         .SPI_WrData( SPI_WrData ),
33.         .SPI_RdData( SPI_RdData ),
34.         .SCL ( SCL ),
35.         .SDO ( SDO ),
36.         .SDI( SDI ),
37.         .SQ_i( SQ_i )
38.     );
39.
40.     /***/
```

```

41.
42.     reg [3:0]i;
43.
44.     always @ ( posedge CLOCK or negedge RESET )
45.         if( !RESET )
46.             begin
47.                 i <= 4'd0;
48.                 SPI_StartSig <= 1'b0;
49.                 SPI_WrData <= 8'd0;
50.             end
51.         else
52.             case( i )
53.
54.                 0:
55.                     if( SPI_DoneSig ) begin SPI_StartSig[1] <= 1'b0; i <= i + 1'b1; end
56.                     else begin SPI_StartSig[1] <= 1'b1; SPI_WrData <= 8'b1001_1001; end
57.
58.                 1:
59.                     if( SPI_DoneSig ) begin SPI_StartSig[0] <= 1'b0; i <= i + 1'b1; end
60.                     else begin SPI_StartSig[0] <= 1'b1; end
61.
62.                 2:
63.                     i <= i;
64.
65.             endcase
66.
67.     /******
68.
69.     always @ ( posedge CLOCK or negedge RESET )
70.         if( !RESET )
71.             begin
72.                 SDI <= 1'b0;
73.             end
74.         else if( SPI_StartSig[0] )
75.             case( SQ_i )
76.
77.                 0: SDI = 1'b1;
78.                 2: SDI = 1'b0;
79.                 4: SDI = 1'b1;
80.                 6: SDI = 1'b0;
81.                 8: SDI = 1'b1;
82.                 10: SDI = 1'b0;
83.                 12: SDI = 1'b1;

```

```

84.             14: SDI = 1'b0;
85.
86.             endcase
87.
88. endmodule

```

上述代码是用来实现局部仿真，第 44~65 行是虚拟输入，第 69~86 行则是虚拟输出，它相较 exp33 多了虚拟输出这段激励内容。虚拟输入，除了步骤 0 除了使能仿真对象写入数据 8'b1001\_1001 以外，步骤 1 也使能仿真对象读取数据。期间，第 69~86 行的虚拟输出则描述虚拟硬件的部分功能，虚拟输出借用 SPI\_StartSig 信号之余，它也借用 SQ\_i 信号，期间反馈输出的数据是 8'b1010\_1010，而且反馈输出也是按照 SQ\_i 指向的过程作出反应。

SQ\_i 指向过程 0~14 ( 偶数 ) 分别是 SCL 信号的上升沿，根据 SPI 传输协议，SCL 上升沿则是设置数据。

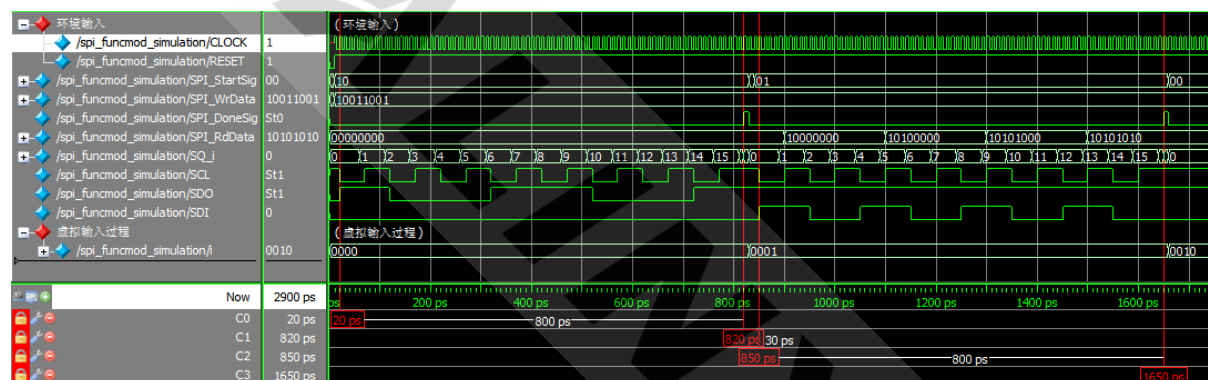


图 6.6.3 spi\_funcmod 的局部仿真结果。

图 6.6.3 是 spi\_funcmod 的局部仿真结果，光标 C0~C1 指向的范围是写操作，C2~C3 指向的范围是读操作。如图 6.6.3 所示，仿真对象的写数据是 8'b1001\_1001，虚拟输出反馈的数据则是 8'b1010\_1010。C3 指向的地方也是仿真对象完成读操作的时候，此刻请注意 SPI\_RdData 的结果，它是 8'b1010\_1010 ... 这个情况表示读操作，还有反馈输出的激励过程都顺利完成。

*ctrlmod.v*

```

1. module ctrlmod
2. (
3.     input CLOCK,RESET,
4.
5.     output [7:0]RAM_Addr,
6.     input [7:0]RAM_RdData,
7.     output RAM_WrEn,

```

```

8.     output [7:0]RAM_WrData,
9.
10.    output [1:0]SPI_StartSig,
11.    input SPI_DoneSig,
12.    output [7:0] SPI_WrData,
13.    input [7:0] SPI_RdData
14. );
15.    reg [3:0]i;
16.    reg [7:0]rAddr,rData;
17.    reg [1:0]isSPI;
18.    reg isRAM;
19.
20.    always @ (posedge CLOCK or negedge RESET)
21.        if( !RESET )
22.            begin
23.                i <= 4'd0;
24.                { rAddr, rData } <= { 8'd0,8'd0 };
25.                isSPI <= 2'b00;
26.                isRAM <= 1'b0;
27.            end
28.    else
29.        case( i )
30.
31.            0,1:
32.                begin rAddr <= 8'd1; i <= i + 1'b1; end
33.
34.            2:
35.                begin rData <= RAM_RdData; i <= i + 1'b1; end
36.
37.            3:
38.                if( SPI_DoneSig ) begin isSPI[1] <= 1'b0; i <= i + 1'b1; end
39.                else begin isSPI[1] <= 1'b1; end
40.
41.            4:
42.                if( SPI_DoneSig ) begin isSPI[0] <= 1'b0; i <= i + 1'b1; end
43.                else begin isSPI[0] <= 1'b1; end
44.
45.            5:
46.                begin rAddr <= 8'd2; isRAM <= 1'b1; i <= i + 1'b1; end
47.
48.            6:
49.                begin isRAM <= 1'b0; i <= i + 1'b1; end
50.

```

```

51.             7:
52.             i <= i;
53.
54.         endcase
55.
56.     assign RAM_Addr = rAddr;
57.     assign RAM_WrEn = isRAM;
58.     assign RAM_WrData = SPI_RdData;
59.     assign SPI_StartSig = isSPI;
60.     assign SPI_WrData = rData;
61.
62. endmodule

```

exp34 的控制模块相较 exp33 的控制模块,它则多了步骤 4 的 SPI 读操作(第 41~43 行),然后就是步骤 5~6 的 ram 写操作(第 45~48 行)。控制模块现在在步骤 0~2 从 ram 模块的地址 1 哪里读取数据,然后在步骤 3 将其送入 SPI 功能模块并且发送出去。时候,它又从 SPI 功能模块哪里读来数据,然后再步骤 5~6 将其写入 ram 模块的地址 2 当中。注意, RAM\_WrData 直接由 SPI\_RdData 驱动(第 58 行)。

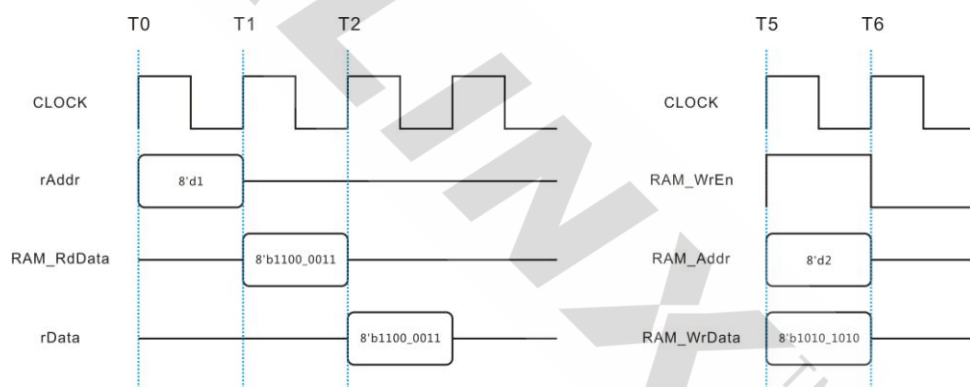


图 6.6.4 笔者脑补的时序图。

假设步骤 0~2, 还有步骤 5~6, 它们分别指向 T0~T2, T5~T6。如图 6.6.4 所示, 那是笔者脑补的时序图, 左图是从 ram 哪里读出数据, 右图则是将数据写入 ram 哪里。根据右图的现实结果, 控制模块在 T5 的时候, 将 RAM\_WrEn 拉高, 并且输出地址 8'd2, 还有数据 8'b1010\_1010, 然后 ram 则会在 T6 将数据 8'b1010\_1010 存入地址 8'd2。

#### exp34\_simulation.vt

```

1. `timescale 1 ps/ 1 ps
2. module exp34_simulation();
3.
4.     reg CLOCK;
5.     reg RESET;
6.

```



```

7.      /*****/
8.
9.      initial
10.     begin
11.         RESET = 0; #10; RESET = 1;
12.         CLOCK = 1; forever #5 CLOCK = ~CLOCK;
13.     end
14.
15.     /*****/
16.
17.     wire [7:0]RAM_Addr,RAM_RdData;
18.     wire RAM_WrEn;
19.     wire [7:0]RAM_WrData;
20.
21.     ram U1
22.     (
23.         .RAM_Addr ( RAM_Addr ),
24.         .CLOCK ( CLOCK ),
25.         .RESET( RESET ),
26.         .RAM_WrData( RAM_WrData ),
27.         .RAM_WrEn ( RAM_WrEn ),
28.         .RAM_RdData ( RAM_RdData )
29.     );
30.
31.     wire [1:0]SPI_StartSig;
32.     wire SPI_DoneSig;
33.     wire [7:0]SPI_WrData;
34.     wire [7:0]SPI_RdData;
35.
36.     ctrlmod U2
37.     (
38.         .CLOCK( CLOCK ),
39.         .RESET( RESET ),
40.         .RAM_Addr( RAM_Addr ),
41.         .RAM_RdData( RAM_RdData ),
42.         .RAM_WrEn( RAM_WrEn ),
43.         .RAM_WrData( RAM_WrData ),
44.         .SPI_StartSig( SPI_StartSig ),
45.         .SPI_DoneSig( SPI_DoneSig ),
46.         .SPI_WrData( SPI_WrData ),
47.         .SPI_RdData( SPI_RdData )
48.     );
49.

```

```

50.     reg SDI;
51.     wire [4:0]SQ_i;
52.
53.     spi_funcmod U3
54.     (
55.         .CLOCK ( CLOCK ),
56.         .RESET( RESET ),
57.         .SPI_StartSig( SPI_StartSig ),
58.         .SPI_DoneSig( SPI_DoneSig ),
59.         .SPI_WrData( SPI_WrData ),
60.         .SPI_RdData( SPI_RdData ),
61.         .SCL ( SCL ),
62.         .SDO ( SDO ),
63.         .SDI( SDI ),
64.         .SQ_i( SQ_i )
65.     );
66.
67.     /*****
68.
69.     always @ ( posedge CLOCK or negedge RESET )
70.         if( !RESET )
71.             begin
72.                 SDI <= 1'b0;
73.             end
74.         else if( SPI_StartSig[0] )
75.             case( SQ_i )
76.
77.                 0: SDI = 1'b1;
78.                 2: SDI = 1'b0;
79.                 4: SDI = 1'b1;
80.                 6: SDI = 1'b0;
81.                 8: SDI = 1'b1;
82.                 10: SDI = 1'b0;
83.                 12: SDI = 1'b1;
84.                 14: SDI = 1'b0;
85.
86.             endcase
87.
88.     endmodule

```

exp34 相较 exp33 之间的不同之处 除了 3 个模块之间组合连线以外( 第 17~65 行 ) ,exp34 也多了反馈输出 ( 第 69~86 行 ) , 其它的内容读者自己看着办吧。

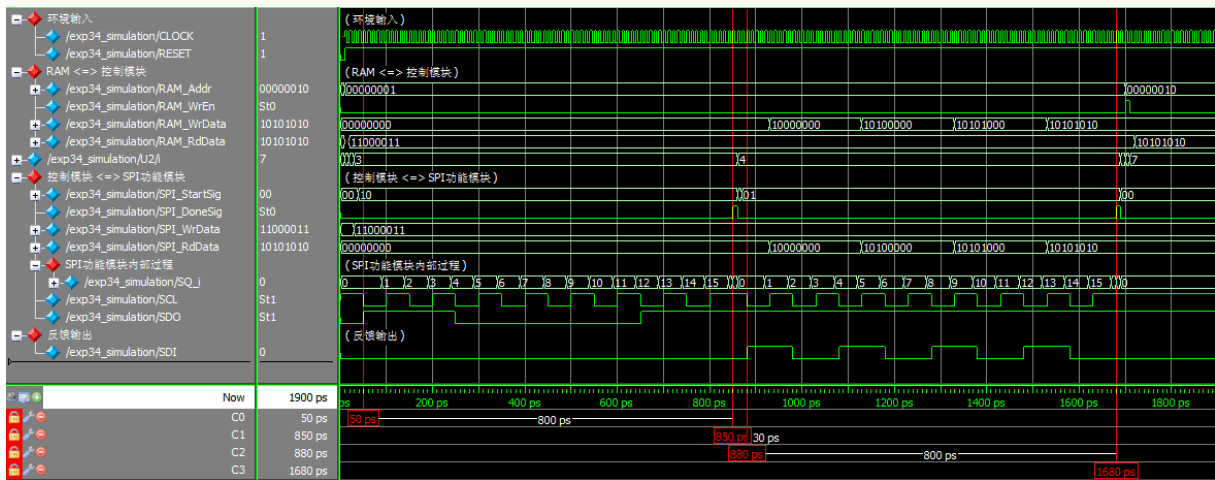


图 6.6.5 exp34 的仿真结果。

图 6.6.5 是 exp34 的仿真结果，初次见面的朋友可能会吓到，还以为遇见切糕。没错，exp34 的仿真结果，在某种程度上来说已经是如假包换的切糕。不过在此之前，我们已经做好许多准备了，所以这块切糕一点也不可怕。光标 C0~C1 指向的地方是 SPI 的写操作，光标 C2~C3 指向的地方则是 SPI 的读操作。

图 6.6.5 基本上是一种拼凑完毕的时序结果而已，全程内容笔者就不详细说了，反之让我们来解析一写小细节吧。U2/i 是用来指向控制模块的内部过程，SQ\_i 则是用来指向 spi 功能模块的过程，此外 SQ\_i 还有 SPI\_StartSig 也用来描述虚拟硬件。当控制模块执行读操作的时候（C2 指向的地方），SPI\_StartSig 的结果是 2'b01，虚拟输出也因此开始执行。

虚拟输出根据 SQ\_i 指向的过程，一步步为 SDO 拉高又拉低，结果输出 8'b1010\_1010 的串行数据。紧接着 C3 指向的地方正好是读操作的结束之际，控制模块接下来会将刚才读出的数据写入 ram 模块的地址 2 当中。

exp34 相较 exp33 虽然稍微刺激一点，然而采用主动设计之后 exp34 也不会难到那里去，因为再大的切糕我们也可以分成好几份消化。如果读者反问笔者，有没有闲情去脑补整个 exp34 的时序，笔者则会笑笑说不。主动设计本来是用来偷懒 ... 啊不！是，用作局部仿真造就整体仿真的设计技巧。笔者脑袋的处理能力不仅有限，而且吞吐量也不够，所以笔者只会执行局部仿真，或者脑补局部时序而已，其余任由想象力填充。

接着，让我们来好好思考一下，这个章节所学到的东西。如果我们按照传统手段去仿真 exp34，或许很多同学会感觉这是一件杀人不偿命的苦差事，笔者对此举四肢赞同。理由很简单，传统手段是一种硬塞硬啃的仿真方法，那些承受能力稍差的同学（笔者也是）一般都会支持不了，然后患上绝望的仿真癌。曾经何时笔者也饱受仿真癌的折磨，所幸有它打救笔者。

为此，笔者才考虑另外的仿真手法。首先是尽量过滤不必要的信息，然后再将整体有规

划地切分为数个个体，期间我们必须做好各种早期准备。接着，再借用自动思想，为个体留下强烈的局部印象，让它深深刻入那记忆之中。事后，我们便可以脑补这个，脑补那个了，然而要不要脑补整体的仿真内容，还是将无数的局部仿真内容结合起来，完全是读者的自由。相对之下，笔者比较喜欢玩拼图，将局部的仿真信息一块一块拼凑起来形成一幅整体的仿真信息。

如果读者感觉不安或者没有信心，读者也可以将脑补的时序图与是实际的时序图进行比较。读者在意哪里就比较哪里，为了避免反脑，读者千万别妄想自己可以吞下整副时序图哦，稍微偏食一点也没关系的。最后笔者奉劝道，做人做事最怕就是三心两意，如果读者选择主动思想，读者必须放手传统流派，让它成为过去，不再回头，然后成为潇洒又勇敢的男人。



## 6.7 仿真的决意

完成 exp33~34 的作业以后，笔者将其交给师兄，不一会师兄露出惊讶的神情，然后急忙呼唤另一位师兄来检查作业。师兄们在商讨期间，笔者用眼角偷瞄周围 ... 这里除了笔者以外，还有它班的同学，然而数量不超过 10 个人，想必它们也是那场噩梦的幸存者吧？它们的身上再也感觉不到常人的气场，应该说它们给人感觉越来越像师兄了，让人觉得非常不舒服。

笔者曾经听过一战期间，斯特勒秘密建立了一只特殊部队，这个部队专门聚集战场的死神。它们虽然怪异，不过它们都是生存率最高的士兵，也是传说的不死军队，那种异于常人的生存率也只能使用奇迹来形容而已。传统流派是不是也使用相似的方法聚集同伴呢？不知为何，笔者浑身忽然觉得恶寒起来。

突如其来“喂”的一声，打破笔者的胡思乱想，笔者立即将意识放在对方的身上。一位年逾 40 的男人正缓缓靠过来，那副似曾相识的面貌，不禁让笔者回忆那起不愉快经历。没错，那位男人正是资深师兄。无形的压力驱使身体站了起来，笔者注视它，它也注视笔者，然而在场所有视线也聚集在这里 ... 就这样，大眼瞪小眼一段时间以后，命运的轮盘开始转动了。

“我的师弟，首先恭喜通过考验，老夫实在非常高兴”，资深师兄客套道。

“是 ... 是吗？”，笔者板着脸答道。

“老夫怎么想都不明白，我的师弟是如何活下来呢？”，资深师兄突然问道。

“ ... 啊！？”，笔者哑然道。

根据笔者的认识，通过考验的人只有啃掉切糕这条选择而已，然而笔者却被切糕啃掉，不过又奇迹生还下来，然后将切糕解决掉。那种状况对传统流派而言，根本毫无先例的，师兄的问题毫无疑问是在质疑笔者“为什么像你这种弱小的蠢货，不仅活了下来，还漂亮解决切糕呢？”很明显，师兄正在怀疑笔者在考验期间出猫。这个问题好比导火线般，不小心将笔者的按钮启动，压抑许久的感情终于爆发了！

“啊哈哈哈哈！啊哈哈哈哈！”，笔者狂笑道。

“传统流派嗨 ... 你到底有多丑陋？”，笔者对着众人叫喊道。

“可恶，小鬼你说谁丑陋！！”，师兄 A 愤怒道。

“难道还有谁吗？”，笔者故意加重语气道。

“蠢货！是不是活得不耐烦了！”，师兄 B 愤怒道。

几名师兄因为经受不了笔者的羞辱，激动地破口大骂道，其中一名师兄还举起拳头冲向笔者。没错“暴力”这就是它们的真面目，来吧！笔者才不怕暴力！此刻，资深师兄忽然举起右手将对方拦住 ... 有一瞬，笔者在隐约听见轮盘的转动声，那股声音好似下定决意一般，越滚越慢，越滚越慢 ...

“丑陋嘛？呵呵，真有意思！”，资深师兄笑道。

“我的师弟，为什么那样觉得？”，资深师兄又继续道。  
“哼！只要细心观察，任谁都会觉得！”，笔者反驳道。  
“有谁这样觉得吗？是你？是你？还是你？”，资深师兄有意将矛头指向其他师兄。  
“没 ... 没有！”，师兄 A 口吃道。  
“不，不是那样的！”，师兄 B 心虚道。  
“那个 ...！那个”，师兄 C 颤栗道。  
“看吧 ... 我的师弟，没有人这样认为”，资深师兄阴笑道。

可恶，这不就是赤裸裸的恐吓吗？它们都畏惧资深师兄，这种情况好比笔者恐惧切糕一样，无法反抗的绝对压力。想着想着，笔者忽然觉得身体有点不停使唤 ... 不，那是颤抖，笔者打从心底也在畏惧。灵魂的左天使说：不服从就会死！右恶魔则说：不屈服，去报复！此刻，理智还有感情正在发生激烈的斗争 ... 头好疼，感觉快要爆炸了，拜托不要再吵了！笔者紧闭双眼，使劲压住太阳穴祈求着。

“孩子，别忘了，白骆驼始终在身边”

温馨的声音忽然响起，笔者不经回忆在梦中出现的它 ... 吵闹声渐渐消去，黑暗中笔者也看见同辈们的笑颜，它们不停为笔者加油打气，告知笔者不能退缩！好温柔呀，好窝心呀，一股前所未有的勇气正逐渐填补心中的懦弱，白骆驼的身影引导笔者踏出人生的决意。此刻，命运的轮盘已经停止转动。

“勇气，已经足够了 ... ”，笔者小声道。  
“怎么了，我的师弟？”，资深师兄问道。  
“各位师兄，谢谢大家一直以来的暴力！小弟决定离开！”，笔者决意道。  
“什么！？”，资深师兄惊讶道。  
“臭小鬼！想走人吗！？”，师兄 A 叫喊道。  
“想来就来，想走就走，以为这里是慈善堂吗？”，师兄 B 叫喊道。  
“不知道也不想知道！总之，各位多保重！”，笔者坚持道。  
“可恶，别自说自话 ... ”，师兄 C 叫喊道。

笔者无视师兄们的叫喊，转身后便立即离开 ... 吧嗒，吧嗒，鞋子与地板的摩擦声也融入背后的叫喊声，然而那些杂音，随着距离的拉长也逐渐消去。笔者抵达门口之际，背后传来有力的叫声，如果没听错的话，那是资深师兄的声音，从刚才开始它就一直保持沉默。

“慢着，我的师弟 ... ”，资深师兄阻止道。  
“.....”，笔者用沉默回应道。  
“再踏出门口一步，您再也不是师弟，而是永远被追缉的敌人！”，资深师兄大喊道。  
“最后一刻还在恐吓对方，传统流派你究竟有多丑陋！”，笔者小声嘀咕道。

笔者头也不回，顺着命运之风向前一方通行，背后仅留下恶意的视线 ...

---

不知不觉，笔者又来到山崖边，笔者顺着崖口正缓缓前进着，然而这次不是寻求解脱才来到这里。笔者在崖口一角忽然停下脚步，百感交集的感情也随之涌上心头 ... 这个地方曾是命运转折点，也是一处契机之地，笔者曾经“绝望”来到这里；笔者也曾经“希望”离开这里；然而，现在又因为回忆来到这里 ...

笔者用怀念的眼光注视远处，哪里曾是解脱的天堂，哪里也曾是希望之光升起的地方。如今仔细一看，哪里不过是一处毫无止境的地平线而已。人真是奇怪的生物，同样的事情会因为不同的心情看到不同的结果，笔者不禁发出自嘲的“呵呵”声。再享受一会风景吧 ... 笔者自言自语道。

时间宛如蒸汽般消失在不知不觉之中，眼见夕阳已经西下，鸟儿也不断发出“吱吱”的归巢声，示意笔者“离开的时候”到了。走吧！笔者带着决意的步伐往家乡出发去。笔者意识到，从今以后，前方会有数之不尽的切糕挡道，背后也有暴力的家伙在追缉。不过，笔者也不是独自一个人默默上路，因为看不见的远处，它无时无刻都在守候笔者，所以笔者不害怕也不寂寞。

笔者认为，盲从才是最可怕的东西，我们宛如丧尸般失去自我，任由别人牵动鼻子前进，不管对方是传统流派还是仿真，结果就像温水煮青蛙，死在不知不觉当中。在未知的土地上，独立又主动的思考非常重要，因为那是唯一保护自己的工具 ... 建模也好，仿真也好，它们都是未知的土地，然而学习就是在未知的土地上冒险。

“读者，您也是不是这样认为呢？”



总结：

- 终于写完了，这个章节的内容比较单调，除了讨论不可仿真模块以外，就是经过综合练习认识主动设计的优劣之处。青春是宝贵的东西，我们不应该任由仿真蹉跎，然而蹉跎青春的三大杀手，它们就是超傻模块，超烦模块，还有超乱模块等不可仿真对象。真可怕，真可怕！

超傻模块物以其名一样，是再也简单不了的家伙，它不仅用时少，而且内容页也非常单纯，基本上都是望上一眼便立即知晓它的用意。根据主动思想，越是简单模块，模块内容越清晰，作为一块个体而言，它已经拥有非常强烈的特征印象，所以我们用不着特意仿真也没问题。但是我们无法阻止寂寞的家伙去仿真它们。

除了超傻模块以外，也有超烦模块这个不可仿真对象。超烦模块并不表示模块它会吱吱喳喳说个不停 ... 换之，它会故意让我们等候，直至我们感觉不耐烦。千万别认为超烦模块是可爱的女人，实际等候起来会产生无尽的破坏力！试问各位绅士，大伙已经浪费多少时间在等候异性呢？超烦模块会拉长 wave 界面，吃尽内存空间，拖慢计算器，让仿真变成更加耗时。

最后一位不可仿真对象就是超乱模块，超乱意指内容不明的模块内容，一般是指没有结构的模块，还有官方插件模块。官方插件模块不是不可仿真，而是仿真起来超级麻烦，因为官方插件模块是隐藏内容，所以我们必须经过各种途径去摸索才能知晓模块内容。模块如果隐藏内容，其一就是驾驭的问题，其二就是阻碍仿真信息的解析工作，因为仿真对象未能解读。

最后的综合实验基本上都是演示主动设计的使用方法。笔者需要强调，主动设计不是东西，主动设计更像技巧这种无形的东西。主动设计是一种讲求个体造就整体的设计技巧，它基于主动思想，相比整体情况它更在乎个体的状况。主动设计有许多内容，然而较为注目就是局部仿真，想象力仿真，还有想象力填充等奇怪的东西。

读者必须知道 modelsim 等仿真工具始终是它人的眼睛，借用它人的力量比哭死爹娘更困难，而且也非常被动 ... 为此，笔者才会尝试运用想象力去执行仿真。期间，读者可能会觉得可笑，认为笔者是不是妄想过头了？不管怎么样，主动设计正逐渐成为笔者的建模，还有仿真习惯 ... 像笔者这样不能一口吞掉切糕的弱小家伙，主动设计会将切糕平均划分以后，再一口一口吃掉。

最后的最后，笔者也写上自己的故事，虽然内容经过武侠化还有夸张化，不过感情还有决意却是如假包换的真东西。这种感觉好比独行侠游走江湖一般，为了明天它必须步步为营，还有拥有一份不可动摇的自我。

、

## 后语

这本书写完以后，笔者浑身都有一种飘飘然的快感，因为那些长期累积在心理深处的问题终于得到解放，感觉除了爽还是爽。当然，笔者不会为了这么不起眼的私人理由才编辑这本书。仿真是什么？这个问题一直以来都是笔者心中的一根刺，无时无刻折磨笔者，让笔者寝食难安。

仿真对常规而言，就是炒饭上面的装饰，或者炸鸡旁边的辣椒酱等，仅是一些跑龙套的角色。极端点说，常规产考书根本不把仿真放在心上，要么就是只言片语的解释，要么就是一笑而过。如果它们那么心不甘情不愿解释仿真，它们应该不碰为好。笔者一直以来都非常讨厌那些半吊子做事不尽心力的家伙，笔者认为这是非常不负责任的行为，后果往往都会害人害己。

笔者于是干脆用自己的认识还有方法去了解仿真，解释仿真。故事中，有一位以笔者为原型的主人公，它为了学习仿真离乡背井来到陌生的土地，然而主人公除了加入传统流派以外便别无选择（传统流派用来比喻常规认识），主人公后来理解常规存在许多无理的行为。对此，主人公只能感到失望和愤怒而已。

然而，它又出现了 ... 不过不是指导 Verilog 而是指导仿真。它为主人公指点迷津之余，也为主人公传授一个不可思议的思想，然后主人公重拾希望，再度鼓起勇气挑战仿真。笔者认为故事最为精彩的情节除了，主人公它과의对话以外，还有主人公与大夫之间的对话，内容仅是一些常规参考书未曾涉及的细节。

不管怎么样，整本书都是围绕仿真而展开，内容从未跑题过。写完这本书以后，笔者自己也不禁感叹道，原来仿真还有那么多不为人知的一面，比起理解原理，理解细节更加容易掌握它。事后笔者也怀疑自己，这本书究竟是不是笔者亲手写的？或者被附体什么的。嗨！可能是连续几天熬夜的关系，脑袋也开始胡思乱想了，再不休息就要发神经。

笔者放下敲击键盘的十指，然后缓缓望向窗外，满月已经高高挂在没有繁星的夜空。最近几个月，常常抬头不见星星，可能是空气污染的关系吧，人类真是不懂珍惜地球，这点好比传统流派般，理性总是被短浅的利益遮住。好了，笔者也是时候休息了，结束之前笔者需要补充道 ... 这本书虽然还有许多不如意的地方，但是用来认识仿真，笔者自认内容已经足够了，最后希望这本书可以帮助读者在那个世界（仿真）生还下去，找到那支唯一的可能性，然后抵达命运之石门。