

Verilog HDL—— 那些事儿 时序篇 v2

黑金FPGA开发板配套教程



黑金动力社区

www.heijin.org

书语

建模不是 Verilog HDL 语言的所有，建模只是使用 Verilog HDL 语言建立一个“像模像样”的“形状”而已。这个“形状”实际上是很粗糙的，还没有经过任何深入的分析。但是我们不可以小看这个“粗糙的形状”，如果没有这个“粗糙的形状”模块的设计根本无法完成。笔者在《Verilog HDL 那些事儿 - 建模篇》的结束语中有这样讲过：“建模是一个粗糙的东西，它还可以继续细化”。

“细化”顾名思义就是进入模块的深层进行分析和优化（如果有需要调试的话）的工作。但是前提，我们必须“更深入 Verilog HDL 语言的世界”才能有效的“细化”模块。这一本起名为《Verilog HDL 那些事儿-时序篇》的笔记分别有两个部分，上半部分和下半部分。上半部分是“步骤和时钟”；下半部分是“综合和仿真”。

“步骤和时钟”主要是深入讨论“步骤”和“时钟”在模块上的作用。宏观上“步骤”是模块执行的“拍子”，“时钟”是模块的“心跳”。微观上“步骤”是模块“操作的过程|状态”，“时钟”是模块“消耗的最小单位”。其实“步骤和时钟”它们是形影不离的兄弟，有“步骤”出现的地方，就有“时钟”的故事，为什么笔者会如此注重“步骤”和“时钟”呢？

当某个模块要完成更多工作的时候，传统的状态机会使得模块的内部臃肿和模块的表达能力下降等问题（这不是笔者的一厢情愿的看法，而是众多初学者都会遇见的问题）。一旦我们用“步骤”来取代“状态机”，那么我们就可以实现如“仿顺序操作”等更多花样的建模技巧来支持设计。“步骤”的优点不仅只是方便了建模的工作，而且“步骤”也有显性指示模块的操作过程和状态。这些好处对模块的“细化”起到很大的帮助。

关于“时钟”它和另一个重点有莫大的联系，就是“模块的沟通”。虽然说“时钟”是“模块最小的消耗单位”，模块之间如果发生了“沟通失误”，这些问题很多时候是模块之间的“沟通”因为慢了一个“时钟”或者快了一个“时钟”而引起的。低级建模是一个多模块的建模，自然而然笔者会非常的重视。“沟通失误”不只是会发生在模块的外部，而且也会发生在模块的内部。了解“时钟”能最大程度的分析模块和“细化”模块。

“综合和仿真”主要是把“综合”和“仿真”放在同一个平台来学习。许多初学者喜欢把“综合”和“仿真”看成两个平台的东西，如一个常见的观点，很多朋友都会认为“建模是用综合；仿真用验证”。这个观点不是不正确，只是有点遗憾而已。当我们把“综合”和“仿真”拆开为两个平台，模块“细化”的可能性不但会降低不少，此外还会对 Verilog HDL 语言的学习带来不少难题。

当我们尝试把这两个东西放在同一个平台上，重新思考，我们会发现到，用在“建模”上的“一套思想”也适合用在“仿真”上。如果用傻瓜的话来说，我们知道“建模”的工作是针对某个资源然后去描述它的形状，最终的目的还是要下载到“现实的环境”中。然而“仿真”比起这个“现实的环境”，它是一个“理想”的“虚拟环境”，在这个“理想的虚拟环境里”不存在任何物理的问题，而且也充满着任何可能性。你要什么输入都

可以创造，模块的任何输出都是显性而且可见的。只要我们明白了这个简单的道理，“建模”和“仿真”的关系是多么“亲近”的，它们的区别只是“在不同的环境执行而已”。

初学者往往都会觉得“仿真”最大的难题就是“如何编辑激励文件”。在这里如果用笔者自己一套的思路重新定义“仿真”的话（从笔者的角度去看“仿真”）。“激励”就是这个整个仿真的执行过程而已，“如何编辑激励文件”等价于“如何安排仿真过程”。在这个时候，建模技巧就会帮到很多大忙，我们可以基于综合语言去编辑这个仿真过程（激励文件）。

当然，“仿真”在这本笔记里的要表达的是“以显性的方式去观察模块的输出，从而以最大程度去细化模块”，然而“如何透过仿真的波形图去执行对模块的优化和调试”就是这本笔记的重点内容。学会编辑激励文件，充其量只是为了让模块达到“预期的输出效果”而已。如果要读懂隐藏在波形图中那些信息，并且用在调试和优化上，那么就必须掌握好 Verilog HDL 语言一定的基础。

前言

在完成《Verilog HDL 那些事儿 - 建模篇》之后，笔者老是觉得物有所不足，所以笔者继续着手了第二篇的《Verilog HDL 那些事儿 - 时序篇》。就如同书语所说的那样，“建模过后的模块是很粗糙，所以我们必须细化它” - 是这一本笔记的主要内容。

为什么这一本笔记要名为“时序篇”呢？

“时序”最容易联想到就是“时序图”，亦即模块的输出。换句话说“时序”是模块执行过程的显性记录。一般在仿真上，模块的时序图都是在理想状态下（没有任何物理上的问题）产生的。时序图里边包含了模块最清洁的执行记录。这些信息对于“细化”模块来说有很大的帮助。然而影响着这些时序就是 Verilog HDL 语言本身。

很多时候，虽然低级建模（建模技巧）已经可以帮助我们完成许多模块设计上的要求，但是低级建模始终是属于“建模”的层次，亦即“塑造”模块一个大概的形状，而且是粗糙的东西而已。这粗糙的东西，效果能不能发挥完全？我们需要经过“细化”它才知道结果。

要完成“细化”的过程一点也不可以马虎。早期的建模必须留下可以“细化”的种子。此外我们还要往模块更深入去了解它，去分析它，如果有模块有问题就调试它。这全部的工作要求，更进一步挑战我们对 Verilog HDL 语言的认识和掌握的程度。有关这一点，再也不是：了解 Verilog HDL 语言相关的语法如何使用？如何使用 Verilog HDL 语言建立一个模块？等这些 Verilog HDL 语言“外观的单纯工作”，而是“深入分析”模块执行时的“内部细节”。关于模块一切的一切过程，我们只能在“时序图”上了解而已。这就是这本笔记命名的初衷。

笔记内容可以说是五花八门：有算法，有建模技巧，有建模，有编辑激励文件，有仿真，有调试……等一大堆的东西。“算法”方面有比较常用的乘法算法和除法算法（算法作为深入理解步骤和时钟的帮手）。“建模技巧”方面有“流水操作”的建模方法。“建模”方面有同步 FIFO …… 等等等。说实话，笔者也觉得笔记的内容有点像“菜市”。

当然，笔记的所有内容都是围绕笔记的重点展开的，和《Verilog HDL 那些事儿-建模篇》一样，每一篇的联系性都很强，谁少了谁都不行。嗯！关于笔记的下半部分“综合和仿真”，笔记在开始写之前笔者考虑了很多问题。在这里，笔者使用了自己另一套的想法重新定义仿真，这方法完全是颠覆网络上一套常用的仿真方法，感觉上笔者在干唱反调的工作 … ⊙_ ⊙b 汗

这也是笔者最担心的事情，笔者怕遭到围观，被淹死在口水中 …… (╯ 3╰)！虽然如此，但是为了使学习有更多选择，故笔者才决定写下来。

这一本笔记的初衷是为了完善《Verilog HDL 那些事儿》系列的笔记。比起“建模篇”，“时序篇”这一本笔记的内容确实是少了不少，但是量少不代表质少。和“建模篇”相

比的话“时序篇”的内容难度事实上是高了不少。“时序篇”和“建模篇”同是一样，都是 Verilog HDL 语言的核心部分，所以笔者加重马力去维护笔记的内容。学习“时序篇”不像学习“建模篇”那样，可以一边学习一边喝牛奶那样轻松，需要用更多的脑力去思考笔记每一章节的重点。所以呀，记得学习勿太过“蜻蜓点水，点过就算”，不然的话就对不起自己了。

akuei2 23-01-2011

目录

书语	02
前言	04
目录	06
软件预备知识	09

上半部分：步骤和时钟

第一章 整数乘法器	18
1.01 整数的概念	18
1.02 传统乘法的概念	20
实验一：传统乘法器	21
1.03 传统乘法器的改进	26
实验二：传统乘法器改进	26
1.04 补码君存在的意义	30
1.05 BOOTH 算法乘法器	31
实验三：传统乘法器改进	34
1.06 笔者情有独钟的步骤 I	39
1.07 BOOTH 算法乘法器的改进	43
实验四：BOOTH 算法乘法器改进	44
1.08 LUT 乘法器	49
实验五：基于 QUARTER SQUARE 的查表乘法器	52
1.09 MODIFIED BOOTH 算法乘法器	65
实验六：MODIFIED BOOTH 乘法器	67
1.10 MODIFIED BOOTH 乘法器 ·改	73
实验七：MODIFIED BOOTH 乘法器 ·改	74
总结	81
第二章 整数除法器	82
2.01 传统的除法器	82
实验八：传统除法器	83

2.02 循环型除法器	88
实验九：传统乘法器改进	90
2.03 循环除法运算的原理	96
实验十：从原理到实现的循环除法器	100
总结	106

第三章 流水操作和建模 107

3.01 流水操作的概念	107
3.02 仿顺序操作向流水操作的转换	109
实验十一：流水式查表乘法器	111
3.03 流水操作和建模	118
实验十二：建模过后的流水式查表乘法器	119
3.04 有包袱的流水操作	125
实验十三：流水式 BOOTH 乘法器	125
实验十四：建模过后的流水式 BOOTH 乘法器	136
3.05 流水操作直接建模	143
实验十五：流水式循环除法器	143
3.06 当不同操作步骤的流水模块并连的时候	151
实验十六：移位寄存器延长工作还威武了	152
总结	156

第四章 模块的沟通 157

4.01 探讨 START_SIG 和 DONE_SIG 的协调性	157
4.02 同步 FIFO	160
实验十七：同步 FIFO	165
4.03 适合同步 FIFO 的控制信号	176
实验十八：同步 FIFO 改进	176
4.04 再建 接口建模	185
实验十九：乘法器接口	185
4.05 混种建模的可能性	196
实验二十：混种建模	199
总结	210

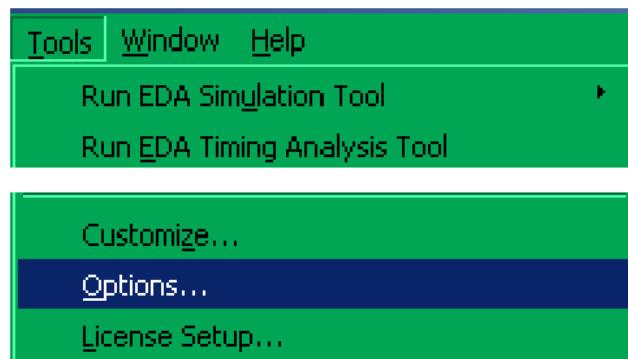
下半部分：综合和仿真

第五章 仿真前的故事	212
5.01 我眼中的仿真	212
5.02 激励的故事	214
5.03 仿真的虚拟环境	216
5.04 综合和仿真	217
总结	219
第六章 刺激和激励过程	220
6.01 精密计数	220
实验二十一：仿真定时器	220
6.02 刺激的各种输入	228
实验二十二之一：虚拟按键	228
实验二十二之二：仿真按键消抖模块	237
实验二十三：PS2 模块仿真	244
6.03 模块相互刺激	249
实验二十四之一：仿真串口发送模块	249
实验二十四之二：仿真串口接收模块	253
6.04 麻烦的 IO 口仿真	259
实验二十五：仿真带有 IO 的模块	260
总结	273
第七章 反应和调试过程	274
7.01 输出的珍贵信息	274
实验二十六：优化 VGA 的同步模块	274
7.02 迟了一步的数据	285
实验二十七：VGA 模块仿真	285
7.03 即时结果和非即时结果	296
实验二十八：即时结果的需要	296
7.04 波形图在我的脑海中	309
总结	314

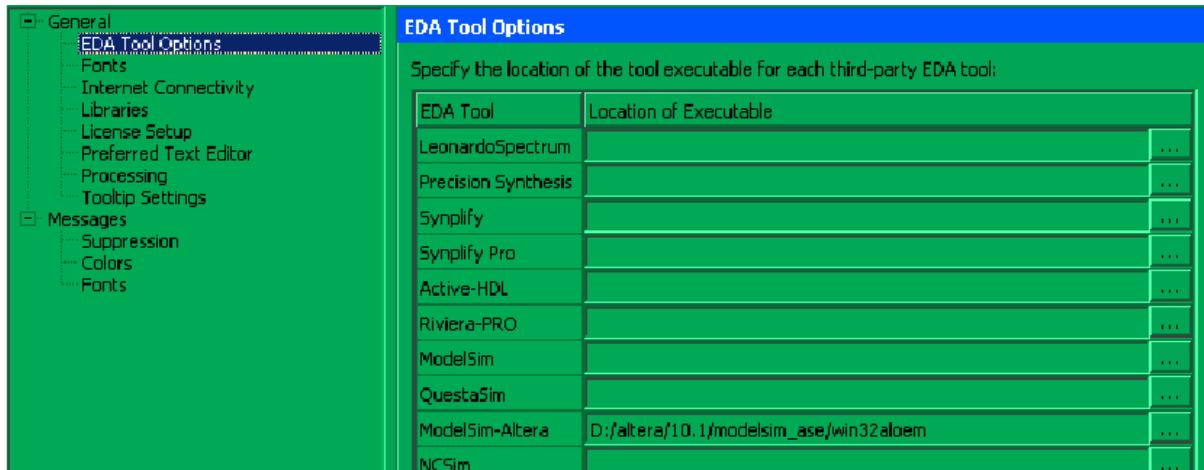
软件预备知识

笔记从第一章开始，实验的内容都是以仿真为主（借助仿真观察输出）。所以读者们必须懂得仿真软件一些简单的配置和使用，在这里我们会用到的软件是 Quartus II 和 Modelsim - Altera（版本任意）。

选择 Modelsim-Altera 的默认路径



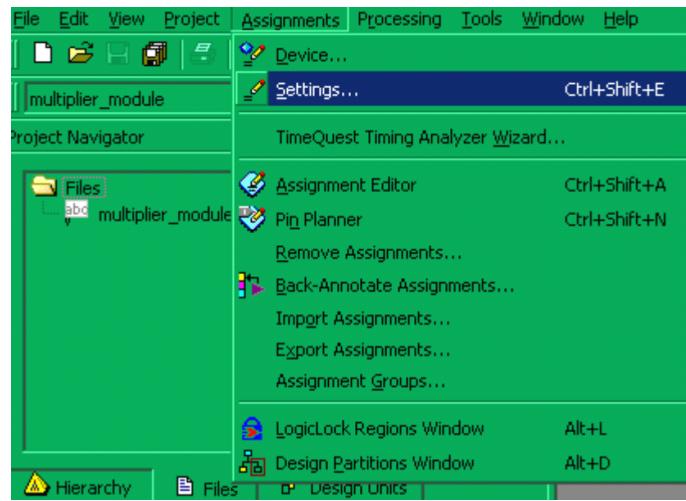
先打开 Quartus II 然后点击 Tools 里的 Options 。



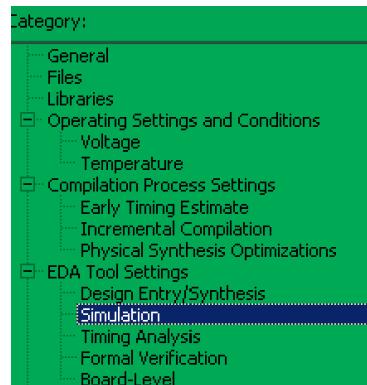
在 General 电子 EDA Tool Options ，然后在 ModelSim-Altera 哪里选择 Modelsim-Altera 的默认路径，如笔者的是 D:/...../win32aloem 。在 10.1 之前的版本路径都是自动配置。但是在版本 10.1 和之后，路径要手动配置了(〒 o 〒)。

项目的预用仿真软件

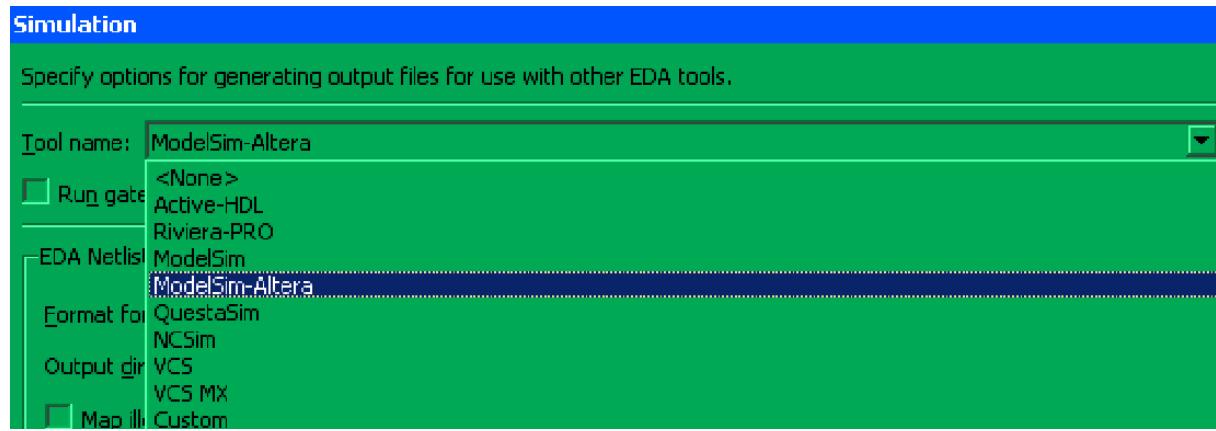
当读者建立一个新的项目以后，如果该项目需要仿真，读者必须为这个项目选择预用的仿真软件。在这里我们的预用仿真软件是 Modelsim-Altera 。



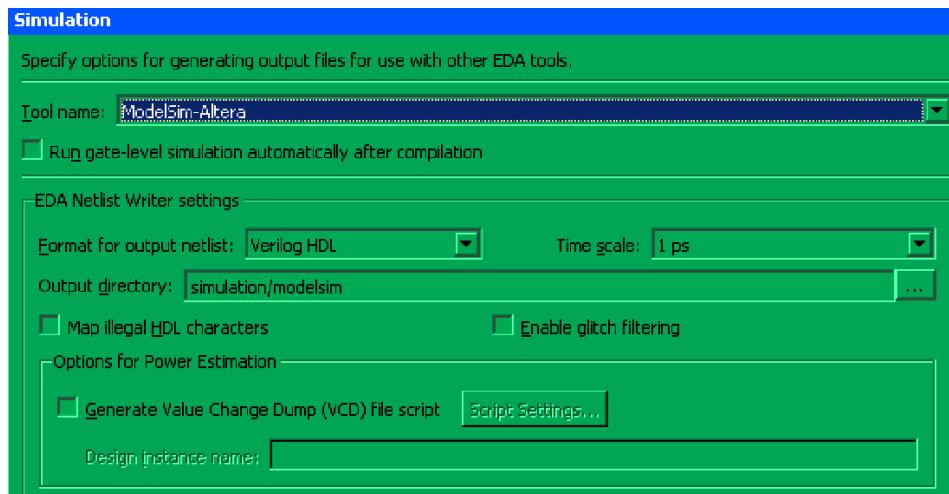
我们以实验一为实例，当打开或者建立某个项目，点击 Settings 。



在 Category 选 Simulation 。



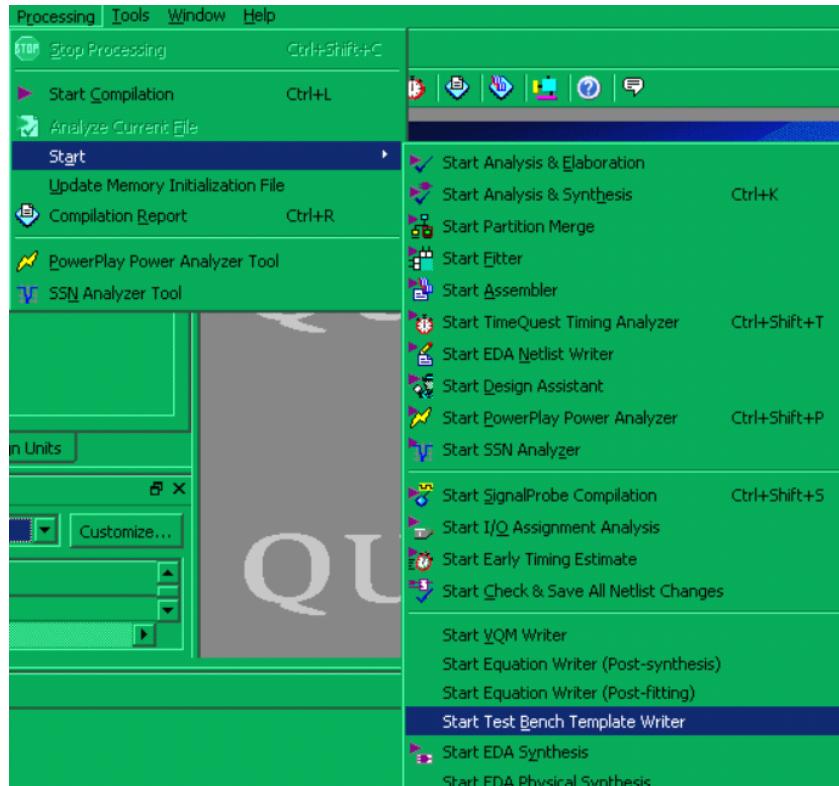
在 Tool name 选在 ModelSim-Altera 。



选择 Tool name 后。 Format for output netlist 选择自己熟悉的 HDL 语言，在这里笔者选择 Verilog HDL 语言。 Time scale （时间最小单位 | 时间刻度）目前先随便填上 1ps 。然而 output directory 的默认选项是 simulation / modelsim。

生成 Testbench 模板

有一些懒人如笔者，常常喜欢直接生成 Testbench 的模板，方便 .vt 文件的编辑。



生成 testbench 模板的步骤如上。要生成 testbench 模板的前提条件是为项目选择御用的仿真软件，然后模块必须编译成功。Testbench 模板生成后的默认路径如下：

项目目录 \simulation\modelsim\ 项目名 .vt

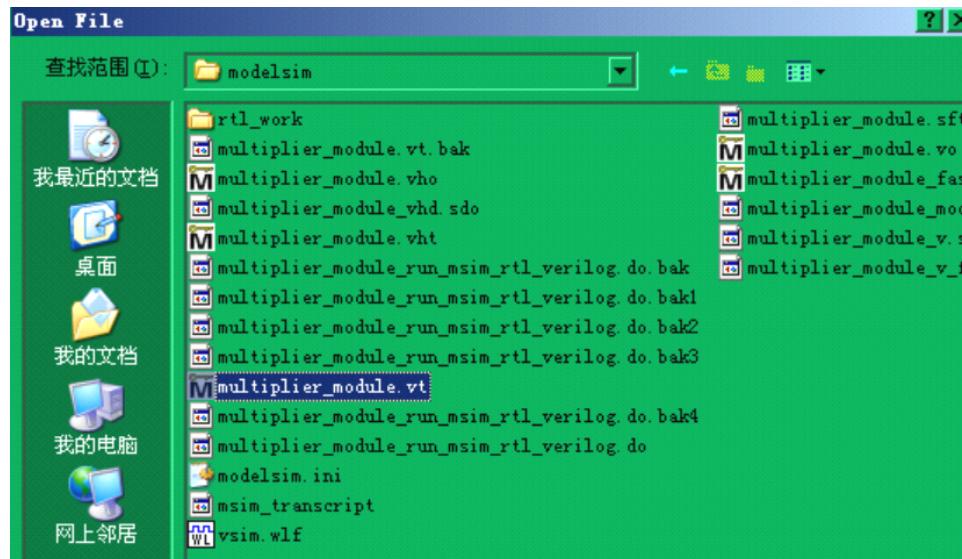
为项目添加 Testbench 文件

我们知道，如果要仿真某个项目，就要添加 testbench 文件，我们以实验一为例：



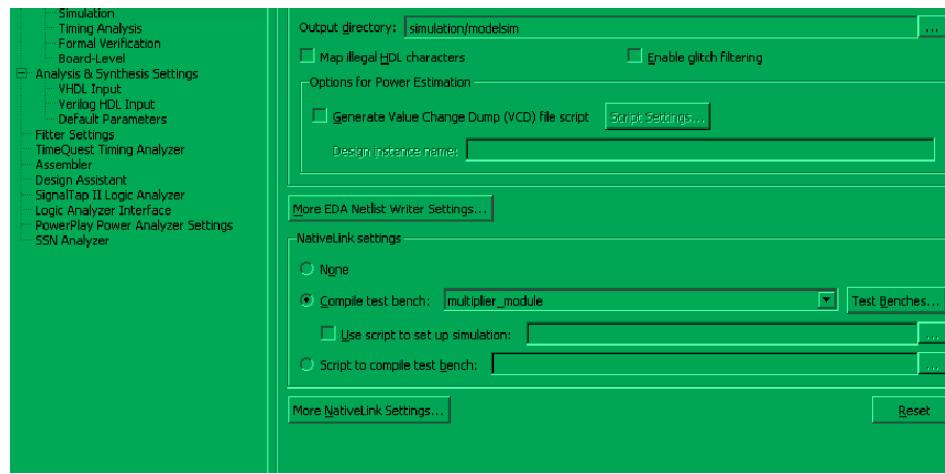
```
1  `timescale 1 ps/ 1 ps
2  module multiplier_module_simulation();
3
4      reg CLK;
5      reg RSTn;
6
7      reg Start_Sig;
8      reg [7:0] Multiplicand;
9      reg [7:0] Multiplier;
10
11     wire Done_Sig;
12     wire [15:0] Product;
13
14     // *****
15
16     initial
17     begin
```

上图是实验一的 testbench 文件。

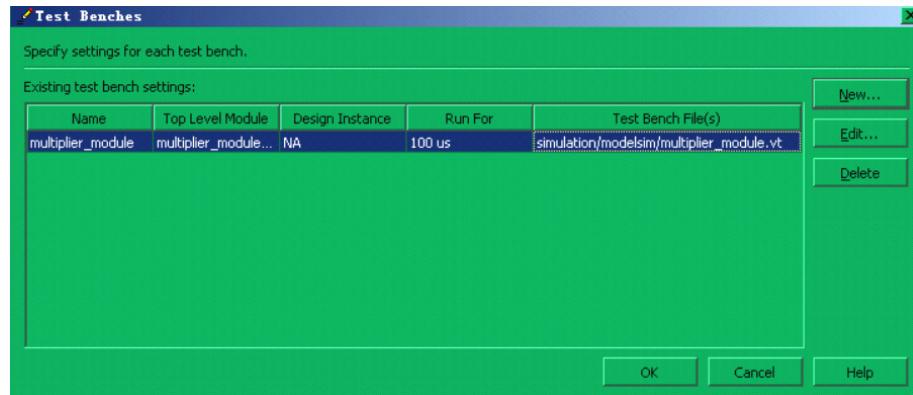


然而它的路径名是：

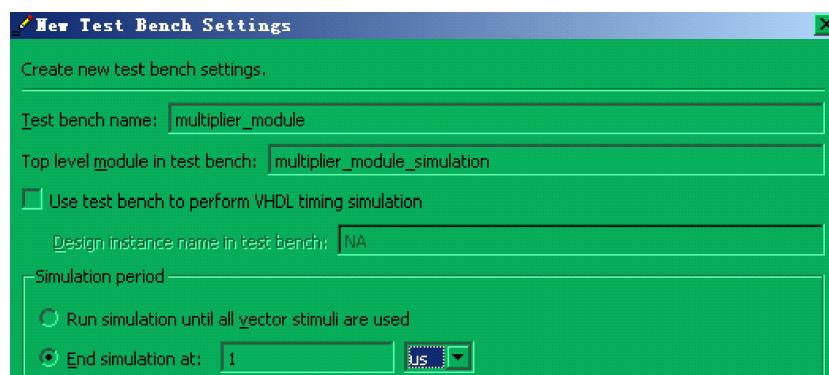
Experiment01\simulation\modelsim\multiplier_module.vt



接下来，我们要为实验一的项目添加 .vt 文件。先打开 Setting 中的 Simulation ， 然后在选择 Compile test bench 。接下来点击 Test Benches ...



然后就会跳出如上的窗口，加下来的工作就是点击 New ...



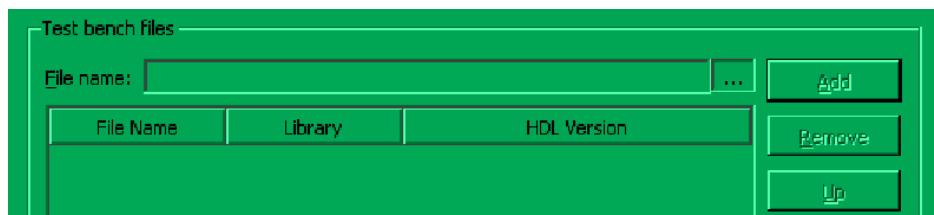
```

1  `timescale 1 ps/ 1 ps
2  module multiplier_module_simulation();
3

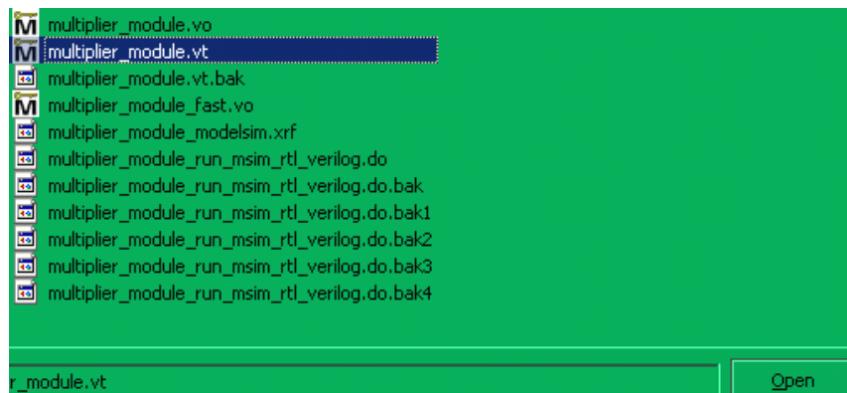
```

这个窗口是用来输入 testbench 的资料和选择 testbench 的路径。 Test bench name 是 testbench 文件的名字，如实验一的 testbench 文件名是 multiplier_module.vt ； Top level module in test bench 是 testbench 文件的顶层模块名，在这里是 mulitplier_module_simulation。

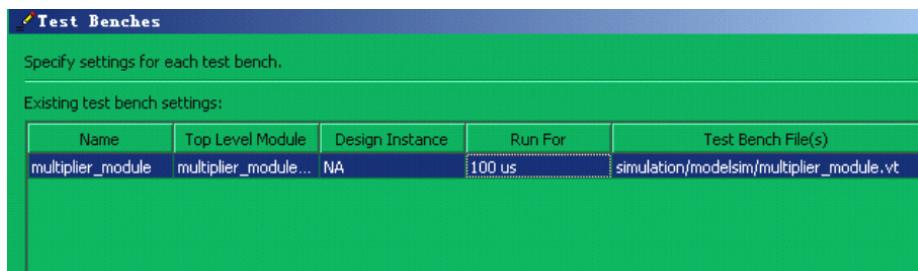
关于 End simulation 是仿真最大的时间，笔者习惯设置 1us（视仿真要求而定）。



接下来的工作，我们就要将 .vt 文件添加进来。点击 File name 最右边的 ... 。



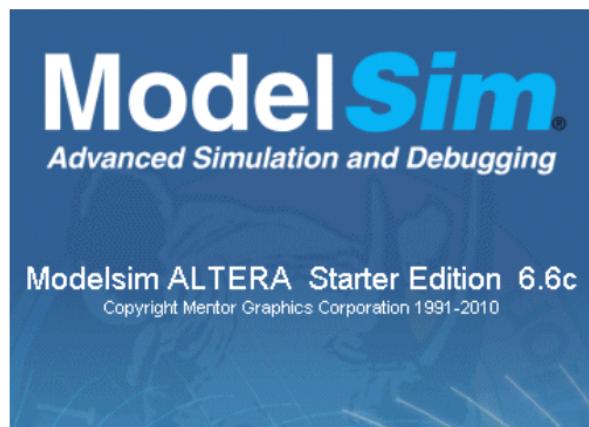
将 .vt 添加进去，然后点击 OK 即可。



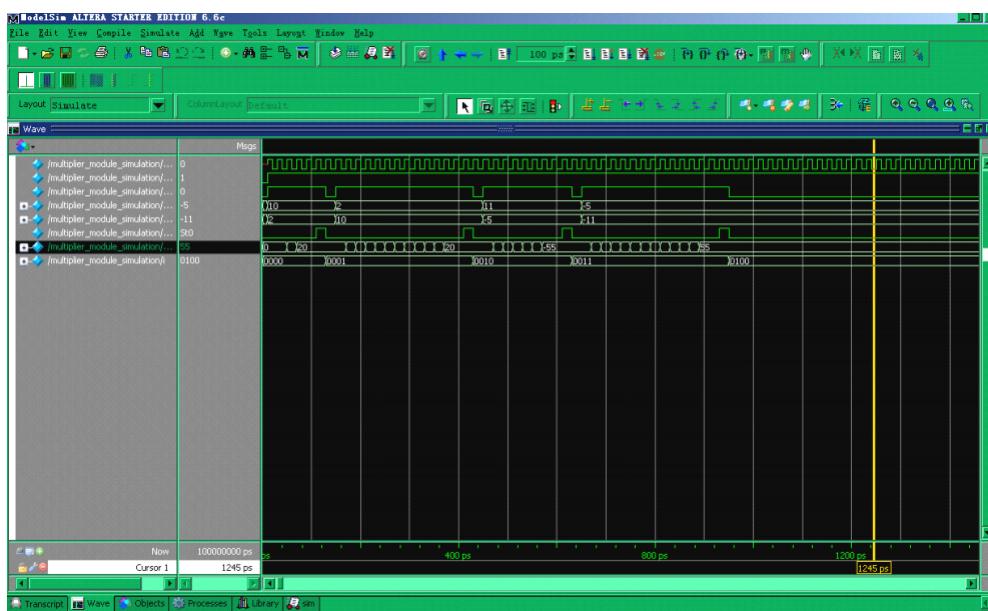
完成后，上面的窗口会显示已经添加 .vt 文件，然后点击 Okay 退出上面的窗口。最后在 Setting 的窗口中点击 Okay 退出 Setting 窗口并且编译项目后就大功告成了。

仿真开始

当项目编译成功而且 testbench 文件正确，那么就可以开始仿真了。



执行如上图的步骤，Modelsim-Altera 会自动打开，而且所有文件都会自动编译，懒惰的人真的赚到了。



当 Modelsim-Altera 编译成功后，读者要怎样 YY 都行。至于 Modelsim 的用法笔者就不涉及了，自己参考相关的资料吧。

上半部分：步骤和时钟

第一章：整数乘法器

1.1 整数的概念

整数在 IEEE 的规定上有，短整数 short integer，中整数 integer 和 长整数 long integer，它们之间的关系如下：

整数	字节空间	取值范围
短整数	一个字节	-127~127
中整数	两个字节	-32767~32767
长整数	和四个字节	-2147483647~2147483647

在这里笔者以短整数为笔记的主角。

短整数的最高位是符号位，符号位的正负表示了该值是“正还是负”？。正值的表示方法很简单，反之负值的表示方法是以补码来表示。

```
+127 亦即 8'b0111_1111;
+4 亦即 8'b0000_0100;
-127 亦即 8'b1000_0001;
-4 亦即 8'b1111_1100;
```

补码在英文又叫 2nd implementation，其实是“正值的求反又加一”的操作。(哎~年轻时的笔者曾经为这个东西头疼过)。一个负值如 -4，是由 +4 求反由加一后而成。

```
8'b0000_0100; // 正值 4
8'b1111_1011; // 求反
8'b1111_1100; // 加 1， 负值 4
```

那么符号位和正值，负值，补码，取值由有什么关系呢？举个例子：A = 8'b0111_1111 (+127) 和 B = 8'b1000_0001 (-127)。

当我们在进行判断一个短整数是正值还是负值的时候，我们可以这样表示：

```
if( !A[7] ) ... // A 是正值
if( B[7] ) ... // B 是负值
```

事实上，我们知道短整数的位宽为 2⁸，亦即取值范围是 0~255。但是符号位的出现吃掉了最高位，所以造成取值范围变成 2⁷=0~171。

你知道吗？在短整数家族里面永远存在一个幽灵成员。该成员很神秘，它不是正值，即不是负值或者 0 值。而且它的能力也不可忽视，它划分了正值和负值的边界，它就是 **8'b1000_0000**。

```
+127  8'b0111_1111;  
划分  8'b1000_0000;  
-127  8'b1000_0001;
```

换句话说，在 **8'b1000_0000** 之前的都是正值，然而在 **8'b1000_0000** 之后是负值。如果读者硬是要说 **8'b1000_0000** 是“负 0”，笔记也无话可说

从上述的内容，我们可以知道：正值可以进行求反又加一之后成为负值。那么负值如何变成正值？同样的一个道理 “负值求反又加一后，成为正值”。

```
8'b1111_1100; // 负 4  
8'b0000_0011; // 求反  
8'b0000_0100; // 加 1 , 正 4
```

1.2 传统乘法的概念

笔者还记得笔者在上小学三年级的时候，老师在黑板上写上 $3 \times 4 = 12$ 。笔者对这神秘的数学公式迷糊了头脑。后来老师解释道：“3 粒苹果重复加上 4 次等于 12 粒苹果”，小时的笔者顿时恍然大悟！

当笔者上了初中，老师在黑板上写上 $3 + -4 = -1$ 。大伙们都明白那是整数，但是初中的笔者脑袋过很迟钝。初中的笔者没有“-3 粒苹果”类似实体的概念，然后老师解释道：“小明欠小黄 4 粒苹果，后来小明还了小黄 3 粒苹果，结果小明还欠小黄一粒苹果”，初中的笔者又恍然大悟。

当老师又在黑板上写上如下的内容：

```
3 x 4 = 12;      " 3 粒苹果重复叠加 4 次，等于 12 粒苹果"
-3 x 4 = -12;    " 欠 3 粒苹果，重复欠 4 次，等于欠 12 粒苹果"
3 x -4 = -12;    " 欠 4 粒苹果，重复欠 3 次，等于欠 12 粒苹果 "
-3 x -4 = 12;    "@#￥%#￥*! %…… "( 嘀咕中 ... )
```

那时候的笔者，嘴巴长得大大，有好一段时间说不出话来。好一段时间笔者都是自己在嘀咕……读者们不要笑，上述的故事确实是笔者的真实故事。那时候的笔者，真的拿不到整数的乘法的门儿，考试还常常满江红，真是悲剧的初中时代……

在传统的概念上乘法等价于“重复几次”。打个比方： $B = 4$ ； $A \times B$ 亦即 A 要重复加四次才能得到答案。

然而在乘法中“负值正值的关系”就是“异或的关系”。

A 值	B 值	结果
正 (0)	正 (0)	正 (0)
正 (0)	负 (1)	负 (1)
负 (1)	正 (0)	负 (1)
负 (1)	负 (1)	正 (0)

```
A x B = C;
3 x 4 = 12;
-3 x 4 = -12;
3 x -4 = -12;
-3 x -4 = 12;
```

从上面的内容看来，无论 A 值和 B 值是什么样的“正值和负值的关系”，结果 C 都是一样。

那么我们可以换一个想法：

“在作乘法的时候只是我们只要对正值进行操作。然而“负值和正值的结果”，我们用“异或”关系来判断 ... ”

实验一：传统的乘法器

该乘法器的大致操作如下：

- (一) 在初始化之际，取乘数和被乘数的正负关系，然后取被乘数和乘数的正值。
- (二) 每一次累加操作，递减一次乘数。直到乘数的值为零，表示操作结束。
- (三) 输出结果根据正负关系取得。

multiplier_module.v

```
1. module multiplier_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]Multiplicand,
8.     input [7:0]Multiplier,
9.
10.    output Done_Sig,
11.    output [15:0]Product
12. );
13.
14.    *****/
15.
16.    reg [1:0]i;
17.    reg [7:0]Mcand;
18.    reg [7:0]Mer;
19.    reg [15:0]Temp;
20.    reg isNeg;
21.    reg isDone;
22.
23.    always @ ( posedge CLK or negedge RSTn )
24.        if( !RSTn )
25.            begin
26.                i <= 2'd0;
```

```

27.          Mcand <= 8'd0;      // Register for Multiplicand
28.          Mer <= 8'd0;      // Register ofr Multiplier
29.          Temp <= 8'd0;     // Sum of prataloc product
30.          isNeg <= 1'b0;
31.          isDone <= 1'b0;
32.      end
33.  else if( Start_Sig )
34.      case( i )
35.
36.          0:
37.          begin
38.
39.              isNeg <= Multiplicand[7] ^ Multiplier[7];
40.              Mcand <= Multiplicand[7] ? ( ~Multiplicand + 1'b1 ) : Multiplicand;
41.              Mer <= Multiplier[7] ? ( ~Multiplier + 1'b1 ) : Multiplier;
42.              Temp <= 16'd0;
43.              i <= i + 1'b1;
44.
45.          end
46.
47.          1:// Multipling
48.          if( Mer == 0 ) i <= i + 1'b1;
49.          else begin Temp <= Temp + Mcand; Mer <= Mer - 1'b1; end
50.
51.          2:
52.          begin isDone <= 1'b1; i <= i + 1'b1; end
53.
54.          3:
55.          begin isDone <= 1'b0; i <= 2'd0; end
56.
57.      endcase
58.
59.  *****/
60.
61. assign Done_Sig = isDone;
62. assign Product = isNeg ? ( ~Temp + 1'b1 ) : Temp;
63.
64. *****/
65.
66. endmodule

```

第 3~11 行是该模块的输入输出。当我们看到 Start_Sig 和 Done_Sig 就会知道这是有仿顺序操作性质模块的结构性标志，不明白的去看笔者之前写的笔记。Multiplicand 和 Multiplier (被乘数和乘数)，都是 8 位位宽，所以输出 Product 是 16 位位宽。

第 16~21 行是该模块所使用的所有寄存器。i 寄存表示步骤，Mcand 用来暂存 Multiplicand 的正值，Mer 用来暂存 Multiplier 的正值，Temp 寄存器是操作空间。然而 isNeg 标志寄存器是用来寄存 Multiplicand 和 Multiplier 之间的正负关系。

在步骤 0 (36~45 行) 是初始化的步骤。第 39 行 isNeg 寄存“乘数和被乘数之间的正负关系”。第 40 行，Mcand 寄存 Multiplicand 的正值，该行表示“如果被乘数的符号位是逻辑 1 的话，就将负值转换为正值，然后 Mcand 寄存该值，否则 Mcand 直接寄存 Multiplicand 的正值”。第 41 行是用来寄存 Multiplier 的正值，该行的操作和 40 行很相似。

在步骤 1 (47~49 行)，是“重复加几次”的操作。Temp 寄存器是叠加空间，Mer 寄存器是寄存递减“重复第几次？”的结果 (49 行)。当 Mer 的值等于 0 (48 行) 这也表示“重复加几次”的操作已经完毕，然后进入下一个步骤。步骤 2~3 是产生完成信号。

在 62 行，Product 信号的输出值是由 isNeg 寄存器作决定，如果 isNeg 是逻辑 1，那么 Temp 的结果从正值转换为负值。否则直接输出 Temp 的值。

multiplier_module.vt

```

1. `timescale 1 ps/ 1 ps
2. module multiplier_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Start_Sig;
8.     reg [7:0] Multiplicand;
9.     reg [7:0] Multiplier;
10.
11.    wire Done_Sig;
12.    wire [15:0]Product;
13.
14.    *****/
15.
16.    initial
17.    begin
18.
19.        RSTn = 0; #10; RSTn = 1;
20.        CLK = 1; forever #10 CLK = ~CLK;
21.
22.    end
23.
24.    *****/

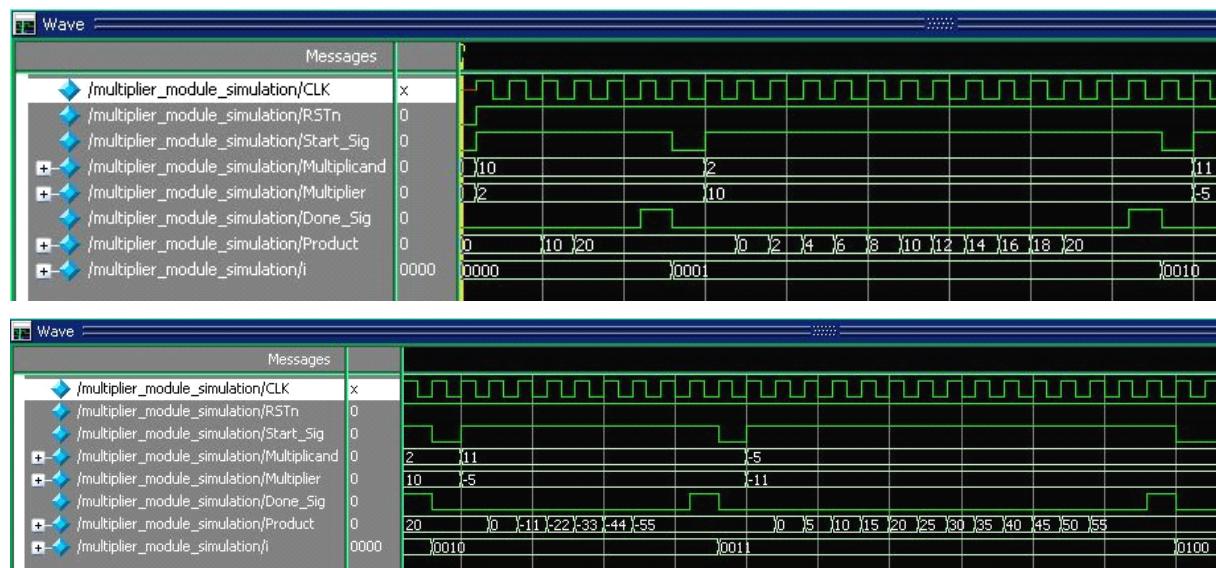
```

```
25.  
26.      multiplier_module U1  
27.      (  
28.          .CLK(CLK),  
29.          .RSTn(RSTn),  
30.          .Start_Sig(Start_Sig),  
31.          .Multiplicand(Multiplicand),  
32.          .Multiplier(Multiplier),  
33.          .Done_Sig(Done_Sig),  
34.          .Product(Product)  
35.      );  
36.  
37.      /**************************************************************************/  
38.  
39.      reg [3:0]i;  
40.  
41.      always @ ( posedge CLK or negedge RSTn )  
42.          if( !RSTn )  
43.              begin  
44.                  i <= 4'd0;  
45.                  Start_Sig <= 1'b0;  
46.                  Multiplicand <= 8'd0;  
47.                  Multiplier <= 8'd0;  
48.              end  
49.          else  
50.              case( i )  
51.  
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'd2; Start_Sig <= 1'b1; end  
55.  
56.                  1: // Multiplicand = 2 , Multiplier = 10  
57.                  if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
58.                  else begin Multiplicand <= 8'd2; Multiplier <= 8'd10; Start_Sig <= 1'b1; end  
59.  
60.                  2: // Multiplicand = 11 , Multiplier = -5  
61.                  if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
62.                  else begin Multiplicand <= 8'd11; Multiplier <= 8'b11111011; Start_Sig <= 1'b1; end  
63.  
64.                  3: // Multiplicand = -5 , Multiplier = -11  
65.                  if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
66.                  else begin Multiplicand <= 8'b11111011; Multiplier <= 8'b11110101; Start_Sig <= 1'b1; end  
67.  
68.                  4:  
69.                  begin i <= 4'd4; end
```

```
70.  
71.  
72.      endcase  
73.  
74.      /*****  
75.  
76.  
77. endmodule
```

第 16~22 行是复位信号和时钟信号的刺激。第 26~35 行是 multiplier_module.v 的实例化。第 39 行以下和普通的仿顺序操作的写法一样，不明白的话请看笔者以往写过的笔记。步骤 0~3，会输入不同的乘数和被乘数来刺激 multiplier_module.v。

仿真结果：



实验一说明：

其实传统的乘法器是很容易的，但是随着整数的出现，负值和正值也随着出现，这也使得设计多了一点难度。但是只要掌握负值和正值的关系以后，乘法只作正值也“无问题”，结果只要在输出之前下一点手脚就行了。

实验一结论：

传统的乘法器虽然简单，但是它有一个致命的问题。就是被乘数越大就越消耗时钟。具体的原因在下一章节解释

1.3 传统乘法器的改进

Verilog HDL 语言所描述的乘法器是以“消耗时钟”作为时间单位。反之组合逻辑所建立的乘法器是以“广播时间”作为时间单位。说简单点就是，Verilog HDL 语言所描述的乘法器“快不快”是根据“时钟消耗”作为评估。

假设 $A = 10, B = 20, A \times B$ ，那么时钟的消耗至少需要 20 个，因为 A 值需要累加 20 次才能得到结果。到底有没有什么办法可以改进这个缺点呢？有学过乘法的朋友都知道 $A(B)$ 等价于 $B(A)$ 。如果以实验一的乘法器作为基础，那么 $A(B)$ 和 $B(A)$ 所消耗的时间就不一样了。结果我们可以这样改进：

如果被乘数小于乘数，那么被乘数和乘数互换。

```
{ Multiplier , Multiplicand } = Multiplicand < Multiplier ? { Multiplicand , Multiplier } :  
{ Multiplier , Multiplicand } ;
```

举个例子：Multiplicand = 2，Multiplicand = 10；

在更换之前，被乘数 2 需要 10 次的累加才能得到结果，亦即需要消耗至少 10 个时钟才能求得结果。更换之后，被乘数为 10 乘数为 2，亦即被乘数 10 只要累加 2 次就能得到结果，所以时钟的消耗是 2 个以上。如此一来，10 次的累加次数和 2 次的累加次数相比，可以减少不少时钟的消耗。

实验二：传统乘法器改进

和实验一相比，实验二在进行累加操作之间多了一个步骤出来，就是被乘数和乘数比较的步骤。

- (一) 在初始化之际，取乘数和被乘数的正负关系，然后取被乘数和乘数的正值。
- (二) 乘数和被乘数比较，如果被乘数小于乘数，结果乘数和被乘数互换。
- (三) 每一次累加操作，递减一次乘数。直到乘数的值为零，表示操作结束。
- (四) 输出结果根据正负关系取得。

multiplier_module_2.v

```
1. module multiplier_module_2  
2. (  
3.     input CLK,  
4.     input RSTn,  
5.
```

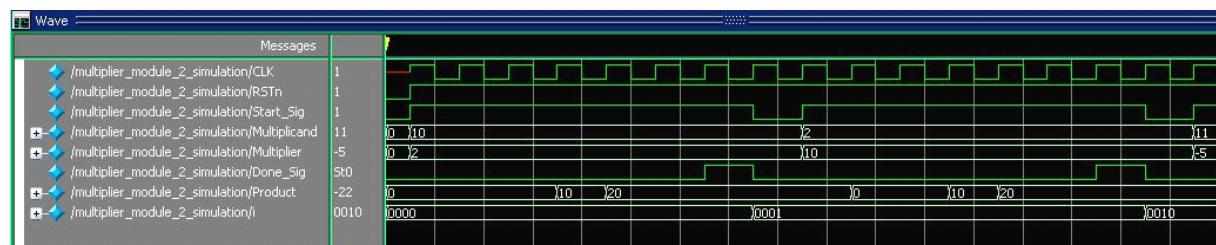
```
6.      input Start_Sig,
7.      input [7:0]Multiplicand,
8.      input [7:0]Multiplier,
9.
10.     output Done_Sig,
11.     output [15:0]Product
12. );
13.
14. ****
15.
16. reg [2:0]i;
17. reg [7:0]Mcand;
18. reg [7:0]Mer;
19. reg [15:0]Temp;
20. reg isNeg;
21. reg isDone;
22.
23. always @ ( posedge CLK or negedge RSTn )
24.   if( !RSTn )
25.     begin
26.       i <= 3'd0;
27.       Mcand <= 8'd0; // Register for Multiplicand
28.       Mer <= 8'd0; // Register ofr Multiplier
29.       Temp <= 8'd0; // Sum of pratocal product
30.       isNeg <= 1'b0;
31.       isDone <= 1'b0;
32.     end
33.   else if( Start_Sig )
34.     case( i )
35.
36.     0:
37.       begin
38.         isNeg <= Multiplicand[7] ^ Multiplier[7];
39.         Mcand <= Multiplicand[7] ? ( ~Multiplicand + 1'b1 ) : Multiplicand;
40.         Mer <= Multiplier[7] ? ( ~Multiplier + 1'b1 ) : Multiplier;
41.         Temp <= 16'd0;
42.         i <= i + 1'b1;
43.       end
44.
45.     1:
46.       begin
47.         { Mcand , Mer } <= Mcand < Mer ? { Mer , Mcand } : { Mcand , Mer };
48.         i <= i + 1'b1;
49.       end
50.
```

```
51.          2: // Multiplying
52.          if( Mer == 0 ) i <= i + 1'b1;
53.          else begin Temp <= Temp + Mcand; Mer <= Mer - 1'b1; end
54.
55.          3:
56.          begin isDone <= 1'b1; i <= i + 1'b1; end
57.
58.          4:
59.          begin isDone <= 1'b0; i <= 3'd0; end
60.
61.      endcase
62.
63.  *****/
64.
65. assign Done_Sig = isDone;
66. assign Product = isNeg ? ( ~Temp + 1'b1 ) : Temp;
67.
68. *****/
69.
70. endmodule
```

和实验一先比，添加了一个比较的步骤（45~49 行）。

仿真结果：

仿真的 .vt 文件和实验一一样。



在仿真的结果上， 10×2 和 2×10 的时钟消耗都一样。

实验二说明：

如果和实验一的乘法器相比较的话，有关时钟的消耗实验二的乘法器多少都有所改进。

实验二结论：

传统的乘法器无论如何改进也好，当遇见如 127×127 的乘数和被乘数，咋样也看不出什么可以优化的地方

1.4 补码存在的意义

每一个人都有存在的意义，有的人用一生的时间去寻找自己的存在意义，有的人则是经过生活的大反转，看到了自己存在意义，有的人则不闻不问 ... 当然补码也有存在的意义，只是在前面的实验被笔者滥用而已。

补码不仅可以执行正值和负值转换，其实补码存在的意义，就是避免计算机去做减法的操作。

$$\begin{array}{r} 1101 \quad -3 \text{ 补} \\ + 1000 \quad 8 \\ \hline 0101 \quad 5 \end{array}$$

假设 $-3 + 8$ ，只要将 -3 转为补码形式，亦即 $0011 \Rightarrow 1101$ ，然后和 8 ，亦即 1000 相加就会得到 5 ，亦即 0101 。至于溢出的最高位可以无视掉。

$$\begin{array}{r} 1101 \quad -3 \text{ 补} \\ + 1110 \quad -2 \text{ 补} \\ \hline 1011 \quad -5 \text{ 补} \end{array}$$

你知道吗？其实 Quartus II 综合器，当我们使用“ $-$ ”算术操作符的时候，其实就是使用补码的形式，具体如下：

```
A = 8'd5;  
B = 8'd9;  
  
A - B 等价于 A + (~B + 1'b1);
```

在实际的操作中，综合器都会如上优化。

1.5: Booth 算法乘法器

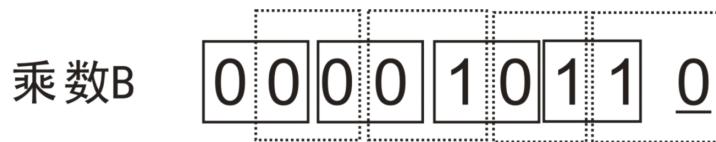
传统的乘法器是有极限的，因此位操作乘法就出现了。笔者在网上冲浪找资源的时候，还常常撞到许多稀奇古怪的位操作乘法器。但是有一种位操作乘法器，吸引了笔者的眼球，它就是 Booth 算法乘法器。实际上 Booth 算法是一种“加码”乘法运算。

Booth 算法的概念也很简单，我们先从数学的角度去理解看看：

B[0]	B[-1]	加码结果
0	0	0 (无操作)
0	1	1 (+被乘数)
1	0	<u>1</u> (-被乘数)
1	1	0 (无操作)

B[-1] 是什么？先假设 B 是 2 的，然而 B 的最右边的后面一位称为“负一位”，那就是 B[-1]。

0010 0 // LSB 右边出现的就是 -1 位



Modified 乘数加码概念

那么上面那个加码表和乘数 B 又有什么关系呢？假设乘数 B 为 2，那么乘数 2 的加码过程会是如下。

1. 一开始的时候在乘数 2 的“负一位”加上一个默认 0 值	0010 0
2. 先判断[0: -1]，结果是 2'b00，表示“0”亦即没有操作	0010 0
3. 判断[1: 0]，结果是 2'b10，表示“1”亦即“-被乘数”操作	0010 0
4. 判断[2: 1]，结果是 2'b01，表示“1”亦即“+被乘数”操作	0010 0
5. 判断[3: 2]，结果是 2'b00，表示“0”亦即没有操作	0010 0

举个例子，被乘数为 7，0111；乘数为 2，0010；结果会是什么？

0111	- A 被乘数
x 0010 0	- B 乘数
<hr/>	
0110	- 乘数加码
<hr/>	
0000	0
111001	1 (-7)
0111	1 (+7)
+ 0000	0
<hr/>	
0001110	14
<hr/>	

从上面的操作过程中，我们可以看到乘数被加码以后，操作的结果是 14。从数学的角度看来，Booth 算法确实是麻烦的存在，如果从位操作的角度来看就不是这么一回事了。实际上，在千奇百怪的位操作乘法中，Booth 算法是唯一几个可以容纳“补码”，亦即 Booth 算法可以容纳“负数”来执行操作。

B[0]	B[-1]	加码结果
0	0	无操作，右移一位
0	1	+被乘数，右移一位
1	0	-被乘数，右移一位
1	1	无操作，右移一位

上面的图表是位操作时候的 **Booth 算法**。Booth 算法在位操作的时候，它使用一个很有个性的空间，就是 P 空间。

先假设：被乘数 A 为 7 (0111)，乘数 B 为 2 (0010)，它们 n 均为 4 位，所以 P 空间的容量是 $n \times 2 + 1$ ，亦即 9 位。

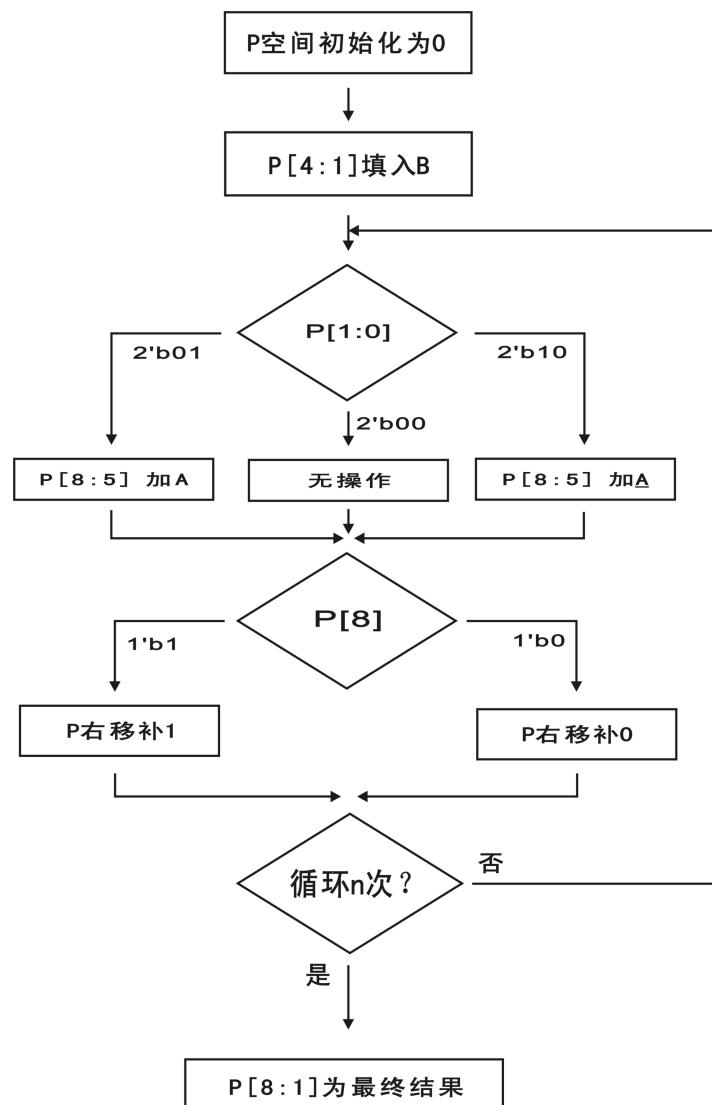
// P 空间

那么 P 空间如何实现乘法的位操作呢？

1. 一开始先求出 -1 (被乘数)	A = 0111, <u>A</u> = 1001
2. 然后初始化 P 空间，默认为 0	P = 0000 0000 0
3. P 空间的 [4..1] 填入乘数	P = 0000 0010 0
4. 判断 P[1:0]，是 2'b00 亦即“无操作”	P = 0000 0010 0
5. 判断 P[8]，如果是逻辑 0 右移一位补 0，反之右移一位补 1。	P = 0000 0001 0
6. 判断 P[1:0]，是 2'b10 亦即“-被乘数”。	P = 0000 0001 0
7. P 空间的[8..5] 和 被乘数 <u>A</u> 相加。	P = 0000 0001 0 + 1001

	P = 1001 0001 0
8. 判断 P[8], 如果是逻辑 0 右移一位, 补 0, 反之右移一位补 1	P = 1100 1000 1
9. 判断 P[1:0], 是 2'b01 亦即 “+被乘数”。	P = 1100 1000 1
10. P 空间的[8..5] 和 被乘数 A 相加。	$ \begin{array}{r} p = 1100 1000 1 \\ + 0111 \\ \hline P = 0011 1000 1 \text{ 无视最高位溢出} \end{array} $
11. 判断 P[8], 如果是逻辑 0 右移一位补 0, 反之右移一位补 1	P = 0001 1100 0
12. 判断 P[1:0], 是 2'b00 亦即 “无操作”	P = 0001 1100 0
13. 判断 P[8], 如果是逻辑 0 右移一位, 补 0, 反之右移一位补 1	P = 0000 1110 0
14. 最终 P 空间的[8..1] 就是最终答案。	P = 0000 1110 0

从上面的操作看来, 由于乘数和被乘数均为 n 位, 所以 “先判断 P[1:0], 然后操作 p 空间, 最后移位 p 空间” 等动作的操作是执行 “4 次”。



如上面的循环图。A 为被乘数， \underline{A} 为被乘数补码形式（ $-1(A)$ ），B 为乘数，n 为乘数和被乘数的位宽，P 为操作空间。一开始 P 空间会初始化，然后 P 空间的[4..1]位会填入 B。然后进入 P[1:0]的判断。每一次的判断过后的操作都会导致 P 空间右移一次，至于右移过后的最高位是补 0 还是补 1，是由当时 P[8]说了算。当循环 n 次以后，最终结果会是 P[8:1]。

实验三：Booth 算法乘法器

实验中建立的 Booth 算法乘法器大致的步骤正如 1.5 章节所描述的那样。

booth_multiplier_module.v

```
1. module booth_multiplier_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]A,
8.     input [7:0]B,
9.
10.    output Done_Sig,
11.    output [15:0]Product,
12.
13.    output [7:0]SQ_a,
14.    output [7:0]SQ_s,
15.    output [16:0]SQ_p
16. );
17.
18. ****
19.
20. reg [3:0]i;
21. reg [7:0]a; // result of A
22. reg [7:0]s; // reverse result of A
23. reg [16:0]p; // operation register
24. reg [3:0]X;
25. reg isDone;
26.
27. always @ ( posedge CLK or negedge RSTn )
28.     if( !RSTn )
29.         begin
```

```
30.          i <= 4'd0;
31.          a <= 8'd0;
32.          s <= 8'd0;
33.          p  <= 17'd0;
34.          X <= 4'd0;
35.          isDone <= 1'b0;
36.      end
37.  else if( Start_Sig )
38.      case( i )
39.
40.          0:
41.              begin a <= A; s <= ( ~A + 1'b1 ); p <= { 8'd0 , B , 1'b0 }; i <= i + 1'b1; end
42.
43.          1:
44.              if( X == 8 ) begin X <= 4'd0; i <= i + 4'd2; end
45.              else if( p[1:0] == 2'b01 ) begin p <= { p[16:9] + a , p[8:0] }; i <= i + 1'b1; end
46.              else if( p[1:0] == 2'b10 ) begin p <= { p[16:9] + s , p[8:0] }; i <= i + 1'b1; end
47.              else i <= i + 1'b1;
48.
49.          2:
50.              begin p <= { p[16] , p[16:1] }; X <= X + 1'b1; i <= i - 1'b1; end
51.
52.          3:
53.              begin isDone <= 1'b1; i <= i + 1'b1; end
54.
55.          4:
56.              begin isDone <= 1'b0; i <= 4'd0; end
57.
58.      endcase
59.
60.  *****/
61.
62.  assign Done_Sig = isDone;
63.  assign Product = p[16:1];
64.
65.  *****/
66.
67.  assign SQ_a = a;
68.  assign SQ_s = s;
69.  assign SQ_p = p;
70.
71.  *****/
72.
73.
74. endmodule
```

第 13~15 行是仿真的输出 (S - Simulation , Q - Output)。第 20~25 行定义了该模块所使用的寄存器。a 寄存器用来寄存 A 值，s 寄存器用来寄存 -1(A) 的值，p 寄存器是 P 空间。输入信号 A 和 B 均为 8 位位宽，所以 p 寄存器是 17 位位宽。至于 X 寄存器是用来表示 n 位，用来指示 n 次循环。

步骤 0 (40~41 行)，初始化了 a, s 寄存器。p[8:1]填入 B 值，亦即乘数，其余的位均为 0 值。

步骤 1 (43~47 行) 是用来判断 p[1:0] 的操作。步骤 2 (49~50 行) 是执行右移一位，是补 0 还是补 1，完全取决于 p[16]。步骤 1~2 会重复交替执行，直到 X 的值达到 8 次(第 44 行)，就会进入下一步步骤。

步骤 3~4(52~56 行)是用来产生完成信号。第 63 行输出信号 product 是由 p 空间的[16..1]来驱动。第 67~69 行是仿真用的输出信号，功能如命名上的意思。

booth_multiplier_module.vt

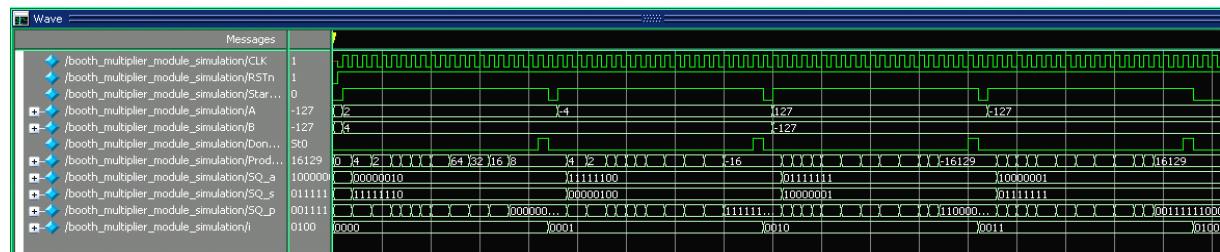
```
1. `timescale 1 ps/ 1 ps
2. module booth_multiplier_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Start_Sig;
8.     reg [7:0]A;
9.     reg [7:0]B;
10.
11.    wire Done_Sig;
12.    wire [15:0]Product;
13.
14.    /*****
15.
16.    wire [7:0]SQ_a;
17.    wire [7:0]SQ_s;
18.    wire [16:0]SQ_p;
19.
20.    *****/
21.
22.    booth_multiplier_module U1
23.    (
24.        .CLK(CLK),
25.        .RSTn(RSTn),
26.        .Start_Sig(Start_Sig),
```

```
27.          .A(A),
28.          .B(B),
29.          .Done_Sig(Done_Sig),
30.          .Product(Product),
31.          .SQ_a(SQ_a),
32.          .SQ_s(SQ_s),
33.          .SQ_p(SQ_p)
34.      );
35.
36.      *****/
37.
38. initial
39. begin
40.     RSTn = 0; #10; RSTn = 1;
41.     CLK = 0; forever #10 CLK = ~CLK;
42. end
43.
44. *****/
45.
46. reg [3:0]i;
47.
48. always @ ( posedge CLK or negedge RSTn )
49.     if( !RSTn )
50.         begin
51.             i <= 4'd0;
52.             A <= 8'd0;
53.             B <= 8'd0;
54.             Start_Sig <= 1'b0;
55.         end
56.     else
57.         case( i )
58.
59.             0: // A = 2, B = 4
60.             if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
61.             else begin A <= 8'd2; B <= 8'd4; Start_Sig <= 1'b1; end
62.
63.             1: // A = -4 , B = 4
64.             if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
65.             else begin A <= 8'b11111100; B <= 8'd4; Start_Sig <= 1'b1; end
66.
67.             2: // A = 127, B = -127
68.             if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
69.             else begin A <= 8'd127; B <= 8'b10000001; Start_Sig <= 1'b1; end
70.
71.             3: // A = -127, B = -127
```

```
72.         if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
73.     else begin A <= 8'b10000001; B <= 8'b10000001; Start_Sig <= 1'b1; end
74.
75.             4:
76.                 i <= 4'd4;
77.
78.         endcase
79.
80.     /*****
81.
82. endmodule
```

在仿真中，从步骤 0~3 (59~73 行)，刺激了不同 A 和 B 的值（被乘和数乘数）。

仿真结果：



(P 空间的详细操作过程，自己代开 modelsim 看吧，界面有限的关系) 从仿真结果上可以看到，4 次的乘法操作所使用的时间都一样，尤其是 -127×-127 的情形，不像传统乘法器那样累加 127 次，才能得到结果。**(p 空间的[Width :1]是用来填入乘数 B，然而 p 空间的 [Width * 2 : Width + 1] 是用来执行和被乘数 A 的操作)**

实验三结论：

按常理来说 8 位的乘数和被乘数，位操作会是使用 8 个时钟而已，但是实验 3 的乘法器，需要先操作后移位的关系，所以多出 8 个时钟的消耗

1.6 笔者情有独钟的步骤

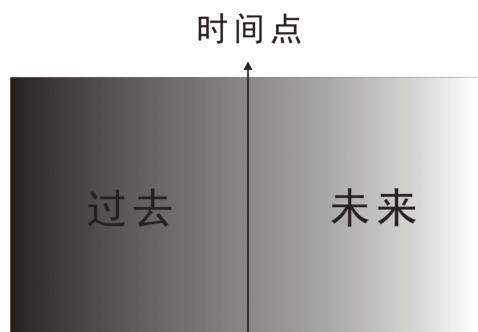
在笔者初学 Verilog HDL 语言，笔者老是捉不好 Verilog HDL 语言，吃了不少苦头。世界就是很巧妙，脑子里就忽然间冒出步骤 i。

步骤 i 是什么？

如果从低级建模的角度去探看骤 i：低级建模里面有一个准则，就是“一个模块一个功能”，步骤 i 好比这个准则的支持者。步骤 i 从 0 开始，表示了这个模块开始工作，直到 i 被清理，这也表示了这个模块已经结束工作。或者可以这样说“一个模块不会出现两个步骤 i”。

有关《Verilog HDL 那些事儿 - 建模篇》那本笔记，几乎所有实验都和“步骤 i”有关。但是在笔记中，笔者只是微微的带过“步骤 i 是仿顺序操作相关的写法 ...”简单的解释而已。步骤 i 的用法很简单，从概念上和“把大象放进冰箱”很类似，实际上这样的认识是一个假像（这样的理解不是错误，只是不完全正确而已）。

Verilog HDL 语言里的“步骤”和 C 语言里的“步骤”的概念是不一样的。C 语言里的“步骤”就好比“把大象放进冰箱需要几个步骤 ...”，相反的 Verilog HDL 语言里的“步骤”，有如“时间点”的观念。（我们常常会把 Verilog HDL 语言的“步骤”看成是“把大象放进冰箱 ...”，这是一种假像，但是这个假像是不会对设计造成很大的影响）



如上面的示意图所示，在这个“时间点”里的所有“决定”会产生不一样的未来。然而“过去”是这个“时间点”可以参考的存在。在这个“时间点”里“可以允许不同的决定在这一刻存在”。举一个例子：A 的初值是 4，B 的初值是 0。

```
case( i )  
 0:  
    begin A <= A + 2'd2; B <= B + 2'd3; i <= i + 1'b1; end  
  
  1:  
    if( A > 3 ) begin B <= A; A = 0; i <= i + 1'b1; end
```

```
else if i <= i + 1'b1;
```

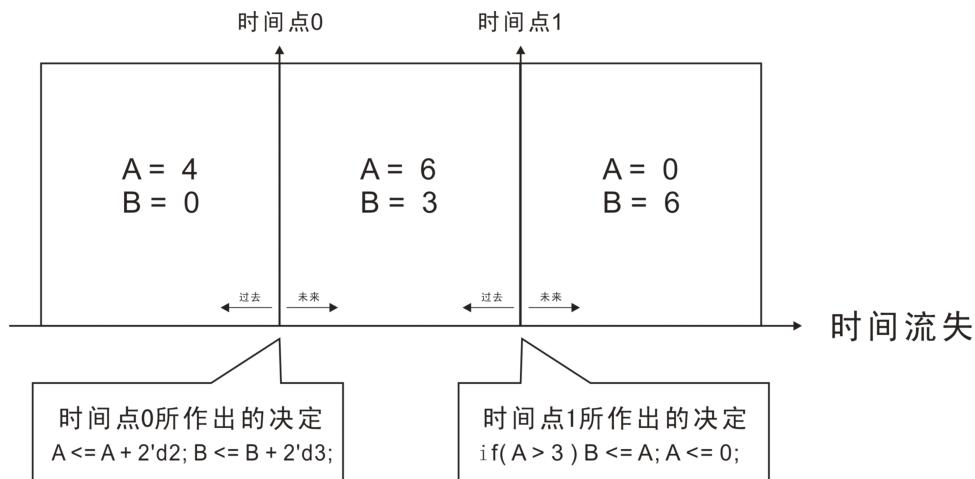
上面的代码看是很贱，但是读者知道里边包含的秘密吗？

在 $i = 0$ 的时候，A “决定” 累加 2，B “决定” 累加 3。
在 $i = 1$ 的时候，如果 A 大于 3，B “决定” 寄存 A 的值，然而将 A 清零。

但是在时间点0，A
值仍为4 B值仍为0

如果用“生动”的话来描述的话。在时间点 0 的时候，这个模块“决定” A 累加 2，B 累加 3。然后在时间点 0 过后“结果”就产生。在时间点 1 的时候，这个模块判断 A 是否大于 3。那么，问题来了“这个模块是以什么作为基础判断 A 大于 3 呢？”。

答案很简单，就是“ A 在时间点 1 过去的结果”或者说“ A 在时间点 0 过后所产生的结果”。



上图完全将上述的内容表达了出来。在这里笔者有一个很在意的问题，那就是“ $<=$ ”赋值操作符。在众多的参考书中“ $<=$ ”赋值操作符被解释为“时间沿有效的赋值操作符”。笔者初学的时候的，完全不知道它是虾米 ... 如果换做时间点的概念来说“ $<=$ ”的操作符，表示了“在这个时间点下决定”的专用赋值操作符。与“ $=$ ”赋值操作符不一样，它没有时间点的概念，所以在 always @ (posedge CLK ...) 区域内它比较少使用。

我们的人生，下错了”决定“只要知错，吸取教训还有从来的机会。但是模块下错了决定，就影响它的一生，所以我们在编辑的时候要特别小心，不然会可能因我们的疏忽，导致了这个模块的一生悲剧。

小时候，笔者学习道德教育的时候，有一句话是笔者一生受用，那就是“先三思而后行”。这个又和上述的内容有什么关系呢？我们知道“时间点”的概念就是“在这个时间点决定了什么，这个时间点的未来会产生什么”。

举个例子，有一个模块他有 A，B 和 C 三个寄存器，它们的初值都是 0：

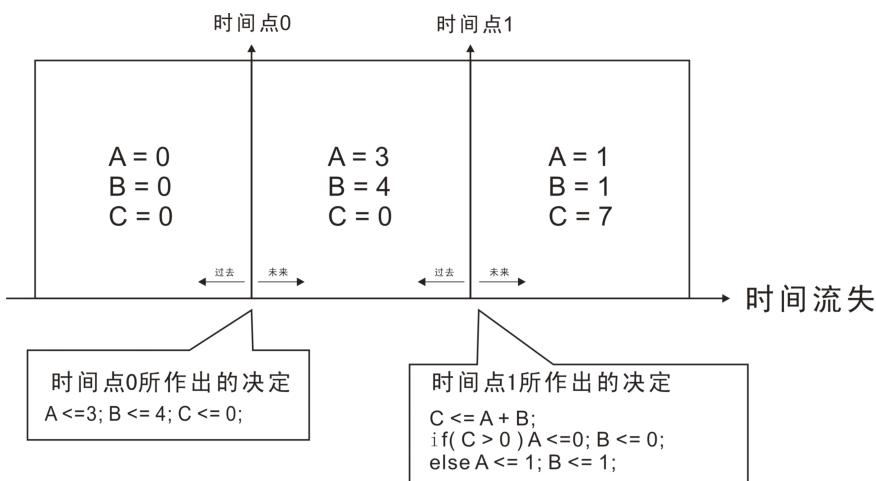
```
case( i )
  0:
    begin A <= 3; B <= 4; C <= 0; i <= i + 1'b1; end

  1:
    begin
      C <= A + B;
      if( C > 0 ) begin A <= 0; B <= 0 ; end
      else begin A <= 1; B <= 1; end

      i <= i + 1'b1;
    end
```

从上面的代码，我们可以知道。在时间点 0，该模块决定了 $A = 3$, $B = 4$, $C = 0$ 。然后到了时间点 1，问题来了“在时间点 1，该模块是以“什么作为基础”去判断 C 呢？**是时间点 1 过去的 C 值，还是在这一个瞬间 $A + B$ 所产生的值？**”。

C



答案如上图所示，if 是以时间点 1 过去的 C 值作为判断的基础。实际上“=”不是不可以出现在 always @ (posedge CLK ...) 里出现，只不过它比较危险。

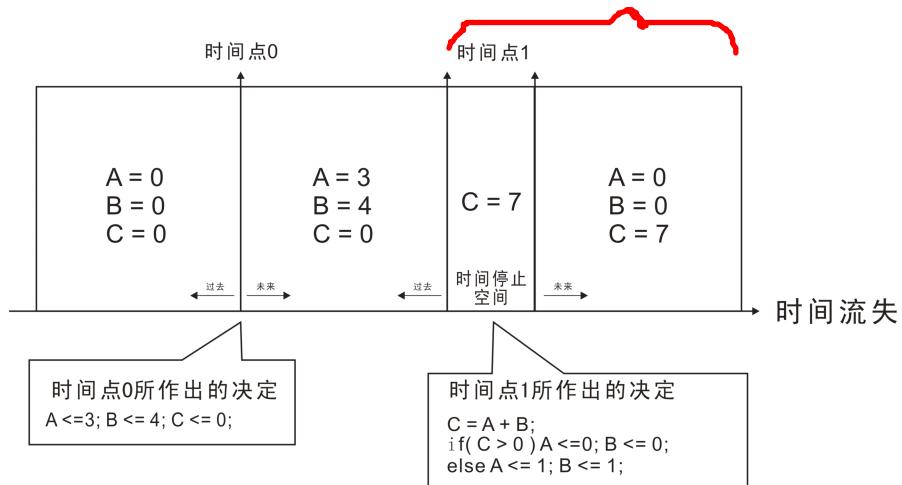
```
case( i )
  0:
    begin A <= 3; B <= 4; C <= 0; i <= i + 1'b1; end

  1:
    begin
      C = A + B
      if( C > 0 ) begin A <= 0; B <= 0 ; end
      else begin A <= 1; B <= 1; end
```

```
i <= i + 1'b1;
end
```

即时间点2将发生这一过程决定的全部结果
即 $C = 7 A = 0 B = 0$

笔者将上面的代码稍微修改了一下，在步骤 1 $C \leq A + B$ 变成了 $C = A + B$ 。



结果会是如上图。在时间点 1，“=” 赋值符号造成了一个而外的时间停止空间，在这个空间里 C 求得了“即时结果”。然而在这个时间点 1 里，“即时结果 C ”会作为判断的基础（而不是时间点 1， C 的过去值作为判断的基础）。

在某种程度上“=”赋值符号的存在会破坏“时间点”的和谐，如果没有有效控制的话，它很容易暴走。笔者在设计模块中，除非出现“不得已”的情况，否则笔者很少在 always @ (posedge CLK ...) 区域内使用它。

具体上，步骤 i 不只是用来支持低级建模的准则而已，此外步骤 i 的“值”还有指示着“第几个时钟沿”或者“模块目前的操作”等功能。这些功能在模块的“细化”过程起到很大的帮助。

1.7 Booth 算法乘法器的改进

在实验三中，所建立的 Booth 算法乘法器，要完成一次乘法计算，至少要消耗 16 个时钟，而且其中 8 个时间就是消耗在移位的方面上。那么有什么办法可以改进，并且将这些“消耗 8 个时钟的移位操作”压缩在同一个步骤内呢？

在 1.6 章节，笔者说了步骤 i 有如“时间点”的概念。假设笔者这样修改实验三的 Booth 乘法器：

```
case ( i )
    0: ... 初始化
    1,2,3,4,5,6,7,8:
        begin
            if( p[1:0] == 2'b01 ) p <= { p[16] , p[16:9] + a , p[8:1] }; 1
            else if( p[1:0] == 2'b10 ) p <= { p[16] , p[16:9] + s , p[8:1] }; 2
            else p <= { p[16] , p[16:1] }; 3
            i <= i + 1'b1;
        end
```

从上面的代码，读者能看出什么破绽吗？我们尝试回忆 Booth 算法的流程图，Booth 算法是先判断 $p[1:0]$ ，然后操作 p 空间，最后 p 空间右移一位，最高位补 0 还是补 1，是取决于经 $p[1:0]$ 操作之后的 $p[16]$ 。

那么问题来了，从上面的代码看来 $p <= \{ p[16] , p[16:9] + a , p[8:1] \}$ ，其中的 p 赋值的内容是以当前时间点的过去值作为基础，而不是以即时结果作为基础。所以上面的代码不行！那么我们可以这样重新修改代码：

改进之处：即时拿到当前时间点，并计算结果

```
case( i )
    0: ... 初始化
    1,2,3,4,5,6,7,8:
        begin
            Diff1 = p[16:9] + a;  Diff2 = p[16:9] + s;

            if( p[1:0] == 2'b01 ) p <= { Diff1[7] , Diff1 , p[8:1] };
            else if( p[1:0] == 2'b10 ) p <= { Diff2[7] , Diff2 , p[8:1] };
            else p <= { p[16] , p[16:1] };
```

```
i <= i + 1'b1;  
end
```

上面的代码表示了，在步骤 1~8 里 Diff1 寄存了 $p[16:9] + a$ 的即时结果，反之 Diff2 寄存了 $p[16:9] + s$ 的即时结果。然后判断 $p[1:0]$ 再来决定 p 的结果是取决于 Diff1，Diff2 或者其他。

在这里有一个重点是，Diff1 和 Diff2 没有使用 “ \leq ” 而是使用 “ $=$ ”，换一句话说，Diff1 和 Diff2 结果是“即时结果”。

具体的操作，还是在实验中明白。

实验四：Booth 算法乘法器改进

基于实验三的 Booth 算法乘法器，从原先的一次乘法需要 16 个时钟，优化至 8 个时钟。

booth_multiplier_module_2.v

```
1. module booth_multiplier_module_2  
2. (  
3.     input CLK,  
4.     input RSTn,  
5.  
6.     input Start_Sig,  
7.     input [7:0]A,  
8.     input [7:0]B,  
9.  
10.    output Done_Sig,  
11.    output [15:0]Product,  
12.  
13.    output [7:0]SQ_a,  
14.    output [7:0]SQ_s,  
15.    output [16:0]SQ_p  
16. );  
17.  
18.    *****/  
19.  
20.    reg [3:0]i;  
21.    reg [7:0]a; // result of A  
22.    reg [7:0]s; // reverse result of A  
23.    reg [16:0]p; // operation register
```

```

24.      reg [7:0]Diff1;
25.      reg [7:0]Diff2;
26.      reg isDone;
27.
28.      always @ ( posedge CLK or negedge RSTn )
29.          if( !RSTn )
30.              begin
31.                  i <= 4'd0;
32.                  a <= 8'd0;
33.                  s <= 8'd0;
34.                  p <= 17'd0;
35.                  Diff1 <= 8'd0;
36.                  Diff2 <= 8'd0;
37.                  isDone <= 1'b0;
38.              end
39.          else if( Start_Sig )
40.              case( i )
41.
42.                  0:
43.                      begin
44.
45.                          a <= A;
46.                          s <= ( ~A + 1'b1 );
47.                          p <= { 8'd0 , B , 1'b0 };
48.                          Diff1 <= 8'd0;
49.                          Diff2 <= 8'd0;
50.
51.                          i <= i + 1'b1;
52.
53.                      end
54.
55.                  1,2,3,4,5,6,7,8:
56.                      begin
57.
58.                          Diff1 = p[16:9] + a;
59.                          Diff2 = p[16:9] + s;
60.
61.                          if( p[1:0] == 2'b01 ) p <= { Diff1[7] , Diff1 , p[8:1] };
62.                          else if( p[1:0] == 2'b10 ) p <= { Diff2[7] , Diff2 , p[8:1] };
63.                          else p <= { p[16] , p[16:1] };
64.
65.                          i <= i + 1'b1;
66.
67.                      end

```

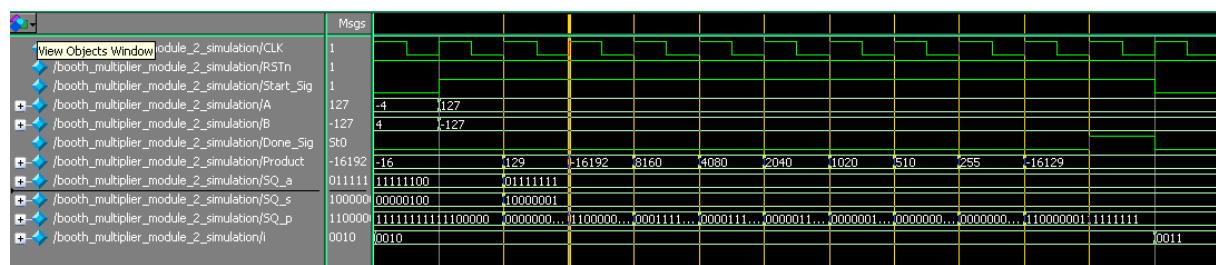
```

68.
69.         9:
70.             begin isDone <= 1'b1; i <= i + 1'b1; end
71.
72.         10:
73.             begin isDone <= 1'b0; i <= 4'd0; end
74.
75.         endcase
76.
77.     /*****
78.
79.     assign Done_Sig = isDone;
80.     assign Product = p[16:1];
81.
82.     *****/
83.
84.     assign SQ_a = a;
85.     assign SQ_s = s;
86.     assign SQ_p = p;
87.
88.     *****/
89.
90.
91. endmodule

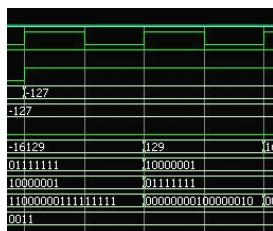
```

同样是 Booth 算法的原理，和实验三不同的是在 55~67 行，步骤 1~8 的循环操作。此外实验四不再使用 X 寄存器作为循环计数，而是直接使用步骤来指示 8 个循环操作（55~67 行）。这样的写法有一个好处，就是 p 空间的操作和 p 空间的移位可以压缩在同一个时钟里完成。

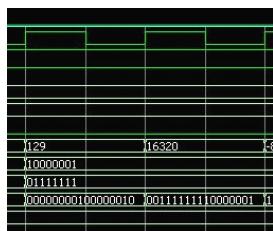
仿真结果：



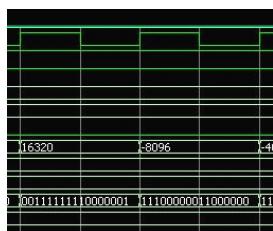
实验四所使用的 .vt 文件和实验三的一样，其中我们以 $127 * -127$ 作为解释。从仿真结果看来，一次的乘法操作只消耗 8 个时钟而已（步骤 0 初始化和步骤 9~10 完成信号产生除外）。现在我们把上面的仿真结果切成一块一块的来看。



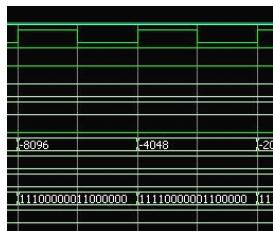
00000000 10000001 0 值左边上升沿开始，即是第一个时间点 $i = 0$ ，亦即步骤 0。步骤 0 之后就是初始化的结果。S 是取反过后的 a 值，并且填充在 p 空间的[8:1]。



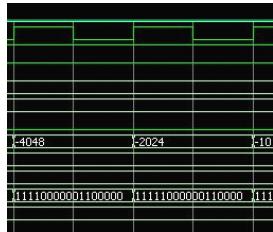
00000000 10000001 0 值右边的上升沿，亦即步骤 1。此时：Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $00000000 + 10000001$ ，结果为 10000001。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $00000000 + 01111111$ ，结果为 01111111。经步骤 1 的“决定”，过去 $p[1:0]$ 是 2'b10，所以 p 值的未来是 { Diff2[7], Diff2, p 过去[8:1] }，亦即 0 01111111 10000001。



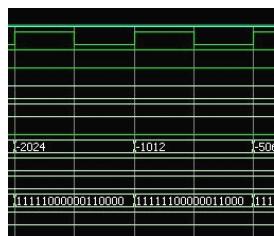
00111111 11000000 1 值右边的上升沿，亦即步骤 2。此时：Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $00111111 + 10000001$ ，结果为 11000000。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $00111111 + 01111111$ ，结果为 10111110。经步骤 2 的“决定”，过去 $p[1:0]$ 是 2'b01，所以 p 值的未来是 { Diff1[7], Diff1, p 过去[8:1] }，亦即 1 11000000 11000000。



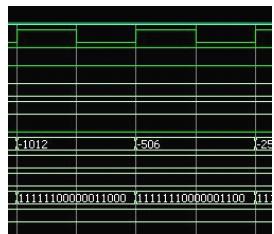
11100000 01100000 0 值右边的上升沿，亦即步骤 3。此时：Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11100000 + 10000001$ ，结果为 01100001。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11100000 + 01111111$ ，结果为 01011111。经步骤 3 的“决定”，过去 $p[1:0]$ 是 2'b00，所以 p 值的未来是 { p 过去[16], p 过去[16:1] }，亦即 1 11100000 01100000。



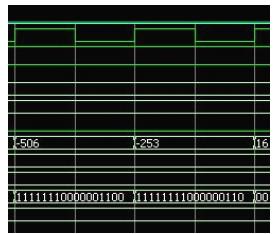
11110000 00110000 0 值右边的上升沿，亦即步骤 4。此时：Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11110000 + 10000001$ ，结果为 01110001。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11110000 + 01111111$ ，结果为 01101111。经步骤 4 的“决定”，过去 $p[1:0]$ 是 2'b00，所以 p 值的未来是 { p 过去[16], p 过去[16:1] }，亦即 1 11110000 00110000。



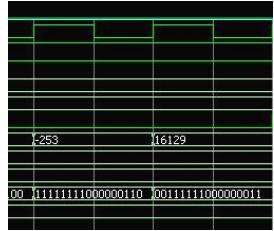
11111000 00011000 0 值右边的上升沿，亦即步骤 5。此时：
Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11111000 + 10000001$ ，结果为 01111001。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11111000 + 01111111$ ，结果为 01110111。经步骤 5 的“决定”，过去 $p[1:0]$ 是 2'b00，所以 p 值的未来是 { p 过去[16], p 过去[16:1]}，亦即 1 11111000 00011000。



11111100 00001100 0 值右边的上升沿，亦即步骤 6。此时：
Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11111100 + 10000001$ ，结果为 01111101。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11111100 + 01111111$ ，结果为 01111011。经步骤 6 的“决定”，过去 $p[1:0]$ 是 2'b00，所以 p 值的未来是 { p 过去[16], p 过去[16:1]}，亦即 1 1111100 00001100。



11111110 000001100 0 值右边的上升沿，亦即步骤 7。此时：
Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11111110 + 10000001$ ，结果为 01111111。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11111110 + 01111111$ ，结果为 01111101。经步骤 7 的“决定”，过去 $p[1:0]$ 是 2'b00，所以 p 值的未来是 { p 过去[16], p 过去[16:1]}，亦即 1 11111110 00000110。



11111111 000000110 值右边的上升沿，亦即步骤 8。此时：
Diff1 寄存过去的 $p[16:9] + a$ ，亦即 $11111111 + 10000001$ ，结果为 10000000。Diff2 寄存过去的 $p[16:9] + s$ ，亦即 $11111111 + 01111111$ ，结果为 01111110。经步骤 8 的“决定”，过去 $p[1:0]$ 是 2'b10，所以 p 值的未来是 {Diff2[7], Diff2, p 过去[8:1]}，亦即 0 01111110 00000011。

最终结果取值未来 $p[16:1]$ ，00111111 00000001 亦即 16129。

实验四说明：

如果以“大象放进冰箱”这样的概念去理解步骤 i，在实验四中可能会产生许多思考逻辑上的矛盾。换一个想法，如果以“时间点”的概念去理解步骤 i 的话，从仿真图看来是绝对逻辑的。（再唠叨的补充一下， p 空间的 [Width : 1] 是用来填入乘数 B，然而 p 空间的 [Width * 2 : Width + 1] 是用来执行和被乘数 A 的操作）

实验四结论：

这一章节笔记的重点不是要“如何实现一个算法”，而是以不同“步骤 i”的概念的，去完成 Booth 算法乘法器的改进。

1.8 LUT 乘法器

1.8 章节以前的乘法器都可以归纳为“慢速乘法器”的家族，当然它们不是真正意义上的慢，只不过它们无法达到急性一族人的任性而已。LUT 乘法器，又成为查表乘法器。用傻瓜的话来说，就是先吧各种各样的结果储存在一个表中，然后乘法的结果以“查表”的方式取得。

举个例子，笔者先建立一个 16×16 正值的查表：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
3	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45
4	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
5	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
6	0	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90
7	0	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105
8	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120
9	0	9	18	27	36	45	54	63	72	81	90	99	108	117	126	135
10	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150
11	0	11	22	33	44	55	66	77	88	99	110	121	132	143	154	165
12	0	12	24	36	48	60	72	84	96	108	120	132	144	156	168	180
13	0	13	26	39	52	65	78	91	104	117	130	143	156	169	182	195
14	0	14	28	42	56	70	84	98	112	126	140	154	168	182	196	210
15	0	15	30	45	60	75	90	105	120	135	150	165	180	195	210	225

假设 $A \times B$ ，它们均为 4 位，A 为 10，B 为 2，那么结果会是 20。查表乘法器之所以被称为快速乘法器，因为查表乘法器只要用些许的时钟，去查表就可以求得乘法的结果。反之，非查表乘法器需要许多的时钟用于乘法的运算，才能求得乘法的结果。如果 $A \times B$ ，它们均为 8 位，那么应该如何呢？难道再建立一个 256×256 乘法器！？这样会死人的。

不知道读者有没有听过 Quarter square 乘法查表呢？

$$(a + b)^2 - (a - b)^2 = a^2 + 2ab + b^2 - (a^2 - 2ab + b^2) \\ = 4ab$$

therefore:

$$ab = ((a + b)^2 / 4) - ((a - b)^2 / 4)$$

上边是 Quarter square 算法的公式，在公式的结束我们可以得到：

$$ab = ((a + b)^2)/4 - ((a - b)^2)/4$$

如果再进一步细分的话，无论是 $(a + b)^2/4$ 或者 $(a - b)^2/4$ ，经过幂运算后，得到的结果都是正值。为什么这么说呢？假设 a 和 b 的位宽都是 8 位的短整数的话，那么 $(127 + 127)^2/4 = (-127 - 127)^2/4$ 。我们可以得到一个这样的结论“ $(a + b)^2/4$ 或者 $(a - b)^2/4$ 使用同样的 $(C)^2/4$ 查表。

下面我们建立一个 C 的范围为 0 ~ 255，并且内容是 $(C)^2/4$ 的查表。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0	0	1	2	4	6	9	12
8	16	20	25	30	36	42	49	56
16	64	72	81	90	100	110	121	132
24	144	156	169	182	196	210	225	240
32	256	272	289	306	324	342	361	380
40	400	420	441	462	484	506	529	552
48	576	600	625	650	676	702	729	756
56	784	812	841	870	900	930	961	992
64	1024	1056	1089	1122	1156	1190	1225	1260
72	1296	1332	1369	1406	1444	1482	1521	1560
80	1600	1640	1681	1722	1764	1806	1849	1892
88	1936	1980	2025	2070	2116	2162	2209	2256

Addr	+0	+1	+2	+3	+4	+5	+6	+7
96	2304	2352	2401	2450	2500	2550	2601	2652
104	2704	2756	2809	2862	2916	2970	3025	3080
112	3136	3192	3249	3306	3364	3422	3481	3540
120	3600	3660	3721	3782	3844	3906	3969	4032
128	4096	4160	4225	4290	4356	4422	4489	4556
136	4624	4692	4761	4830	4900	4970	5041	5112
144	5184	5256	5329	5402	5476	5550	5625	5700
152	5776	5852	5929	6006	6084	6162	6241	6320
160	6400	6480	6561	6642	6724	6806	6889	6972
168	7056	7140	7225	7310	7396	7482	7569	7656
176	7744	7832	7921	8010	8100	8190	8281	8372
184	8464	8556	8649	8742	8836	8930	9025	9120

Addr	+0	+1	+2	+3	+4	+5	+6	+7
192	9216	9312	9409	9506	9604	9702	9801	9900
200	10000	10100	10201	10302	10404	10506	10609	10712
208	10816	10920	11025	11130	11236	11342	11449	11556
216	11664	11772	11881	11990	12100	12210	12321	12432
224	12544	12656	12769	12882	12996	13100	13225	13340
232	13456	13572	13689	13806	13924	14042	14161	14280
240	14400	14520	14641	14762	14884	15006	15129	15252
248	15376	15500	15625	15750	15876	16002	16129	16256

这个查表的寻址虽然是 0~255，但是实际上下限是 254 而已。因为我们知道两个短整数最大值相加仅有 $-127 + -127 = -254$ 或者 $127 + 127 = 254$ ，所以查表的 255 空间是一个没有用处的空间。

那么问题来了，如果短整数的最大取值范围是 $-127 \sim 127$ 而已，何来寄存 $-254 \sim 254$ 呢？在这里我们就涉及了“**整数从小容量向大容量的转换**”的问题。假设 C 是 9 位位宽的不正规整数，然而 A 和 B 都是 8 位位宽的正规整数，那么 $C = A + B$ 会是：

$C = A + B$	等价于	$C = \{ A[7], A \} + \{ B[7], B \}$
$A = 127 (0111\ 1111)$		$A = 127 (00111\ 1111)$
$B = 127 (0111\ 1111)$		$B = 127 (00111\ 1111)$
等价于		
$\begin{array}{r} A \quad 0111\ 1111 \\ B \quad 0111\ 1111 \\ \hline C \quad \underline{01111\ 1110} \end{array}$		$\begin{array}{r} A \quad 00111\ 1111 \\ B \quad 00111\ 1111 \\ \hline C \quad \underline{01111\ 1110} \end{array}$
$A = -127 (1000\ 0001)$		$A = -127 (11000\ 0001)$
$B = -127 (1000\ 0001)$		$B = -127 (11000\ 0001)$
等价于		
$\begin{array}{r} A \quad 1000\ 0001 \\ B \quad 1000\ 0001 \\ \hline C \quad \underline{10000\ 0010} \end{array}$		$\begin{array}{r} A \quad 11000\ 0001 \\ B \quad 11000\ 0001 \\ \hline C \quad \underline{10000\ 0010} \end{array}$

接下来，我们来看一看下面一段 Quarter square 乘法查表的核心功能代码：

```

reg [8:0]I1,I2;

case( i )
    0:
        begin
            I1 <= { A[7], A } + { B[7], B };          // C = A + B;
            I2 <= { A[7], A } + { ~B[7], (~B + 1'b1) }; // C = A - B;
            i <= i + 1'b1;
        end
    1: // 取正值
        begin
            I1 <= I1[8] ? ( ~I1 + 1'b1 ) : I1;
            I2 <= I2[8] ? ( ~I2 + 1'b1 ) : I2;
        end
endcase

```

```
i <= i + 1'b1;  
end
```

I1 和 I2 均为 9 位位宽。在步骤 0 的时候, I1 表示了 $C = A + B$, 相反的 I2 表示了 $C = A - B$ 。由于短整数的赋值采用补码的表示方式, 所以大大简化了正负转换的操作。

假设 $A = -1 (1111\ 1111)$, $B = -3 (1111\ 1101)$, 经过上面步骤 0 的操作:

$$\begin{aligned}I1 &= \{1\ 11111111\} + \{1\ 1111\ 1101\} = 1\ 1111\ 1100 (-4) \text{ 等价于 } I1 = -1 + -3 = -4 \\I2 &= \{1\ 11111111\} + \{0\ 0000\ 0011\} = 0\ 0000\ 0010 (2) \text{ 等价于 } I2 = -1 - (-3) = -1 + 3 = 2\end{aligned}$$

步骤 1 是 I1 和 I2 从负值转换为正值。

假设 $I1 = -4 (1\ 111\ 1100)$, $I2 = 2 (0\ 0000\ 0010)$, 经过步骤 1 的操作:

$$\begin{aligned}I1 &= 0\ 0000\ 0011 + 1 = 0\ 0000\ 0100; \\I2 &= 0\ 0000\ 0010;\end{aligned}$$

为什么在步骤 1 中, 要特意将负值转换为正值呢? 笔者在前面已经说过, 无论是 $(-C)^2$ 还是 $(C)^2$ 取得的结果都是一致。为了使两者 I1 和 I2 能共用相同的查表, 这是必须采取的步骤。

如果用 I1 和 I2 来表达 Quarter square 公式, 那么:

$$(|I1|^2 / 4) - (|I2|^2 / 4)$$

实验五：基于 Quarter square 的查表乘法器

首先笔者必须手动建立 0~255 关于 $(C)^2/4$ 结果的 lut_module.v (rom 模块)。因为用 Quartus II 建立 ip 的 rom 在仿真的时候, 很不给力很别扭, 为了避免诸多的麻烦, 自己手动建立的 rom 才是最明智的决策。

lut_module.v

```
1. module lut_module  
2. (  
3.     input CLK,  
4.     input RSTn,  
5.     input [7:0]Addr,  
6.     output [15:0]Q
```

```
8. );
9.
10. *****/
11.
12. reg [15:0]rQ;
13.
14. always @ ( posedge CLK or negedge RSTn )
15.     if( !RSTn )
16.         rQ <= 16'd0;
17.     else
18.         case( Addr )
19.
20.             0,1   : rQ <= 16'd0;
21.             2      : rQ <= 16'd1;
22.             3      : rQ <= 16'd2;
23.             4      : rQ <= 16'd4;
24.             5      : rQ <= 16'd6;
25.             6      : rQ <= 16'd9;
26.             7      : rQ <= 16'd12;
27.             8      : rQ <= 16'd16;
28.             9      : rQ <= 16'd20;
29.             10     : rQ <= 16'd25;
30.             11     : rQ <= 16'd30;
31.             12     : rQ <= 16'd36;
32.             13     : rQ <= 16'd42;
33.             14     : rQ <= 16'd49;
34.             15     : rQ <= 16'd56;
35.             16     : rQ <= 16'd64;
36.             17     : rQ <= 16'd72;
37.             18     : rQ <= 16'd81;
38.             19     : rQ <= 16'd90;
39.             20     : rQ <= 16'd100;
40.             21     : rQ <= 16'd110;
41.             22     : rQ <= 16'd121;
42.             23     : rQ <= 16'd132;
43.             24     : rQ <= 16'd144;
44.             25     : rQ <= 16'd156;
45.             26     : rQ <= 16'd169;
46.             27     : rQ <= 16'd182;
47.             28     : rQ <= 16'd196;
48.             29     : rQ <= 16'd210;
49.             30     : rQ <= 16'd225;
50.             31     : rQ <= 16'd240;
51.             32     : rQ <= 16'd256;
```

```
52.          33  :  rQ <= 16'd272;
53.          34  :  rQ <= 16'd289;
54.          35  :  rQ <= 16'd306;
55.          36  :  rQ <= 16'd324;
56.          37  :  rQ <= 16'd342;
57.          38  :  rQ <= 16'd361;
58.          39  :  rQ <= 16'd380;
59.          40  :  rQ <= 16'd400;
60.          41  :  rQ <= 16'd420;
61.          42  :  rQ <= 16'd441;
62.          43  :  rQ <= 16'd462;
63.          44  :  rQ <= 16'd484;
64.          45  :  rQ <= 16'd506;
65.          46  :  rQ <= 16'd529;
66.          47  :  rQ <= 16'd552;
67.          48  :  rQ <= 16'd576;
68.          49  :  rQ <= 16'd600;
69.          50  :  rQ <= 16'd625;
70.          51  :  rQ <= 16'd650;
71.          52  :  rQ <= 16'd676;
72.          53  :  rQ <= 16'd702;
73.          54  :  rQ <= 16'd729;
74.          55  :  rQ <= 16'd756;
75.          56  :  rQ <= 16'd784;
76.          57  :  rQ <= 16'd812;
77.          58  :  rQ <= 16'd841;
78.          59  :  rQ <= 16'd870;
79.          60  :  rQ <= 16'd900;
80.          61  :  rQ <= 16'd930;
81.          62  :  rQ <= 16'd961;
82.          63  :  rQ <= 16'd992;
83.          64  :  rQ <= 16'd1024;
84.          65  :  rQ <= 16'd1056;
85.          66  :  rQ <= 16'd1089;
86.          67  :  rQ <= 16'd1122;
87.          68  :  rQ <= 16'd1156;
88.          69  :  rQ <= 16'd1190;
89.          70  :  rQ <= 16'd1225;
90.          71  :  rQ <= 16'd1260;
91.          72  :  rQ <= 16'd1296;
92.          73  :  rQ <= 16'd1332;
93.          74  :  rQ <= 16'd1369;
94.          75  :  rQ <= 16'd1406;
95.          76  :  rQ <= 16'd1444;
```

```
96.          77  :  rQ <= 16'd1482;
97.          78  :  rQ <= 16'd1521;
98.          79  :  rQ <= 16'd1560;
99.          80  :  rQ <= 16'd1600;
100.         81  :  rQ <= 16'd1640;
101.         82  :  rQ <= 16'd1681;
102.         83  :  rQ <= 16'd1722;
103.         84  :  rQ <= 16'd1764;
104.         85  :  rQ <= 16'd1806;
105.         86  :  rQ <= 16'd1849;
106.         87  :  rQ <= 16'd1892;
107.         88  :  rQ <= 16'd1936;
108.         89  :  rQ <= 16'd1980;
109.         90  :  rQ <= 16'd2025;
110.         91  :  rQ <= 16'd2070;
111.         92  :  rQ <= 16'd2116;
112.         93  :  rQ <= 16'd2162;
113.         94  :  rQ <= 16'd2209;
114.         95  :  rQ <= 16'd2256;
115.         96  :  rQ <= 16'd2304;
116.         97  :  rQ <= 16'd2352;
117.         98  :  rQ <= 16'd2401;
118.         99  :  rQ <= 16'd2450;
119.        100  :  rQ <= 16'd2500;
120.        101  :  rQ <= 16'd2550;
121.        102  :  rQ <= 16'd2601;
122.        103  :  rQ <= 16'd2652;
123.        104  :  rQ <= 16'd2704;
124.        105  :  rQ <= 16'd2756;
125.        106  :  rQ <= 16'd2809;
126.        107  :  rQ <= 16'd2862;
127.        108  :  rQ <= 16'd2916;
128.        109  :  rQ <= 16'd2970;
129.        110  :  rQ <= 16'd3025;
130.        111  :  rQ <= 16'd3080;
131.        112  :  rQ <= 16'd3136;
132.        113  :  rQ <= 16'd3192;
133.        114  :  rQ <= 16'd3249;
134.        115  :  rQ <= 16'd3306;
135.        116  :  rQ <= 16'd3364;
136.        117  :  rQ <= 16'd3422;
137.        118  :  rQ <= 16'd3481;
138.        119  :  rQ <= 16'd3540;
139.        120  :  rQ <= 16'd3600;
```

```
140.          121 : rQ <= 16'd3660;
141.          122 : rQ <= 16'd3721;
142.          123 : rQ <= 16'd3782;
143.          124 : rQ <= 16'd3844;
144.          125 : rQ <= 16'd3906;
145.          126 : rQ <= 16'd3969;
146.          127 : rQ <= 16'd4032;
147.          128 : rQ <= 16'd4096;
148.          129 : rQ <= 16'd4160;
149.          130 : rQ <= 16'd4225;
150.          131 : rQ <= 16'd4290;
151.          132 : rQ <= 16'd4356;
152.          133 : rQ <= 16'd4422;
153.          134 : rQ <= 16'd4489;
154.          135 : rQ <= 16'd4556;
155.          136 : rQ <= 16'd4624;
156.          137 : rQ <= 16'd4692;
157.          138 : rQ <= 16'd4761;
158.          139 : rQ <= 16'd4830;
159.          140 : rQ <= 16'd4900;
160.          141 : rQ <= 16'd4970;
161.          142 : rQ <= 16'd5041;
162.          143 : rQ <= 16'd5112;
163.          144 : rQ <= 16'd5184;
164.          145 : rQ <= 16'd5256;
165.          146 : rQ <= 16'd5329;
166.          147 : rQ <= 16'd5402;
167.          148 : rQ <= 16'd5476;
168.          149 : rQ <= 16'd5550;
169.          150 : rQ <= 16'd5625;
170.          151 : rQ <= 16'd5700;
171.          152 : rQ <= 16'd5776;
172.          153 : rQ <= 16'd5852;
173.          154 : rQ <= 16'd5929;
174.          155 : rQ <= 16'd6006;
175.          156 : rQ <= 16'd6084;
176.          157 : rQ <= 16'd6162;
177.          158 : rQ <= 16'd6241;
178.          159 : rQ <= 16'd6320;
179.          160 : rQ <= 16'd6400;
180.          161 : rQ <= 16'd6480;
181.          162 : rQ <= 16'd6561;
182.          163 : rQ <= 16'd6642;
183.          164 : rQ <= 16'd6724;
```

```
184.          165 : rQ <= 16'd6806;
185.          166 : rQ <= 16'd6889;
186.          167 : rQ <= 16'd6972;
187.          168 : rQ <= 16'd7056;
188.          169 : rQ <= 16'd7140;
189.          170 : rQ <= 16'd7225;
190.          171 : rQ <= 16'd7310;
191.          172 : rQ <= 16'd7396;
192.          173 : rQ <= 16'd7482;
193.          174 : rQ <= 16'd7569;
194.          175 : rQ <= 16'd7656;
195.          176 : rQ <= 16'd7744;
196.          177 : rQ <= 16'd7832;
197.          178 : rQ <= 16'd7921;
198.          179 : rQ <= 16'd8010;
199.          180 : rQ <= 16'd8100;
200.          181 : rQ <= 16'd8190;
201.          182 : rQ <= 16'd8281;
202.          183 : rQ <= 16'd8372;
203.          184 : rQ <= 16'd8464;
204.          185 : rQ <= 16'd8556;
205.          186 : rQ <= 16'd8649;
206.          187 : rQ <= 16'd8742;
207.          188 : rQ <= 16'd8836;
208.          189 : rQ <= 16'd8930;
209.          190 : rQ <= 16'd9025;
210.          191 : rQ <= 16'd9120;
211.          192 : rQ <= 16'd9216;
212.          193 : rQ <= 16'd9312;
213.          194 : rQ <= 16'd9409;
214.          195 : rQ <= 16'd9506;
215.          196 : rQ <= 16'd9604;
216.          197 : rQ <= 16'd9702;
217.          198 : rQ <= 16'd9801;
218.          199 : rQ <= 16'd9900;
219.          200 : rQ <= 16'd10000;
220.          201 : rQ <= 16'd10100;
221.          202 : rQ <= 16'd10201;
222.          203 : rQ <= 16'd10302;
223.          204 : rQ <= 16'd10404;
224.          205 : rQ <= 16'd10506;
225.          206 : rQ <= 16'd10609;
226.          207 : rQ <= 16'd10712;
227.          208 : rQ <= 16'd10816;
```

```
228.          209  :  rQ <= 16'd10920;
229.          210  :  rQ <= 16'd11025;
230.          211  :  rQ <= 16'd11130;
231.          212  :  rQ <= 16'd11236;
232.          213  :  rQ <= 16'd11342;
233.          214  :  rQ <= 16'd11449;
234.          215  :  rQ <= 16'd11556;
235.          216  :  rQ <= 16'd11664;
236.          217  :  rQ <= 16'd11772;
237.          218  :  rQ <= 16'd11881;
238.          219  :  rQ <= 16'd11990;
239.          220  :  rQ <= 16'd12100;
240.          221  :  rQ <= 16'd12210;
241.          222  :  rQ <= 16'd12321;
242.          223  :  rQ <= 16'd12432;
243.          224  :  rQ <= 16'd12544;
244.          225  :  rQ <= 16'd12656;
245.          226  :  rQ <= 16'd12769;
246.          227  :  rQ <= 16'd12882;
247.          228  :  rQ <= 16'd12996;
248.          229  :  rQ <= 16'd13100;
249.          230  :  rQ <= 16'd13225;
250.          231  :  rQ <= 16'd13340;
251.          232  :  rQ <= 16'd13456;
252.          233  :  rQ <= 16'd13572;
253.          234  :  rQ <= 16'd13689;
254.          235  :  rQ <= 16'd13806;
255.          236  :  rQ <= 16'd13924;
256.          237  :  rQ <= 16'd14042;
257.          238  :  rQ <= 16'd14161;
258.          239  :  rQ <= 16'd14280;
259.          240  :  rQ <= 16'd14400;
260.          241  :  rQ <= 16'd14520;
261.          242  :  rQ <= 16'd14641;
262.          243  :  rQ <= 16'd14762;
263.          244  :  rQ <= 16'd14884;
264.          245  :  rQ <= 16'd15006;
265.          246  :  rQ <= 16'd15129;
266.          247  :  rQ <= 16'd15252;
267.          248  :  rQ <= 16'd15376;
268.          249  :  rQ <= 16'd15500;
269.          250  :  rQ <= 16'd15625;
270.          251  :  rQ <= 16'd15750;
271.          252  :  rQ <= 16'd15876;
```

```

272.          253  :  rQ <= 16'd16002;
273.          254  :  rQ <= 16'd16129;
274.          255  :  rQ <= 16'd16256;
275.
276.      endcase
277.
278.      /*****
279.
280.      assign Q = rQ;
281.
282.      *****/
283.
284.endmodule

```

这是笔者目前贴过最长的 .v 文件了 ...

lut_multiplier_module.v

这个模块的功能很简单。首先，先取得 $I1 = A + B$ ， $I2 = A - B$ ，然后 $I1$ 和 $I2$ 都正值呼，接下来将 $I1$ 和 $I2$ 送至各自的查表，再然后将得出的查表结果 $Q1_Sig$ ($I1$ 的结果) 和 $Q2_Sig$ ($I2$ 的结果) 执行相减。

我们知道硬件是不怎么适合作相减的操作，所以 $Q2$ 必须以负值的补码形式和 $Q1$ 相加。亦即 $Q1_Sig + (\sim Q2_Sig + 1'b1)$ 。那么 Quarter square 的公式可以重新修改为：

$$(a + b)^2/4 - (a - b)^2/4 = (|I1|)^2/4 + [(|I2|)^2/4]_{\text{补}} \\ = Q1_Sig + [Q2_Sig]_{\text{补}}$$

```

1. module lut_multiplier_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]A,
8.     input [7:0]B,
9.
10.    output Done_Sig,
11.    output [15:0]Product,
12.
13.    *****/
14.
15.    output [8:0]SQ_I1_Sig,

```

```

16.      output [8:0]SQ_I2_Sig,
17.      output [15:0]SQ_Q1_Sig,
18.      output [15:0]SQ_Q2_Sig
19.
20.      *****/
21.
22. );
23.
24. *****/
25.
26.     wire [15:0]Q1_Sig;
27.     wire [15:0]Q2_Sig;
28.
29. *****/
30.
31.     reg [3:0]i;
32.     reg [8:0]I1;
33.     reg [8:0]I2;
34.     reg [15:0]Data;
35.     reg isDone;
36.
37.     always @ ( posedge CLK or negedge RSTn )
38.         if( !RSTn )
39.             begin
40.                 i <= 4'd0;
41.                 I1 <= 9'd0;
42.                 I2 <= 9'd0;
43.                 Data <= 16'd0;
44.                 isDone <= 1'b0;
45.             end
46.         else if( Start_Sig )
47.             case( i )
48.
49.             0:
50.             begin
51.                 I1 <= { A[7], A } + { B[7], B };
52.                 I2 <= { A[7], A } + { ~B[7], ( ~B + 1'b1 ) };
53.                 i <= i + 1'b1;
54.             end
55.
56.             1:
57.             begin
58.                 I1 <= I1[8] ? ( ~I1 + 1'b1 ) : I1;
59.                 I2 <= I2[8] ? ( ~I2 + 1'b1 ) : I2;

```

```
60.          i <= i + 1'b1;
61.      end
62.
63.      2:
64.          begin i <= i + 1'b1; end
65.
66.      3:
67.          begin Data <= Q1_Sig + ( ~Q2_Sig + 1'b1 ); i <= i + 1'b1; end
68.
69.      4:
70.          begin isDone <= 1'b1; i <= i + 1'b1; end
71.
72.      5:
73.          begin isDone <= 1'b0; i <= 4'd0; end
74.
75.
76.
77.      endcase
78.
79. ****
80.
81. lut_module U1
82. (
83.     .CLK ( CLK ),
84.     .RSTn( RSTn ),
85.     .Addr ( I1[7:0] ),
86.     .Q ( Q1_Sig )
87. );
88.
89. ****
90.
91. lut_module U2
92. (
93.     .CLK ( CLK ),
94.     .RSTn( RSTn ),
95.     .Addr ( I2[7:0] ),
96.     .Q ( Q2_Sig )
97. );
98.
99. ****
100.
101. assign Done_Sig = isDone;
102. assign Product = Data;
103.
```

```

104.      *****/
105.
106.      assign SQ_I1_Sig = I1;
107.      assign SQ_I2_Sig = I2;
108.      assign SQ_Q1_Sig = Q1_Sig;
109.      assign SQ_Q2_Sig = Q2_Sig;
110.
111.      *****/
112.
113.endmodule

```

第 15~18 行是仿真的输出。

从 37~77 行是该模块的主功能。步骤 0 (49~54 行) 是取 I1 和 I2 的值。步骤 1 (56~61 行) 是 I1 和 I2 的正值化操作。步骤 2 (63~64 行) 是延迟一个时钟，给予足够的时间从 lut_module.v 读出结果。步骤 3 (66~67 行)，是 Quarter square 公式操作的最后一步，亦即 Q1_Sig - Q2_Sig。

89~99 行是 lut_module.v 的实例化，U1 是给 I1 使用的查表，U2 是给 I2 使用的查表，它们的输出连线分别是 Q1_Sig (26 行) 和 Q2_Sig (27 行)。102 行的 Product 信号由 Data 寄存器驱动。然而 106~109 行是仿真的输出，分别有 I1, I2, Q1_Sig 和 Q2_Sig。

lut_multiplier_module.vt

```

1.  `timescale 1 ps/ 1 ps
2.  module lut_multiplier_module_simulation();
3.
4.      reg CLK;
5.      reg RSTn;
6.
7.      reg Start_Sig;
8.      reg [7:0]A;
9.      reg [7:0]B;
10.
11.     wire Done_Sig;
12.     wire [15:0]Product;
13.
14.     *****/
15.
16.     wire [8:0]SQ_I1_Sig;
17.     wire [8:0]SQ_I2_Sig;
18.     wire [15:0]SQ_Q1_Sig;
19.     wire [15:0]SQ_Q2_Sig;
20.

```

```
21.      *****/
22.
23.      lut_multiplier_module U1
24.      (
25.          .CLK(CLK),
26.          .RSTn(RSTn),
27.          .Start_Sig(Start_Sig),
28.          .A(A),
29.          .B(B),
30.          .Done_Sig(Done_Sig),
31.          .Product(Product),
32.          .SQ_I1_Sig(SQ_I1_Sig),
33.          .SQ_I2_Sig(SQ_I2_Sig),
34.          .SQ_Q1_Sig(SQ_Q1_Sig),
35.          .SQ_Q2_Sig(SQ_Q2_Sig)
36.      );
37.
38.      *****/
39.
40.      initial
41.      begin
42.          RSTn = 0; #10; RSTn = 1;
43.          CLK = 0; forever #10 CLK = ~CLK;
44.      end
45.
46.      *****/
47.
48.      reg [3:0]i;
49.
50.      always @ ( posedge CLK or negedge RSTn )
51.          if( !RSTn )
52.              begin
53.                  i <= 4'd0;
54.                  Start_Sig <= 1'b0;
55.                  A <= 8'd0;
56.                  B <= 8'd0;
57.              end
58.          else
59.              case( i )
60.
61.                  0: // A = -127 , B = 127
62.                  if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
63.                  else begin A <= 8'b10000001; B <= 8'd127; Start_Sig <= 1'b1; end
64.
65.                  1: // A = 2 , B = - 4
```

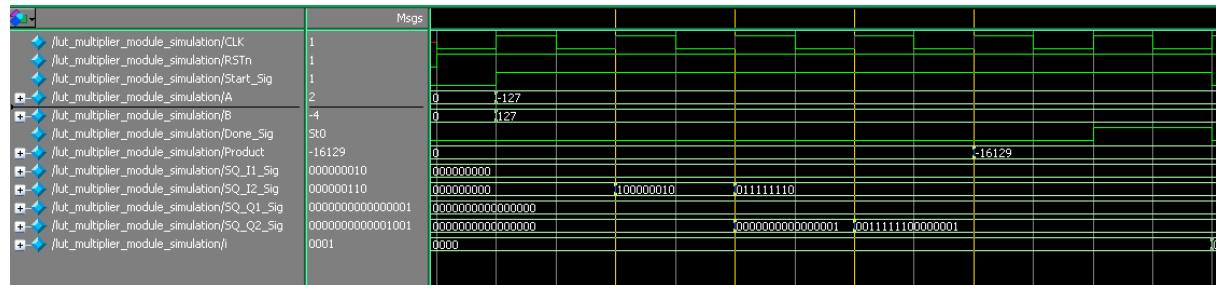
```

66.           if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
67.           else begin A <= 8'd2; B <= 8'b11111100; Start_Sig <= 1'b1; end
68.
69.           2: // A = 10 , B = 100
70.           if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
71.           else begin A <= 8'd10; B <= 8'd100; Start_Sig <= 1'b1; end
72.
73.           3: // A = -127 , B = -127
74.           if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
75.           else begin A <= 8'b10000001; B <= 8'b10000001; Start_Sig <= 1'b1; end
76.
77.           4:
78.           i <= 4'd4;
79.
80.       endcase
81.
82.
83.   endmodule

```

.vt 文件的写法和之前的实验都一样，如果真的不知道笔者在写什么，就得好好看笔者之前写的笔记。

仿真结果：



看吧！一次的乘法操作仅需 4 个时钟而已。比起改进的 Booth 算法减少了一半的时钟消耗。真不愧是查表式的乘法器，佩服佩服。

实验五结论：

说实话，查表式的乘法器是“以空间换时间”的乘法器，所以说查表式的乘法器是很消耗空间。到底有什么乘法器“可以节约空间，又节省时钟”呢？小知识：读者知道吗？传统查表的乘法器都有一个僵局，假设 $A \times B$ ，那么其中一个变量需要是“恒数”，否则查表的建立是很庞大的。但是 Quarter square 公式的出现把这个僵局给打破。

1.9 Modified Booth 算法乘法器

事先声明 modified booth 算法 和 改进的 booth 算法乘法器（实验四）是没有任何关系的。如字面上的意思 modified booth 算法是 booth 算法的升级版。我们稍微回味一下 booth 算法：

假设 B 是 4 位位宽的乘数，那么 booth 算法会对 $B[0:-1], B[1:0], B[2:1], B[3:2]$ 加码，而使得乘法运算得到简化。booth 算法有典型数学做法，也有位操作的做法。Modified booth 算法比起 booth 算法，对于 B 乘数的加码范围会更广，而使得 $n/2$ 运算次数的优化。再假设 B 是 4 位位宽的倍数，那么 modified booth 算法会对 $B[1:-1], B[3:1]$ 执行加码。



Modified Booth 乘数加码概念

如果站在位操作的角度上：

B[1]	B[0]	B[-1]	操作结果
0	0	0	无操作，右移两位
0	0	1	+被乘数，右移两位
0	1	0	+被乘数，右移两位
0	1	1	右移一位，+被乘数，右移一位
1	0	0	右移一位，-被乘数，右移一位
1	0	1	-被乘数，右移两位
1	1	0	-被乘数，右移两位
1	1	1	无操作，右移两位

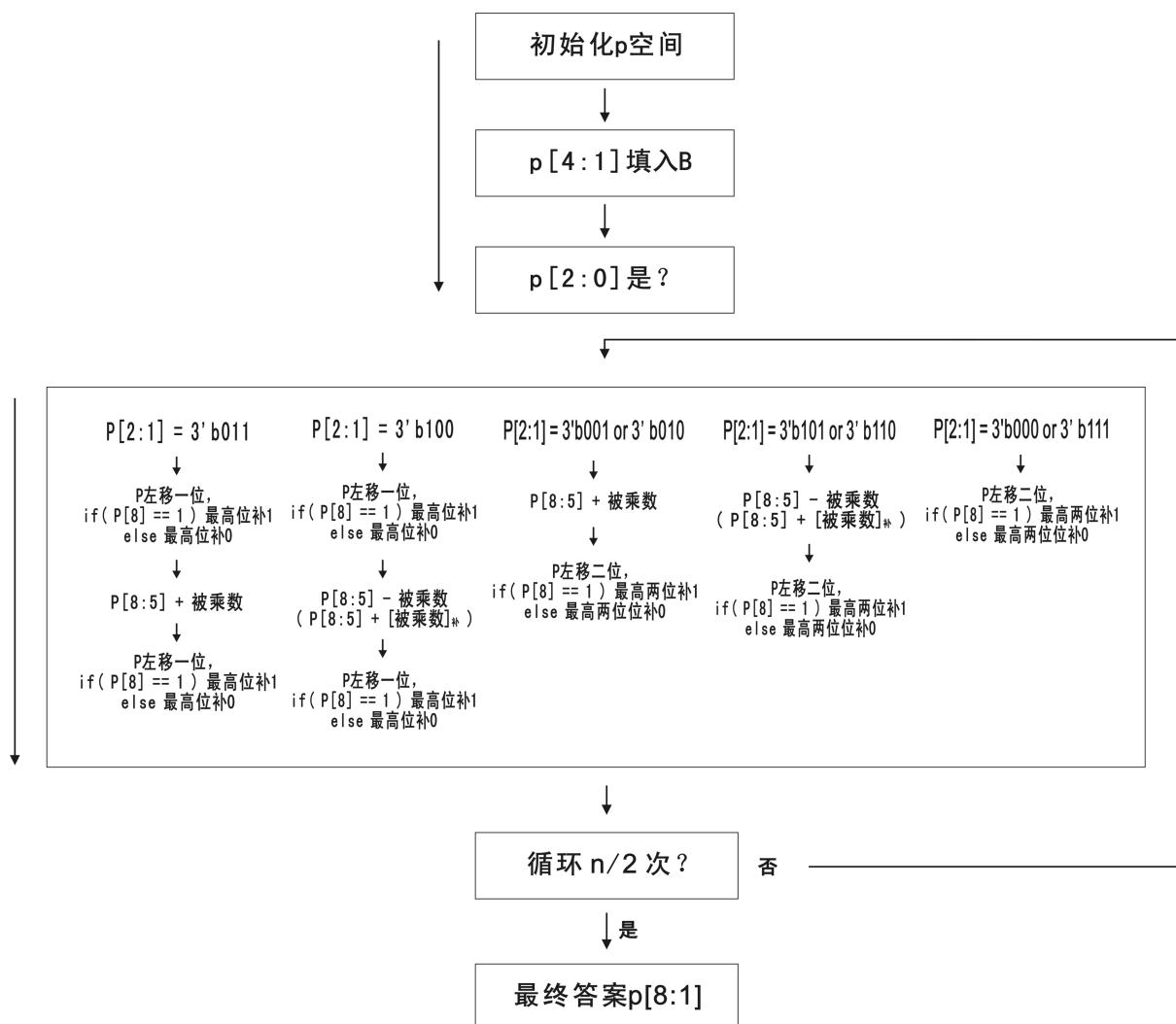
Modified booth 算法同样也有使用 p 空间，假设乘数 A，和被乘数 B，均为 4 位，那么 p 空间的大小 $n \times 2 + 1$ ，亦即 9 位。假设一个例子：乘数 A 为 7 (0111)，被乘数 B 为 2 (0010)。

1. 先求出 +被乘数 和 -被乘数，亦即 A 和 <u>A</u> 。	$A = 0111, \underline{A} = 1001$
2. P 空间初始化为 0，然后 P 空间的[4..1] 填入乘数	$P = 0000\ 0000\ 0$
3. 亦即 B。	$P = 0000\ 0010\ 0$
4. 先判断 p[2:0]，结果是 3'b100	$P = 0000\ 0010\ 0$
5. 亦即“右移一位，-被乘数，右移一位”。	
6. 右移一位	$P = 0000\ 0001\ 0$
7. p[8:5] 加上 <u>A</u>	$P = 0000\ 0001\ 0$

	$\begin{array}{r} + 1001 \\ \hline P = 1001\ 0001\ 0 \end{array}$
8. 右移一位	$p = 1100\ 1000\ 1$
9. 判断 $p[2:0]$, 结果是 3'b001 亦即“+被乘数, 右移二位”。	$p = 1100\ 1000\ 1$

10. $p[8:5]$ 加上 A	$\begin{array}{r} P = 1100\ 1000\ 1 \\ + 0111 \\ \hline P = 0011\ 1000\ 1 \end{array}$
11. 右移二位	$P = 0000\ 1110\ 0$
12. 最终取出 $p[8:1]$ 就是最终答案 8'b00001110 , 亦即 14。	$P = 0000\ 1110\ 0$

关于 4 位为位宽的乘数和被乘数操作流程图如下：



说实话 modified booth 算法的位操作是很不规则的，从上面的流程图可以看到，不同的 $p[2:0]$ 操作都有“不同的操作步骤”。

实验六：Modified Booth 乘法器

这个模块大致的操作如上述的流程图。

modified_booth_module.v

```
1. module modified_booth_multiplier_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]A,
8.     input [7:0]B,
9.
10.    output Done_Sig,
11.    output [15:0]Product,
12.
13.    /*****
14.
15.    output [7:0]SQ_a,
16.    output [7:0]SQ_s,
17.    output [16:0]SQ_p
18.
19.    *****/
20.
21. );
22.
23. *****/
24.
25. reg [3:0]i;
26. reg [7:0]a; // multipicand register
27. reg [7:0]s; // reverse result of rA
28. reg [16:0]p; // operation register
29. reg [3:0]X;
30. reg isDone;
31.
32. always @ ( posedge CLK or negedge RSTn )
33.     if( !RSTn )
34.         begin
35.             i <= 4'd0;
```

```
36.          a <= 8'd0;
37.          s <= 8'd0;
38.          p  <= 17'd0;
39.          X <= 4'd0;
40.          isDone <= 1'b0;
41.      end
42.  else if( Start_Sig )
43.    case( i )
44.
45.      0:
46.        begin
47.          a <= A;
48.          s <= ( ~A + 1'b1 );
49.          p <= { 8'd0 , B , 1'b0 };
50.          i <= i + 1'b1;
51.        end
52.
53.      1:
54.        if( X == 4 ) begin X <= 4'd0; i <= 4'd9; end
55.        else if( p[2:0] == 3'b001 || p[2:0] == 3'b010 ) begin p <= { p[16:9] + a , p[8:0] }; i <= i + 1'b1; end
56.        else if( p[2:0] == 3'b011 ) begin i <= 4'd3; end
57.        else if( p[2:0] == 3'b100 ) begin i <= 4'd6; end
58.        else if( p[2:0] == 3'b101 || p[2:0] == 3'b110 ) begin p <= { p[16:9] + s , p[8:0] }; i <= i + 1'b1; end
59.        else i <= i + 1'b1;
60.
61.      2:
62.        begin p <= { p[16] , p[16] , p[16:2] }; X <= X + 1'b1; i <= 4'd1; end
63.
64.      *****/
65.
66.      3:
67.        begin p <= { p[16] , p[16:1] }; i <= i + 1'b1; end
68.
69.      4:
70.        begin p <= { p[16:9] + a , p[8:0] }; i <= i + 1'b1; end
71.
72.      5:
73.        begin p <= { p[16] , p[16:1] }; X <= X + 1'b1; i <= 4'd1; end
74.
75.      *****/
76.
77.      6:
78.        begin p <= { p[16] , p[16:1] }; i <= i + 1'b1; end
79.
80.      7:
```

```

81.          begin p <= { p[16:9] + s , p[8:0] }; i <= i + 1'b1; end
82.
83.          8:
84.          begin p <= { p[16] , p[16:1] }; X <= X + 1'b1; i <= 4'd1; end
85.
86.          *****/
87.
88.          9:
89.          begin isDone <= 1'b1; i <= i + 1'b1; end
90.
91.          10:
92.          begin isDone <= 1'b0; i <= 4'd0; end
93.
94.          *****/
95.
96.
97.      endcase
98.
99.      *****/
100.
101.     assign Done_Sig = isDone;
102.     assign Product = p[16:1];
103.
104.     *****/
105.
106.     assign SQ_a = a;
107.     assign SQ_s = s;
108.     assign SQ_p = p;
109.
110.    *****/
111.
112.
113. endmodule

```

15~17 行是仿真的输出。43~94 行是该模块的主功能。在步骤 0 (45~51 行) 取得被乘数 A 并且寄存在 a 寄存器，此外取得 -1(被乘数 A) 并且寄存在 s 寄存器。在初始化 p 空间的同时，将乘数 B 填入 p[8:1]。

由于被乘数 A 和乘数 B 的位宽为 8，所以 p 空间是 $n \times 2 + 1$ 亦即 17。p 空间的 [Width : 1] 是用来填入乘数 B，然而 p 空间的 [Width * 2 : Width + 1] 是用来执行被乘数 A 的操作。

由于 modified booth 算法的关系，不同的 p[2:0] 结果都有不同的操作步骤。步骤 1 和 2 (53~62 行) 是 p[2:0] 等于 3'b000 | 111 | 001 | 010 | 101 | 110 的操作。步骤 3~5 (66~73 行) 是 p[2:0] 等于 3'b011 的操作 (56 行)。反之步骤 6~8 (77~84 行) 是针对 p[2:0] 3'b100

的操作 (57 行)。

步骤 9~10 是产生完成信号。第 102 行的 product 输出信号是由 p[16:1] 来驱动。第 106~109 的仿真输出信号，分别由寄存器 a，s 和 p 来驱动。

modified_booth_multiplier_module.vt

```
1. `timescale 1 ps/ 1 ps
2. module modified_booth_multiplier_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Start_Sig;
8.     reg [7:0]A;
9.     reg [7:0]B;
10.
11.    wire Done_Sig;
12.    wire [15:0]Product;
13.
14.    /*****
15.
16.    wire [7:0]SQ_a;
17.    wire [7:0]SQ_s;
18.    wire [16:0]SQ_p;
19.
20.    *****/
21.
22.    modified_booth_multiplier_module i1
23.    (
24.        .CLK(CLK),
25.        .RSTn(RSTn),
26.        .Start_Sig(Start_Sig),
27.        .A(A),
28.        .B(B),
29.        .Done_Sig(Done_Sig),
30.        .Product(Product),
31.        .SQ_a(SQ_a),
32.        .SQ_s(SQ_s),
33.        .SQ_p(SQ_p)
34.    );
35.
36.    *****/
37.
```

```
38.      initial
39.      begin
40.          RSTn = 0; #10; RSTn = 1;
41.          CLK = 0; forever #10 CLK = ~CLK;
42.      end
43.
44.      *****/
45.
46.      reg [3:0]i;
47.      reg [7:0]X;
48.      reg [7:0]Y;
49.
50.      always @ ( posedge CLK or negedge RSTn )
51.          if( !RSTn )
52.              begin
53.                  i <= 4'd0;
54.                  A <= 8'd0;
55.                  B <= 8'd0;
56.                  X <= 8'd0;
57.                  Y <= 8'd0;
58.                  Start_Sig <= 1'b0;
59.              end
60.          else
61.              case( i )
62.
63.                  0:
64.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
65.                      else begin A <= 8'd2; B <= 8'd4; Start_Sig <= 1'b1; end
66.
67.                  1:
68.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
69.                      else begin A <= 8'b11111100; B <= 8'd4; Start_Sig <= 1'b1; end
70.
71.                  2:
72.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
73.                      else begin A <= 8'd127; B <= 8'b10000001; Start_Sig <= 1'b1; end
74.
75.                  3:
76.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
77.                      else begin A <= 8'b10000001; B <= 8'b10000001; Start_Sig <= 1'b1; end
78.
79.                  4:
80.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
81.                      else begin A <= X; B <= Y; Start_Sig <= 1'b1; end
82.
```

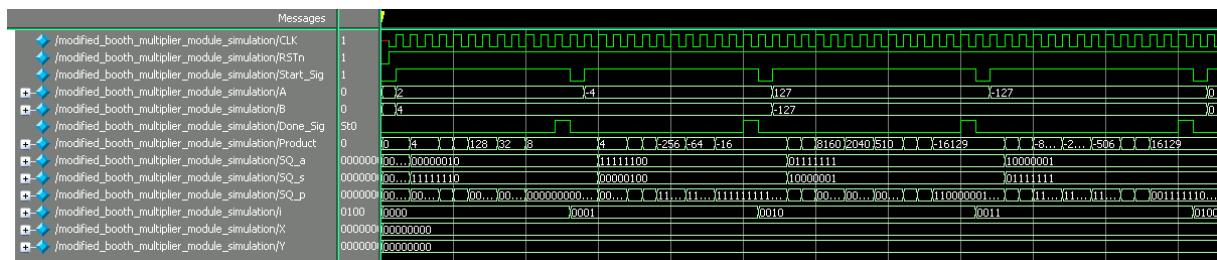
```

83.      5:
84.      if( X == 8'b01111111 ) begin X <= 8'd0; i <= i + 1'b1; end
85.      else if( Y == 8'b10000001 ) begin Y <= 8'd11111111; X <= X + 1'b1; end
86.      else begin Y <= Y + 8'b11111111; i <= i - 1'b1; end
87.
88.      6:
89.      i <= 4'd6;
90.
91.      endcase
92.
93.      ****
94.
95.
96.
97. endmodule

```

上面是激励文件，在步骤 5 加入了类似 for 嵌套循环的东西，以递增的值去刺激 .v 文件。写法都是大同小异 ~ 自己看着办吧。

仿真结果：



在仿真结果中，可以很明显的看到当 2(4) 和 127 (-127) 有明显的时钟消耗差异。

实验六结论：

Modified booth 算法用“位操作”虽然它可以提升乘法运算的速度（使用更少时钟），由于它的操作步骤是不规则的，很多时候实验五的乘法器是很别扭的。换句话说，用它还要图运气，因为不同的乘数和被乘数都有不同的时钟消耗

1.10 Modified Booth 乘法器 • 改

如果要把 Modified Booth 乘法器别扭的性格去掉，我们不得站在“数学的角度”去使用 modified booth 算法。下表是从数学的角度去使用 modified booth。

乘数B	0	0	0	1	0	1	1	0
-----	---	---	---	---	---	---	---	---

Modified Booth 乘数加码概念

B[1]	B[0]	B[n-1]	操作结果
0	0	0	无操作
0	0	1	+被乘数
0	1	0	+被乘数
0	1	1	+2(被乘数)
1	0	0	-2(被乘数)
1	0	1	-被乘数
1	1	0	-被乘数
1	1	1	无操作

我们假设 被乘数 A 和乘数 B 均为 4 位位宽 : A=7 (0111), B=2 (0010)。

在这里我们必须注意一下，4 位的被乘数 A 的取值范围最大是 $-7 \sim 7$ 。当 $B[1:-1]$ 等于 011 或者 100 的时候，然而，+2(被乘数) 或者 -2(被乘数) 都会使得 A 的最大值突破取值范围。所以需要从 4 位位宽的空间向更大的位位宽哦空间转换。这里就选择向 8 位位宽的空间转换吧。

$$A = (7) 0000\ 0111; \quad 2A = (14) 0000\ 1110; \quad -2A = (-14) 1111\ 0010.$$

B 乘数加码为 $B[1:-1] = 3'b100$ ，亦即 $-2(\text{被乘数})$ 和 $B[3:1] = 3'b100$ ，亦即 $+(\text{被乘数})$ 。

A	0 1 1 1	
B	0 0 1 0	<u>0</u>
	=====	
	+1 -2	B 乘数加码
	=====	
	1 1 1 1 0 0 1 0	
+ 0 0 0 0 0 1 1 1		<< 2 左移两位
	=====	
	1 0 0 0 0 1 1 1 0	无视超过 8 位最高位的益处
	=====	

还记得 booth 算法在数学角度上的运算吗？4 位的乘数和被乘数相乘，乘数必须加码 n 次，而且乘积也是 n 次，亦即 4 次加码操作，和 4 次的乘积操作。相反的 modified booth 算法在数学的角度上使用的话，4 位位宽的乘数和被乘数相乘，乘数加码为 $n/2$ 次，而且乘积也是 $n/2$ 的次数，亦即 2 次加码操作，和 2 次的乘积操作

实验七：Modified Booth 乘法器 • 改

modified_booth_multiplier_module_2.v

```
1. module modified_booth_multiplier_module_2
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]A,
8.     input [7:0]B,
9.
10.    output Done_Sig,
11.    output [15:0]Product,
12.
13.    /*****
14.
15.    output [15:0]SQ_a,
16.    output [15:0]SQ_a2,
17.    output [15:0]SQ_s,
18.    output [15:0]SQ_s2,
19.    output [3:0]SQ_i,
20.    output [8:0]SQ_N
21.
22.    *****/
23. );
24.
25. );
26.
27. *****/
28.
29. reg [3:0]i;
30. reg [15:0]a; // multipicand register
31. reg [15:0]a2;
32. reg [15:0]s; // reverse result of rA
33. reg [15:0]s2;
```

```

34.      reg [15:0]p; // operation register
35.      reg [3:0]M;
36.      reg [8:0]N;
37.      reg isDone;
38.
39.      always @ ( posedge CLK or negedge RSTn )
40.          if( !RSTn )
41.              begin
42.                  i  <= 4'd0;
43.                  a  <= 8'd0;
44.                  a2 <= 9'd0;
45.                  s  <= 8'd0;
46.                  s2 <= 9'd0;
47.                  p  <= 16'd0;
48.                  M <= 4'd0;
49.                  N  <= 9'd0;
50.                  isDone <= 1'b0;
51.              end
52.          else if( Start_Sig )
53.              case( i )
54.
55.                  0:
56.                      begin
57.                          a <= A[7] ? { 8'hFF ,A } : { 8'd0 ,A };
58.                          a2 <= A[7] ? { 8'hFF ,A + A } : { 8'd0 ,A + A };
59.                          s  <= ~A[7] ? { 8'hFF ,(~A + 1'b1) } : { 8'd0 ,(~A + 1'b1) };
60.                          s2 <= ~A[7] ? { 8'hFF ,(~A + 1'b1) + (~A + 1'b1) } : { 8'd0 ,(~A + 1'b1) + (~A + 1'b1) };
61.                          p <= 16'd0;
62.                          M <= 4'd0;
63.                          N <= { B ,1'b0 };
64.                          i <= i + 1'b1;
65.                      end
66.
67.                  1,2,3,4:
68.                      begin
69.
70.                          if( N[2:0] == 3'b001 || N[2:0] == 3'b010 ) p <= p + ( a << M );
71.                          else if( N[2:0] == 3'b011 ) p <= p + ( a2 << M );
72.                          else if( N[2:0] == 3'b100 ) p <= p + ( s2 << M );
73.                          else if( N[2:0] == 3'b101 || N[2:0] == 3'b110 ) p <= p + ( s << M );
74.
75.                          M <= M + 2'd2;
76.                          N <= ( N >> 2 );
77.                          i <= i + 1'b1;
78.
```

```
79.           end
80.
81.           5:
82.           begin isDone <= 1'b1; i <= i + 1'b1; end
83.
84.           6:
85.           begin isDone <= 1'b0; i <= 4'd0; end
86.
87.       endcase
88.
89.   /*****
90.
91.   assign Done_Sig = isDone;
92.   assign Product = p;
93.
94.   *****/
95.
96.   assign SQ_a = a;
97.   assign SQ_a2 = a2;
98.   assign SQ_s = s;
99.   assign SQ_s2 = s2;
100.  assign SQ_i = i;
101.  assign SQ_N = N;
102.
103. *****/
104.
105. endmodule
```

第 29~27 行是该模块所使用的寄存器。a 是用来寄存 A，a2 是用来寄存 2A，s 是用来寄存 -A，s2 是用来寄存 -2A。M 是用来表示每次乘积的偏移量。

由于这个实验不是站在位操作的角度上，所以 P 空间仅是作为累加空间的存在。寄存器 N 用来判别 booth 加码操作，所以寄存器 N 用于寄存乘数 B 的值。乘数 B 是 8 位位宽，所以 N 空间的大小是 “乘数 B 的大小 + 1”。多出来的 1 个空间是用来寄存 B[-1] 的值。

在步骤 0 (55~65 行)，是用来初始化所有相关的寄存器。寄存器 a, a2, s, s2 在初始化的同时也进行 8 位 向 16 位 空间转换。寄存器 p 和 M 都清零，至于寄存器 N[8:1] 是用来填充乘数 B, N[0] 填入 0 值。

步骤 1~4 (67~79)，也就是 4 次的乘积次数，因为受到 n/2 的关系。每一次的乘积操作都是先判别 N[2:0]，然后累加相关的值。

我们知道传统的乘法，每一次的乘积操作，都有偏移量。打个比方： $123_{10} * 1111_{10}$

123	
1111	
=====	
123	<= 十进制的第一个乘积是 偏移 0，没有左移位操作。
123	<= 十进制的第二个乘积是 偏移 10，也就是左移 1 位。
123	<= 十进制的第三个乘积是 偏移 100，也就是左移 2 位。
123	<= 十进制的第四个乘积是 偏移 1000，也就是左移 3 位。
=====	

同样的道理，寄存器 M 是用于记录二进制的每一次乘积的偏移量，但是 modified booth 乘法的乘积偏移量是普通 2 进制乘法乘积偏移量的 2 倍。也就是说，每一次乘积操作结束都必须左移+2。

至于寄存器 N 它寄存了 $B[7:0] + B[-1]$ 的值。然而每一次的加码判别都是 $N[2:0]$ ，所以每一次的乘积之后，N 都需要右移两位。

假设 $B = 1101\ 0010$ ，N 必然是 $1101\ 0010\ 0$ 。

乘积 1	乘积 2	乘积 3	乘积 4
$B[1:-1] = 100$	$B[3:1] = 001$	$B[5:3] = 010$	$B[7:5] = 110$
$N = 1101\ 0010\ 0$	$N = 0011\ 0100\ 1$	$N = 0000\ 1101\ 0$	$N = 0000\ 0011\ 0$

为什么 8 位位宽的数据相乘，乘积运算次数是 $n / 2$ ，亦即 4 呢？这是 Modified booth 算法的一个特点。如果站在数学的角度上，Modified booth 算法可以节省“乘积次数 / 2”。

第 92 行的 product 输出是由寄存器 p 驱动。前面笔者说过了，如果站在数学的角度，p 空间只是累加空间的作用而已。然而 p 空间的大小是“乘数和被乘数位宽大小的相加”。

第 96~101 行是仿真信号的驱动。有一点值得注意的是，除了寄存 a, a2, s, s2 和 N 以外，笔者还故意将该步骤 i 引出，这是为了观察“Modified booth 乘法使得乘积次数减半”这一事实。

modified_booth_multiplier_module_2.vt

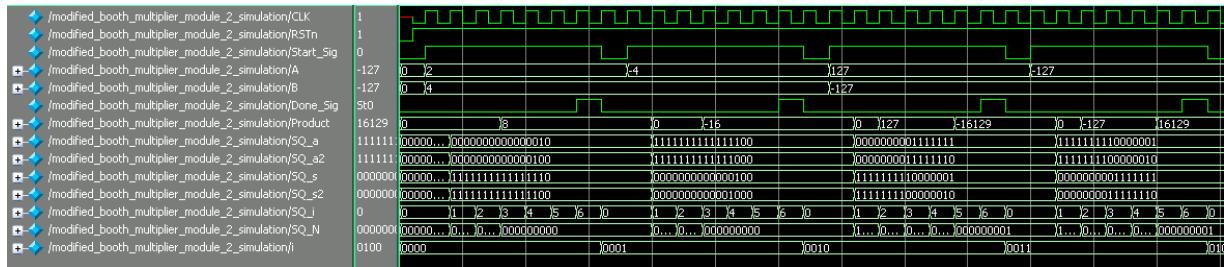
```

1. `timescale 1 ps/ 1 ps
2. module modified_booth_multiplier_module_2_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
```

```
6.  
7.      reg Start_Sig;  
8.      reg [7:0]A;  
9.      reg [7:0]B;  
10.  
11.     wire Done_Sig;  
12.     wire [15:0]Product;  
13.  
14.     /*****  
15.  
16.     wire [15:0]SQ_a;  
17.     wire [15:0]SQ_a2;  
18.     wire [15:0]SQ_s;  
19.     wire [15:0]SQ_s2;  
20.     wire [3:0]SQ_i;  
21.     wire [8:0]SQ_N;  
22.  
23.     /*****  
24.  
25. modified_booth_multiplier_module_2 U1  
26.  (  
27.      .CLK(CLK),  
28.      .RSTn(RSTn),  
29.      .Start_Sig(Start_Sig),  
30.      .A(A),  
31.      .B(B),  
32.      .Done_Sig(Done_Sig),  
33.      .Product(Product),  
34.      .SQ_a(SQ_a),  
35.      .SQ_a2(SQ_a2),  
36.      .SQ_s(SQ_s),  
37.      .SQ_s2(SQ_s2),  
38.      .SQ_i(SQ_i),  
39.      .SQ_N(SQ_N)  
40.  );  
41.  
42.  /*****  
43.  
44. initial  
45. begin  
46.     RSTn = 0; #10; RSTn = 1;  
47.     CLK = 0; forever #10 CLK = ~CLK;  
48. end  
49.  
50.  *****/
```

```
51.  
52.      reg [3:0]i;  
53.  
54.      always @ ( posedge CLK or negedge RSTn )  
55.          if( !RSTn )  
56.              begin  
57.                  i <= 4'd0;  
58.                  A <= 8'd0;  
59.                  B <= 8'd0;  
60.                  Start_Sig <= 1'b0;  
61.              end  
62.          else  
63.              case( i )  
64.  
65.                  0:  
66.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
67.                      else begin A <= 8'd2; B <= 8'd4; Start_Sig <= 1'b1; end  
68.  
69.                  1:  
70.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
71.                      else begin A <= 8'b11111100; B <= 8'd4; Start_Sig <= 1'b1; end  
72.  
73.                  2:  
74.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
75.                      else begin A <= 8'd127; B <= 8'b10000001; Start_Sig <= 1'b1; end  
76.  
77.                  3:  
78.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end  
79.                      else begin A <= 8'b10000001; B <= 8'b10000001; Start_Sig <= 1'b1; end  
80.  
81.                  4:  
82.                      i <= 4'd4;  
83.  
84.              endcase  
85.  
86.      /*****  
87.  
88.  
89. endmodule
```

仿真结果：



从仿真结果上，我们可以看到，每一个乘法操作都消耗同样的时钟。此外还有一点，当 SQ_i 等于 4 之后，就会得到正确的答案。这也证明说，Modified booth 算法使得乘积的次数优化位 $n/2$ 。

实验七结论：

实验七和实验六相比，实验七乘法运算更快了 1 倍了。所以我们可以承认 modified booth 算法确实拥有减半乘积次数的特点。

总结:

从实验一到实验七当中，笔者详细描述出四种乘法器的各种千秋，其中还有几种乘法器笔者还特意去优化它们。从四种乘法器之中“传统乘法器，Booth 乘法器，LUT 查表乘法器，和 Modified Booth 乘法器”，LUT 乘法器拥有最少的时钟消耗（最快的运算速度），但是 LUT 乘法器却暴露出消耗资源的弱点。

如果将 LUT 乘法器排外，自然而然 Modified Booth 乘法器成为第二候选人，但是要建立 Modified Booth 乘法器需要很好的理论基础，故很多新手都很怕它。至于 Booth 乘法和是最受欢迎的，如果设计的要求不像 DSP 那么任性，估计会有很多人喜欢它，因为它中庸，简单，容易亲近。

剩下的传统的乘法器，它什么都不比上后者，难道我们就要鄙视它吗？这个不然，笔者可以成功接触各种各样的乘法，都是托它的福，不然笔者是不可能如此深入研究整数乘法器。传统的乘法器，最主要的功能是传达“乘法运算”的概念。整数乘法器所涉及的知识可真不小，Verilog HDL 语言的基础先姑且不说，而且还涉及诸如补码，整数的表示方法，不同位空间的整数转换等等 … 都是一些电脑基础的知识。

“步骤”对于 Verilog HDL 语言的建模是绝对有帮助（建模篇已经秀得非常清楚了），尤其是对“多步骤”的模块而言。虽然状态机也可以实现多状态来完成上述的乘法运算，但是结果会使得“模块表达能力”和“代码结构”变成不堪入目，这是笔者不敢想象的后果。

此外在这一章笔记里，笔者所强调的“步骤”除了是“把大象放进冰箱”的概念以外，（把大象放进冰箱”是认识“步骤”的一个假象，但是这个假象不会对建模造成很大的影响。）“**步骤**”真正概念是**“时间点”**。在这里笔者非建议读者把“步骤”看成“时间点”。因为这样的理解方式，对往后“模块的沟通”又或者“波形图”的理解都有很大的帮助。

第二章：整数除法器

2.1 传统的除法器

整数除法器没有像整数乘法器那样丰富的种类，整数除法器的分类仅有传统型的循环型之分。老实说笔者也真的有点郁闷，翻了很多文章，论文，参考书，然后再衡量与 Verilog HDL 语言，笔者得到的都是零碎的线索。说一句真心话，真的真的有够郁闷“好想抽根烟，看夕阳”这样的心情。

时光又回到笔者的小学时候，在数学这门课中，笔者最喜欢就是减法，最讨厌就是除法。喜欢减法的原因，因为小学的减法没有整数的概念，任何被减数小于减数都是零，所以笔者特别钟爱。但是当数学课本出现“除法”的字眼，小学的笔者忽然间觉得“它”很碍眼。

传统除法的概念和传统乘法的概念是一样的，乘法是累加过程，反之除法是递减过程，直到被除数小于除数。但是那时候的笔者不晓得“只要有减法就能攻略除法”这一知识。

先来简单的扫盲：除数 Divisor，被除数 Dividend，商数 Quotient，余数 Reminder。

再来是苹果的故事：

假设有一只 72HP（被除数）的苹果怪兽，勇者 akuei2，一次的攻击是减少 10 HP（除数）。但是勇者它不杀生，当苹果怪兽余下的 HP 小于勇者的攻击容量，勇者的攻击就结束。

勇者对苹果怪兽攻击的次数	苹果上一次的 HP 余积	苹果下一次的 HP 余积
勇者的第 1 次对苹果怪兽的攻击	72	62
勇者的第 2 次对苹果怪兽的攻击	62	52
勇者的第 3 次对苹果怪兽的攻击	52	42
勇者的第 4 次对苹果怪兽的攻击	42	32
勇者的第 5 次对苹果怪兽的攻击	32	22
勇者的第 6 次对苹果怪兽的攻击	22	12
勇者的第 7 次对苹果怪兽的攻击	12	2
攻击结束。	2	

在上面“攻击过程中”，勇者一共攻击了 7 次，那么苹果怪兽也伤（商数）了 7 次，苹果怪兽余下（余数）的 HP 是 2。

上面有关勇者大战苹果怪兽的故事就是传统除法器的概念。反之，在我们小时候所学习的除法方式有莫大的差别，那是一种“心算”的除法方式。我们必须清楚，计算器是一个笨蛋，硬件比计算机更笨蛋，它什么都不会。

硬件除法器和硬件乘法器同样也会遇上“除数和被除数的正负关系”的问题。

```
13 / 2 = 商 6 , 余 1
-13 / 2 = 商 -6, 余 1 -1
13 / -2 = 商 -6, 余 1
-13 / -2 = 商 6, 余 1 -1
```

```
4 - 2 = 4 + 2 补
```

除数和被除数的正负关系反映了它们是“异或”的关系。为了设计的方便，在作除法的时候被除数取正值，除数取负值的补码，因为传统的除法就是被除数递减的概念。此外最后求出的商是负值还是正值，是根据原来除数和原来被除数的正负关系作决定。

实验八：传统除法器

传统除法器的设计非常单纯：

- 一、先取除数和被除数的正负关系，然后正值化被除数。传统除法器因为需要递减的关系，所以除数取负值和补码形式。
- 二、被除数递减与除数，每一次的递减，商数递增。
- 三、直到被除数小于除数，递减过程剩下的是余数。
- 四、输出的结果根据除数和被除数的正负关系。

divider_module.v

```
1. module divider_module
2. (
3.
4.     input CLK,
5.     input RSTn,
6.
7.     input Start_Sig,
8.     input [7:0]Dividend,
9.     input [7:0]Divisor,
10.
11.    output Done_Sig,
12.    output [7:0]Quotient,
```

```
13.      output [7:0]Reminder
14.
15. );
16.
17. ****
18.
19. reg [3:0]i;
20. reg [7:0]Dend;
21. reg [7:0]Dsor;
22. reg [7:0]Q;
23. reg [7:0]R;
24. reg isNeg;
25. reg isDone;
26.
27. always @ ( posedge CLK or negedge RSTn )
28.     if( !RSTn )
29.         begin
30.             i <= 4'd0;
31.             Dend <= 8'd0;
32.             Dsor <= 8'd0;
33.             Q <= 8'd0;
34.             isNeg <= 1'b0;
35.             isDone <= 1'b0;
36.         end
37.     else if( Start_Sig )
38.         case( i )
39.
40.             0:
41.                 begin
42.                     Dend <= Dividend[7] ? ~Dividend + 1'b1 : Dividend;
43.                     Dsor <= Divisor[7] ? Divisor : ( ~Divisor + 1'b1 );
44.                     isNeg <= Dividend[7] ^ Divisor[7];
45.                     Q <= 8'd0;
46.                     i <= i + 1'b1;
47.                 end
48.
49.             1:
50.                 if( Divisor > Dend ) begin Q <= isNeg ? ( ~Q + 1'b1 ) : Q; i <= i + 1'b1; end
51.                 else begin Dend <= Dend + Dsor; Q <= Q + 1'b1; end
52.
53.             2:
54.                 begin isDone <= 1'b1; i <= i + 1'b1; end
55.
56.             3:
57.                 begin isDone <= 1'b0; i <= 4'd0; end
```

```
58.  
59.         endcase  
60.  
61.     /*****  
62.  
63.     assign Done_Sig = isDone;  
64.     assign Quotient = Q;  
65.     assign Reminder = Dend;  
66.  
67.     /*****  
68.  
69. endmodule
```

第 7~13 行是该模块的输入和输出，采用仿顺序操作的结构（Start_Sig 和 Done_Sig）。第 19~25 行是该模块所使用的寄存器。Dend 用来寄存被除数的正值，Dsor 用来寄存除数负值的补码，Q 用来寄存商数，isNeg 用来寄存除数和被除数的正负关系。

在步骤 0 (40~47 行)，取得被除数的正值和除数负值的补码 (42~43 行)，44 行取得除数和被除数的正负关系。在 45 行清理 Q 寄存器后，i 递增以示下一个步骤。

在步骤 1 (49~51 行)，第 51 行表示了除法的递减操作。被除数每一次被除数递减，Q 寄存器都会递增。第 50 行的 if 条件用于判断“除法操作是否已经结束？”（当被除数小于除数）。当第 50 行的 if 条件成立后，根据 isNeg 的逻辑（除数和被除数的正负关系）来决定 Q 的赋值结果是正值还是负值，最后步骤 i 递增。

步骤 2~3 是产生完成信号 (53~57 行)。

divider_module.vt

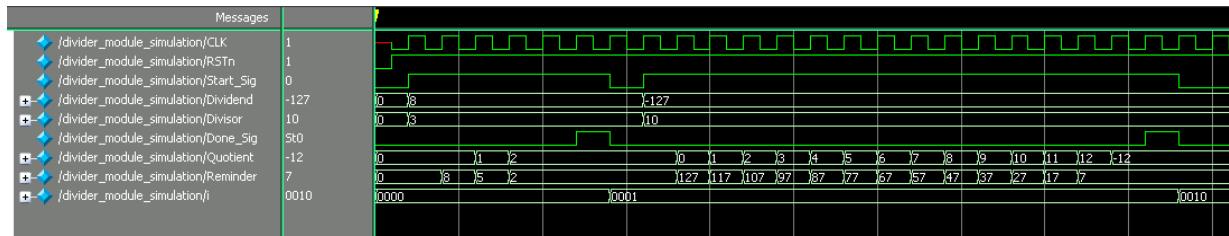
```
1. `timescale 1 ps/ 1 ps  
2. module divider_module_simulation();  
3.  
4.     reg CLK;  
5.     reg RSTn;  
6.  
7.     reg Start_Sig;  
8.     reg [7:0]Dividend;  
9.     reg [7:0]Divisor;  
10.  
11.    wire Done_Sig;  
12.    wire [7:0]Quotient;  
13.    wire [7:0]Reminder;  
14.  
15.    /*****
```

```
16.
17.      divider_module U1
18.      (
19.          .CLK(CLK),
20.          .RSTn(RSTn),
21.          .Start_Sig(Start_Sig),
22.          .Dividend(Dividend),
23.          .Divisor(Divisor),
24.          .Done_Sig(Done_Sig),
25.          .Quotient(Quotient),
26.          .Reminder(Reminder)
27.      );
28.
29.      *****/
30.
31.      initial
32.      begin
33.          RSTn = 0; #10; RSTn = 1;
34.          CLK = 0; forever #10 CLK = ~CLK;
35.      end
36.      *****/
37.
38.      reg [3:0]i;
39.
40.      always @ ( posedge CLK or negedge RSTn )
41.          if( !RSTn )
42.              begin
43.                  i <= 4'd0;
44.                  Dividend <= 8'd0;
45.                  Divisor <= 8'd0;
46.                  Start_Sig <= 1'b0;
47.              end
48.          else
49.              case( i )
50.
51.                  0: // Dividend = 8 , Divisor = 3
52.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
53.                      else begin Dividend <= 8'd8; Divisor <= 8'd3; Start_Sig <= 1'b1; end
54.
55.                  1: // Dividend = -127 , Divisor = 10
56.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
57.                      else begin Dividend <= 8'b10000001; Divisor <= 8'd10; Start_Sig <= 1'b1; end
58.
59.                  2:
```

```
60.           i <= 4'd2;
61.
62.       endcase
63.
64.   endmodule
```

.vt 文件的很简单，笔者就不多说了，自己看着办吧。

仿真结果：



仿真结果如上。哎~仿真图暴露了传统除法器的弱点。不一样的被除数，就有不一样的时钟消耗。

实验八说明：

无论是传统的乘法器还是传统的除法器。它们都有不好的一面，只要除数（乘数）稍微大了一点，它就要“多吃”时钟。但是传统的乘法器还好，还可以优化，相反的传统的除法器就有点不妙了 ... 它无法改进也无法优化。

实验八结论：

传统除法器的好处就是简单，但是应用能力却非常低。因为它对时钟的消耗是根据除数的“数量”来作决定 当除数的“数量”越大，它越消耗时钟。从另一个角度来说，传统除法器和传统乘法器都有同样的弱点，就是运算不规则。

2.2 循环型除法器

循环型的除法器，如果用笔者的话来说，就是位操作的除法器。循环型的除法器是典型的硬件除法器，假设除数和被除数的位宽为 N 位，那么除法器就需要循环 N 次完成除法操作，结果取得“N 位商和 N 位余”。

典型的循环型除法器，有分为可恢复和不可恢复。我们知道无论是乘法器还是除法器，都有一个操作空间，对吧？所谓可恢复，当它触发某种条件，操作空间的值又恢复到初始化的状态，反之不可恢复的意思则是相反。

循环型的除法器，可恢复也好还是不可恢复也好 ... 这一点也不重要，因为在 Verilog HDL 语言的眼里，也是几段代码的故事。最重要的是那一种最“养眼”，笔者就选谁。循环型的除法器的种类在网络上尽是千奇百怪。笔者在很偶然的情况下，发现其中一种的循环型除法器。故，是老外的东西，话说老外真的不简单。为了尊重原创者，就直接引用该除法器的原名字吧！Streamlined divider 。

该除法器的思路很简单：

假设被除数 A = 7 (0111)，除数 B = 2 (0010)，它们均为 4 位位宽。那么操作空间就是 Temp 就是 $2 * \text{Width}$ 。Temp[Width - 1 : 0] 是用来填充被除数，Temp[Width * 2 - 1 : Width - 1] 是用与除数递减操作。为了方便操作，我们建立 5 位位宽的 s 空间用来寄存除数 B 的负值补码形式。此外还要考虑，把移位操作和除法运算压缩在同一个步骤里面。

```
reg [7:0]Temp;
reg [7:0]Diff;
reg [4:0}s;
```

Temp 是操作空间，Diff 是临时操作空间（求得即时结果），s 是用来寄存除数 B 的负值（补码形式）。

```
Temp <= { 4'd0 , A };
s <= B[3] ? { B[3] , B } ? { ~B[3] , ~B + 1'b0 }; //如果除数 B 为负值，就直接填入，
//否则转换为负值后填入。
Diff <= 8'd0;
```

被

首先初始化 Temp 空间和 s 空间。Temp 空间的[Width - 1 : 0] 填入 被除数 A，然而 s 空间用于寄存 除数 B 负值的补码形式。最后顺便清零 Diff 空间。

接下来的动作，是核心的部分。

```
case( i )
```

```
.....  
1,2,3,4: // 因为除数 B 和被除数 A 的位宽均为 4 位，所以循环 4 次。  
begin
```

```
Diff = Temp + { s , 3'b0 } ; // “=” 表示当前步骤取得结果
```

```
if( Diff[7] ) Temp <= { Temp [6:0], 1'b0 };  
else Temp <= { Diff[6:0] , 1'b1 };
```

```
i <= i + 1'b1;
```

```
end
```

每一次的操作，

- 一、在 Diff 空间先取 Temp[Width * 2 - 1 : Width - 1] - B 或者 Temp[7:3] + s 的即时结果。
- 二、然后判断 Diff 空间的“最高位”，亦即符号位，是逻辑 1 还是逻辑 0。
- 三、如果是逻辑 1，Temp 空间左移一位，最低位补 0。
反之如果是逻辑 0，Temp 空间被赋予 Diff 的即时结果，并且左移一位，最低位补 1。

当经过 4 次的循环操作后。Temp 空间的 [Width - 1 : 0] 是商数，[Width * 2 - 1 : Width] 是余数。在这里“=”赋值运算符是重点，Diff 要求取得 Temp[7:3] + s 的即时结果。

```
assign Quotient = Temp[3:0];  
assign Reminder = Temp[7:4];
```

Quotient 和 Reminder 的输出驱动分别是 Temp[3:0] 和 Temp[7:4]。

我们先假设一个情况，除数 A = 7 (0111)，除数 B = 2 (0010)，它们均为 4 位位宽。

以下是操作过程：

1. 初始化	<p>$\text{Temp} = 0000\ 0111$</p> <p>$s = 1110$</p> <p>$\text{Diff} = 0000\ 0000$</p>
2. 第一次操作 $\text{Diff} = \text{Temp} + \{ s , 3'b0 \} ;$	<p>$\text{Diff} = 0000\ 0000 + 1110\ 000 \rightarrow 0000\ 0111$</p> <p>$= 1111\ 0000$</p>
3. 判断 $\text{Diff}[7]$ ，是逻辑 1， Temp 左移一位然后，最低位补 0。	<p>$\text{Temp} = 0000\ 1110$</p> <p>$0000\ 0111$ 左移 1 位 补 0</p>
4. 第二次操作 $\text{Diff} = \text{Temp} + \{ s , 3'b0 \} ;$	<p>$\text{Diff} = 0000\ 1110 + 1110\ 000$</p> <p>$= 1111\ 1110$</p>

此时，上一个步骤的决定全部生效，即
 $\text{Temp} = 0000\ 1110$
 $s = 1\ 1110$
 $\text{Diff} = 1111\ 0111$

第二次操作后决定
 Temp = 0001 1100
 S = 1 1110
 Diff = 1111 1110
 即第三次操作的当前值

5. 判断 Diff[7]，是逻辑 1， Temp 左移一位然后，最低位补 0。	Temp = 0001 1100
6. 第三次操作 Diff = Temp + { s, 3'b0 } ;	第三次操作后决定 Temp = 0001 1001 S = 1 1110 Diff = 0000 1100 即第四次操作的当前值
7. 判断 Diff[7]，是逻辑 0， Temp 赋予 Diff，然后左移一位，最低位补 1。	Diff = 0001 1100 + 11110 000 = 0000 1100 Temp = Diff = 0001 1001
8. 第四次操作 Diff = Temp + { s, 3'b0 } ;	Diff = 0001 1001 + 11110 000 = 0000 1001
9. 判断 Diff[7]，是逻辑 0， Temp 赋予 Diff，然后左移一位，最低位补 1。	Temp = Diff = 0001 0011
10. 最终结果 商数 = Temp[3:0] 亦即 0011(3) 余数 = Temp[7:4] 亦即 0001(1)。	Temp[3:0] = 0011 Temp[7:4] = 0001

注：在操作的过程中，无视最高位的溢出

第四次操作后决定
 Temp = 0001 0011
 S = 1 1110
 Diff = 0000 1001
 此为结果

Streamlined divider 的除法过程大致上是这样，具体的内容还是直接看代码吧。

实验九：循环型除法器

除法器的大致操作如上述内容，在这里就不重复了。不同的只是除数和被除数从 4 位位宽变成 8 位位宽而已。

streamlined_divider_module.v

```

1. module streamlined_divider_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]Dividend,
8.     input [7:0]Divisor,
9.
10.    output Done_Sig,
11.    output [7:0]Quotient,
12.    output [7:0]Reminder,
13.
14.    /*****
15.    output [15:0]SQ_Diff,
```

```

17.      output [15:0]SQ_Temp
18. );
19.
20.      ****
21.
22.      reg [3:0]i;
23.      reg [8:0]s;
24.      reg [15:0]Temp;
25.      reg [15:0]Diff;
26.      reg isNeg;
27.      reg isDone;
28.
29.      always @ ( posedge CLK or negedge RSTn )
30.          if( !RSTn )
31.              begin
32.                  i <= 4'd0;
33.                  s <= 9'd0;
34.                  Temp <= 16'd0;
35.                  Diff <= 16'd0;
36.                  isNeg <= 1'b0;
37.                  isDone <= 1'b0;
38.              end
39.          else if( Start_Sig )
40.              case( i )
41.
42.                  0:
43.                      begin
44.
45.                          isNeg <= Dividend[7] ^ Divisor[7];
46.                          s <= Divisor[7] ? { 1'b1, Divisor } : { 1'b1 , ~Divisor + 1'b1 };
47.                          Temp <= Dividend[7] ? { 8'd0 , ~Dividend + 1'b1 } : { 8'd0 , Dividend };
48.                          Diff <= 16'd0;
49.                          i <= i + 1'b1;
50.
51.                      end
52.
53.                  1,2,3,4,5,6,7,8:
54.                      begin
55.
56.                          Diff = Temp + { s , 7'd0 };
57.
58.                          if( Diff[15] ) Temp <= { Temp[14:0] , 1'b0 };
59.                          else Temp <= { Diff[14:0] , 1'b1 };
60.
61.                          i <= i + 1'b1;

```

```

62.
63.           end
64.
65.           9:
66.           begin isDone <= 1'b1; i <= i + 1'b1; end
67.
68.           10:
69.           begin isDone <= 1'b0; i <= 2'd0; end
70.
71.
72.       endcase
73.
74.   *****/
75.
76.   assign Done_Sig = isDone;
77.   assign Quotient = isNeg ? (~Temp[7:0] + 1'b1) : Temp[7:0];
78.   assign Reminder = Temp[15:8]; ← 应加上对余数的正负判断，否则是错的
79.
80.   *****/
81.
82.   assign SQ_Diff = Diff;
83.   assign SQ_Temp = Temp;
84.
85.   *****/
86.
87.
88. endmodule

```

第 16~17 行是仿真输出，分别针对操作空间 Temp 和 Diff。该除法器和传统除法器有几分相识。在步骤 0 (42~51 行) isNeg 寄存器取除数和被除数的正负关系 (45 行)。s 是寄存除数从 8 位空间向 9 位位宽空间的转换，并且取负值-补码形式 (46 行)。47 行的 Temp 空间 [Width : 0] 填入被除数的正值化。48 行是 Diff 空间的清零。

步骤 1~8 (53~63 行) 是该除法器的循环操作，具体的操作过程上述内容已经讲过了。不同的只是除数和被除数的位宽变成 8 而已，所以循环操作的次数也变成 8 次。步骤 9~10 (65~69) 是产生完成信号。

streamlined_divider_module.vt

```

1. `timescale 1 ps/ 1 ps
2. module streamlined_divider_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;

```

```
6.  
7.     reg Start_Sig;  
8.     reg [7:0]Dividend;  
9.     reg [7:0]Divisor;  
10.  
11.    wire Done_Sig;  
12.    wire [7:0]Quotient;  
13.    wire [7:0]Reminder;  
14.  
15.    /*****  
16.  
17.    wire [15:0]SQ_Diff;  
18.    wire [15:0]SQ_Temp;  
19.  
20.    /*****  
21.  
22.    streamlined_divider_module U1  
23.    (  
24.        .CLK(CLK),  
25.        .RSTn(RSTn),  
26.        .Start_Sig(Start_Sig),  
27.        .Dividend(Dividend),  
28.        .Divisor(Divisor),  
29.        .Done_Sig(Done_Sig),  
30.        .Quotient(Quotient),  
31.        .Reminder(Reminder),  
32.        .SQ_Diff(SQ_Diff),  
33.        .SQ_Temp(SQ_Temp)  
34.    );  
35.  
36.  
37.    initial  
38.    begin  
39.        RSTn = 0; #10; RSTn = 1;  
40.        CLK = 0; forever #10 CLK = ~CLK;  
41.    end  
42.  
43.    /*****  
44.  
45.    reg [3:0]i;  
46.  
47.    always @ ( posedge CLK or negedge RSTn )  
48.        if( !RSTn )  
49.            begin  
50.                i <= 4'd0;
```

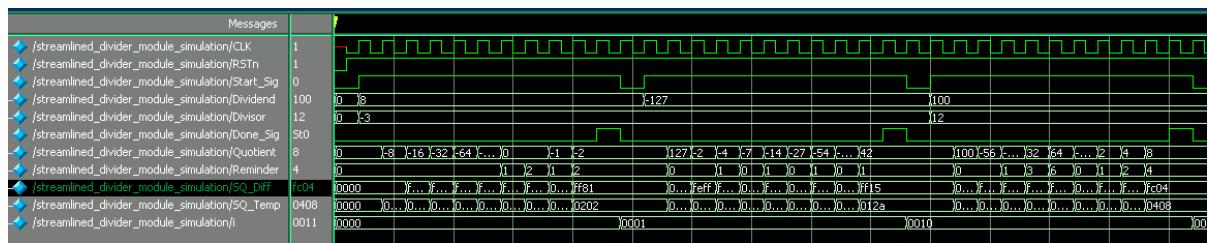
```

51.           Dividend <= 8'd0;
52.           Divisor <= 8'd0;
53.           Start_Sig <= 1'b0;
54.       end
55.   else
56.       case( i )
57.
58.           0: // Dividend = 8 , Divisor = -3
59.               if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
60.               else begin Dividend <= 8'd8; Divisor <= 8'b11111101; Start_Sig <= 1'b1; end
61.
62.           1: // Dividend = -127 , Divisor = -3
63.               if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
64.               else begin Dividend <= 8'b10000001; Divisor <= 8'b11111101; Start_Sig <= 1'b1; end
65.
66.           2: // Dividend = 100 , Divisor = 12
67.               if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
68.               else begin Dividend <= 8'd100; Divisor <= 8'd12; Start_Sig <= 1'b1; end
69.
70.           3:
71.               i <= 4'd3;
72.
73.       endcase
74.
75.
76.   endmodule

```

.vt 文件的编写方法还是一如既往的简单，自己看着办吧！

仿真结果：



仿真结果显示了 3 次的除法操作。3 次除法操作的过程都很和谐，时钟的消耗也是一致，很好很好。

实验九说明:

循环操作的除法器（或者称为位操作的除法器），实际上有很多种类。不知为什么，笔者就是喜欢它，啊~有多废话了。从仿真图中，除了初始化使用一个时钟以外，然后再加上 8 个时钟的循环操作，那么我们可以知道，实验九的除法一次性的除法操作使用了 9 个时钟（无视产生完成信号的步骤）。在某种程度上是可以接受的。

（实验九和实验四有许多相似的地方。）

实验九结论:

实验九的除法器，在一般的应用上还是切切有余。如果读者要它和 DSP 般一样任性，那么它会有点力不从心

2.3 循环除法运算的原理

说一句实话，笔者是一个不喜欢“吃”课本的家伙，当然更不喜欢课本的一套作风。整数除法器不像整数乘法器那样，拥有多姿多彩的算法支持，所以笔者不得不把课本中的循环除法原理往茶几上摆放。用一句傻瓜的话来说，只要了解了这些简单的原理，就算没有其他算法的补助，也能轻松的实现整数除法器。

来看一看一个公式：

$$\text{被除数} = \text{商数} * \text{除数} + \text{余数}$$

对，就是让笔者觉得厌恶的公式。如果要把这个公式以循环的方式去实现除法运算，这个公式需要再变化：

$$\text{被除数} - \text{商数} * \text{除数} = \text{余数}$$

然后再加上简单的修改：

$$\text{被除数} - \text{除数} = \text{余数} \quad // \text{ 商数被除外}$$

再加上一个约束：

$$\text{除数必须大于被除数} : \text{ 被除数} < \text{除数}$$

读者可能会产生疑问？如果这个约束成立，那么这个公式还用得了呢？其实它是有苦衷的……继续修改公式：

$$\text{被除数} - \text{除数} * 2^M = \text{余数} * 2 ;$$

条件：

如果 被除数小于等于 除数* 2^M ，余数不变而且倍增，以备下一次运算用。商为 0。
如果 被除数大于 除数* 2^M ，余数等于结果并且倍增，以备下一次运算用。商为 1。

实例 1：我们先不固定除数和被除数的取值范围，假设被除数 A = 10，除数 B = 3。为了使除数 B 大于被除数 A，让 $B' = B \cdot 2^m$ ，假设 m = 3 … (R 为余数，Q 为商)

A	$B' (24)$	R	Q	
10	$- 3 \cdot 2^3$	$= 10 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
20	$- 3 \cdot 2^3$	$= 20 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
40	$- 3 \cdot 2^3$	$= 16 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。
32	$- 3 \cdot 2^3$	$= 8 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。

循环结束 $R = 8 * 2 = 16$, 正确结果的 R , Q 亦即 :

正确的 $R' = R / 2^{m+1} = 16 / 16 = 1$

$Q = 0011$

结论: 当 m 等于 3, $R' = R / 2^{m+1} = R / 16$, 循环次数等于 $m + 1$, 亦即 4。

实例 2: 再假设被除数 $A = 10$, 除数 $B = 3$ 。为了使除数 B 大于被除数 A , 让 $B' = B \cdot 2^m$, 假设 $m = 4 \dots$ (R 为余数 , Q 为商)

$3 * 2^4 = 48$

A	$B' (48)$	R	Q	
10	$- 3 \cdot 2^4$	$= 10 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
20	$- 3 \cdot 2^4$	$= 20 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
40	$- 3 \cdot 2^4$	$= 40 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
80	$- 3 \cdot 2^4$	$= 32 * 2$	1	A 大于 B' , R 等于结果并且倍增。商为 1。
64	$- 3 \cdot 2^4$	$= 16 * 2$	1	A 大于 B' , R 等于结果并且倍增。商为 1。

循环结束 $R = 16 * 2 = 32$, 正确结果的 R , Q 亦即 :

正确的 $R' = R / 2^{m+1} = 32 / 32 = 1$

$Q = 00011$

结论: 当 m 等于 4, $R' = R / 2^{m+1} = R / 32$, 循环次数等于 $m + 1$, 亦即 5。

实例 3: 再假设被除数 $A = 10$, 除数 $B = 3$ 。为了使除数 B 大于被除数 A , 让 $B' = B \cdot 2^m$, 假设 $m = 5 \dots$ (R 为余数 , Q 为商)

$3 * 2^5 = 96$

A	$B' (96)$	R	Q	
10	$- 3 \cdot 2^5$	$= 10 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
20	$- 3 \cdot 2^5$	$= 20 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
40	$- 3 \cdot 2^5$	$= 40 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
80	$- 3 \cdot 2^5$	$= 80 * 2$	0	A 小于 B' , R 不变并且倍增。商为 0。
160	$- 3 \cdot 2^5$	$= 64 * 2$	1	A 大于 B' , R 等于结果并且倍增。商为 1。
128	$- 3 \cdot 2^5$	$= 32 * 2$	1	A 大于 B' , R 等于结果并且倍增。商为 1。

循环结束 $R = 32 * 2 = 64$, 正确结果的 R , Q 亦即 :

正确的 $R' = R / 2^{m+1} = 64 / 64 = 1$

$Q = 000011$

结论: 当 m 等于 5, $R' = R / 2^{m+1} = R / 64$, 循环次数等于 $m + 1$, 亦即 6。

实例 4：再假设被除数 $A = 10$ ，除数 $B = 3$ 。为了使除数 B 大于被除数 A , 让 $B' = B \cdot 2^m$ ，假设 $m = 6 \dots$ (R 为余数， Q 为商)

A	$B' (192)$	R	Q	
10	$- 3 \cdot 2^6$	$= 10 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
20	$- 3 \cdot 2^6$	$= 20 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
40	$- 3 \cdot 2^6$	$= 40 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
80	$- 3 \cdot 2^6$	$= 80 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
160	$- 3 \cdot 2^6$	$= 160 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
320	$- 3 \cdot 2^6$	$= 128 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。
256	$- 3 \cdot 2^6$	$= 64 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。

循环结束 $R = 64 * 2 = 128$, 正确结果的 R ， Q 亦即：

正确的 $R' = R / 2^{m+1} = 128 / 128 = 1$

$Q = 000\ 0011$

结论：当 m 等于 6, $R' = R / 2^{m+1} = R / 128$, 循环次数等于 $m + 1$, 亦即 7。

实例 5：再假设被除数 $A = 10$ ，除数 $B = 3$ 。为了使除数 B 大于被除数 A , 让 $B' = B \cdot 2^m$ ，假设 $m = 7 \dots$ (R 为余数， Q 为商)

A	$B' (384)$	R	Q	
10	$- 3 \cdot 2^7$	$= 10 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
20	$- 3 \cdot 2^7$	$= 20 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
40	$- 3 \cdot 2^7$	$= 40 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
80	$- 3 \cdot 2^7$	$= 80 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
160	$- 3 \cdot 2^7$	$= 160 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
320	$- 3 \cdot 2^7$	$= 640 * 2$	0	A 小于 B' ，R 不变并且倍增。商为 0。
640	$- 3 \cdot 2^7$	$= 256 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。
512	$- 3 \cdot 2^7$	$= 128 * 2$	1	A 大于 B' ，R 等于结果并且倍增。商为 1。

循环结束 $R = 128 * 2 = 256$, 正确结果的 R ， Q 亦即：

正确的 $R' = R / 2^{m+1} = 256 / 256 = 1$

$Q = 0000\ 0011$

结论：当 m 等于 7, $R' = R / 2^{m+1} = R / 256$, 循环次数等于 $m + 1$, 亦即 8。

从上面 5 个实例中，被除数 A 和除数 B 不变，均为是 10 和 3，反之 m 从 3~7 递增。但是求得的结果也是一样。从中我们可以了解一个事实：

如果以 “**被除数 = 商数*除数 + 余数**” 公式实现循环除法运算，那么它必须 “**修改**” 符合循环除法运算的形式。此外为了符合 “**被除数 < 除数**” 的约束，“**除数*2^m**” 是必须的。m 的影响力有 “**正确的 R' = R / 2^{m+1}**，然而循环次数是 m + 1。”

先给自己 5 分钟消化吧，笔者也消化了 3 天（笔者是笨蛋的关系）。比起笔者给出的例子实际上的原理还要猥琐的多。

===== 5 分钟 =====

在循环除法运算在原理上（纯数学上），我们不会去考虑被除数 A 和除数 B 的空间（位宽）。相反的，如果把它放在 Verilog HDL 语言上，大伙也不得不谨慎。假设被除数 A 和除数 B 均为 8 位 位宽。考虑到 “**被除数 < 除数**” 的约束，m 到底要取多大呢？

我们知道 8 位空间的整数，取值范围是 -127~127，如果抛开负值（笔者一般上都不喜欢把负值往里边算），那么取值范围会是 0~127。为了使除数 B “**完全**” 大于被除数 A，m 取值 Width - 1 为最佳，亦即 7。所以 A = { 0~127 } 是完全小于 B = { 0 ~ 127 } * 2⁷ 而使约束成立。

除此之外，我们还要考虑另三个问题：第一个问题就是操作空间的大小；第二个问题是循环次数；第三个问题是正确的 R'（余数）结果：

- 1) 在这里，余数 R 寄存的空间便会成为操作空间，操作空间最理想的大小是 Width * 2，亦即 16 位位宽。
- 2) 在这里我们知道，为了成立约束，m 取值为 7，那么循环次数会是 m + 1 = 8。
- 3) 根据上面的 5 个实例我们可以得到一个事实，正确的余数 R' 等价于 $R' = R / 2^{(m+1)}$ 如果 m 的取值是 7 的话，那么 $R' = R / 2^{(7+1)} = R / 2^8$ 。

当一切就绪以后，我们还面临了最大的一个问题 ... 就是如何把移位操作和除法操作压缩在同一个步骤里？

```
case( i )
  1,2,3,4,5,6,7,8; // 循环次数等于 m + 1
begin
  // " Result << 1" 倍增，亦即乘以 2 的意思
  if( R <= (Divisor << 7) ) begin R <= R << 1; Q[8-i] = 1'b0; end
  else begin R <= ( R - ( Divisor << 7 ) ) << 1; Q[8-i] = 1'b1; end
```

```
i <= i + 1'b1;
```

```
end
```

上述一段代码就是最关键的！

if(R <= (Divisor << 7)) 表示了“R”是否小于等于“经过增值后的 Divisor”。如果“是”R 倍增，商为 0。否则 R 赋予“相减之后然后倍增的结果”，商为 1。我们知道我们的 m 取值为 7，所以有 8 次的循环操作。当完成 8 次的循环运算过后，那么 $R' = R / 2^{(m+1)}$ ， $R / 2^{(7+1)} = R / 2^{(8)}$ ，亦即 $R' = R / 256$ ，当然我们也可以使用最快捷的方法求得 R' ，方法如下：

```
assign Reminder = R[15:8];
```

这段代码和 $R' = R / 256$ 是等价的。

好了，再给自己 5 分钟休息吧。如果上面的内容，读者不能完全明白的话，这也是人之常情，因为到目前为止笔者都是在强调一些零星的重点。关键还是在实验。具体的内容还是在试验中直接了解。

实验十：从原理到实现的循环除法器

other_divider_module.v

```
1. module other_divider_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Start_Sig,
7.     input [7:0]Dividend,
8.     input [7:0]Divisor,
9.
10.    output Done_Sig,
11.    output [7:0]Quotient,
12.    output [7:0]Reminder,
13.
14.    *****/
15.
16.    output [15:0]SQ_R
17.
18.    *****/
19.
```

```

20. );
21.
22. *****/
23.
24. reg [3:0]i;
25. reg [7:0]Q;
26. reg [7:0]Dsor;
27. reg [15:0]_Dsor;
28. reg [15:0]R;
29. reg isNeg;
30. reg isDone;
31.
32. always @ ( posedge CLK or negedge RSTn )
33.     if( !RSTn )
34.         begin
35.             i <= 4'd0;
36.             Q <= 8'd0;
37.             Dsor <= 8'd0;
38.             _Dsor <= 16'd0;
39.             R <= 16'd0;
40.             isNeg <= 1'b0;
41.             isDone <= 1'b0;
42.         end
43.     else if( Start_Sig )
44.         case( i )
45.
46.             0:
47.                 begin
48.
49.                     isNeg <= Divisor[7] ^ Dividend[7];
50.                     Dsor <= Divisor[7] ? ~Divisor + 1'b1 : Divisor;
51.                     _Dsor <= Divisor[7] ? { 8'hff , Divisor } : { 8'hff , ~Divisor + 1'b1 };
52.                     R <= Dividend[7] ? { 8'd0 , ~Dividend + 1'b1 } : { 8'd0 , Dividend };
53.                     Q <= 8'd0;
54.                     i <= i + 1'b1;
55.
56.                 end
57.
58.             1,2,3,4,5,6,7,8: // m + 1
59.                 begin
60.
61.                     if( R <= (Dsor << 7) ) begin R <= { R[14:0] , 1'b0 }; Q[8-i] = 1'b0; end
62.                     else begin R <= ( R + ( _Dsor << 7 ) ) << 1; Q[8-i] = 1'b1; end
63.
64.                     i <= i + 1'b1;

```

```
65.
66.           end
67.
68.           9:
69.           begin isDone <= 1'b1; i <= i + 1'b1; end
70.
71.           10:
72.           begin isDone <= 1'b0; i <= 4'd0; end
73.
74.
75.           endcase
76.
77.   *****/
78.
79.   assign Done_Sig = isDone;
80.   assign Quotient = isNeg ? (~Q + 1'b1) : Q;
81.   assign Reminder = R[15:8];
82.
83.   *****/
84.
85.   assign SQ_R = R;
86.
87.   *****/
88.
89. endmodule
```

在 16 行是仿真输出用的 SQ_R。24~30 行是该模块所使用的寄存器，Dsor 是用来寄存除数的正值。_Dsor 是用来寄存除数的负值的补码形式，_Dsor 寄存器的位宽和操作空间 R 一样。

在步骤 0 (46~56 行)，isNeg 用来寄存除数和被除数的正负关系 (49 行)。Dsor 用来寄存除数的正值 (50 行)。_Dsor 用来寄存除数的负值的补码形式，为了方便 A - B' 的操作 (51 行)。操作空间 R 在初始化中，[7..0]是用来填入被除数的正值。

步骤 1~8 (58~66 行)，是该模块的核心部分，也是循环除法操作。在每一次的循环中，先判断 R 的值 是否小于等于 Dsor << 7 ($R \leq B'$)。如果 if 条件成立 (R 的值小于等于 B') 操作空间 R 的值翻倍，Q[8-i] 等于 0。如果 if 条件不成立 (R 的值大于 B') R 赋予“相减过后的翻倍值”并且 Q[8-i] 等于 1。

步骤 9~10 (68~72 行) 是完成信号的产生。

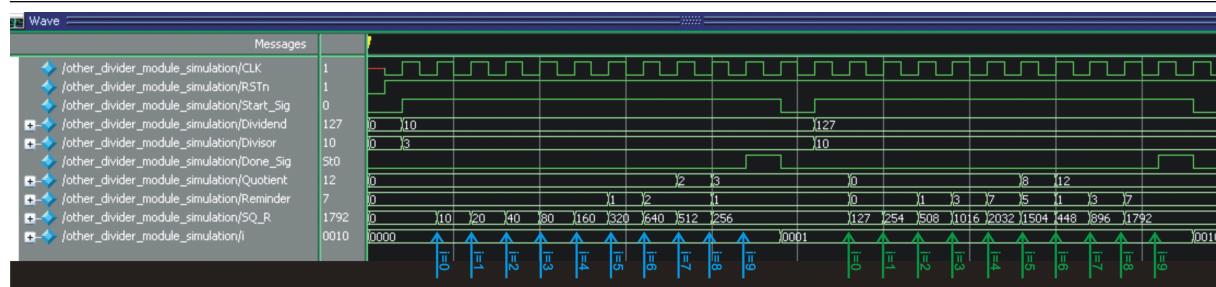
在 80 行 Quotient 的输出是根据 isNeg，亦即除数和被除数的正负关系。在 81 行 Reminder 的输出是由 R[15:8] 驱动 ($R' = R / 2^{m+1}$ 的简化写法)。85 行是操作空间 R 的仿真输出。

other_divider_module.vt

```
1. `timescale 1 ps/ 1 ps
2. module other_divider_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Start_Sig;
8.     reg [7:0]Dividend;
9.     reg [7:0]Divisor;
10.
11.    wire Done_Sig;
12.    wire [7:0]Quotient;
13.    wire [7:0]Reminder;
14.
15.    /*****
16.
17.    wire [15:0]SQ_R;
18.
19.    *****/
20.
21.    other_divider_module U1
22.    (
23.        .CLK(CLK),
24.        .RSTn(RSTn),
25.        .Start_Sig(Start_Sig),
26.        .Dividend(Dividend),
27.        .Divisor(Divisor),
28.        .Done_Sig(Done_Sig),
29.        .Quotient(Quotient),
30.        .Reminder(Reminder),
31.
32.        *****/
33.
34.        .SQ_R( SQ_R )
35.
36.        *****/
37.    );
38.
39.    *****/
40.
41.    initial
```

```
42.      begin
43.          RSTn = 0; #10; RSTn = 1;
44.          CLK = 0; forever #10 CLK = ~CLK;
45.      end
46.
47.      *****/
48.
49.      reg [3:0]i;
50.
51.      always @ ( posedge CLK or negedge RSTn )
52.          if( !RSTn )
53.              begin
54.
55.                  i <= 4'd0;
56.                  Dividend <= 8'd0;
57.                  Divisor <= 8'd0;
58.                  Start_Sig = 1'b0;
59.
60.              end
61.          else
62.              case( i )
63.
64.                  0:
65.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
66.                      else begin Dividend <= 8'd10; Divisor <= 8'd3; Start_Sig <= 1'b1; end
67.
68.                  1:
69.                      if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
70.                      else begin Dividend <= 8'd127; Divisor <= 8'd10; Start_Sig <= 1'b1; end
71.
72.                  2:
73.                      i <= 4'd2;
74.
75.              endcase
76.
77.      *****/
78.
79.
80.  endmodule
```

仿真结果：



根据上图的仿真结果，当 Dividend 为 10，Divisor 为 3，得到的结果 Quotient 和 Remainder 是 3 与 1。(左边的箭头) 当步骤 $i = 0$ 的时候，该模块“决定”初始化所有相关的寄存器，当步骤 $i = 1 \sim 8$ ，在每一个时间点里执行 8 次的循环除法操作。

SQ_R 输出的值和前面实例 5 的结果完全是一样。不同的只是实例中用“把大象放进冰箱”的步骤概念，实验所使用的是“时间点”的步骤概念。

实验十结论：

实验九的除法器实际上是实验十除法器的简化版，为什么这样说呢？还记得实验九的除法器是如何判断 Diff 空间的嘛？它是判断 Diff 空间的最高位，亦即符号位。我们知道只要符号位为逻辑 1，必定是负值。换言之，这也表示 $\text{Temp} < (\text{Divisor} \ll 7)$ ，其中 Temp 是余数 R， $\text{Divisor} \ll 7$ 是 s 经过正直化后的左移 7 位的 B' 。（Diff 是 $R - B'$ 之后的结果）

实验九和实验十的乘法器，它们所使用的原理都一样。但是实验九的除法器不及实验十的除法器来得稳定，因为它简化太多了。

总结：

说实话，笔者对于这篇笔记也非常郁闷。笔者花了很多天的时间查找和分析资料，得到结果就是这些而已。许多资料都和“整数除法器”扯不上关系。在除法器的世界，最常见莫过于循环型的除法器。此外还有一种常见的除法器，它基于 SRT 算法的除法器。

SRT 算法实际上是属于小数的除法器。宏观上是和整数乘法器扯不上关系。在微观上 SRT 算法也可以实现整数的除法，但是作为代价，要遵守大量的条件，只要除数和被除数的取值范围稍微不同，那么要遵守的条件也会跟谁变动。你说烦不烦！？

如果读者问：“整数除法器到底有什么作用？”

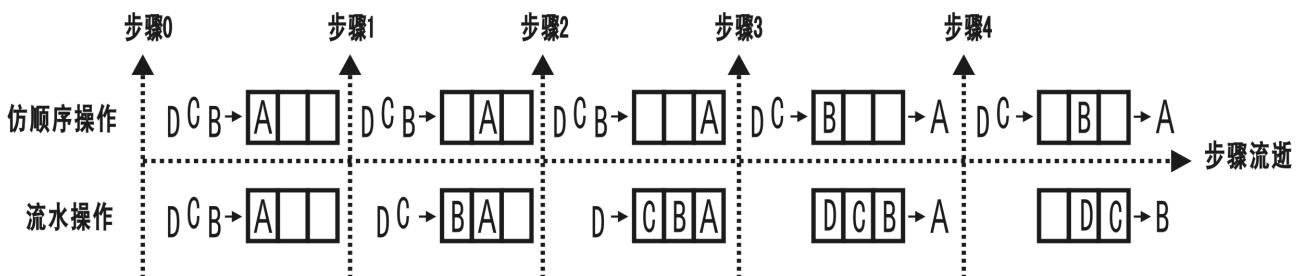
除了普通的除法运算要求之外，整数乘法器最常见就有“取位操作”。一个很典型的实验就是数码管驱动实验。在这个试验中，常常要从计数器中取百位，十位，个位，然后送往相关的数码管中。笔者还记得，笔者在早期的时候，还特意记录了一遍有关数码管实验的笔记。嗯~现在回想起来，也有几分真实感。那时候接触 Verilog HDL 语言不久，写不出什么好除法器，用的是 Quartus II 自带的除法器，然后作成取位器。

无论整数乘法器还是除法器，从第一章到第二章的所有实验都是为后面几个章节在作准备。在这一章笔记中，笔者继续强化读者对“步骤”的认识，期间也建议了使用“时间点”的概念去理解“步骤”。

第三章：流水操作和建模

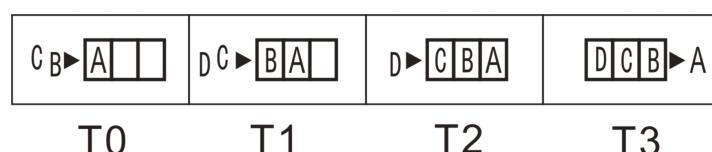
3.1 流水操作的概念

流水操作可以说是 Verilog HDL 语言（硬件描述语言）的特权，顺序操作如 C 语言是很难实现流水操作。一些同学听到“流水”两个字，不知为什么会莫名的胆怯起来，类似的感受笔者真的很了解。流水操作的概念看似简单，但是要实现它确实是不容易。但是读者们知道吗，当了解了步骤 i 的相关概念“把大象放进冰箱”或者“时间点”，我们可以从“仿顺序操作”转换为“流水操作”。

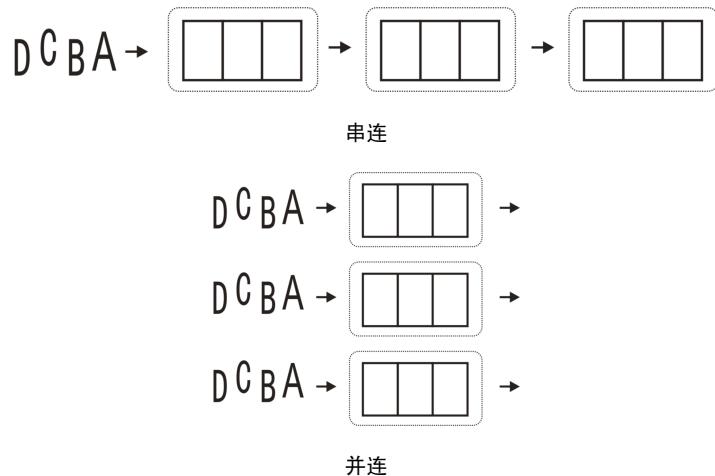


上图表示了 仿顺序操作 和 流水操作 的区别，步骤采用时间点的概念。上面的 3 个小格子表示了 3 个不同的操作，每一个字母（数据）都要经过这些小格子（经过处理）。仿顺序操作的概念很简单，3 个小格子，每一个字母经过都要 3 个步骤（处理一个数据需要消耗 3 个时钟），当上一个字母从格子出来之后（处理完毕），下一个字母才能进入。反之，流水操作在每一个步骤，字母只要有“空格子”，字母都可以陆陆续续的经过（每一个时钟都有数据被读入，处理和被处理完毕）。

在步骤 0~步骤 4 中，仿顺序操作只有一个字母出来（处理完毕）而已，然而流水操作在步骤 0~4 中，已经有两个字母出来（处理完毕），在接下来的 2 个步骤之中，字母 C 和 D 都会陆续出来（处理完毕）。从优点上来评价流水操作，它就有执行效率高这个优点而已，可是在许多方便流水操作也不是很“在行”。



我们先假设有一个流水操作模块，它有 3 个操作步骤，在 A 进入之际，直到 A 离开这个模块之后，期间有 3 个空挡的时间（T0~T2），称为 **潜伏时间**，在这段时间内没有任何字母从这个模块出来。一旦经过这段时间，字母就会远远不断从这个模块出来。换句话说，流水操作的“潜伏时间”是根据“操作步骤”而定。预测潜伏时间，是流水操作不好控制的原因之一。



此外，无论是并连还是串连的流水操作，读者会发现流水操作的操作方向（连线向量）都是“向前走”，这是导致流水操作不好控制的原因之二。

所以说，“驱动”呀，“控制”呀，都不适合流水操作，因为在驱动和控制方面，连线向量（连线关系），不可能只有一个方向而已。相反的，如果是“数据处理”或者“算法”之类的话，流水操作再适合不过了。

3.2 仿顺序操作向流水操作的转换

坏话当前，流水操作是非常不适合设计“驱动”和“控制”，反之它非常适合“数据处理”。然而我们在前面所学过的整数乘法器和整数除法器实验就是“数据处理”的其中一个家族。

可能读者们曾经经历过，空手建立一个拥有流水操作的模块，初期会有一种无法入手的莫名感觉。如果用笔者的话来说，第一：可能是习惯顺序操作的关系，第二：可能不了解“时间点”的概念关系。要建立流水操作的模块，其实很多地方我们都可以从顺序操作接签。

假设有一块电路板 A，它的初值状态是 4'b0000。

- A[0]如果是逻辑一，表示已经安装电阻。
- A[1]如果是逻辑一，表示已经安装二极管。
- A[2]如果是逻辑一，表示已经安装电容。
- A[3]如果是逻辑一，表示已经完成封装。

那么一块完整的电路板 A，它的最终值是 4'b1111。如果这块电路板 A，要完成操作，它必须经过以下的步骤。

- (一) 读入 A 的初值。
- (二) 将 A[0]设置 1。
- (三) 将 A[1]设置 1。
- (四) 将 A[2]设置 1。
- (五) 将 A[3]设置 1。

在仿顺序操作上，可以这样写：

```
reg [3:0]A;  
  
always @ ( posedge CLK )  
.....  
    case( i )  
        0: begin A = A_input; i <= i + 1'b1; end;  
        1: begin A[0] = 1'b1; i <= i + 1'b1; end  
        2: begin A[1] = 1'b1; i <= i + 1'b1; end  
        3: begin A[2] = 1'b1; i <= i + 1'b1; end  
        4: begin A[3] = 1'b1; i <= i + 1'b1; end  
    .....
```

上面的代码中，步骤 i 从 0~4，这也表示电路板 A 要完成封装就需要经过的步骤。

如果换做流水操作，我们可以这样写：

```
reg [3:0]A [4:0]; // 建立一维数组的 A，放顺操作中有 5 个步骤，所以建立 5 个 A。  
  
always @ ( posedge CLK )  
begin  
    //A[m]表示，一维数组 A 中第 m 个元素。  
    //A[m][n]表示，一维数组 A 中第 m 个元素，第 n 位。  
  
    A[0] <= A_input;          // 从外部读取新的 A 值  
    A[1] <= A[0] | 4'b0001;    // 读取 A[0]值，并且设置第 0 位，然后赋予 A[1]  
    A[2] <= A[1] | 4'b0010;    // 读取 A[1]值，并且设置第 1 位，然后赋予 A[2]  
    A[3] <= A[2] | 4'b0100;    // 读取 A[2]值，并且设置第 2 位，然后赋予 A[3]  
    A[4] <= A[3] | 4'b1000;    // 读取 A[3]值，并且设置第 3 位，然后赋予 A[4]  
  
end
```

从上面的代码，我们可以知道。如果电路板 A 要完成封装，就要经过 5 个步骤。所以在寄存器 A，必须扩展为 5 个元素，最简单的办法就是建立储存器。然而储存器的 words 是 5 个，位宽和寄存器 A 一样。

全部 5 个操作步骤都挤在同一个 begin ... end 之间，这也意味着每一个时钟，这 5 个操作步骤都在发生着。在每一个时钟中...

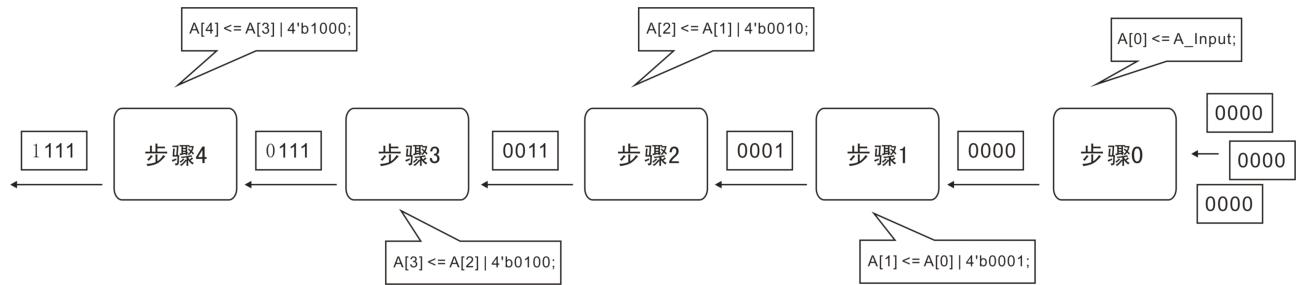
A[0] 读取一个新的值（读入一块 4'b0000 的电路板）。

A[1]读取 A[0]的值，然后设置第 0 位（A[1]空间从 A[0]空间引入一块电路板，并且安装电阻 4'b0001）。

A[2]读取 A[1]的值，然后设置第 1 位（A[2]空间从 A[1]空间引入一块电路板，并且安装二极管 4'b0011）。

A[3]读取 A[2]的值，然后设置第 2 位（A[3]空间从 A[2]空间引入一块电路板，并且安装电容 4'b0111）。

A[4]读取 A[3]的值，然后设置第 3 位（A[4]空间从 A[3]空间引入一块电路板，并且封装 4'b1111）。



如果用图形来表示，会是如同上图。

实验十一：流水式查表乘法器

在试验十一里，我们要基于实验五的查表乘法器，建立一个流水操作的查表乘法器。在这里简单回顾一下查表乘法器的操作步骤：

- (一) 取得 I1，取得 I2。
- (二) 正值化 I1 和 I2。
- (三) 正值化后的 I1 和 I2 送往查表。
- (四) 取得查表结果 Q1_Sig 和 Q2_Sig 然后相减。

将上面的操作步骤，流水化后会变成什么样子呢？

pipeline_lut_multiplier_module.v

```

1. module pipeline_lut_multiplier_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [7:0]A,
7.     input [7:0]B,
8.
9.     output [15:0]Product,
10.
11.    /*****
12.
13.    output [8:0]SQ_I1_0,
14.    output [8:0]SQ_I1_1,
15.    output [8:0]SQ_I2_0,
  
```

```
16.      output [8:0]SQ_I2_1,
17.      output [15:0]SQ_Q1,
18.      output [15:0]SQ_Q2
19.
20.      *****/
21. );
22.
23. *****/
24.
25.      wire [15:0]Q1_Sig;
26.      wire [15:0]Q2_Sig;
27.
28.      *****/
29.
30.      reg [8:0]I1 [1:0];
31.      reg [8:0]I2 [1:0];
32.
33.      always @ ( posedge CLK or negedge RSTn )
34.          if( !RSTn )
35.              begin
36.                  I1[0] <= 9'd0; I1[1] <= 9'd0;
37.                  I2[0] <= 9'd0; I2[1] <= 9'd0;
38.              end
39.          else
40.              begin
41.
42.                  I1[0] <= { A[7], A } + { B[7], B };
43.                  I2[0] <= { A[7], A } + { ~B[7], ( ~B + 1'b1 ) };
44.
45.                  *****/
46.
47.                  I1[1] <= I1[0][8] ? ( ~I1[0] + 1'b1 ) : I1[0];
48.                  I2[1] <= I2[0][8] ? ( ~I2[0] + 1'b1 ) : I2[0];
49.
50.                  *****/
51.
52.          // read file from rom
53.
54.          *****/
55.
56.      end
57.
58.
59.      *****/
60.
```

```

61.      lut_module  U1
62.      (
63.          .CLK ( CLK ),
64.          .Addr ( I1[1][7:0] ),
65.          .Q ( Q1_Sig )
66.      );
67.
68.      *****/
69.
70.      lut_module  U2
71.      (
72.          .CLK ( CLK ),
73.          .Addr ( I2[1][7:0] ),
74.          .Q ( Q2_Sig )
75.      );
76.
77.      *****/
78.
79.      assign Product = Q1_Sig + ( ~Q2_Sig + 1'b1 );
80.
81.      *****/
82.
83.      assign SQ_I1_0 = I1[0];
84.      assign SQ_I1_1 = I1[1];
85.      assign SQ_I2_0 = I2[0];
86.      assign SQ_I2_1 = I2[1];
87.      assign SQ_Q1 = Q1_Sig;
88.      assign SQ_Q2 = Q2_Sig;
89.
90.      *****/
91.
92.
93.
94. endmodule

```

第 13~18 行是仿真用的输出。第 30~31 行，建立了 I1 和 I2 的储存器，它们均是 2 个。第 61~75 行声明了两个 LUT，这两个查表主要是用于步骤 3。第 79 行是用于最后一个步骤，亦即步骤 4。

第 42~43 行是用于第 1 步骤，亦即读取 I1 和 I2 的值，然而 I1[0] 和 I2[0] 作为暂存空间。第 47~48 行是用于第 2 步骤，主要是从 I1[0] 和 I2[0] 取值，正值化后赋予 I1[1] 和 I2[1]。

第 50~54 行，用了一段注释，目的是显性指示步骤 3。在 61~75 行，是从 LUT 模块从取

值。

第 79 行，是第四步骤，笔者使用了组合逻辑的方式将这一步骤简了化。这一步骤的工作，就是将 LUT 模块输出的值 Q1_Sig 和 Q2_Sig 相减，并且输出。第 83~88 行是仿真输出与驱动源。

实际上，流水化后的查表乘法器只需 3 个步骤而已，因为步骤 4 在 79 行被简化了。

pipeline_lut_multiplier_module.vt

```
1. `timescale 1 ps/ 1 ps
2. module pipeline_lut_multiplier_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]A;
8.     reg [7:0]B;
9.
10.    wire [15:0]Product;
11.
12.    /*****
13.
14.    wire [8:0]SQ_I1_0;
15.    wire [8:0]SQ_I1_1;
16.    wire [8:0]SQ_I2_0;
17.    wire [8:0]SQ_I2_1;
18.    wire [15:0]SQ_Q1;
19.    wire [15:0]SQ_Q2;
20.
21.    *****/
22.
23.    pipeline_lut_multiplier_module U1
24.    (
25.        .CLK(CLK),
26.        .RSTn(RSTn),
27.        .A(A),
28.        .B(B),
29.        .Product(Product),
30.        .SQ_I1_0( SQ_I1_0 ),
31.        .SQ_I1_1( SQ_I1_1 ),
32.        .SQ_I2_0( SQ_I2_0 ),
33.        .SQ_I2_1( SQ_I2_1 ),
```

```

34.      .SQ_Q1( SQ_Q1 ),
35.      .SQ_Q2( SQ_Q2 )
36. );
37.
38. ****
39.
40. initial
41. begin
42.     RSTn = 0; #10; RSTn = 1;
43.     CLK = 0; forever #10 CLK = ~CLK;
44. end
45.
46. ****
47.
48. reg [3:0]i;
49.
50. always @ ( posedge CLK or negedge RSTn )
51.     if( !RSTn )
52.         begin
53.
54.             i <= 4'd0;
55.             A <= 8'd0;
56.             B <= 8'd0;
57.
58.         end
59.     else
60.         case( i )
61.
62.             0: // A = 127, B= 127
63.                 begin A <= 8'd127; B <= 8'd127; i <= i + 1'b1; end
64.
65.             1: // A = 10, B= 12
66.                 begin A <= 8'd10; B <= 8'd12; i <= i + 1'b1; end
67.
68.             2: // A = 32, B= 74
69.                 begin A <= 8'd32; B <= 8'd74; i <= i + 1'b1; end
70.
71.             3: // A = -127, B= 20
72.                 begin A <= 8'b10000001; B <= 8'd20; i <= i + 1'b1; end
73.
74.             4:
75.                 begin A <= 8'd0; B <= 8'd0; i <= 4'd4; end
76.
77.         endcase

```

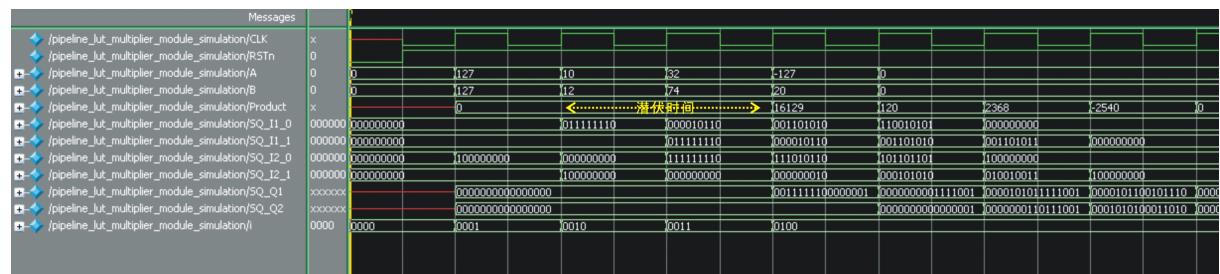
```

78.
79.      ****
80.
81.
82. endmodule

```

第 62~72 行表示，在每一个时钟 .v 文件都刺激不同数据，这些数据分别是 $127 * 127$, $10 * 12$, $32 * 74$, $-127 * 20$ 。流水化以后的查表乘法器，和仿顺序操作的查表乘法器的激励文件写法有点不同，这一点请注意。

仿真结果：



流水化后的查表乘法器，有三个操作步骤，所以说这个流水操作的模块的潜伏时间是 3 个时钟。在上面的仿真图中，箭头的范围表示了流水查式查表乘法器的潜伏时间。在激励 .vt 文件中，步骤 0~3 分别对 A 和 B 输入了 $127 * 127$, $10 * 12$, $32 * 74$ 和 $-127 * 20$ 。一旦潜伏时间过后，这些经过处理的数据就会源源不断的出来。

实验十一说明：

流水操作最大的问题就是潜伏时间。假设笔者在第一次的时间，输入大量的数据，经过潜伏时间后，处理之后的数据会源源不断的出来。然后笔者等待一段时间，然后又输入大量的数据，又要重新等待潜伏时间的经过 ...

这就是为什么笔者说，流水操作很野很难驯服的原因。

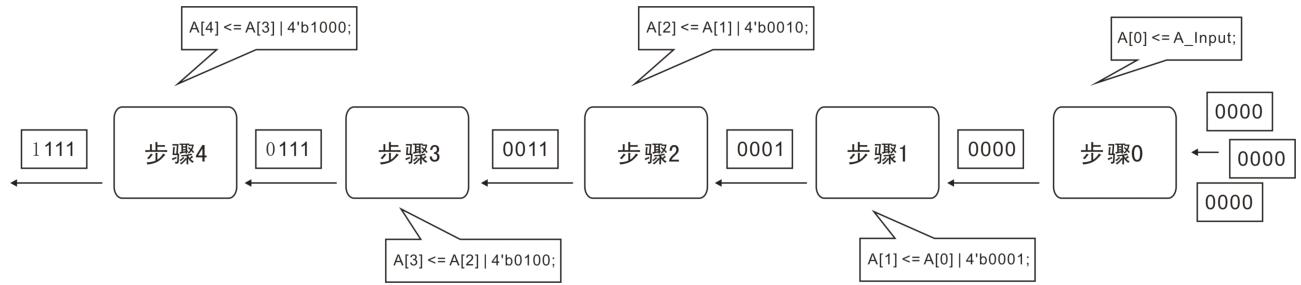
实验十一结论：

流水操作不像仿顺序操作那样，设计逻辑那么容易掌握。笔者在初期的时候，面对流水操作都也是摸不着头脑，不知道要从哪一步开始编辑。最后，不知不觉得习惯了，先设计出仿顺序操作，然后再转换为流水操作。

此外流水操作，还有一个头疼的问题，就是编辑 .v 文件的时候，因为流水化的关系，会把 Verilog HDL 语言的代码风格破坏得一塌糊涂，最终使得 .v 文件不容读懂。(笔者

在《Verilog HDL 那些事儿-建模篇》已经说过，读不懂的 .v 文件是最致命的)

到底有没有好办法更能，显性的，有结构的，容易理解的，来表达流水操作？



上面的图形告诉了我们一个事实，如果流水操作能用图形来表达的话，稍微逆向思考，那么流水操作是不是也能使用“建模”来实现？

3.3 流水操作和建模

看到“建模”大伙儿不禁会联想与“低级建模”吧？低级建模的出现就是为了使 Verilog HDL 语言的建模更有结构和更有层次。从另一个方面而言“建模”提升了 Verilog HDL 语言的表达能力。在“单文件”主义下，Verilog HDL 语言的表达能力往往是最致命的，其外还有许多隐藏着的小问题。如果流水操作的书写，亦是单文件，亦是没有建模，后果不是一般的严重。最终结果就是“你自己的东西只有你自己看得懂而已！”没有更多的价值。

言归正主，在上一章节中，实验十一留下的问题是：“如果流水操作可以用图形来表达，逆向思考的话，流水操作是不是也能用建模来表达呢？”，答案是大力点头的。稍微回顾一下实验十一，流水式查表乘法器的操作步骤：

在每一个时钟中：

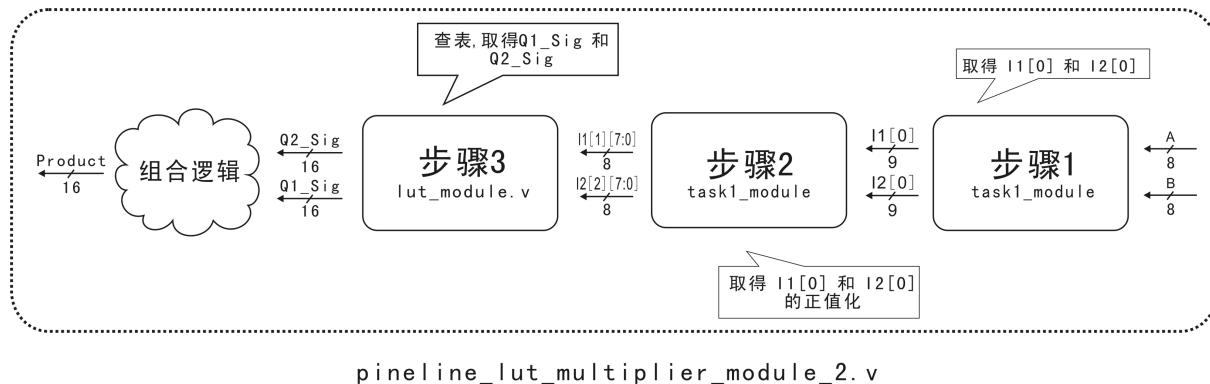
I1[0] 和 I2[0] 取值；

I1[1] 和 I2[1] 从 I1[0] 和 I2[0] 取值并且正值化；

LUT 从 I1[1] 和 I2[1] 取值，并且查表；

LUT 的输出信号 Q1_Sig 和 Q2_Sig，相减并且驱动 Product 输出。

流水式查表乘法器，实际上有 4 个步骤，但是最后一个步骤笔者使用组合逻辑将它简化了。所以仅剩下 3 个步骤而已。如果把以上的流水操作建模化的话，会是如下图：



在这里我们不得不再借助低级建模的力量了。笔者接签了低级建模的图形特征，将流水式查表乘法器的建模表达出来。上面的图形，已经将流水式查表乘法器表达得非常清楚，我们只要跟着“连线”的方向，就会知道它是干什么的。但是有一点必须注意的是，上面的图形是“每一个时钟所有模块都操作一次”。

实验十二：建模过后的流水式查表乘法器

pipeline_lut_multiplier_module_2.v

```
1. module pipeline_lut_multiplier_module_2
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [7:0]A,
7.     input [7:0]B,
8.
9.     output [15:0]Product
10.
11. );
12.
13. ****
14.
15. wire [8:0]U1_I1_Out;
16. wire [8:0]U1_I2_Out;
17.
18. task1_module U1( CLK, RSTn, A, B, U1_I1_Out, U1_I2_Out );
19.
20. wire [7:0]U2_I1_Out;
21. wire [7:0]U2_I2_Out;
22.
23. task2_module U2( CLK, RSTn, U1_I1_Out, U1_I2_Out, U2_I1_Out, U2_I2_Out );
24.
25. wire [15:0]Q1_Sig;
26. wire [15:0]Q2_Sig;
27.
28. lut_module U3a( CLK, U2_I1_Out, Q1_Sig );
29. lut_module U3b( CLK, U2_I2_Out, Q2_Sig );
30.
31. ****
32.
33. assign Product = Q1_Sig + ( ~Q2_Sig + 1'b1 );
34.
35. ****
36.
37. endmodule
```

```
38.  
39. module task1_module  
40. (  
41.  
42.     input CLK,  
43.     input RSTn,  
44.  
45.     input [7:0]A,  
46.     input [7:0]B,  
47.  
48.     output [8:0]I1_Out,  
49.     output [8:0]I2_Out  
50.  
51. );  
52.  
53. ****  
54.  
55. reg [8:0]I1;  
56. reg [8:0]I2;  
57.  
58. always @ ( posedge CLK or negedge RSTn )  
59.     if( !RSTn )  
60.         begin  
61.  
62.             I1 <= 9'd0;  
63.             I2 <= 9'd0;  
64.  
65.         end  
66.     else  
67.         begin  
68.  
69.             I1 <= { A[7], A } + { B[7], B };  
70.             I2 <= { A[7], A } + { ~B[7], (~B + 1'b1) };  
71.  
72.         end  
73.  
74. ****  
75.  
76. assign I1_Out = I1;  
77. assign I2_Out = I2;  
78.  
79. ****  
80.  
81. endmodule
```

```
82.  
83.  
84.  
85. module task2_module  
86. (  
87.     input CLK,  
88.     input RSTn,  
89.  
90.     input [8:0]I1_In,  
91.     input [8:0]I2_In,  
92.  
93.     output [7:0]I1_Out,  
94.     output [7:0]I2_Out  
95. );  
96.  
97. /*****  
98.  
99. reg [8:0]I1;  
100. reg [8:0]I2;  
101.  
102. always @ ( posedge CLK or negedge RSTn )  
103.     if( !RSTn )  
104.         begin  
105.  
106.             I1 <= 9'd0;  
107.             I2 <= 9'd0;  
108.  
109.         end  
110.     else  
111.         begin  
112.  
113.             I1 <= I1_In[8] ? ( ~I1_In + 1'b1 ) : I1_In;  
114.             I2 <= I2_In[8] ? ( ~I2_In + 1'b1 ) : I2_In;  
115.  
116.         end  
117.  
118. /*****  
119.  
120. assign I1_Out = I1[7:0];  
121. assign I2_Out = I2[7:0];  
122.  
123. /*****  
124.  
125.endmodule
```

笔者采用比较奇怪的写法，第 1~37 是 pipeline_lut_multiplier_module_2.v 组合模块，其中实例化了步骤 1 功能模块（18 行），步骤 2 功能模块（23 行），步骤 3 查表模块（28~29 行）。步骤 1~2 的功能模块是调用内部的模块，然而步骤 3，亦即 LUT 模块是调用外部的模块。

第 39~81 行，是步骤 1 的功能模块，该模块的功能很简单，从组合模块的外部读入 A 和 B 的值，取得 I1 和 I2 后输出它们。

第 85~125 行，是步骤 2 的功能模块，该模块的功能同样也很简单。从步骤 1 的功能模块取得 I1 和 I2 的值然后正值化，并且输出。

在 28~29 行实例化的 LUT 模块，是扮演者步骤 3 的功能模块。它从步骤 2 的功能模块，取得正值化后的 I1 和 I2，然后经查表，将结果输出至 Q1_Sig 和 Q2_Sig。

最后的工作就是发生在 33 行。Product 输出信号，是由组合逻辑 $Q1_Sig + Q2_Sig$ 补来驱动。这样的写法可以再简化一个步骤。

pipeline_lut_multiplier_module_2.vt

```
1. `timescale 1 ps/ 1 ps
2. module pipeline_lut_multiplier_module_2_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]A;
8.     reg [7:0]B;
9.
10.    wire [15:0]Product;
11.
12.    /*****
13.
14.    pipeline_lut_multiplier_module_2 U1
15.    (
16.        .CLK(CLK),
17.        .RSTn(RSTn),
18.        .A(A),
19.        .B(B),
20.        .Product(Product)
21.    );
22.
23.    *****/
```

```
24.  
25.     initial  
26.     begin  
27.         RSTn = 0; #10; RSTn = 1;  
28.         CLK = 0; forever #10 CLK = ~CLK;  
29.     end  
30.  
31.     /*****  
32.  
33.     reg [3:0]i;  
34.  
35.     always @ ( posedge CLK or negedge RSTn )  
36.         if( !RSTn )  
37.             begin  
38.  
39.                 i <= 4'd0;  
40.                 A <= 8'd0;  
41.                 B <= 8'd0;  
42.  
43.             end  
44.         else  
45.             case( i )  
46.  
47.                 0:  
48.                     begin A <= 8'd127; B <= 8'd127; i <= i + 1'b1; end  
49.  
50.                 1:  
51.                     begin A <= 8'd10; B <= 8'd12; i <= i + 1'b1; end  
52.  
53.                 2:  
54.                     begin A <= 8'd32; B <= 8'd74; i <= i + 1'b1; end  
55.  
56.                 3:  
57.                     begin A <= 8'b10000001; B <= 8'd20; i <= i + 1'b1; end  
58.  
59.                 4:  
60.                     begin A <= 8'd0; B <= 8'd0; i <= 4'd4; end  
61.  
62.             endcase  
63.  
64.     /*****  
65.  
66.  
67. endmodule
```

仿真结果：



仿真的结果和实验十一的仿真结果既是一模一样。

实验十二说明：

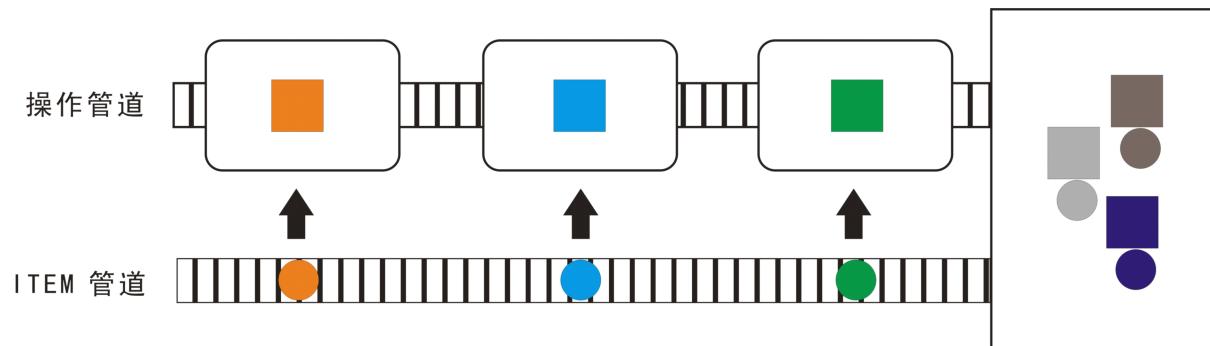
实验十二和实验十一相比，Verilog HDL 语言的表达是不是更直接，更容易理解呢？这样的建模方法，不但更能清楚表达流水操作的目的，它还使得代码更容易维护。

实验十二结论：

无论是实验十一还是实验十二，该流水操作的只有数据而没有“携带包袱 ... ”，这句话又是什么意思呢？

3.4 有包袱的流水操作

“有包袱的流水操作”看到这句话，笔者莫名的笑了起来。我们往往会遇见，一些数据被处理的时候，常常要参考该数据“携带的恒定值”。在流水操作中，假设每一个数据都有一个恒定值，当这一个数据从一个步骤进入另一个步骤，该恒定值也必须同步移动，类似的数据称为“包袱”或者“包袱数据”



如果形象一点来表示，有包袱的流水操作基本上会分为“操作管道”和“ITEM 管道”。操作管道负责“载运”要操作的数据，反之 ITEM 管道“载运”要参考的数据（要操作的数据不是恒定，要参考的数据是恒定的）。

上图中，圆角的矩形都是操作步骤，途中有 3 个圆角矩形，所以以上的流水操作需要 3 个步骤。左边矩形和圆形的集合体是原始数据，矩形表示要操作的数据，圆形表示要参考的数据。一旦这些集合体送入流水操作，它们都会被分离。

矩形（要操作的数据）被载运到操作管道，圆形（要参考的数据）被载运到 ITEM 管道。矩形移动或者被操作，圆形一定会同步跟随者。图中黑色向上的箭头，表示矩形被操作的时候，参考圆形。

废话少说还是直接上一个实验更直接。

实验十三：流水式 BOOTH 乘法器

在这里就使用实验四已经改进的 BOOTH 乘法器作为实验十三的基础。我们稍微回忆一下该乘法器的操作步骤：

- (一) 初始化 p 空间而填入乘数，取得 a (被乘数 A) 和 s (被乘数 A 负值的补码形式)。
- (二) 重复 8 次的循环操作。

我们知道在实验四中，寄存器 a 和寄存器 s 的值是不变，反之 p 空间的值会改变。所以说 a 和 s 都是 p 的包袱。p 空间要送往操作管道，a 和 s 要送往 ITEM 管道。

如果要把实验四的仿顺序操作转化为流水操作，该如何是好？实验四中的 Booth 算法，大致上有 9 个步骤。照原理来说，必须建立一维 9 个元素的 p 空间，a 寄存器和 s 寄存器。但是我们知道 a 寄存器和 s 寄存器的值是恒定，所以它们很乐意被集合在 Item 的寄存器里。

```
reg [16:0]p [8:0];
reg [15:0]Item [8:0]; // Item[15:8] 被乘数 A 负值补码 , Item[7:0] 被乘数 A

always @ ( posedge CLK )
begin

    p[0] <= { 8'd0 , B , 1'b0 };
    Item[0] <= { ~A + 1'b1 , A }; // { 被乘数 A 负值补码, 被乘数 A }

    .....

```

pipeline_booth_multiplier_module.v

```
1. module pipeline_booth_multiplier_module
2. (
3.     input CLK,
4.
5.     input [7:0]A,
6.     input [7:0]B,
7.
8.     output [15:0]product,
9.
10.    /*****
11.
12.    output [16:0]SQ_p0,
13.    output [16:0]SQ_p1,
14.    output [16:0]SQ_p2,
15.    output [16:0]SQ_p3,
16.    output [16:0]SQ_p4,
17.    output [16:0]SQ_p5,
18.    output [16:0]SQ_p6,
19.    output [16:0]SQ_p7,
20.
21.    output [15:0]SQ_Item0,
22.    output [15:0]SQ_Item1,
```

```

23.      output [15:0]SQ_Item2,
24.      output [15:0]SQ_Item3,
25.      output [15:0]SQ_Item4,
26.      output [15:0]SQ_Item5,
27.      output [15:0]SQ_Item6,
28.      output [15:0]SQ_Item7
29.
30.      *****/
31.
32. );
33.
34. *****/
35.
36. reg [16:0]p [8:0];
37. reg [15:0]Item [7:0];
38. reg [7:0]Diff1 [7:0];
39. reg [7:0]Diff2 [7:0];
40.
41. always @ ( posedge CLK )
42.     begin
43.
44.     *****/ // Step Initial
45.
46.     p[0] <= { 8'd0, B , 1'b0 };
47.     Item[0] <= { ~A + 1'b1 , A };
48.
49.     *****/ // Step 0
50.
51.     Diff1[0] = p[0][16:9] + Item[0][7:0];
52.     Diff2[0] = p[0][16:9] + Item[0][15:8];
53.
54.     if( p[0][1:0] == 2'b01 ) p[1] <= { Diff1[0][7] , Diff1[0] , p[0][8:1] };
55.     else if( p[0][1:0] == 2'b10 ) p[1] <= { Diff2[0][7] , Diff2[0] , p[0][8:1] };
56.     else p[1] <= { p[0][16] , p[0][16:1] };
57.
58.     Item[1] <= Item[0];
59.
60.     *****/ // Step 1
61.
62.     Diff1[1] = p[1][16:9] + Item[1][7:0];
63.     Diff2[1] = p[1][16:9] + Item[1][15:8];
64.
65.     if( p[1][1:0] == 2'b01 ) p[2] <= { Diff1[1][7] , Diff1[1] , p[1][8:1] };
66.     else if( p[1][1:0] == 2'b10 ) p[2] <= { Diff2[1][7] , Diff2[1] , p[1][8:1] };
67.     else p[2] <= { p[1][16] , p[1][16:1] };

```

```
68.
69.           Item[2] <= Item[1];
70.
71.           **** // Step 2
72.
73.           Diff1[2] = p[2][16:9] + Item[2][7:0];
74.           Diff2[2] = p[2][16:9] + Item[2][15:8];
75.
76.           if( p[2][1:0] == 2'b01 ) p[3] <= { Diff1[2][7] , Diff1[2] , p[2][8:1] };
77.           else if( p[2][1:0] == 2'b10 ) p[3] <= { Diff2[2][7] , Diff2[2] , p[2][8:1] };
78.           else p[3] <= { p[2][16] , p[2][16:1] };
79.
80.           Item[3] <= Item[2];
81.
82.           **** // Step 3
83.
84.           Diff1[3] = p[3][16:9] + Item[3][7:0];
85.           Diff2[3] = p[3][16:9] + Item[3][15:8];
86.
87.           if( p[3][1:0] == 2'b01 ) p[4] <= { Diff1[3][7] , Diff1[3] , p[3][8:1] };
88.           else if( p[3][1:0] == 2'b10 ) p[4] <= { Diff2[3][7] , Diff2[3] , p[3][8:1] };
89.           else p[4] <= { p[3][16] , p[3][16:1] };
90.
91.           Item[4] <= Item[3];
92.
93.           **** // Step 4
94.
95.           Diff1[4] = p[4][16:9] + Item[4][7:0];
96.           Diff2[4] = p[4][16:9] + Item[4][15:8];
97.
98.           if( p[4][1:0] == 2'b01 ) p[5] <= { Diff1[4][7] , Diff1[4] , p[4][8:1] };
99.           else if( p[4][1:0] == 2'b10 ) p[5] <= { Diff2[4][7] , Diff2[4] , p[4][8:1] };
100.          else p[5] <= { p[4][16] , p[4][16:1] };
101.
102.         Item[5] <= Item[4];
103.
104.         **** // Step 5
105.
106.         Diff1[5] = p[5][16:9] + Item[5][7:0];
107.         Diff2[5] = p[5][16:9] + Item[5][15:8];
108.
109.         if( p[5][1:0] == 2'b01 ) p[6] <= { Diff1[5][7] , Diff1[5] , p[5][8:1] };
110.         else if( p[5][1:0] == 2'b10 ) p[6] <= { Diff2[5][7] , Diff2[5] , p[5][8:1] };
111.         else p[6] <= { p[5][16] , p[5][16:1] };
112.
```

```
113.           Item[6] <= Item[5];
114.
115.           /***** // Step 6
116.
117.           Diff1[6] = p[6][16:9] + Item[6][7:0];
118.           Diff2[6] = p[6][16:9] + Item[6][15:8];
119.
120.           if( p[6][1:0] == 2'b01 ) p[7] <= { Diff1[6][7] , Diff1[6] , p[6][8:1] };
121.           else if( p[6][1:0] == 2'b10 ) p[7] <= { Diff2[6][7] , Diff2[6] , p[6][8:1] };
122.           else p[7] <= { p[6][16] , p[6][16:1] };
123.
124.           Item[7] <= Item[6];
125.
126.           /***** // Step 7
127.
128.           Diff1[7] = p[7][16:9] + Item[7][7:0];
129.           Diff2[7] = p[7][16:9] + Item[7][15:8];
130.
131.           if( p[7][1:0] == 2'b01 ) p[8] <= { Diff1[7][7] , Diff1[7] , p[7][8:1] };
132.           else if( p[7][1:0] == 2'b10 ) p[8] <= { Diff2[7][7] , Diff2[7] , p[7][8:1] };
133.           else p[8] <= { p[7][16] , p[7][16:1] };
134.
135.           /***** // Step end
136.
137.
138.       end
139.
140.   *****/
141.
142.   assign product = p[8][16:1];
143.
144.   *****/
145.
146.   assign SQ_p0 = p[0];
147.   assign SQ_p1 = p[1];
148.   assign SQ_p2 = p[2];
149.   assign SQ_p3 = p[3];
150.   assign SQ_p4 = p[4];
151.   assign SQ_p5 = p[5];
152.   assign SQ_p6 = p[6];
153.   assign SQ_p7 = p[7];
154.
155.   assign SQ_Item0 = Item[0];
156.   assign SQ_Item1 = Item[1];
157.   assign SQ_Item2 = Item[2];
```

```

158. assign SQ_Item3 = Item[3];
159. assign SQ_Item4 = Item[4];
160. assign SQ_Item5 = Item[5];
161. assign SQ_Item6 = Item[6];
162. assign SQ_Item7 = Item[7];
163.
164. ****
165.
166.
167. endmodule

```

(好长呀 ...) 第 12~28 行, 是仿真输出。在 36 行, 和原理一样, 流水操作有 9 个操作步骤, 所以 p 空间建立一维为 9 个元素。Item 是用来寄存恒定数据, 如果按照原理来说, 它也必须建立一维为 9 个元素才对, 可是 “包袱数据的最后一个元素往往是无用的” 所以把它 “舍掉”, 成为 8 个元素。

Diff1 和 Diff2 空间是用来寄存 $a + p[16:9]$, $s + p[16:9]$ 的计时结果, 如果按照原理, 流水操作有 9 个步骤那么它们必须建立 9 个元素才对 ... 可是, 实际上在循环操作中它们才会出现, 如果排除初始化步骤, 那么 Diff1 和 Diff2 空间会是一维为 8 个元素。

ah 接下来长长的代码会看到使人心烦。

第 44~47 行是 **初始化步骤**, p[0] 空间的 [8..1] 填入乘数 B。Item[0] 的 [15..8] 用来寄存 s (或者被乘数-1A 的补码形式), [7..0] 用来寄存被乘数 A。

49~58 行是循环操作的 **第 1 次操作**。Diff1[0] 和 Diff2[0] 取得 $p[0][16:9] + Item[0][7:0]$ 的即时结果 (注意 “=”) 和 $p[0][16:9] + Item[0][15:8]$ 的即时结果 (51~52 行)。在 54~56 行, 取得 p[0] 的值, 并且判断 [1:0], 来执行相关的 Booth 加码操作, 然后赋值与 p[1]。58 行是 Item[0] 值移动到 Item[1] 去, 这是目前操作数据的参考数据, 所以必须同步更随着。

60~69 行是循环操作的 **第 2 次操作**。Diff1[1] 和 Diff2[1] 取得 $p[1][16:9] + Item[1][7:0]$ 的即时结果 (注意 “=”) 和 $p[1][16:9] + Item[1][15:8]$ 的即时结果 (62~63 行)。在 65~67 行, 取得 p[1] 的值, 并且判断 [1:0], 来执行相关的 Booth 加码操作, 然后赋值与 p[2]。69 行是 Item[1] 值移动到 Item[2] 去, 这是目前操作数据的参考数据, 所以必须同步更随着。

71~80 行是循环操作的 **第 3 次操作**。Diff1[2] 和 Diff2[2] 取得 $p[2][16:9] + Item[2][7:0]$ 的即时结果 (注意 “=”) 和 $p[2][16:9] + Item[2][15:8]$ 的即时结果 (73~74 行)。在 76~78 行, 取得 p[2] 的值, 并且判断 [1:0], 来执行相关的 Booth 加码操作, 然后赋值与 p[3]。80 行是 Item[2] 值移动到 Item[3] 去, 这是目前操作数据的参考数据, 所以必须同步更随着。

82~91 行是循环操作的 **第 4 次操作**。Diff1[3] 和 Diff2[3] 取得 $p[3][16:9] + Item[3][7:0]$

的即时结果（注意“=”）和 $p[3][16:9] + Item[3][15:8]$ 的即时结果（84~85 行）。在 87~89 行，取得 $p[3]$ 的值，并且判断 [1:0]，来执行相关的 Booth 加码操作，然后赋值与 $p[4]$ 。91 行是 $Item[3]$ 值移动到 $Item[4]$ 去，这是目前操作数据的参考数据，所以必须同步更随着。

93~102 行是循环操作的第 5 次操作。Diff1[4] 和 Diff[4] 取得 $p[4][16:9] + Item[4][7:0]$ 的即时结果（注意“=”）和 $p[4][16:9] + Item[4][15:8]$ 的即时结果（95~96 行）。在 98~100 行，取得 $p[4]$ 的值，并且判断 [1:0]，来执行相关的 Booth 加码操作，然后赋值与 $p[5]$ 。102 行是 $Item[4]$ 值移动到 $Item[5]$ 去，这是目前操作数据的参考数据，所以必须同步更随着。

104~113 行是循环操作的第 6 次操作。Diff1[5] 和 Diff[5] 取得 $p[5][16:9] + Item[5][7:0]$ 的即时结果（注意“=”）和 $p[5][16:9] + Item[5][15:8]$ 的即时结果（106~107 行）。在 109~111 行，取得 $p[5]$ 的值，并且判断 [1:0]，来执行相关的 Booth 加码操作，然后赋值与 $p[6]$ 。113 行是 $Item[5]$ 值移动到 $Item[4]$ 去，这是目前操作数据的参考数据，所以必须同步更随着。

115~124 行是循环操作的第 7 次操作。Diff1[6] 和 Diff[6] 取得 $p[6][16:9] + Item[6][7:0]$ 的即时结果（注意“=”）和 $p[6][16:9] + Item[6][15:8]$ 的即时结果（117~118 行）。在 120~122 行，取得 $p[6]$ 的值，并且判断 [1:0]，来执行相关的 Booth 加码操作，然后赋值与 $p[7]$ 。124 行是 $Item[6]$ 值移动到 $Item[7]$ 去，这是目前操作数据的参考数据，所以必须同步更随着。

126~133 行是循环操作的第 8 次操作。Diff1[7] 和 Diff[7] 取得 $p[7][16:9] + Item[7][7:0]$ 的即时结果（注意“=”）和 $p[7][16:9] + Item[7][15:8]$ 的即时结果（128~129 行）。在 131~133 行，取得 $p[7]$ 的值，并且判断 [1:0]，来执行相关的 Booth 加码操作，然后赋值与 $p[8]$ 。这是最后一步的循环操作，目前操作数据的参考数据，再也没有用途了，所以可以废除掉（好现实 ...）。

在这里笔者需要给自己打一个提醒针，42~138 行，每一个时钟每一个步骤都在执行着！

在 142 行，Product 的数据是由 $p[8][16:1]$ 驱动。因为 $p[8]$ 的数据，是已经完成操作的数据。第 146~162 行是仿真输出。146~153 行是 p 空间 [0..7] 的仿真输出（ $p[8]$ 已经驱动 Product 信号了 - 142 行）。155~162 行是“Item 管道”的仿真输出。

pipeline_booth_multiplier_module.vt

```

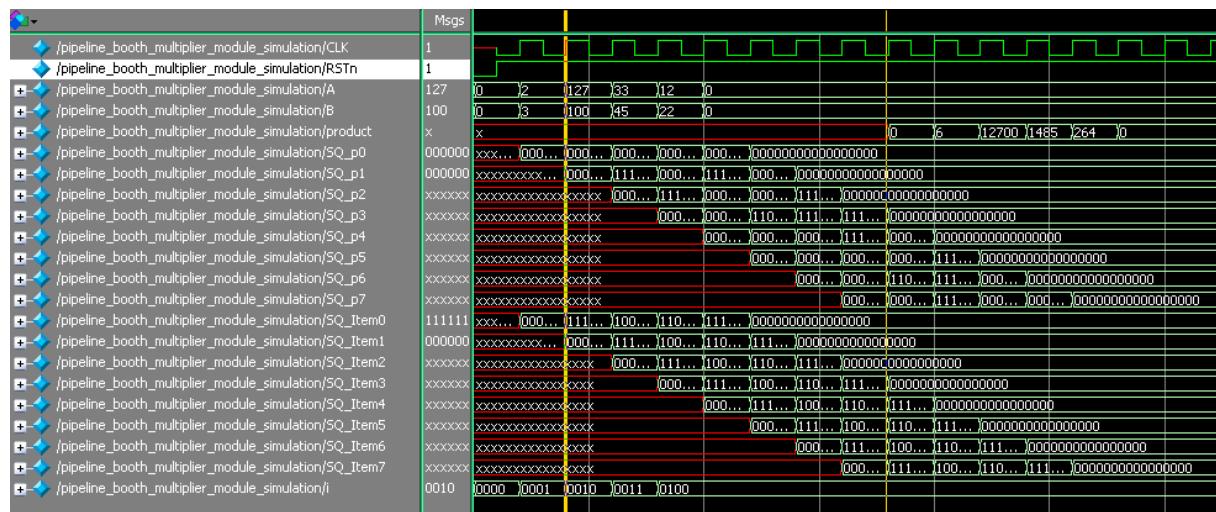
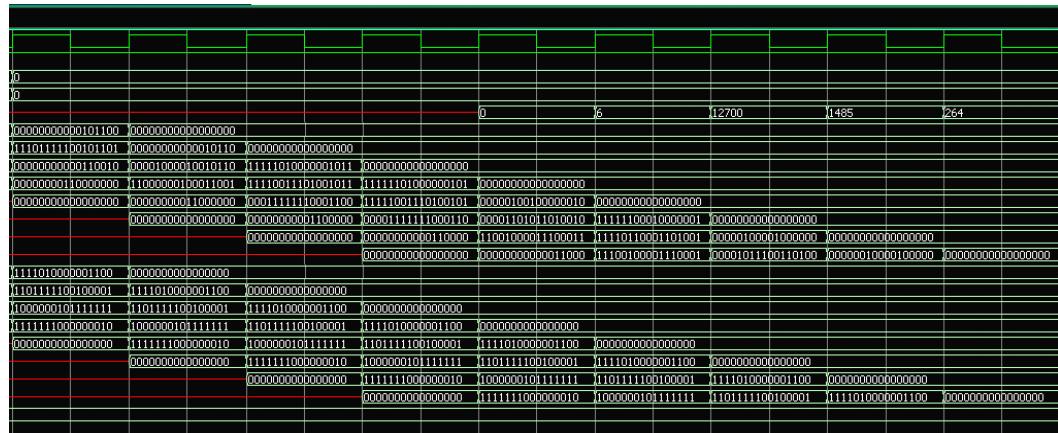
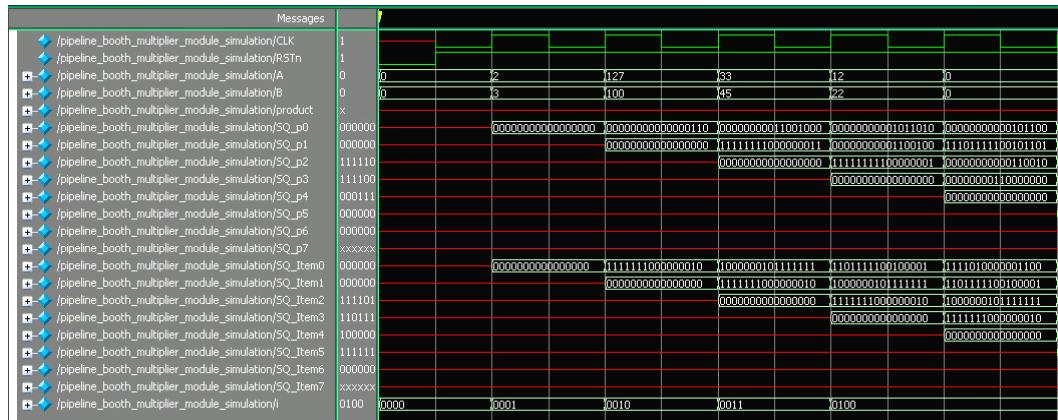
1. `timescale 1 ps/ 1 ps
2. module pipeline_booth_multiplier_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]A;
```

```
8.      reg [7:0]B;
9.
10.     wire [15:0]product;
11.
12.     wire [16:0]SQ_p0;
13.     wire [16:0]SQ_p1;
14.     wire [16:0]SQ_p2;
15.     wire [16:0]SQ_p3;
16.     wire [16:0]SQ_p4;
17.     wire [16:0]SQ_p5;
18.     wire [16:0]SQ_p6;
19.     wire [16:0]SQ_p7;
20.
21.     wire [15:0]SQ_Item0;
22.     wire [15:0]SQ_Item1;
23.     wire [15:0]SQ_Item2;
24.     wire [15:0]SQ_Item3;
25.     wire [15:0]SQ_Item4;
26.     wire [15:0]SQ_Item5;
27.     wire [15:0]SQ_Item6;
28.     wire [15:0]SQ_Item7;
29.
30.   *****/
31.
32. pipeline_booth_multiplier_module i1
33. (
34.     .CLK(CLK),
35.     .A(A),
36.     .B(B),
37.     .product(product),
38.     .SQ_p0( SQ_p0 ),
39.     .SQ_p1( SQ_p1 ),
40.     .SQ_p2( SQ_p2 ),
41.     .SQ_p3( SQ_p3 ),
42.     .SQ_p4( SQ_p4 ),
43.     .SQ_p5( SQ_p5 ),
44.     .SQ_p6( SQ_p6 ),
45.     .SQ_p7( SQ_p7 ),
46.     .SQ_Item0( SQ_Item0 ),
47.     .SQ_Item1( SQ_Item1 ),
48.     .SQ_Item2( SQ_Item2 ),
49.     .SQ_Item3( SQ_Item3 ),
50.     .SQ_Item4( SQ_Item4 ),
51.     .SQ_Item5( SQ_Item5 ),
52.     .SQ_Item6( SQ_Item6 ),
```

```
53.           .SQ_Item7( SQ_Item7 )
54.       );
55.
56.   ****
57.
58.   initial
59.   begin
60.       RSTn = 0; #10; RSTn = 1;
61.       CLK = 0; forever #10 CLK = ~CLK;
62.   end
63.
64.   ****
65.
66.   reg [3:0]i;
67.
68.   always @ ( posedge CLK or negedge RSTn )
69.       if( !RSTn )
70.           begin
71.
72.               i <= 4'd0;
73.               A <= 8'd0;
74.               B <= 8'd0;
75.
76.           end
77.       else
78.           case( i )
79.
80.               0:
81.                   begin A <= 8'd2; B <= 8'd3; i <= i + 1'b1; end
82.
83.               1:
84.                   begin A <= 8'd127; B <= 8'd100; i <= i + 1'b1; end
85.
86.               2:
87.                   begin A <= 8'd33; B <= 8'd45; i <= i + 1'b1; end
88.
89.               3:
90.                   begin A <= 8'd12; B <= 8'd22; i <= i + 1'b1; end
91.
92.               4:
93.                   begin A <= 8'd0; B <= 8'd0; end
94.
95.           endcase
96.
97.   endmodule
```

激励文件还是一如既往的风格，自己看着办吧。

仿真结果：



哦！仿真图太长了，需要宰掉几段才行。如果要看完美的还是自行打开 Modelsim 来看吧。第一和第二张仿真结果是流水式 BOOTH 乘法器的详细执行过程，然而第三章仿真结果是全程的执行过程，两个直线之间是该乘法器的潜伏时间。

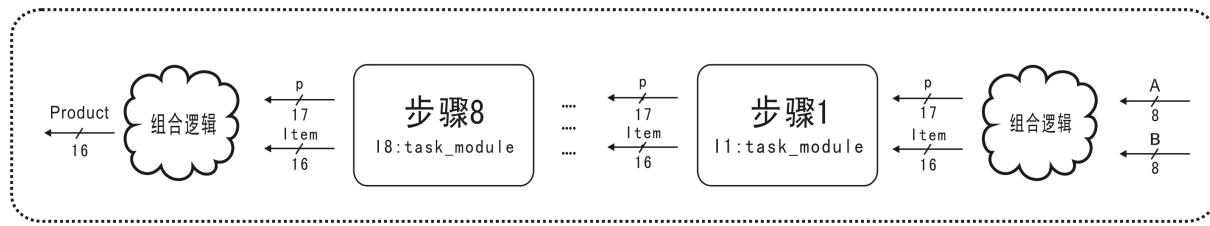
实验十三说明：

实验十三中 p 空间被载运至操作管道，反之 Item 空间被载运至 Item 管道。

实验十三结论：

嗯！没有建模化的流水操作，代码看得眼睛很疼，感觉很长很臭 ...

实验十四：建模过后的流水式 BOOTH 乘法器



`pipeline_booth_multiplier_module_2.v`

在这里我们再借用一下低级建模的力量。`pipeline_booth_multiplier_module_2.v` 是组合模块，里边包含了 I1~I8 (这里 i 的意思不是 instance 而是步骤 i 的 i) 八个同样的 `task_module` 功能模块。

在实验十三，我们知道 Booth 算法需要 9 个步骤，其中第一个步骤就是初始化 p 空间和 Item 寄存器，然而实验十四的初始化步骤由组合逻辑简化了。换句话说，初始化步骤被组合逻辑取代了，余下的就有 8 个循环操作步骤而已。

此外我们还知道 8 个循环操作步骤都是一样的“内容”，在这里笔者只是建立一个功能模块，然后实例化 8 次。最终结果如上面的图形（还是看源码比较直接）。

`pipeline_booth_multiplier_module_2.v`

```

1. module pipeline_booth_multiplier_module_2
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [7:0]A,
7.     input [7:0]B,
8.
9.     output [15:0]Product
10. );
11.
12. ****
13.
14. wire [16:0]I1_P_Out;
15. wire [15:0]I1_Item_Out;
16.
17. task_module I1( CLK, RSTn, { 8'd0, B , 1'b0}, { ~A + 1'b1, A }, I1_P_Out, I1_Item_Out );
18.
19. ****

```

```
20.  
21.     wire [16:0]I2_P_Out;  
22.     wire [15:0]I2_Item_Out;  
23.  
24.     task_module I2( CLK, RSTn, I1_P_Out, I1_Item_Out, I2_P_Out, I2_Item_Out );  
25.  
26.     /*****  
27.  
28.     wire [16:0]I3_P_Out;  
29.     wire [15:0]I3_Item_Out;  
30.  
31.     task_module I3( CLK, RSTn, I2_P_Out, I2_Item_Out, I3_P_Out, I3_Item_Out );  
32.  
33.     /*****  
34.  
35.     wire [16:0]I4_P_Out;  
36.     wire [15:0]I4_Item_Out;  
37.  
38.     task_module I4( CLK, RSTn, I3_P_Out, I3_Item_Out, I4_P_Out, I4_Item_Out );  
39.  
40.     /*****  
41.  
42.     wire [16:0]I5_P_Out;  
43.     wire [15:0]I5_Item_Out;  
44.  
45.     task_module I5( CLK, RSTn, I4_P_Out, I4_Item_Out, I5_P_Out, I5_Item_Out );  
46.  
47.     /*****  
48.  
49.     wire [16:0]I6_P_Out;  
50.     wire [15:0]I6_Item_Out;  
51.  
52.     task_module I6( CLK, RSTn, I5_P_Out, I5_Item_Out, I6_P_Out, I6_Item_Out );  
53.  
54.     /*****  
55.  
56.     wire [16:0]I7_P_Out;  
57.     wire [15:0]I7_Item_Out;  
58.  
59.     task_module I7( CLK, RSTn, I6_P_Out, I6_Item_Out, I7_P_Out, I7_Item_Out );  
60.  
61.     /*****  
62.  
63.     wire [16:0]I8_P_Out;  
64.     wire [15:0]I8_Item_Out;
```

```
65.  
66.      task_module I8( CLK, RSTn, I7_P_Out, I7_Item_Out, I8_P_Out, I8_Item_Out );  
67.  
68.      /*****  
69.  
70.  
71.      assign Product = I8_P_Out[16:1];  
72.  
73.      /*****  
74.      /*****  
75.  
76.  
77.  
78. endmodule  
79.  
80.  
81.  
82. module task_module  
83. (  
84.  
85.     input CLK,  
86.     input RSTn,  
87.  
88.     input [16:0]P_In,  
89.     input [15:0]Item_In,  
90.  
91.     output [16:0]P_Out,  
92.     output [15:0]Item_Out  
93.  
94. );  
95.  
96.      /*****  
97.  
98.      reg [16:0]p;  
99.      reg [15:0]Item;  
100.     reg [7:0]Diff1;  
101.     reg [7:0]Diff2;  
102.  
103.    always @ ( posedge CLK or negedge RSTn )  
104.        if( !RSTn )  
105.            begin  
106.  
107.                p <= 17'd0;  
108.                Item <= 16'd0;  
109.
```

```

110.          end
111.      else
112.          begin
113.
114.              Diff1 = P_In[16:9] + Item_In[7:0];
115.              Diff2 = P_In[16:9] + Item_In[15:8];
116.
117.              if( P_In[1:0] == 2'b01 ) p <= { Diff1[7] , Diff1 , P_In[8:1] };
118.              else if( P_In[1:0] == 2'b10 ) p <= { Diff2[7] , Diff2 , P_In[8:1] };
119.              else p <= { P_In[16] , P_In[16:1] };
120.
121.              Item <= Item_In;
122.
123.          end
124.
125.      /*****
126.
127.      assign P_Out = p;
128.      assign Item_Out = Item;
129.
130.  *****/
131.
132.
133. endmodule

```

(笔者再强调一次, 这里的 I1 不是 instance 的 i 而是步骤 i 的 i, 这样记的方法可以使逻辑思维更清晰。)

在 17 行中的 $\{ 8'd0, B, 1'b0 \}$ 和 $\{ \sim A + 1'b1, A \}$ (上图图形右边的组合逻辑)。该组合逻辑简化了初始化的步骤。然而这两组合逻辑的驱动对象是 i1 的 p 和 Item。

82~133 行是 task_module, 亦即循环操作的功能模块。在 85~92 行的模块输入输出端中, 88 行是 p 空间“操作管道”的进口(task_module 功能模块其中之一输入口), 然而 89 行是 Item 寄存器的“Item 管道”的进口(task_module 功能模块其中之一输入口)。相反的 91~92 行是“操作管道”和“Item 管道”的输出口(task_module 功能模块的输出口)。

98~101 行分别建立了相关的寄存器, 寄存器 p 用来驱动 P_out 的输出 (127 行), 寄存器 Item 则用来驱动 Item_Out 的输出 (128 行)。112~123 行是该功能模块的主功能, 具体的操作和实验十三非常类似, 不同的只是实验十四时用建模来表达流水操作。

14~66 行是按照图形来实例化模块的, 具体的连线关系和图形是大同小异。第 71 行 Product 输出的驱动是 I8_P_Out[16:1] (是图形中左边的组合逻辑)。在 66 行, 读者有没有注意到 I8_Item_Out 引出后就被抛弃了, 因为第 8 个 task_module, 是最后一个流水操作所以“包袱”再也不需要了 ...

pipeline_booth_multiplier_module_2.vt

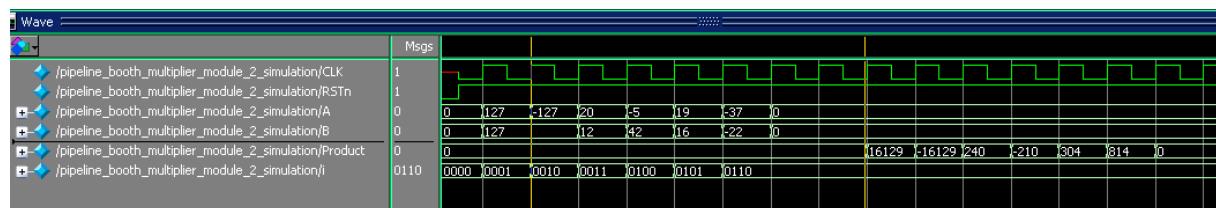
```
1. `timescale 1 ps/ 1 ps
2. module pipeline_booth_multiplier_module_2_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]A;
8.     reg [7:0]B;
9.
10.    wire [15:0]Product;
11.
12.    /*****
13.
14.    pipeline_booth_multiplier_module_2 U1
15.    (
16.        .CLK(CLK),
17.        .RSTn(RSTn),
18.        .A(A),
19.        .B(B),
20.        .Product(Product)
21.    );
22.
23.    *****/
24.
25.    initial
26.    begin
27.        RSTn = 0; #10; RSTn = 1;
28.        CLK = 0; forever #10 CLK = ~CLK;
29.    end
30.
31.    *****/
32.
33.    reg [3:0]i;
34.
35.    always @ ( posedge CLK or negedge RSTn )
36.        if( !RSTn )
37.            begin
38.
39.                i <= 4'd0;
40.                A <= 8'd0;
```

```

41.          B <= 8'd0;
42.
43.      end
44.  else
45.      case( i )
46.
47.          0: // A = 127 , B = 127
48.          begin A <= 8'd127; B <= 8'd127; i <= i + 1'b1; end
49.
50.          1: // A = -127 , B = 127
51.          begin A <= 8'b10000001; B <= 8'd127; i <= i + 1'b1; end
52.
53.          2: // A = 20 , B = 12
54.          begin A <= 8'd20; B <= 8'd12; i <= i + 1'b1; end
55.
56.          3: // A = -5 , B = 42
57.          begin A <= 8'b11111011; B <= 8'd42; i <= i + 1'b1; end
58.
59.          4: // A = 19 , B = 16
60.          begin A <= 8'd19; B <= 8'd16; i <= i + 1'b1; end
61.
62.          5: // A = -37 , B = -22
63.          begin A <= 8'b11011011; B <= 8'b11101010; i <= i + 1'b1; end
64.
65.          6:
66.          begin A <= 8'd0; B <= 8'd0; i <= 4'd6; end
67.
68.      endcase
69.
70.  *****/
71.
72. endmodule

```

仿真结果：



上图是仿真结果。比起还没有建模化的流水式 Booth 乘法器，建模化过后的流水式 Booth 乘法器更简洁不少。

实验十四说明:

再一次见证建模威武！建模过后的流水操作，表达能力不但提高，而且还简化了步骤和
减少时钟，真是可惜可贺！

实验十四结论:

```
reg [16:0]Item[7:0];
.....
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        begin
            Item[0] <= 16'd0; Item[1] <= 16'd0; .....
            ..... // 呜呜 (ㄒ o ㄒ), 饶了我吧 , 好多的储存器元素要初始化。
```

在实验十三，由于流水操作没有建模化，要初始化一大堆的储存器，真的是一场悲剧，所以笔者就索性不要加入 RSTn 复位信号。反之经过建模化后的流水操作，减少了对储存器的依赖，寄存器的初始化也变得方便许多。因为流水操作经过建模化后，我们只要针对“某个模块的源实例”初始化相关的寄存器即可。

3.5 流水操作直接建模

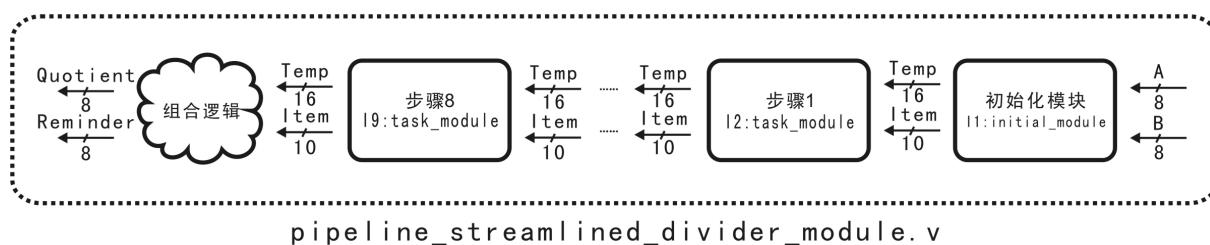
从实验十一到实验十四，流水操作的建模都是先经过“仿顺序操作 => 建立流水操作”的转换，然后再将“流水操作建模化”。这一章我们就干直接一点事情，我们直接从流水操作向建模执行。

实验十五：流水式循环除法器

实验十五主要是基于实验九-循环型除法器的改进。实验九的循环型除法器主要步骤如下：

- (一) 初始化要作的工作：取得除数和被除数的正负关系。取得除数负值化后的补码形式。取得被除数正值化，并且初始化 Temp 空间。
- (二) 执行 8 次的循环除法运算操作。

如果我们深入分析实验九的话，我们可以发现除数和被除数的正负关系 isNeg 和除数负值化的补码 s 都是不变的“恒定值”，亦即“包袱数据”。然而 Temp 空间里的数据是“操作数据”



上图的图形是组合模块 `pipeline_streamlined_divider_module.v`，里边包含了 1 个 `initial_module` 和 8 个 `task_module` 功能模块，最后输出由组合逻辑驱动。各个功能模块的功能如命名般一样，非常直接。`initial_module` 是建立“操作数据”和“参考数据”，然而 `task_module` 是循环操作的功能，实验中有 8 个的 `task_module` 的实例，亦即有 8 次的循环操作。

`pipeline_streamlined_divider_module.v`

```
1. module pipeline_streamlined_divider_module
2. (
3.
4.     input CLK,
5.     input RSTn,
```

```
6.  
7.      input [7:0]Dividend,  
8.      input [7:0]Divisor,  
9.  
10.     output [7:0]Quotient,  
11.     output [7:0]Reminder  
12.  
13. );
14.  
15. ****  
16.  
17.     wire [15:0]I0_Temp_Out;  
18.     wire [9:0]I0_Item_Out;  
19.  
20.     initial_module I0( CLK, RSTn, Dividend, Divisor, I0_Temp_Out, I0_Item_Out );  
21.  
22. ****  
23.  
24.     wire [15:0]I1_Temp_Out;  
25.     wire [9:0]I1_Item_Out;  
26.  
27.     task_module I1( CLK, RSTn, I0_Temp_Out, I0_Item_Out, I1_Temp_Out, I1_Item_Out );  
28.  
29. ****  
30.  
31.     wire [15:0]I2_Temp_Out;  
32.     wire [9:0]I2_Item_Out;  
33.  
34.     task_module I2( CLK, RSTn, I1_Temp_Out, I1_Item_Out, I2_Temp_Out, I2_Item_Out );  
35.  
36. ****  
37.  
38.     wire [15:0]I3_Temp_Out;  
39.     wire [9:0]I3_Item_Out;  
40.  
41.     task_module I3( CLK, RSTn, I2_Temp_Out, I2_Item_Out, I3_Temp_Out, I3_Item_Out );  
42.  
43. ****  
44.  
45.     wire [15:0]I4_Temp_Out;  
46.     wire [9:0]I4_Item_Out;  
47.  
48.     task_module I4( CLK, RSTn, I3_Temp_Out, I3_Item_Out, I4_Temp_Out, I4_Item_Out );  
49.  
50. ****
```

```
51.  
52.     wire [15:0]I5_Temp_Out;  
53.     wire [9:0]I5_Item_Out;  
54.  
55.     task_module I5( CLK, RSTn, I4_Temp_Out, I4_Item_Out, I5_Temp_Out, I5_Item_Out );  
56.  
57.     /*****  
58.  
59.     wire [15:0]I6_Temp_Out;  
60.     wire [9:0]I6_Item_Out;  
61.  
62.     task_module I6( CLK, RSTn, I5_Temp_Out, I5_Item_Out, I6_Temp_Out, I6_Item_Out );  
63.  
64.     /*****  
65.  
66.     wire [15:0]I7_Temp_Out;  
67.     wire [9:0]I7_Item_Out;  
68.  
69.     task_module I7( CLK, RSTn, I6_Temp_Out, I6_Item_Out, I7_Temp_Out, I7_Item_Out );  
70.  
71.     /*****  
72.  
73.     wire [15:0]I8_Temp_Out;  
74.     wire [9:0]I8_Item_Out;  
75.  
76.     task_module I8( CLK, RSTn, I7_Temp_Out, I7_Item_Out, I8_Temp_Out, I8_Item_Out );  
77.  
78.     /*****  
79.  
80.     assign Quotient = I8_Item_Out[9] ? (~I8_Temp_Out[7:0] + 1'b1) : I8_Temp_Out[7:0];  
81.     assign Reminder = I8_Temp_Out[15:8];  
82.  
83.     /*****  
84.  
85. endmodule  
86.  
87.  
88. /*****  
89.  
90. module initial_module  
91. (  
92.     input CLK,  
93.     input RSTn,  
94.  
95.     input [7:0]Dividend,
```

```
96.      input [7:0]Divisor,
97.
98.      output [15:0]Temp_Out,
99.      output [9:0]Item_Out
100.
101. );
102.
103.      reg [15:0]Temp;
104.      reg [9:0]Item;
105.
106.      always @ ( posedge CLK or negedge RSTn )
107.          if( !RSTn )
108.              begin
109.                  Temp <= 16'd0;
110.                  Item <= 10'd0;
111.              end
112.          else
113.              begin
114.
115.                  Item[9] <= Dividend[7] ^ Divisor[7];
116.                  Item[8:0] <= Divisor[7] ? { 1'b1, Divisor } : { 1'b1 , ~Divisor + 1'b1 };
117.                  Temp <= Dividend[7] ? { 8'd0 , ~Dividend + 1'b1 } : { 8'd0 , Dividend };
118.
119.              end
120.
121.      *****/
122.
123.      assign Temp_Out = Temp;
124.      assign Item_Out = Item;
125.
126.      *****/
127.
128. endmodule
129.
130.
131. module task_module
132. (
133.     input CLK,
134.     input RSTn,
135.
136.     input [15:0]Temp_In,
137.     input [9:0]Item_In,
138.
139.     output [15:0]Temp_Out,
140.     output [9:0]Item_Out
```

```

141. );
142.
143. *****/
144.
145. reg [15:0]Diff;
146. reg [15:0]Temp;
147. reg [9:0]Item;
148.
149. always @ ( posedge CLK or negedge RSTn )
150.     if( !RSTn )
151.         begin
152.
153.             Diff <= 16'd0;
154.             Temp <= 16'd0;
155.             Item <= 10'd0;
156.
157.         end
158.     else
159.         begin
160.
161.             Diff = Temp_In + { Item_In[8:0] , 7'd0 };
162.
163.             if( Temp_In <= ( (~Item_In[8:0] + 1'b1) << 7 ) ) Temp <= { Temp_In[14:0] , 1'b0 };
164.             else Temp <= { Diff[14:0] , 1'b1 };
165.
166.             Item <= Item_In;
167.
168.         end
169.
170. *****/
171.
172. assign Temp_Out = Temp;
173. assign Item_Out = Item;
174.
175. *****/
176.
177. endmodule

```

第 90~128 行是 initial_module，它的工作主要是区分和建立“操作数据”和“参考数据”。它直接从顶层模块读入 Divisor 和 Dividend。

在 103~104 行，建立了 Temp 和 Item 寄存器。Item 的最高位 [9] 用来记录除数和被除数的正负关系（115 行），Item 的低九位 [8..1] 用来寄存 除数负值化的补码（116 行）。Temp 空间的[7..1] 用来填入被除数的正值化结果（117 行）。在 123~124 行输出信号

Temp_Out 和 Item_Out 由寄存器 Temp 和 Item 驱动。

第 131~177 行是循环操作的功能模块。在 136~137 行引入了 Temp_In 和 Item_In 的输入。第 145 行块声明了 Diff 寄存器，用于取得即时结果（161 行）。第 146~147 行声明寄存器 Temp 和 Item 寄存器，Temp 寄存器是用于循环除法操作的暂存空间（163~164 行），而 Item “包袱数据”的暂存空间，此外该寄存器们驱动着 Temp_Out 和 Item_Out 输出信号（172~173 行）。

在这里有一点必须注意就是 163 行 ($\sim\text{Item_In}[8:0] + 1'b1) << 7$) 的内容表示了，从 Item_In 的 [8..0] 取得 B 的负值，然后正直化它，然后取得 B'，最有用用于比较 Temp_In，亦即 R。在前面笔者已经说过，streamlined divider 是经过简化的除法器，它是基于 2.3 章的原理实现的，此外它的“简化”也造成它的不稳定，所以笔者才如此修改。（具体的内容请看 2.2 章和 2.3 章）

从 15~83 行是模块实例化和连线过程，具体和图形大同小异，笔者不罗嗦了，自己看着办吧。（80~81 行是图形中右边所指的组合逻辑）。

pipeline_streamlined_divider_module.vt

```

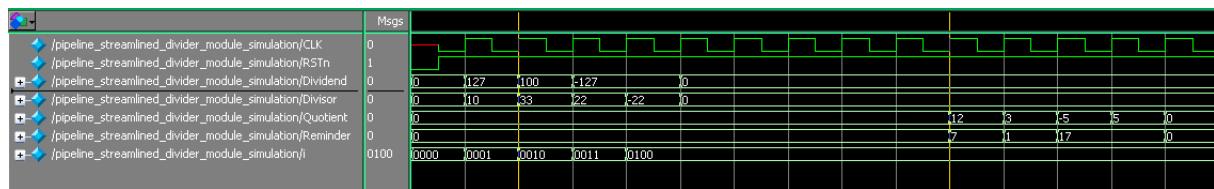
1. `timescale 1 ps/ 1 ps
2. module pipeline_streamlined_divider_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]Dividend;
8.     reg [7:0]Divisor;
9.
10.    wire [7:0]Quotient;
11.    wire [7:0]Reminder;
12.
13.    /*****
14.
15.    pipeline_streamlined_divider_module U1
16.    (
17.        .CLK(CLK),
18.        .RSTn(RSTn),
19.        .Dividend(Dividend),
20.        .Divisor(Divisor),
21.        .Quotient(Quotient),
22.        .Reminder(Reminder)
23.    );
24.
25.    *****/

```

```
26.  
27.      initial  
28.      begin  
29.  
30.          RSTn = 0; #10; RSTn = 1;  
31.          CLK = 0; forever #10 CLK = ~CLK;  
32.  
33.      end  
34.  
35.      /*****  
36.  
37.      reg [3:0]i;  
38.  
39.      always @ ( posedge CLK or negedge RSTn )  
40.          if( !RSTn )  
41.              begin  
42.                  i <= 4'd0;  
43.                  Dividend <= 8'd0;  
44.                  Divisor <= 8'd0;  
45.              end  
46.          else  
47.              case( i )  
48.  
49.                  0: // Dividend = 127, Divisor = 10  
50.                  begin Dividend <= 8'd127; Divisor <= 8'd10; i <= i + 1'b1; end  
51.  
52.                  1: // Dividend = 100, Divisor = 33  
53.                  begin Dividend <= 8'd100; Divisor <= 8'd33; i <= i + 1'b1; end  
54.  
55.                  2: // Dividend = -127, Divisor = 22  
56.                  begin Dividend <= 8'b10000001; Divisor <= 8'd22; i <= i + 1'b1; end  
57.  
58.                  3: // Dividend = -127, Divisor = -22  
59.                  begin Dividend <= 8'b10000001; Divisor <= 8'b11101010; i <= i + 1'b1; end  
60.  
61.                  4:  
62.                  begin Dividend <= 8'd0; Divisor <= 8'd0; i <= 4'd4; end  
63.  
64.              endcase  
65.  
66.      /*****  
67.  
68.  endmodule
```

还是一如既往的写法

仿真结果:



根据原理，该组合模块拥有 9 个操作步骤，所以潜伏时间是 9 个时钟。

实验十五说明:

嗯！只要明白了 3.1~3.4 章的原理，直接性的流水操作建模化是一件非常简单的事。

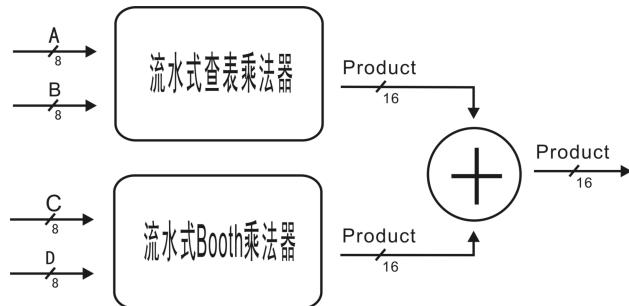
实验十五结论:

实验十一到实验十五，笔者都是在建立单个流水操作的模块而已。如果笔者打算把两个流水模块并连起来的话

3.6 当不同操作步骤的流水操作模块并连的时候 ...

$$(a \times b) + (c \times d)$$

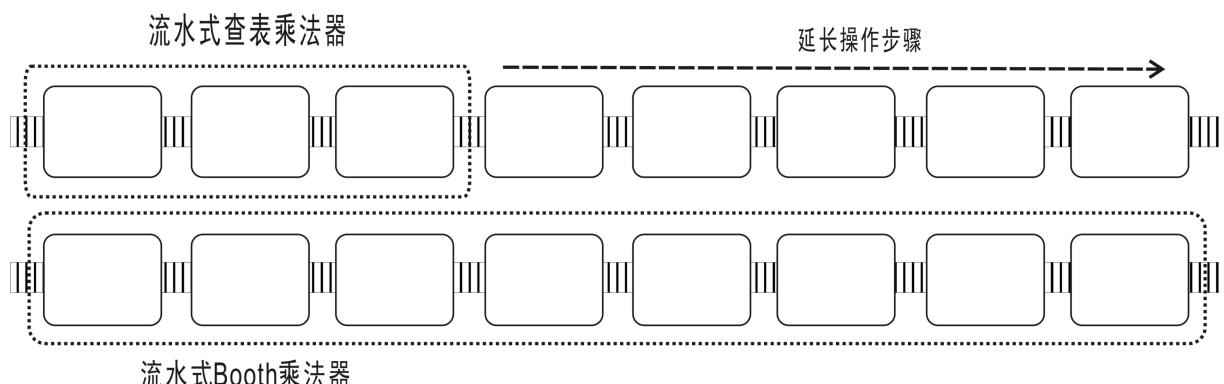
假设笔者要求，求出上面的公式，那么笔者就需要两个乘法器。为了减少时钟的消耗，笔者决定执行并行操作，亦即两个除法器一同工作，所以两个乘法器需要并连起来。



笔者稍微顽皮一点，一个乘法器使用流水式查表乘法器，另一个乘法器采用流水式 Booth 乘法器。哎呀！问题来了，前者有 3 个步骤，后者有 8 个步骤，那么该怎么办呢？

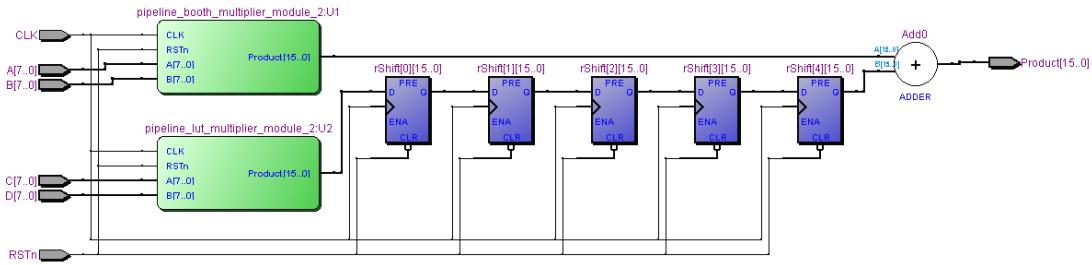
在很久很久以前，笔者一直对“移位寄存器”充满好奇。从 74 系列 IC 到 Verilog HDL 语言，笔者始终无法知晓它是干什么用的，果真是笔者想多了，它的最终作用就是移位用嘛 ...

移位寄存器在流水操作里，它扮演着延长操作步骤的作用。我们知道流水式查表乘法器的操作步骤有 3 个，然而流水式 Booth 乘法器有 8 个，那么流水式查表乘法器就必须向流水式 Booth 乘法器看齐，它自身必须“延长 5 个空步骤”，使得两个乘法器都同步。



如果形象一点有如上图的感觉。流水式查表乘法器一方的前面延长 5 了个无用的操作步骤。（忽然间，笔者觉得移位寄存器很没有用 ... 活着的意义就是延长工作）

实验十六：移位寄存器延长工作，还威武了



这一实验的建模结果如上图（笔者开始偷懒话图形了，直接载个扩展图算了）。

exp16_top_module.v

```

1. module exp16_top_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [7:0]A,
7.     input [7:0]B,
8.     input [7:0]C,
9.     input [7:0]D,
10.
11.    output [15:0]Product
12. );
13.
14. ****
15.
16.    wire [15:0]U1_Product;
17.
18.    pipeline_booth_multiplier_module_2 U1( CLK, RSTn, A, B, U1_Product );
19.
20. ****
21.
22.    wire [15:0]U2_Product;
23.
24.    pipeline_lut_multiplier_module_2 U2( CLK, RSTn, C, D, U2_Product );
25.

```

```
26.
27.     reg [15:0]rShift [4:0];
28.
29.     always @ ( posedge CLK or negedge RSTn )
30.         if( !RSTn )
31.             begin
32.                 rShift[0] <= 16'd0;
33.                 rShift[1] <= 16'd0;
34.                 rShift[2] <= 16'd0;
35.                 rShift[3] <= 16'd0;
36.                 rShift[4] <= 16'd0;
37.             end
38.         else
39.             begin
40.
41.                 rShift[0] <= U2_Product;
42.                 rShift[1] <= rShift[0];
43.                 rShift[2] <= rShift[1];
44.                 rShift[3] <= rShift[2];
45.                 rShift[4] <= rShift[3];
46.             end
47.
48.     /*****
49.
50.     assign Product = U1_Product + rShift[4];
51.
52.     *****/
53.
54. endmodule
```

第 18 行是流水式 Booth 乘法器的实例化，24 行是流水式查表乘法器的实例化。27~46 行是移位寄存器的声明和建立。

(看吧，是不是很单纯。连线关系和图形一样。)

exp16_top_module.vt

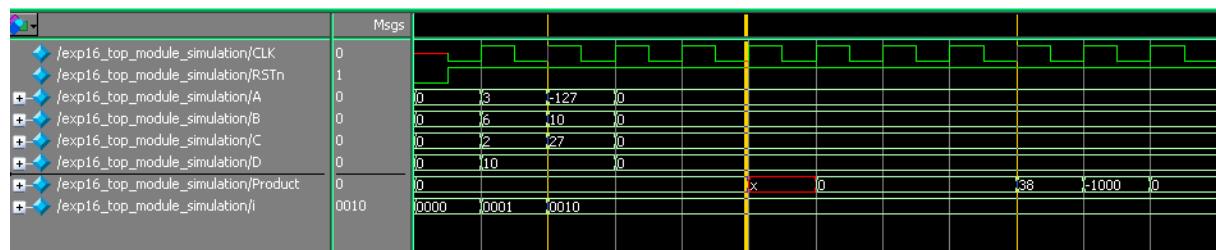
```
1. `timescale 1 ps/ 1 ps
2. module exp16_top_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [7:0]A;
```

```
8.      reg [7:0]B;
9.      reg [7:0]C;
10.     reg [7:0]D;
11.
12.     wire [15:0]Product;
13.
14.     /*****/
15.
16.     exp16_top_module U1
17.     (
18.         .CLK(CLK),
19.         .RSTn(RSTn),
20.         .A(A),
21.         .B(B),
22.         .C(C),
23.         .D(D),
24.         .Product(Product)
25.     );
26.
27.     /*****
28.
29.     initial
30.     begin
31.         RSTn = 0; #10; RSTn = 1;
32.         CLK = 0; forever #10 CLK = ~CLK;
33.     end
34.
35.     /*****
36.
37.     reg [3:0]i;
38.
39.     always @ ( posedge CLK or negedge RSTn )
40.         if( !RSTn )
41.             begin
42.                 i <= 4'd0;
43.                 A <= 8'd0;
44.                 B <= 8'd0;
45.                 C <= 8'd0;
46.                 D <= 8'd0;
47.             end
48.         else
49.             case( i )
50.
51.                 0: // A = 3, B = 6, C = 2, D = 10 , answer = 38
52.                 begin A <= 8'd3; B <= 8'd6; C <= 8'd2; D <= 8'd10; i <= i + 1'b1; end
```

```
53.  
54.           1: // A = -127, B = 10, C = 27, D = 10 , answer = -1000  
55.           begin A <= 8'b10000001; B <= 8'd10; C <= 8'd27; D <= 8'd10; i <= i + 1'b1; end  
56.  
57.           2:  
58.           begin A <= 8'd0; B <= 8'd0; C <= 8'd0; D <= 8'd0; i <= 4'd2; end  
59.  
60.  
61.       endcase  
62.  
63.   /*****  
64.  
65. endmodule
```

.vt 文件还是一如既往的清一色，呵呵！见笑了。

仿真结果：



流水式查表乘法器经过移位寄存器的延长，它和流水式 Booth 乘法器同步了。上图的仿真结果显示，潜伏时间有 8 个时钟，其中 5 个时钟是受到移位寄存器的延迟。经过延迟以后，两个乘法器可以同步操作了。

实验十六结论：

移位寄存器的用途，到目前为止笔者只是知道它能延长操作步骤而已。

总结：

每当写到这里，笔者不知道为什么特别安心。

对于许多初学者来说，流水操作一直是很神秘，而且深不可测。其实这是不然的，流水操作之所以会给初学者一种模糊的感觉，是因为初学者不清楚要建立流水操作之前，必须做好的准备而已。基本上，流水操作和仿顺序操作它们之间有微妙的关系。笔者一直都很建议，如果要建立某个流水操作，我们必须先了解它在“顺序操作”中的操作步骤。然后使用 Verilog HDL 语言建立起仿顺序操作，最后再向流水操作转换，就如实验十一~十四那样。

在这一章节中，笔者所强调的除了“仿顺序操作 => 流水操作的转换”以外，笔者还强调流水操作的建模化，建模化以后的流水操作，在许多方面都有提高。此外还有一点，非常另笔者在意的地方是“流水操作的用途”。我们知道流水操作的执行方向，仿佛像一支直射的箭矢，永远只会向前而已。**换言之，流水操作的处理方向只有向前，类似的数据处理有“算法”。**

当然，这一章节的重点，不可能只是在讨论“流水操作”而已。要掌握好“流水操作”，我们需要两个前提条件：一是步骤，二是时钟。要实现“仿顺序操作 => 流水操作的转换”，我们必须明白“什么是步骤”。要预测流水操作的潜伏时间，我们必须明白“什么是时钟”。

“步骤”好比是模块的“执行拍子”，然而“时钟”是“拍子”的停留时间。在第三章中，所有流水操作的实验，都是一个步骤一个时钟，换句话说每一个步骤都消耗一个时钟。时钟在模块的定义上，它是“模块最小的消耗单位”，明白这个道理有助于我们对模块的细化，这也是实验十六所隐藏的信息。

在实验十六中它包含了两个乘法模块，一个乘法模块是消耗 3 个时钟，另一个模块是消耗 8 个时钟。如果要这两个模块并联操作的话，“小的必须向大的看齐”，所以其中一个乘法模块必须延迟 5 个时钟。这样作的目的就有一个，就是“数据的对齐性”。假设小的一方是延迟 4 个时钟或者延迟 5 个时钟，而不是延迟 5 个时钟。那么结果会发生“数据对齐的错误”：延迟 4 个时钟会使数据快了一个时钟；延迟 6 个时钟会使慢了一个时钟。

这种错误对于细化模块来说是最普遍的问题。在细化的过程中，其中一项工作就是要排除这种错误的可能性。

第四章：模块的沟通

4.1 探索 Start_Sig 和 Done_Sig 的协调性

Start_Sig 和 Done_Sig 是仿顺序操作中模块的象征性的信号，如果能掌握它们，那么利用 Verilog HDL 模仿顺序操作再也不是梦。Start_Sig 和 Done_Sig 顾名思义就是模仿顺序操作语言中的“函数调用”和“函数返回”指令。如果从某个角度来看，前者是启动信号，后者是反馈信号。作为初学者，可以允许在不了解它们的情况下使用它们。相反的，如果想进一步的深入，就不得不对 Start_Sig 和 Done_Sig 时序之间的协调性去探讨。

Start_Sig 和 Done_Sig 当两个模块互相作用的时候，这两个信号的时序会如此协调与和谐，笔者也觉得很不可思议。笔者还记得在写这一本《Verilog HDL 建模技巧 · 仿顺序操作 思路篇》笔记的时候，也是笔者学习 Verilog HDL 语言不久，当时的笔者对时钟和步骤的掌握还不成熟，笔者既然会创作出这样的东西，只能说是奇迹或者是神明帮助 ...

接下来，我们需要实验一的帮助来理解 Start_Sig 和 Done_Sig 在时序上的协调性。在这里笔者重新粘贴 multiplier_module.v 和 multiplier_module.vt 相关的步骤，帮助刷新刷新读者们已经发霉的脑袋，同时也使得读者省去翻页的麻烦。

multiplier_module.v

```

33.    else if( Start_Sig )
34.        case( i )
35.
36.            0:
37.                begin
38.
39.                    isNeg <= Multiplicand[7] ^ Multiplier[7];
40.                    Mcand <= Multiplicand[7] ? ( ~Multiplicand + 1'b1 ) : Multiplicand;
41.                    Mer <= Multiplier[7] ? ( ~Multiplier + 1'b1 ) : Multiplier;
42.                    Temp <= 16'd0;
43.                    i <= i + 1'b1;
44.
45.                end
46.
47.            1: // Multipling
48.                if( Mer == 0 ) i <= i + 1'b1;
49.                else begin Temp <= Temp + Mcand; Mer <= Mer - 1'b1; end
50.
```

```

51.          2:
52.          begin isDone <= 1'b1; i <= i + 1'b1; end
53.
54.          3:
55.          begin isDone <= 1'b0; i <= 2'd0; end
56.
57.      endcase
58.

```

multiplier_module.vt

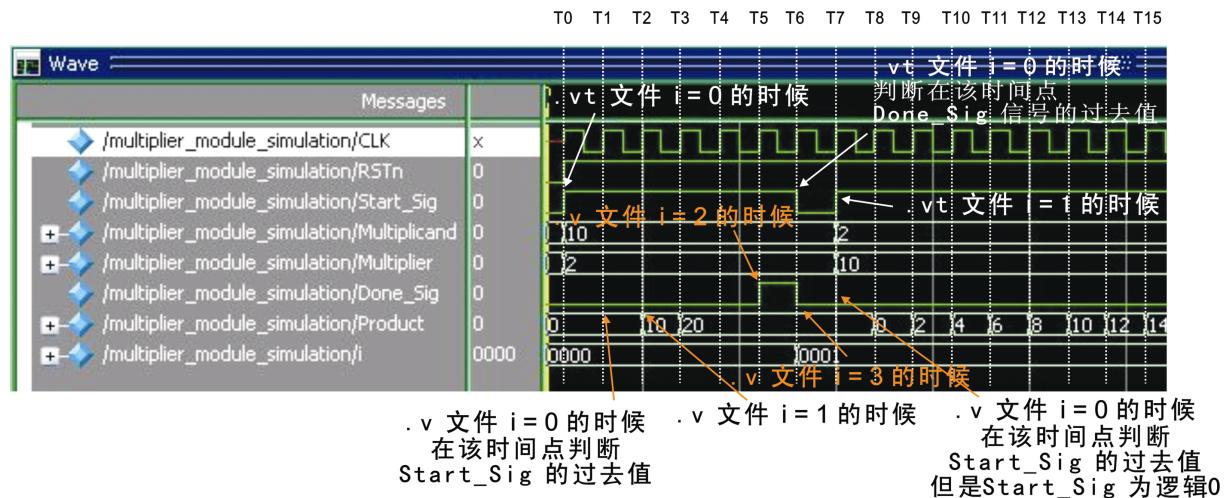
```

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'd2; Start_Sig <= 1'b1; end
55.
56.          1: // Multiplicand = 2 , Multiplier = 10
57.          if( Done_Sig ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
58.          else begin Multiplicand <= 8'd2; Multiplier <= 8'd10; Start_Sig <= 1'b1; end

```

在这里我们仅需要 *multiplier_module.v* 和 *multiplier_module.vt* 相关的几个步骤而已。

仿真结果：



Done_Sig 和 Start_Sig 时序的协调性不是什么复杂的东西，但是对于新手而言是一件苦差事。上面的仿真图是实验一的仿真结果，这一张图已经说明了一切。*.vt* 文件代表实验一的 *multiplier_divider_module.vt*，亦即激励文件。*.v* 文件代表实验一的 *multiplier_divider_module.v*，亦即乘法模块。

在 T0 的时候 .vt 文件是步骤 0 (亦即 $i=0$)，它“决定”拉高 Start_Sig，并且“决定”发送乘数和被乘数 $10 * 2$ 。但是这时候 .v 文件判断到 Start_Sig 过去值是逻辑 0，所以 .v 文件没有“决定什么”。在 T0 的未来 Start_Sig 拉高， $10 * 2$ 发送在 Multiplicand 和 Multiplier 信号上。

当 T1 的时候 .vt 文件还是停留在步骤 0，它在等待 .v 文件的反馈。在同一个时候 .v 文件判断到 Start_Sig 的过去值是逻辑 1，结果 .v 被启动并且进入步骤 0 (亦即乘法模块的初始化步骤)。所以在 T1 的未来 .v 文件的内部开始初始化，当然初始化的取值 (参考值) 是来自 T1 的 Multiplicand 和 Multiplier 过去值，亦即 $10 * 2$ 。

当时钟是 T2 的时候，.vt 文件依然等待 .v 文件的反馈。反之是 .v 文件进入步骤 1 并且已经完成第一次的乘法运算操作。所以在 T2 时间点 Product 的未来值是 10。

在 T3 的时候 .vt 文件还是老样子。.v 文件已近完成第二次的乘法运算操作。所以在 T3 时间点，未来的 Product 值是 20。

在 T4 的时候 .vt 文件还是老样子。.v 文件这时候已经完成乘法运算操作，.v 内部的 if 条件成立 (使用了一个时钟) 然后“决定”进入下一个步骤。在 T4 的未来 .v 没有任何改变，只不过 .v 内部的步骤 i 递增了。

在 T5 的时候 .vt 文件还是老样子。.v 文件这时候已近进入步骤 2，对于 .v 文件来说，在步骤 2 它“决定”拉高完成信号。所以在 T5 时间点 Done_Sig 的未来值会是逻辑 1。

在 T6 的时候 .vt 文件不再是老样子了，它检查到 Done_Sig 过去值是逻辑 1，所以它“决定”拉低 Start_Sig 并且进入下一个步骤。在同一个时间 .v 文件处于步骤 3，它“决定”是拉低 Done_Sig 并且步骤回到 0。所以在 T6 时间点 Done_Sig 和 Start_Sig 的未来值都是逻辑 0。

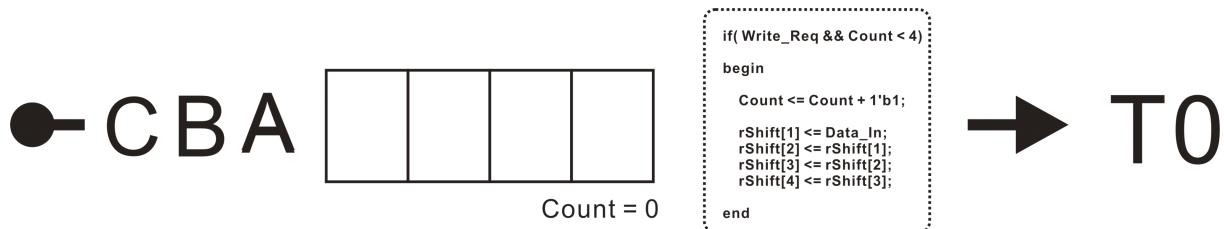
当时间是 T7 的时候，.vt 文件会进入步骤 1，它会重新“决定”拉高 Start_Sig，和发送乘数和被乘数 ... 上述的动作会再一次的重复

Done_Sig 和 Start_Sig 是控制信号的一员，站在仿顺序操作上的它们，可以视为是两个模块之间沟通的桥梁。笔者也是因为这个“桥梁”才会继续深入探索模块之间的控制或者协调作用。事实上 Done_Sig 和 Start_Sig 只要掌握好使用办法，不明白时钟和步骤的相互作用也不要紧。但是想要深入了解 Verilog HDL 语言的话，一定要好好它们之间的协调作用。

在这里稍微反思一下流水操作和仿顺序操作的差别。在前章笔者说过，流水操作是“永远向前走”操作方式，因为它只要将完成的工作丢个下一方，用不着像仿顺序操作那样，为了与其他模块沟通，需要配备控制信号。所以流水操作在某种程度上，比仿顺序操作单调多了并且更难控制。

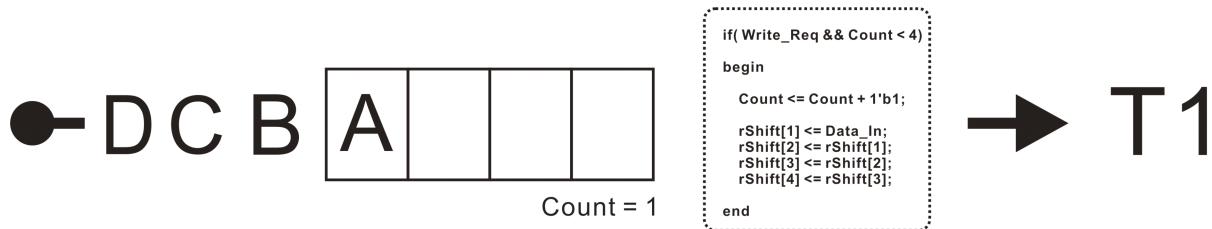
4.2 同步 FIFO

为了写这章笔记，笔者真的花了不少前期的准备。同步 FIFO 实际上就是一个被加工后的移位寄存器，用文字来表达可能会抽象一点，在这里笔者稍微借用画图的力量：

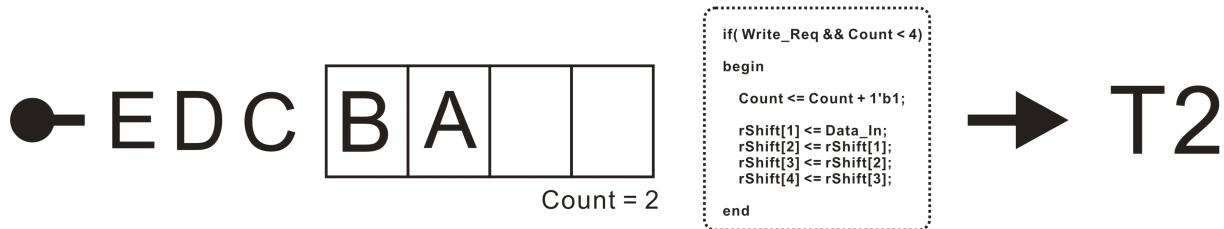


在上图中，四个小格子就是深度为 4 的移位寄存器，格子左边的字母表示数据，格子右边的代码代表“写操作”的“决定”。下边是移位寄存器的数据存入数目，最右边是当前的时间（当前的时间点）。该操作是使用“时间点”的概念来解读。我们先从写操作开始：

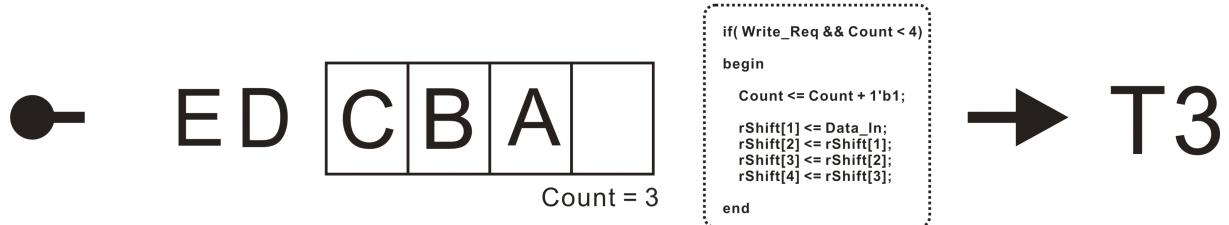
在 T0 的时候，移位寄存器的当前数据数目是 0，然而在该时间点所做的“决定”是“写入数据”。假设 Write_Req 从初始的时候就一直被拉高，那么在 T0 的未来，Count 的值会递增为 1，然后移位寄存器的第一个格子就会读入 A。



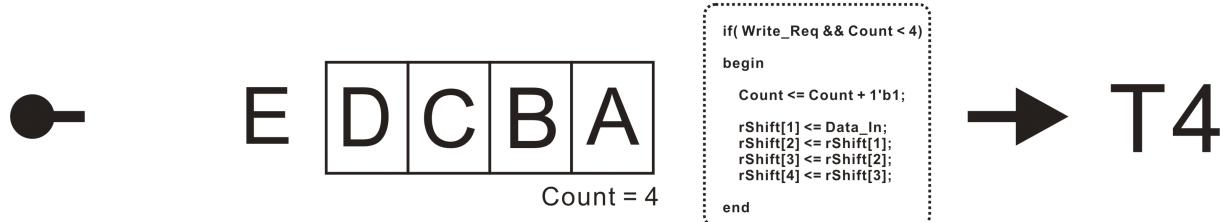
在 T1 的时候，由于 T0 的决定，移位寄存器的当前数据数目是 1，然而第一个格子的数据是 A。然后在 T1 依然重复同样的“决定”。假设 Write_Req 从初始的时候就一直被拉高，那么在 T1 的未来，Count 的值会递增为 2，然后 A 会向第二个格子移动，则第一个格子会读入数据 B。



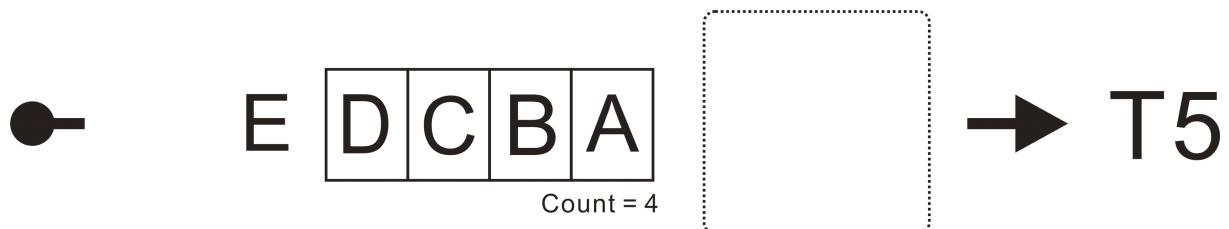
在 T2 的时候，由于 T1 的决定，移位寄存器的当前数据数目是 2，然而第一个格子的数据是 B，第二个格子则是 A。然后在 T2 重复同样的“决定”。假设 Write_Req 从初始的时候就一直被拉高，那么在 T2 的未来，Count 的值会递增为 3，然后 A 会向第三个格子移动，B 会向第三个格子移动，则第一个格子会读入数据 C。



在 T3 的时候，由于 T2 的决定，移位寄存器的当前数据数目是 3，然而第一个格子是 C，第二个格子是 B，第三个格子则是 A。然后在 T3 重复同样的“决定”。假设 Write_Req 从初始的时候就一直被拉高，那么在 T3 的未来，Count 的值会递增为 4，然后 A 会向第四个格子移动，B 会向第三个格子移动，C 会向第二个格子移动，则第一个格子会读入数据 D。

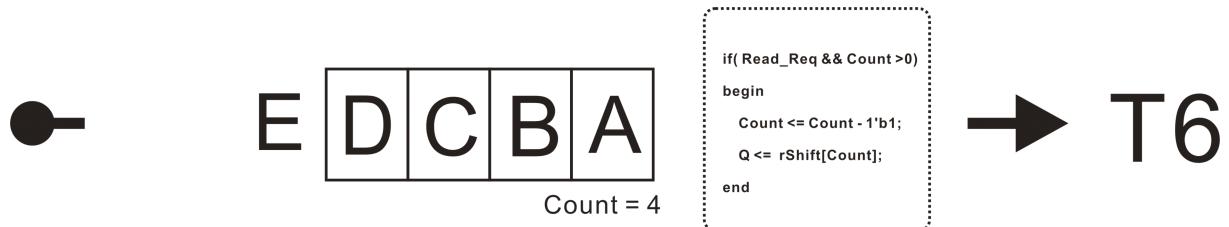


在 T4 的时候，由于 T3 的决定，移位寄存器的当前数据数目是 4，然而第一个格子是 D，第二个格子是 C，第三个格子是 B，第四个格子则是 A。然后在 T4 重复同样的“决定”。假设 Write_Req 从初始的时候就一直被拉高，由于写操作的“决定”有条件约束，就是 Count < 4，所以 T4 的未来，没有任何改变。

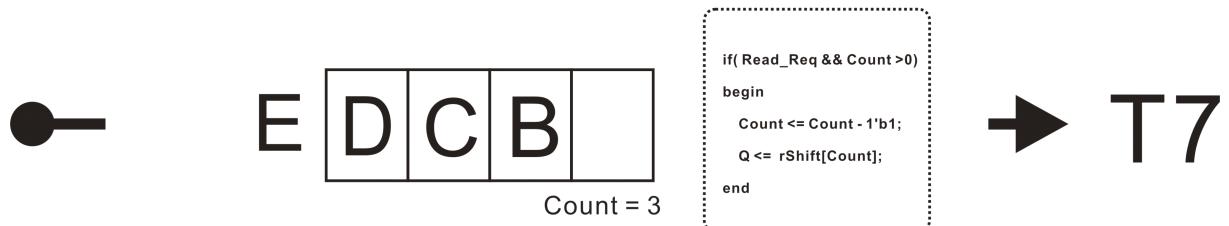


在 T5 的时候，由于受到写操作的条件约束，所以移位寄存器的结果和 T4 的时候一模一样。数据存入数也是 4 个。

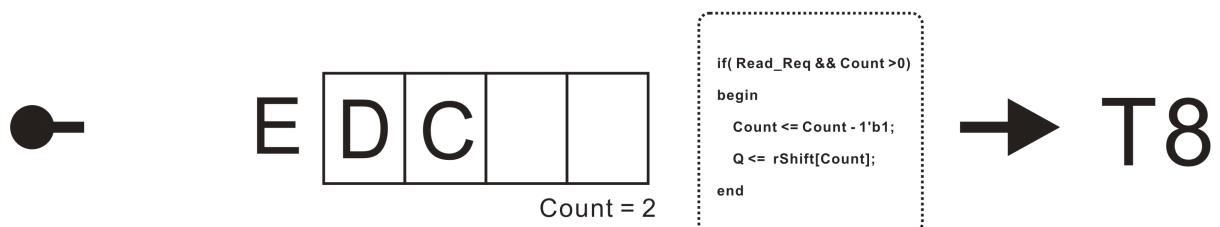
啊~写操作已经被笔者玩厌了，借下来执行读操作看看。



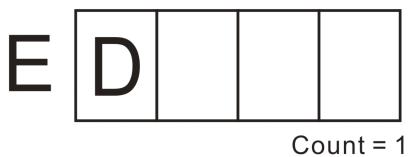
在 T6 的时候，由于 T5 没有任何“决定”，所以 T6 保持 T5 未来的结果。再重复说明一下移位寄存器的状况：目前的输入存入数依然保持 4 个，然而 T6 做了读操作的“决定”。假设从 T5 开始 Read_Req 一直被拉高，T6 的未来会是，数据 A 会被读出，然后 Count 的值会递减为 3。



在 T7 的时候，由于 T6 的“决定”，移位寄存的第四个格子已经被读出，第三个格子寄存 B，第二个格子寄存 C，第一个格子寄存 D。目前的输入存入数是 3 个。然而 T7 重复一样的读操作“决定”。假设从 T5 开始 Read_Req 一直被拉高，T7 的未来会是，数据 B 会被读出，然后 Count 的值会递减为 2。



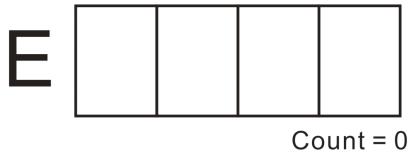
在 T8 的时候，由于 T7 的“决定”，移位寄存的第三格子已经被读出，第二个格子寄存 C，第一个格子寄存 D。目前的输入存入数是 2 个。然而 T8 重复一样的读操作“决定”。假设从 T5 开始 Read_Req 一直被拉高，T8 的未来会是，数据 C 会被读出，然后 Count 的值会递减为 1。



```
if( Read_Req && Count >0)
begin
    Count <= Count - 1'b1;
    Q <= rShift[Count];
end
```

→ T9

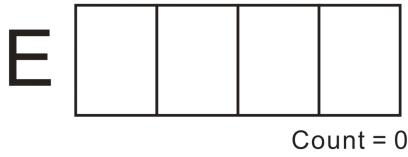
在 T9 的时候，由于 T8 的“决定”，移位寄存的第二个格子已经被读出，第一个格子寄存 D。目前的输入存入数是 1 个。然而 T9 重复一样的读操作“决定”。假设从 T5 开始 Read_Req 一直被拉高，T9 的未来会是，数据 D 会被读出，然后 Count 的值会递减为 0。



```
if( Read_Req && Count >0)
begin
    Count <= Count - 1'b1;
    Q <= rShift[Count];
end
```

→ TA

在 TA 的时候，由于 T9 的“决定”，移位寄存的第一个格子已经被读出，。目前的输入存入数是 0 个。如果 T9 重复一样的读操作“决定”，会由于读操作“决定”的条件约束，Count > 0，那么 TA 的未来保持不变。

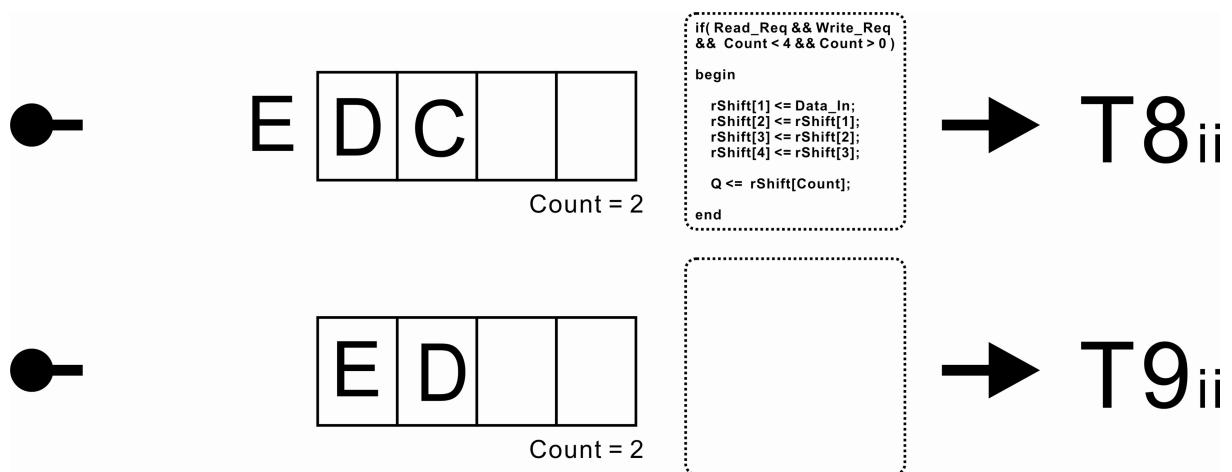


```
if( Read_Req && Count >0)
begin
    Count <= Count - 1'b1;
    Q <= rShift[Count];
end
```

→ TB

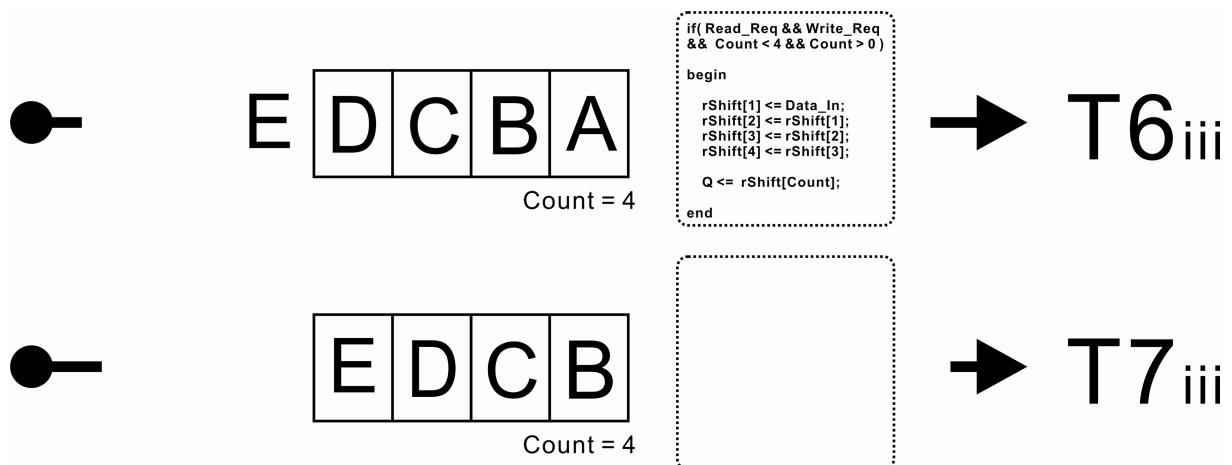
在 TB 的时候，由于 TA 的“决定”受到条件约束的影响，所以 TB 的未来也会保持不变。

如果写操作和读操作同时发生的话 那么我们需要乘坐时光机返回到 T8



时间回到 T8。这时候所做的“决定”是同时读写操作（同时读写操作的“决定”就像是读操作和写操作的总和，其中 Count 相互抵消）。那么在 T8 的未来（T9），C 会被读出，D 会移入第二个格子，E 会被读入第一个格子，Count 保持不变。

如果我们又坐时光机回到 T6，然后执行同样的决定（同时读写操作），看看会发生怎样的结果 ...



上图是在 T6 的时候，四个格子中的字母是 D, C, B, A，而 Count 是 4。T6 所“决定”的操作是“读写同时执行”，所以在 T6 的未来，亦即 T7 之际，A 从移位寄存器中被读出，B, C, D 相续移位，然而第一个格子读入 E，并且 Count 保持不变。

同步 FIFO 的操作有，读操作，写操作，和读写同时操作，这三个操作而已。同步 FIFO 操作的原理很简单，但是要从外部调用同步 FIFO，并且要实现这些操作，就必须伤一点脑筋。典型的 FIFO 控制信号有，Write_Req, Read_Req, Full_Sig 和 Empty_Sig，每一个信号的功能如命名般一样，但是这些控制信号，仅适合异步的 FIFO 而已。如果“死马当活马”把它用在同步 FIFO 的身上，调用就会发生问题。

实验十七：同步 FIFO

fifo_module.v

```
1. module fifo_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [7:0]FIFO_Write_Data,
8.
9.     input Read_Req,
10.    output [7:0]FIFO_Read_Data,
11.
12.    output Full_Sig,
13.    output Empty_Sig,
14.
15.    /*****
16.
17.    output [7:0]SQ_rS1,
18.    output [7:0]SQ_rS2,
19.    output [7:0]SQ_rS3,
20.    output [7:0]SQ_rS4,
21.    output [2:0]SQ_Count
22.
23.    *****/
24. );
25.
26. *****/
27.
28. parameter DEEP = 3'd4;
29.
30. *****/
31.
32. reg [7:0]rShift [DEEP:0];
33. reg [2:0]Count;
34. reg [7:0]Data;
35.
36. always @ ( posedge CLK or negedge RSTn )
37.     if( !RSTn )
```

```

38.         begin
39.
40.             rShift[0] <= 8'd0; rShift[1] <= 8'd0; rShift[2] <= 8'd0;
41.             rShift[3] <= 8'd0; rShift[4] <= 8'd0;
42.             Count <= 3'd0;
43.             Data <= 8'd0;
44.
45.         end
46.     else if( Read_Req && Write_Req && Count < DEEP && Count > 0 )
47.         begin
48.             rShift[1] <= FIFO_Write_Data;
49.             rShift[2] <= rShift[1];
50.             rShift[3] <= rShift[2];
51.             rShift[4] <= rShift[3];
52.             Data <= rShift[ Count ];
53.         end
54.     else if( Write_Req && Count < DEEP )
55.         begin
56.
57.             rShift[1] <= FIFO_Write_Data;
58.             rShift[2] <= rShift[1];
59.             rShift[3] <= rShift[2];
60.             rShift[4] <= rShift[3];
61.
62.             Count <= Count + 1'b1;
63.         end
64.     else if( Read_Req && Count > 0 )
65.         begin
66.             Data <= rShift[Count];
67.             Count <= Count - 1'b1;
68.         end
69.
70.
71.     *****/
72.
73.     assign FIFO_Read_Data = Data;
74.     assign Full_Sig = ( Count == DEEP ) ? 1'b1 : 1'b0;
75.     assign Empty_Sig = ( Count == 0 ) ? 1'b1 : 1'b0;
76.
77.     *****/
78.
79.     assign SQ_rS1 = rShift[1];
80.     assign SQ_rS2 = rShift[2];
81.     assign SQ_rS3 = rShift[3];

```

```

82.     assign SQ_rS4 = rShift[4];
83.     assign SQ_Count = Count;
84.
85.     /*****
86.
87. endmodule

```

第 3~13 行，是 FIFO 的输入输出，其中（12~13 行）是 Full_Sig 和 Empty_Sig，它们的驱动条件在 74~75 行。Full_Sig 拉高的条件就是“Count 等价于深度”，然而 Empty_Sig 的拉高条件就是“Count 等价于零”的时候。第 17~21 行是仿真输出。

在 28 行声明了常量 DEEP 为 4，也就说笔者打算建立深度为 4 的同步 FIFO。

在 32 行，声明了位宽为 8 个字（words）为 5，也就是位宽为 8，深度为 5 的存储器。笔者建立同步 FIFO 都有一个坏习惯，**第 0 个深度的储存器是视为不见，目的只有一个就是为了方便同步 FIFO 的设计。**具体的原因往下看，就会明白了。

第 33~34 行声明的 Count 寄存器和 Data 寄存器，前者用来计数数据存入数，后者用来驱动 FIFO_Read_Data 输出（73 行）。

第 37~45 初始化的动作，这一步对于 FIFO 来说是最重要的，就是将所有相关的寄存器都初始化为 0。（如果建立的同步 FIFO 深度不多的话，可以考虑类似初始化方式。反之，如果同步 FIFO 的深度的数目很大的话，必须考虑利用 .mif 文件来初始化了。具体的方法请参考 VerilogHDL 那些事儿 - 建模篇的第五章。）

第 46~53 行是同步 FIFO 同时读写操作。第 54~63 行是对同步 FIFO 的写操作。第 64~68 行是对同步 FIFO 的读操作。有一个重点不得不注意是，“读操作”，“写操作”，“同时读写操作”它们有优先级之分。**“同时读写操作”的优先级永远都是最高的。**

在这里，第 74~75 行的 Full_Sig 和 Empty_Sig 驱动条件看是很有道理，很有逻辑。其实这是初学者对 FIFO 的错觉。如果用于异步 FIFO 的话，74~75 行的驱动条件绝对没有错误，但是把它们用于同步 FIFO 的话，这就大错特错，具体的原因看了仿真结果就会知道。

fifo_module.vt

```

1. `timescale 1 ps/ 1 ps
2. module fifo_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Write_Req;
8.     reg [7:0]FIFO_Write_Data;

```

```
9.  
10.    reg Read_Req;  
11.  
12.    wire [7:0]FIFO_Read_Data;  
13.  
14.    wire Empty_Sig;  
15.    wire Full_Sig;  
16.  
17.    /*****  
18.  
19.    wire [7:0]SQ_rS1;  
20.    wire [7:0]SQ_rS2;  
21.    wire [7:0]SQ_rS3;  
22.    wire [7:0]SQ_rS4;  
23.    wire [2:0]SQ_Count;  
24.  
25.    /*****  
26.  
27. fifo_module i1  
28. (  
29.     .CLK(CLK),  
30.     .RSTn( RSTn ),  
31.     .Write_Req(Write_Req),  
32.     .FIFO_Write_Data(FIFO_Write_Data),  
33.     .Read_Req(Read_Req),  
34.     .FIFO_Read_Data(FIFO_Read_Data),  
35.     .Empty_Sig(Empty_Sig),  
36.     .Full_Sig(Full_Sig),  
37.     .SQ_rS1( SQ_rS1 ),  
38.     .SQ_rS2( SQ_rS2 ),  
39.     .SQ_rS3( SQ_rS3 ),  
40.     .SQ_rS4( SQ_rS4 ),  
41.     .SQ_Count( SQ_Count )  
42. );  
43.  
44.    /*****  
45.  
46. initial  
47. begin  
48.     RSTn = 0; #10; RSTn = 1;  
49.     CLK = 0; forever #10 CLK = ~CLK;  
50. end  
51.  
52.    /*****  
53.
```

```
54.      reg [3:0]i;
55.
56.      always @ ( posedge CLK or negedge RSTn )
57.          if( !RSTn )
58.              begin
59.
60.                  i <= 4'd0;
61.                  Write_Req <= 1'b0;
62.                  Read_Req <= 1'b0;
63.                  FIFO_Write_Data <= 8'd0;
64.
65.              end
66.          else
67.              case( i )
68.
69.                  /*****
70.
71.                  0:
72.                  begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd5; i <= i + 1'b1; end
73.
74.                  1:
75.                  begin Write_Req <= 1'b0; i <= i + 1'b1; end
76.
77.                  *****/
78.
79.                  2:
80.                  begin
81.                      Write_Req <= 1'b1; FIFO_Write_Data <= 8'd6;
82.                      Read_Req <= 1'b1;
83.                      i <= i + 1'b1;
84.                  end
85.
86.                  3:
87.                  begin Write_Req <= 1'b0; Read_Req <= 1'b0; i <= i + 1'b1; end
88.
89.                  *****/
90.
91.                  4:
92.                  begin Read_Req <= 1'b1; i <= i + 1'b1; end
93.
94.                  5:
95.                  begin Read_Req <= 1'b0; i <= i + 1'b1; end
96.
97.                  *****/
98.
```

```
99.          6:
100.         begin
101.           Write_Req <= 1'b1; Read_Req <= 1'b0;
102.           FIFO_Write_Data <= 8'd100; i <= i + 1'b1;
103.         end
104.
105.          7:
106.         begin
107.           Write_Req <= 1'b1; Read_Req <= 1'b1;
108.           FIFO_Write_Data <= 8'd33; i <= i + 1'b1;
109.         end
110.
111.          8:
112.         begin
113.           Write_Req <= 1'b0; Read_Req <= 1'b1;
114.           i <= i + 1'b1;
115.         end
116.
117.         *****/
118.
119.          9:
120.         begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd99; i <= i + 1'b1; end
121.
122.          10:
123.         begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end
124.
125.         *****/
126.
127.          11:
128.         begin Read_Req <= 1'b0; i <= i + 1'b1; end
129.
130.         *****/
131.
132.          12:
133.         if( Full_Sig ) begin Write_Req <= 1'b0; i <= i + 1'b1;end
134.         else begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= FIFO_Write_Data + 1'b1; end
135.
136.          13:
137.         if( Empty_Sig )begin Read_Req <= 1'b0; i <= i + 1'b1; end
138.         else begin Write_Req <= 1'b0; Read_Req <= 1'b1; end
139.
140.         *****/
141.
142.          14:
143.         begin i <= 4'd14; end
```

```
144.  
145.      ****  
146.  
147.  
148.      endcase  
149.  
150.  
151. endmodule
```

.vt 文件的风格还是一如既往一样清一色。

69~97 行的动作是先写一个数据（71~75 行），然后同时间写一个数据再读一个数据（79~87 行），最后读一个数据（91~95 行）。为简单起见，笔者把每一个动作都消耗两个时钟。

71~75 行的动作是在步骤 0 拉高 Write_Req 然后发送数据 8'd5，然后在步骤 1 拉低 Write_Req。79~87 行的动作，是在步骤 2 的时候同时拉高 Write_Req 和 Read_Req 并且发送数据 8'd6，然后再步骤 3 同时拉低 Write_Req 和 Read_Req。91~95 行的动作是在步骤 4 的时候拉高 Read_Req，然后在步骤 5 拉低 Read_Req。

99~115 行所执行的动作大致上和 69~97 一样，但是时钟消耗是一个时钟而已。在步骤 6 拉高 Write_Req 并且发送数据 8'd100。步骤 7 的动作是同时拉高 Write_Req 和 Read_Req 并且发送数据 8'd33。步骤 8 的动作是拉高 Read_Req。

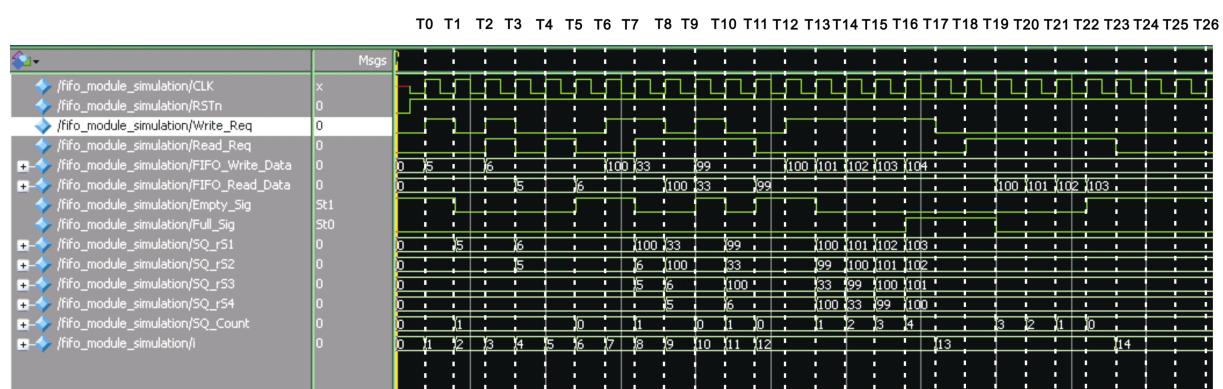
在 111~115 行（步骤 8）的结果，我们忘记了把 Read_Req 拉低，当进入步骤 9（119~120 行）的时候它依然保持拉高。步骤 9 是拉高 Write_Req，并且发送数据 8'd99。

122~123 行（步骤 10）是读操作而已（Write_Req 被拉低了），然后在 127~128 行（步骤 11）只是单纯的拉低 Read_Req。（注意步骤 8~10）

第 132~138 行是用来测试使用 Full_Sig 和 Empty_Sig 控制信号对同步 FIFO 造成的结果。在步骤 12 会陆续向 FIFO 写入数据，直到 Full_Sig 信号拉高为止。反之步骤 13 会陆续从 FIFO 读取数据，直到 Empty_Sig 拉高为止（意外也开始发生了）。

这些动作到底会产生怎样的结果，我们从仿真结果中拭目以待。

仿真结果：



在 T0 的时候 .vt 进入步骤 0, Write_Req “决定” 被拉高和数据 8'd5 “决定” 被发送。所以在 T0 的未来, FIFO_Write_Data 的未来值是 8'd5, Write_Req 会被拉高。

(注意: 由于初始状态, FIFO 为空 Empty_Sig 是拉高状态。)

在 T1 的时候 .vt 进入步骤 1, Write_Req “决定” 被拉低。在同一个时间 .v 检测到 Write_Req 的过去值是逻辑 1, 所以 .v “决定” 写操作。在 T1 的未来, Write_Req 会拉低, 而且 FIFO_Write_Data 的过去值 8'd5 会被写入 FIFO 的第一格子。

(注意: T1 未来的 SQ_rS1。)

在 T2 的时候 .vt 是步骤 2, Read_Req 和 Write_Req“决定”拉高, 然后 FIFO_Write_Data “决定”发送数据 8'd6。在同一个时候 .v 检查到 Write_Req 和 Read_Req 的过去值都是逻辑 0, 所以 .v 乖乖的待命。在 T2 的未来 Read_Req 和 Write_Req 的会被拉高, 数据 8'd6 会发送在 FIFO_Write_Data。

在 T3 的时候 .vt 进入步骤 3, 它“决定”拉低 Read_Req 和 Write_Req。在同一个时候 .v 文件检测到 Read_Req 和 Write_Req 的过去值都是逻辑 1, 所以 .v 决定“同时读写操作”。T3 的未来 Read_Req 和 Write_Req 会被拉低; 数据 8'd5 会从 FIFO 的第一个格子被读出, 然后 FIFO_Write_Data 的过去值 8'd6 会被写入 FIFO 的第一个格子。

(注意: 同时读写操作使得 Count 不变, 此外第一个格子的数据 8'd5 被读出的同时, 数据 8'd6 别写入。)

在 T4 的时候 .vt 进入步骤 4, 它“决定”拉高 Read_Req。在同一个时候 .v 检查到 Read_Req 和 Write_Req 的过去值都是逻辑 0, 所以 .v “决定”沉默。在 T4 的未来, Read_Req 被拉高而已。

在 T5 的时候 .vt 进入步骤 5, 它“决定”拉低 Read_Req。在同一个时候 .v 检查到 Read_Req 的过去值是逻辑 1, 所以它“决定”读操作。在 T5 的未来, Read_Req 被拉低, FIFO 第一个格子的数据 8'd6 会被读出。

(注意: 由于 FIFO 已经空了, 所以 Empty_Sig 会在这个时间的未来被拉高。则 Count 也成为 0。)

在 T6~T8 重复 T0~T5 同样的动作，只是时钟消耗从 2 个变成 1 个。

在 T8 的时候 .vt 进入步骤 8，它“决定”拉高 Read_Req 和拉低 Write_Req。在同一个时候 .v 检测到，Write_Req 和 Read_Req 的过去值是逻辑 1，所以 .v “决定”同时读写操作。所以在 T8 的未来 Write_Req 被拉低，Read_Req 依然保持拉高的状态，数据 8'd33 被读入 FIFO，然后数据 8'd100 从 FIFO 读出。

在 T9 的时候 .vt 进入步骤 9，它“决定”拉高 Write_Req。在同一个时间 .v 检测到，Read_Req 的过去值是逻辑 1，所以 .v “决定”读操作。所以在 T9 的未来，Read_Req 保持不变，Write_Req 被拉高，则 FIFO 会把第一个格子中的数据 8'd33 吐出来。

(注意：这时候的 FIFO 已经为空，所以在 T9 的未来 Empty_Sig 会被拉高)。

T10 的过去态：读写均为使能 故 t10 应该写入数据同时读出数据；但是由于 empty 过去为空 使得同时读写不成立，造成此步的操作仅为写

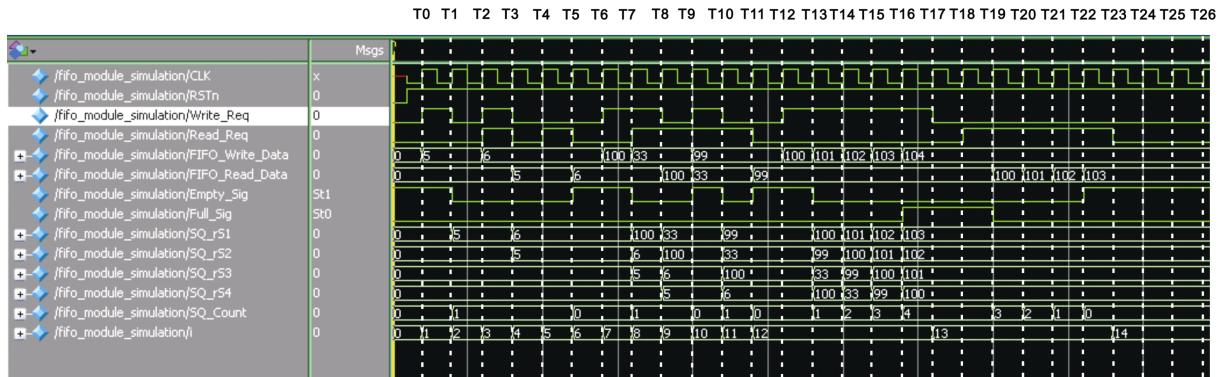
在 T10 的时候意外发生了。在这个时候 .vt 进入步骤 10，它“决定”拉高 Read_Req，并且“决定”拉低 Write_Req。在同一个时间 .v 检查到 Write_Req 和 Read_Req 的过去同是逻辑 1，但是 FIFO 已经为空了（注意 Empty_Sig 的过去值），所以同时读写操作的“决定”不成立，反而写操作的“决定”成立 (.v 文件内部的优先级关系)。

在 T10 的未来，Read_Req 被拉高，Write_Req 被拉低，数据 8'd99 被写入 FIFO 的第一个格子。（注意：T10 的未来里 Empty_Sig 被拉低了。）

在 T11 的时候 .vt 进入步骤 11，它“决定”拉低 Read_Req。在同一个时间 .v 检测到 Read_Req 的过去值是逻辑 1，所以它“决定”读操作。在 T11 的未来，Read_Req 被拉低，数据 8'd99 被读出。

嗯！暂时冷静一下脑袋吧。在这里我们只是讨论，同步 FIFO 的读操作，写操作，和读写操作而已，我们还没有进入控制信号 Empty_Sig 和 Full_Sig 的应用（不要被笔者吓到，真正的好戏现在才要开始。）

事实上，在 T10 的意外是决定不会发生的，因为我们忽略了控制信号的使用，所以才会发生这样的意外。FIFO 在实际的调用中，如果好好的使用控制信号，这样的困境是绝对不会发生。



在这里，笔者重新粘贴仿真结果，为了避免翻页的麻烦。

在 T12 的时候 .vt 进入步骤 12，它“决定”不停的向 FIFO 写入数据，直到 Full_Sig 拉高。在 T12~T16 之前数据的写入还算成功，但是在 T16 后和在 T17 之间，意外发生了。

在 T16 的时候，由于 .vt 文件对 Full_Sig 的判断失误 (Full_Sig 为逻辑 0)，它继续“决定”拉高 Write_Req 和发送数据 8'd104。在同一个时间 .v 文件判断 Write_Req 的过去值是逻辑 1，所以 .v “决定”写入数据 8'd103。在 T16 的未来，Write_Req 被拉高，数据 8'd104 被发送在 FIFO_Write_Data，而且 FIFO_Write_Data 的过去值 8'd103 被写入 FIFO。(注意 T16，Full_Sig 的未来值，已经是逻辑 1)

在 T17 的时候 .vt 文件检查到 Full_Sig 的过去值是逻辑 0，所以它“决定”拉低 Write_Req。在同一个时间 .v 检测到 Write_Req 的过去值是逻辑 1，但是 .v 的内部已经饱和了，所以它“决定”无视这一次的读取操作。在 T17 的未来，数据 8'd104 会被作废，Full_Sig 持续拉高 (意外又发生)。

在 T18~T22 之间 .vt 根据 Empty_Sig 陆续成功从 FIFO 中读出数据。

实验十七说明：

从仿真结果中，我们可以知道几个事实。

第一，FIFO 的调用绝对需要控制信号，不然会发生像在 T10 的意外。

第二，Full_Sig 和 Empty_Sig 是不适合同步 FIFO 的写操作 (T17 的意外)。

我们知道同步 FIFO，它的行为时遵守“时间点”的概念，“决定”的结果是产生在该时间点的未来，参考值是来至改时间的过去。然而控制信号 Full_Sig 和 Empty_Sig 在同步 FIFO 的内部产生了问题，导致写操作失败 (Full_Sig 和同步 FIFO 写入数据不一致的关系)。

实验十七结论:

同步 FIFO 和异步 FIFO，虽然它们只是相差了一个字，但是它们全然是不同的。Quartus II 所生成的是异步 FIFO，但是它可以兼容为同步 FIFO。

现在稍微把头脑冷静一下，在《Verilog HDL 那些事儿 - 建模篇》中 FIFO 的目的是用来缓冲信息和独立化接口模块。然而同步 FIFO 完全可以胜任这一点，为何还要执着于异步 FIFO 呢？但是余下的问题是控制信号 Full_Sig 和 Empty_Sig 同步 FIFO 用不了，我们需要其他的控制信号。

实际上 Full_Sig 和 Empty_Sig 同步 FIFO 不是用不了，如果写操作和读操作均用两个时钟的话，这个问题可以漂亮的解决。但是，这样的做法会消耗额外的时钟，也使得同步 FIFO 的调用特别别扭。在最后我们还知道一个事实，对同步 FIFO 的调用，绝对需要控制信号，不然的话会发生像在 T10 的意外。

4.3 适合同步 FIFO 的控制信号

实验十七我们留下了这样一个问题：“**同步 FIFO 不适合 Empty_Sig 和 Full_Sig**，那么有什么样的控制信号适合它呢？”

在这里笔者稍微刷新一下 Empty_Sig 和 Full_Sig 在同步 FIFO 的作用。Empty_Sig 和 Full_Sig 都是 FIFO 典型的控制信号，它们用来判断 FIFO 的状态。如果 Empty_Sig 拉高，这表示 FIFO 已经为空。反之 Full_Sig 拉高的话，则表示 FIFO 已经为满。但是 Empty_Sig 和 Full_Sig 在同步 FIFO 的中，尤其是在写操作中，出现了情况。为了解决一个问题，我们需要适合同步 FIFO 的控制信号。

实验十八：同步 FIFO 改进

在这个实验中，我们放弃了 Empty_Sig 和 Full_Sig，取而代之的是 Left_Sig。作用如名字般，该信号用来反馈出 FIFO 目前的“空格数目”。

fifo_module_2.v

```
1. module fifo_module_2
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [7:0]FIFO_Write_Data,
8.
9.     input Read_Req,
10.    output [7:0]FIFO_Read_Data,
11.
12.    output [2:0]Left_Sig
13. );
14.
15.     ****
16.
17. parameter DEEP = 3'd4;
18.
19.     ****
20.
21. reg [7:0]rShift [DEEP:0];
```

```

22.    reg [2:0]Count;
23.    reg [7:0]Data;
24.
25.    always @ ( posedge CLK or negedge RSTn )
26.        if( !RSTn )
27.            begin
28.
29.                rShift[0] <= 8'd0; rShift[1] <= 8'd0; rShift[2] <= 8'd0;
30.                rShift[3] <= 8'd0; rShift[4] <= 8'd0;
31.                Count <= 3'd0;
32.                Data <= 8'd0;
33.
34.            end
35.        else if( Read_Req && Write_Req && Count < DEEP && Count > 0 )
36.            begin
37.                rShift[1] <= FIFO_Write_Data;
38.                rShift[2] <= rShift[1];
39.                rShift[3] <= rShift[2];
40.                rShift[4] <= rShift[3];
41.                Data <= rShift[ Count ];
42.            end
43.        else if( Write_Req && Count < DEEP )
44.            begin
45.
46.                rShift[1] <= FIFO_Write_Data;
47.                rShift[2] <= rShift[1];
48.                rShift[3] <= rShift[2];
49.                rShift[4] <= rShift[3];
50.
51.                Count <= Count + 1'b1;
52.            end
53.        else if( Read_Req && Count > 0 )
54.            begin
55.                Data <= rShift[Count];
56.                Count <= Count - 1'b1;
57.            end
58.
59.
60.    *****/
61.
62.    assign FIFO_Read_Data = Data;
63.    assign Left_Sig = DEEP - Count;
64.
65.    *****/

```

```
66.  
67. endmodule
```

同样是深度为 4 的同步 FIFO。但是在 12 行中 Left_Sig 取代了 Empty_Sig 和 Full_Sig。在 63 行是 Left_Sig 输出信号的驱动条件“**反馈出 FIFO 目前的空格数目**”。其余的地方和实验十七的没有什么两样。

fifo_module_2.vt

```
1. `timescale 1 ps/ 1 ps  
2. module fifo_module_2_simulation();  
3.  
4.     reg CLK;  
5.     reg RSTn;  
6.  
7.     reg Write_Req;  
8.     reg [7:0]FIFO_Write_Data;  
9.  
10.    reg Read_Req;  
11.  
12.    wire [7:0]FIFO_Read_Data;  
13.  
14.    wire [2:0]Left_Sig;  
15.  
16.    /*****  
17.  
18.    fifo_module_2 U1  
19.    (  
20.        .CLK(CLK),  
21.        .RSTn( RSTn ),  
22.        .Write_Req(Write_Req),  
23.        .FIFO_Write_Data(FIFO_Write_Data),  
24.        .Read_Req(Read_Req),  
25.        .FIFO_Read_Data(FIFO_Read_Data),  
26.        .Left_Sig(Left_Sig)  
27.    );  
28.  
29.    /*****  
30.  
31.    initial  
32.    begin  
33.        RSTn = 0; #10; RSTn = 1;  
34.        CLK = 0; forever #10 CLK = ~CLK;  
35.    end
```

```
36.  
37.      /*****  
38.  
39.      reg [4:0]i;  
40.  
41.      always @ ( posedge CLK or negedge RSTn )  
42.          if( !RSTn )  
43.              begin  
44.  
45.                  i <= 5'd0;  
46.                  Write_Req <= 1'b0;  
47.                  Read_Req <= 1'b0;  
48.                  FIFO_Write_Data <= 8'd0;  
49.  
50.              end  
51.          else  
52.              case( i )  
53.  
54.                  /*****  
55.  
56.                  0:  
57.                      begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= 8'd1; i <= i + 1'b1; end  
58.  
59.                  1:  
60.                      begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= 8'd2; i <= i + 1'b1; end  
61.  
62.                  2:  
63.                      begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= 8'd3; i <= i + 1'b1; end  
64.  
65.                  3:  
66.                      begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= 8'd4; i <= i + 1'b1; end  
67.  
68.                  /*****  
69.  
70.                  4:  
71.                      begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end  
72.  
73.                  5:  
74.                      begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end  
75.  
76.                  6:  
77.                      begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end  
78.  
79.                  7:  
80.                      begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end
```

```
81.  
82.      ****  
83.  
84.      8: // 0 + 1 < 1  
85.      if( Left_Sig <= 1 ) begin Write_Req <= 1'b0; i <= i + 1'b1; end  
86.      else begin Write_Req <= 1'b1; Read_Req <= 1'b0; FIFO_Write_Data <= FIFO_Write_Data + 1'b1; end  
87.  
88.      9: //> DEEP - 1  
89.      if( Left_Sig >= 3 )begin Read_Req <= 1'b0; i <= i + 1'b1; end  
90.      else begin Write_Req <= 1'b0; Read_Req <= 1'b1; end  
91.  
92.      ****  
93.  
94.      10:  
95.      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd5; i <= i + 1'b1; end  
96.      else begin Write_Req <= 1'b0; i <= i + 1'b1; end  
97.  
98.      11:  
99.      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd6; i <= i + 1'b1; end  
100.     else begin Write_Req <= 1'b0; i <= i + 1'b1; end  
101.  
102.      12:  
103.      begin  
104.  
105.          if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd7; end  
106.          else Write_Req <= 1'b0;  
107.  
108.          if( Left_Sig <= 3 ) begin Read_Req <= 1'b1; end  
109.          else Read_Req <= 1'b0;  
110.  
111.          i <= i + 1'b1;  
112.  
113.      end  
114.  
115.      13:  
116.      begin  
117.  
118.          if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'd8; end  
119.          else Write_Req <= 1'b0;  
120.  
121.          if( Left_Sig <= 3 ) begin Read_Req <= 1'b1; end  
122.          else Read_Req <= 1'b0;  
123.  
124.          i <= i + 1'b1;  
125.
```

```

126.           end
127.
128.           14:
129.             if( Left_Sig <= 3 ) begin Write_Req <= 1'b0; Read_Req <= 1'b1; i <= i + 1'b1; end
130.             else begin Read_Req <= 1'b0; i <= i + 1'b1; end
131.
132.           15:
133.             if( Left_Sig <= 3 ) begin Read_Req <= 1'b1; i <= i + 1'b1; end
134.             else begin Read_Req <= 1'b0; i <= i + 1'b1; end
135.
136.           16:
137.             begin Read_Req <= 1'b0; i <= 5'd16; end
138.
139.         endcase
140.
141.
142. endmodule

```

上面是激励文件。第 56~80 行（步骤 0~7）是不使用控制信号的情况下对同步 FIFO 调用。
_{<1>}前四个时钟是写入 4 个数据，亦即 8'd1, 8'd2, 8'd3, 8'd4。然而后四个时钟则是陆续
_{<2>}读出 FIFO 的数据。

第 84~86 行（步骤 8）是利用控制信号，对 FIFO 陆续写入数据，直到 if 条件满足。
 第 88~90 行（步骤 9）同样是利用控制信号，对 FIFO 陆续读出数据，直到 if 条件满足。
_{<4>}

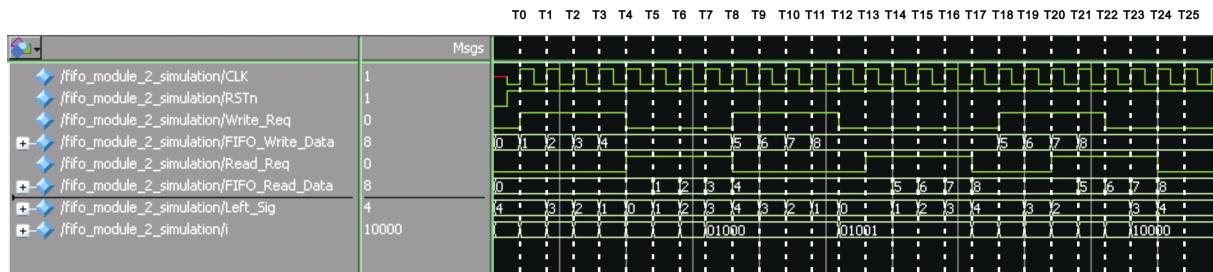
步骤 10~15 的操作，主要是把同步 FIFO 想象为“它同时被两方调用”，一方写另一方读。
 在步骤 10~15 中，同步 FIFO 的调用同样也是用控制信号。
_{<5>}在步骤 10~13 被想象为 A 方向同步 FIFO 写入数据，然而在步骤 12~15 被想象为 B 方从同步 FIFO 读取数据。
_{<6>}

94~134 行的动作如下：

步骤 10，A 对 FIFO 写入数据 8'd5 (94~96 行)。
 步骤 11，A 对 FIFO 写入数据 8'd6 (98~100)
 步骤 12，A 对 FIFO 写入数据 8'd7 (105~106)，同时 B 对 FIFO 读出数据 (108~109)。
 步骤 13，A 对 FIFO 写入数据 8'd8 (118~119)，同时 B 对 FIFO 读出数据 (121~122)。
 步骤 14，B 对 FIFO 读出数据 (128~130)。
 步骤 15，B 对 FIFO 读出数据 (132~134)。

在这里，可能读者会产生疑问：“控制信号为什么是 `Left_Sig >= 1` 或者 `Left_Sig <= 1` 呢？”这不是笔者随便填的，具体的原因请看仿真的结果。

仿真结果：



上图是仿真结果。从 T0~T7 是 .vt 的步骤 0~7，它的操作是写入四个数据，然后读出 4 个数据，没有使用控制信号。

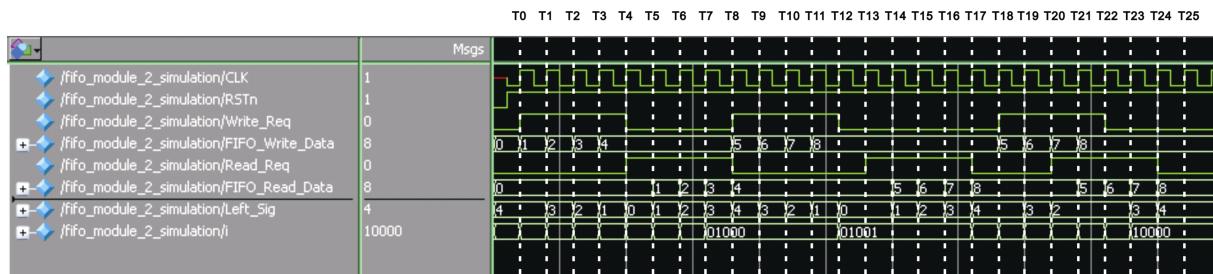
在 T8 的时候 .vt 是步骤 8，它“决定”陆续写入数据，直到 if 条件成立。在 T8~T11 之间，利用控制信号（Left_Sig）对 FIFO 的写入还蛮顺利的。那么重点来了，在 T12 的时候 .vt 还是保持在步骤 8，它检测到 Left_Sig 的过去值是 1，这时候 if 条件成立了 ($\text{Left_Sig} \leq 1$)，它“决定”拉低 Write_Req。在同一个时候，.v 检测到 Write_Req 的过去值是逻辑 1，所以 .v “决定”写操作。在 T12 的未来 Write_Req 被拉低，数据 8'd8 被写入 FIFO。

在 T9 的时候 .vt 进入步骤 9，它“决定”陆续读出数据，直到 if 条件成立。在 T13~16 之间，利用控制信号从 FIFO 读取数据的工作还蛮顺利的。在 T17 这时候 .vt 还保持在步骤 9，它检测到 Left_Sig 的过去值是 3，所以 if 条件成立 ($\text{Left_Sig} \geq 3$) 并且它“决定”拉低 Read_Req。在同一个时候 .v 检查到 Read_Req 的过去值是逻辑 1，它“决定”读取数据。所以在 T17 的未来 Read_Req 被拉低，并且数据 8'd8 被读出。

.vt 接下来的动作可以想象为“有两方同时对同步 FIFO 的调用，一方是写操作，另一方是读操作，它们均利用控制信号”。

在 T18 的时候 .vt 是步骤 10，亦即 A 方对 FIFO 写操作。A 方先检测到 Left_Sig 的过去值是 4，if 条件成立 ($\text{Left_Sig} \geq 1$)，它“决定”拉高 Write_Req，并且“决定”发送数据 8'd5。在同一个时候 .v 检测到 Write_Req 和 Read_Req 的过去值均为逻辑 0，结果 .v 安静的等待。所以在 T18 的未来，Write_Req 被拉高，数据 8'd5 发送在 FIFO_Write_Data。

在 T19 的时候 .vt 是步骤 11，亦即 A 方对 FIFO 写操作。A 检测到 Left_Sig 的过去值是 4，if 条件成立 ($\text{Left_Sig} \geq 1$)，它“决定”拉高 Write_Req，并且“决定”发送数据 8'd6。在同一个时候 .v 检测到 Write_Req 过去值是逻辑 1，它“决定”写入 FIFO_Write_Data 的过去值，亦即 8'd5。所以在 T19 的未来，Write_Req 持续被拉高，数据 8'd6 被发送，数据 8'd5 被写入 FIFO。



在 T20 的时候 .vt 是步骤 12，亦即 A 方对 FIFO 写操作，B 方对 FIFO 读操作。A 方检测到 Left_Sig 的过去值是 3，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”拉高 Write_Req 并且“决定”发送数据 8'd7。在同样的时间，B 方检测到 Left_Sig 的过去值是 3，if 条件成立 ($\text{Left_Sig} \leq 3$) 它“决定”拉高 Read_Req。

在同一个时候 .v 检测到 Write_Req 的过去值是逻辑 1，它“决定”将数据 8'd6 读入。所以在 T20 的未来，Write_Req 持续被拉高，Read_Req 被拉高，数据 8'd7 被发送，数据 8'd6 被读入 FIFO。

在 T21 的时候 .vt 是步骤 13，亦即 A 方对 FIFO 写操作，B 方对 FIFO 读操作。A 方检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”拉高 Write_Req 并且“决定”发送数据 8'd8。在同样的时间，B 方检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \leq 3$) 它“决定”拉高 Read_Req。

在同一个时候 .v 检测到 Write_Req 和 Read_Req 的过去值均为逻辑 1，它“决定”将数据 8'd7 读入，将 8'd5 读出。所以在 T21 的未来，Write_Req 持续被拉高，Read_Req 持续被拉高，数据 8'd8 被发送至 FIFO_Writa_Data，数据 8'd7 被写入 FIFO，数据 8'd5 被读出 FIFO。

在 T22 的时候 .vt 进入步骤 14，亦即 A 方结束写入操作，B 方继续对 FIFO 读操作。A 方“决定”拉低 Write_Req (注: .vt 文件的 129 行)。在同样的时间，B 方检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \leq 3$)，它“决定”拉高 Read_Req。

在同一个时候 .v 检测到 Write_Req 和 Read_Req 的过去值均为逻辑 1，它“决定”读入数据 8'd8 和读出数据 8'd6。所以在 T22 的未来，Write_Req 拉低，Read_Req 持续拉高，数据 8'd8 被写入 FIFO，数据 8'd6 被读出。

在 T23 的时候 .vt 进入步骤 15，亦即 B 方对 FIFO 读操作。B 方检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \leq 3$)，它“决定”拉高 Read_Req。在同一个时候 .v 检测到 Read_Req 的过去值为逻辑 1，它“决定”读出数据 8'd7。所以在 T22 的未来，Read_Req 持续拉高，数据 8'd7 被读出。

在 T24 的时候 .vt 进入步骤 16，B 方读操作结束，并且“决定”拉低 Read_Req。在同一个时候 .v 检测到 Read_Req 的过去值为逻辑 1，它“决定”读出数据 8'd8。所以在 T24 的未来，Read_Req 被拉低，数据 8'd8 被读出。

实验十八说明:

(呼 ~ 先给笔者松一口气 ...) 在仿真结果中 T8~T16 对于使用 Left_Sig 针对同步 FIFO 写入和读出操作，但是没有发生写入失败的意外。在 T18~T23，是一个假想的状态，假设用两方，A 方和 B 方，一方对 FIFO 写操作，一方对 FIFO 读操作，它们都是用控制信号。结果也是没有发生任何意外，真是可喜可贺。

此实验的本质

在仿真结果里我们可以看到一个信息。如果调用同步 FIFO 没有出现意外的话，.vt (激励文件) 和 .v (同步 FIFO) 之间的沟通使用了一个时钟，亦即 .vt 上一个时钟发信号，.v 下一个时钟才了解。

实验十八结论:

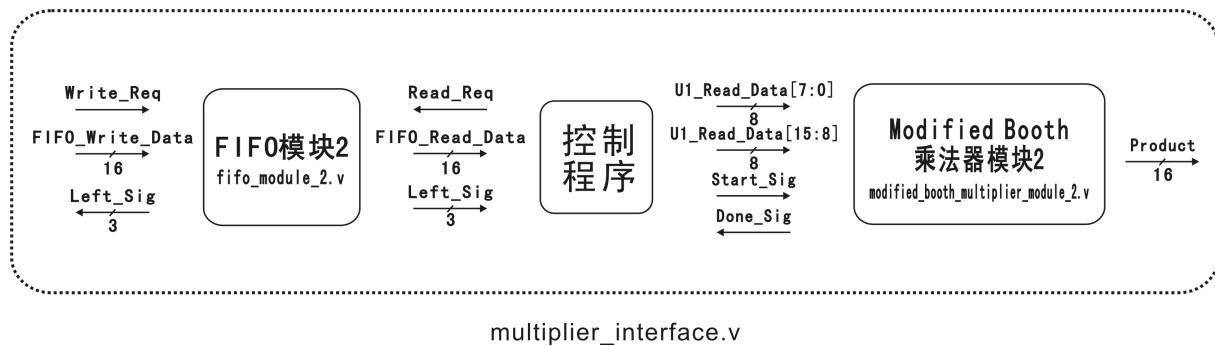
Full_Sig 和 Empty_Sig 既然不适合同步 FIFO，反之 Left_Sig 却可以帮助到同步 FIFO。但是 Left_Sig 的使用方法比较别扭一点，使用者必须清楚 FIFO 的深度为前提。

事实上实验十七和十八是围绕着“模块之间的沟通与控制信号的关系”故事，不过是利用同步 FIFO 来借题发挥而已。我们知道仿顺序操作典型的控制信号，有 Start_Sig 和 Done_Sig，然而同步 FIFO 中的 Write_Read, Read_Req 对于 Start_Sig; Full_Sig, Empty_Sig 和 Left_Sig 对于 Done_Sig；它们有几分相似。

说实话，同步 FIFO 的调用有几分难度，但是只要了解了，掌握了，又是向前踏出一大步

4.4 再建接口模块

在这一章节当中，我们要测试在实验十八中所建立的 FIFO 模块。



上图是久违的建模，该组合模块 `multiplier_interface.v` 是一个乘法器接口。接口的输入缓冲是基于实验十八中的同步 FIFO，它的深度为 4，位宽为 16。其中 `FIFO_Write_data` 的高八位是被乘数 A 驱动，低八位是乘数 B 驱动。

组合模块中的“控制程序”担任 FIFO 和乘法器之间的协调控制。它从 FIFO 读出数据，过滤数据，送往乘法器，然后启动乘法器（`U1_Read_Data[15:8]` 是被乘数 A，`U1_Read_Data[7:0]` 是乘数 B）。乘法器是实验七的 Modified Booth 乘法器 • 改。

（在这里涉及许多“低级建模”的基础，如果读者不明白笔者在说什么，请好好的复习《Verilog HDL 那些事儿 - 建模篇》第五章- 接口建模。）

实验十九：乘法器接口

`fifo_module_2.v`

```

1. module fifo_module_2
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [15:0]FIFO_Write_Data,
8.
9.     input Read_Req,
10.    output [15:0]FIFO_Read_Data,
11.

```

```
12.      output [2:0]Left_Sig
13. );
14.
15. ****
16.
17. parameter DEEP = 3'd4;
18.
19. ****
20.
21. reg [15:0]rShift [DEEP:0];
22. reg [2:0]Count;
23. reg [15:0]Data;
24.
25. always @ ( posedge CLK or negedge RSTn )
26.     if( !RSTn )
27.         begin
28.
29.             rShift[0] <= 15'd0; rShift[1] <= 15'd0; rShift[2] <= 15'd0;
30.             rShift[3] <= 15'd0; rShift[4] <= 15'd0;
31.             Count <= 3'd0;
32.             Data <= 15'd0;
33.
34.         end
35.     else if( Read_Req && Write_Req && Count < DEEP && Count > 0 )
36.         begin
37.             rShift[1] <= FIFO_Write_Data;
38.             rShift[2] <= rShift[1];
39.             rShift[3] <= rShift[2];
40.             rShift[4] <= rShift[3];
41.             Data <= rShift[ Count ];
42.         end
43.     else if( Write_Req && Count < DEEP )
44.         begin
45.
46.             rShift[1] <= FIFO_Write_Data;
47.             rShift[2] <= rShift[1];
48.             rShift[3] <= rShift[2];
49.             rShift[4] <= rShift[3];
50.
51.             Count <= Count + 1'b1;
52.         end
53.     else if( Read_Req && Count > 0 )
54.         begin
55.             Data <= rShift[Count];
```

```
56.           Count <= Count - 1'b1;
57.       end
58.
59.
60.   *****/
61.
62.   assign FIFO_Read_Data = Data;
63.   assign Left_Sig = DEEP - Count;
64.
65.   *****/
66.
67. endmodule
```

该 FIFO 模块和实验十八中的 FIFO 模块没有什么区别，只是位宽为 16 位罢了。

multiplier_interface.v

```
1. module multiplier_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [15:0]FIFO_Write_Data,
8.     output [2:0]Left_Sig, 为何接口需要此信号输出?
9.                         给调用此模块往FIFO写入数据的模块使用
10.    output [15:0]Product
11. );
12.
13. *****/
14.
15. reg isRead;
16.
17. wire [2:0]U1_Left_Sig;
18. wire [15:0]U1_Read_Data;
19.
20. fifo_module_2 U1
21. (
22.     .CLK( CLK ),
23.     .RSTn( RSTn ),
24.     .Write_Req( Write_Req ),
25.     .FIFO_Write_Data( FIFO_Write_Data ),
26.     .Read_Req( isRead ),
```

```

27.      .FIFO_Read_Data( U1_Read_Data ),
28.      .Left_Sig( U1_Left_Sig )
29. );
30.
31. ****
32.
33. reg isStart;
34.
35. wire U2_Done_Sig;
36. wire [15:0]U2_Product;
37.
38. modified_booth_multiplier_module_2 U2
39.
40. (
41.     .CLK( CLK ),
42.     .RSTn( RSTn ),
43.     .Start_Sig( isStart ),
44.     .A( U1_Read_Data[15:8] ),
45.     .B( U1_Read_Data[7:0] ),
46.     .Done_Sig( U2_Done_Sig ),
47.     .Product( U2_Product )
48. );
49. ****
50.
51. reg i;
52.
53. always @ ( posedge CLK or negedge RSTn )
54.   if( !RSTn )
55.     begin
56.       isRead <= 1'b0;
57.       isStart <= 1'b0;
58.       i <= 1'b0;
59.     end
60.   else
61.     case( i )
62.
63.       0:
64.         if( U1_Left_Sig <= 3 ) begin isRead <= 1'b1; i <= i + 1'b1; end
65.
66.       1:
67.         if( U2_Done_Sig ) begin isStart <= 1'b0; i <= i - 1'b1; end
68.         else begin isRead <= 1'b0; isStart <= 1'b1; end
69.
70.

```

```

71.         endcase
72.
73.     *****/
74.
75.     assign Left_Sig = U1_Left_Sig;
76.
77.     assign Product = U2_Product;
78.
79. *****/
80.
81. endmodule

```

multiplier_interface.v 是一个组合模块。第 20~29 行实例了 FIFO 模块，38~47 行实例化了乘法器。在 15, 33 分别声明了寄存器 isRead, isStart (哎 ... modelsim 的编译器不给力的关系, 寄存器必须在调用之前声明, 不然 modelsim 编译不通过。), isRead 用来驱动 FIFO 的 Read_Req (26 行), isStart 用来启动乘法器 (42 行)。

第 51~71 是控制程序。控制程序在步骤 0 先检测 FIFO 的 Left_Sig (U1_Left_Sig), 如果 FIFO 不为空, 就从 FIFO 读出数据, 然后步骤 i 递增。在 66~68 行是步骤 1, 控制程序启动乘法器, 并且等待直到乘法器完成操作, 然后返回步骤 0 (67 行)。

(注意: 乘法器 U2 的 A 和 B 输入口, 是直接由 U1_Read_Data 驱动, 27 行, 43~44 行, 所以从步骤 0 到步骤 1, 乘法器的操作信息就已经准备好了)

在 75 行, 笔者把 FIFO 的 Left_Sig 直接引出来。

multiplier_interface.vt

```

1. `timescale 1 ps/ 1 ps
2. module multiplier_interface_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Write_Req;
8.     reg [15:0]FIFO_Write_Data;
9.     wire [2:0]Left_Sig;
10.
11.    wire [15:0]Product;
12.
13.    *****/
14.
15.    multiplier_interface U1
16.    (

```

```
17.          .CLK(CLK),
18.          .RSTn(RSTn),
19.          .Write_Req(Write_Req),
20.          .FIFO_Write_Data(FIFO_Write_Data),
21.          .Left_Sig(Left_Sig),
22.          .Product(Product)
23.      );
24.
25.      *****/
26.
27.      initial
28.      begin
29.          RSTn = 0; #10; RSTn = 1;
30.          CLK = 0; forever #10 CLK = ~CLK;
31.      end
32.
33.      *****/
34.
35.      reg [3:0]i;
36.
37.      always @ ( posedge CLK or negedge RSTn )
38.          if( !RSTn )
39.              begin
40.                  Write_Req <= 1'b0;
41.                  FIFO_Write_Data <= 16'd0;
42.                  i <= 4'd0;
43.              end
44.          else
45.              case( i )
46.
47.                  0:
48.                      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd12 , 8'd9 }; i <= i + 1'b1; end
49.                      else Write_Req <= 1'b0;
50.
51.                  1:
52.                      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd33 , 8'd10 }; i <= i + 1'b1; end
53.                      else Write_Req <= 1'b0;
54.
55.                  2:
56.                      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd40 , 8'd5 }; i <= i + 1'b1; end
57.                      else Write_Req <= 1'b0;
58.
59.                  3:
60.                      if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd127 , 8'd127 }; i <= i + 1'b1; end
61.                      else Write_Req <= 1'b0;
```

```
62.
63.          4:
64.          if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd37 , 8'd21 }; i <= i + 1'b1; end
65.          else Write_Req <= 1'b0;
66.
67.          5,6,7,8:
68.          begin Write_Req <= 1'b0; i <= i + 1'b1; end
69.
70.          9:
71.          if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd9 , 8'd8 }; i <= i + 1'b1; end
72.          else Write_Req <= 1'b0;
73.
74.          10:
75.          begin Write_Req <= 1'b0; i <= 4'd10; end
76.
77.          endcase
78.
79.          *****/
80.
81. endmodule
```

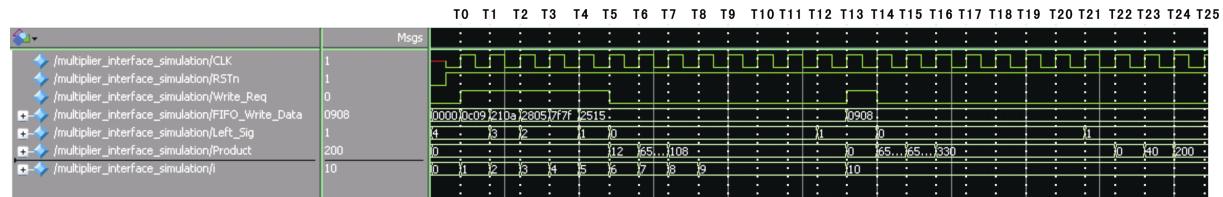
上面是 .vt 文件。在步骤 0~4 (47~65 行) 利用控制信号 Left_Sig 分别向 FIFO 写入数据 12*9, 33*10, 40*5, 127*127, 37*21。步骤 0~4 的写法都有一个共同点, if 条件先判断 FIFO 是否不为满, 如果是就写入数据, 并且递增 i 进入下一个步骤。否则拉低 Write_Req, 直到 FIFO 不未满为止。

步骤 5~8 (67~68 行) 是空置 4 个时钟。

步骤 9 (70~72 行), if 先判断 FIFO 是否不为满, 如果是就写入数据 9*8。如果不是, 就拉低 Write_Req 直到 FIFO 不未满为止。

具体的结果还是要看仿真结果。

仿真结果：



上图是仿真结果。在 T0 的时候 .vt 是步骤 0，它检测到 Left_Sig 的过去值是 4，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”发送数据 $12*9$ 。在同一个时间，控制程序在步骤 0，它检测到 Left_Sig 的过去值是 4，if 条件不成立 ($\text{Left_Sig} \leq 3$)，所以它“决定”作罢。在 T0 的未来，数据 $12*9$ 被发送在 FIFO_Write_Data。

在 T1 的时候 .vt 是步骤 1，它检测到 Left_Sig 的过去值是 4，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”向 FIFO 发送数据 $33*10$ 。在同一个时候，控制程序在步骤 0，它检测到 Left_Sig 的过去值依然是 4，if 条件不成立 ($\text{Left_Sig} \leq 3$)，它“决定”作罢。所以在 T1 的未来，数据 $33*10$ 发送在 FIFO_Write_Data，数据 $12*9$ 被写入 FIFO。

在 T2 的时候 .vt 是步骤 2，它检测到 Left_Sig 的过去值是 3，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”向 FIFO 发送数据 $40*5$ 。在同一个时候，控制程序在步骤 0，它检测到 Left_Sig 的过去值是 3，if 条件成立 ($\text{Left_Sig} \leq 3$)，它“决定”从 FIFO 读取数据。所以在 T2 的未来，数据 $40*5$ 发送在 FIFO_Write_Data，数据 $33*10$ 被写入 FIFO，数据 $12*9$ 会被读出。

在 T3 的时候 .vt 是步骤 3，它检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”向 FIFO 发送数据 $127*127$ 。在同一个时候，控制程序在步骤 1，它“决定”启动乘法器。所以在 T3 的未来，数据 $127*127$ 发送在 FIFO_Write_Data，数据 $40*5$ 被写入 FIFO，数乘法器开始执行操作信息为 $12*9$ 。

在 T4 的时候 .vt 是步骤 4，它检测到 Left_Sig 的过去值是 2，if 条件成立 ($\text{Left_Sig} \geq 1$) 它“决定”向 FIFO 发送数据 $37*21$ 。在同一个时候，控制程序依然在步骤 1，它正等待乘法器的完成信号。所以在 T4 的未来，数据 $37*21$ 发送在 FIFO_Write_Data，数据 $127*127$ 被写入 FIFO，数乘法器依然执行中。

在 T5 的时候 .vt 是步骤 5 它什么都不干。在同一个时候，控制程序依然在步骤 1，它正等待乘法器的完成信号。在 T5 的未来，数据 $37*21$ 被写入 FIFO。

在 T6,T7,T8 的时候 .vt 是步骤 6,7,8，它什么都不干。在同一个时候，控制程序依然在步骤 1，它正等待乘法器的完成信号。

在 T9, T10, T11 的时候 .vt 是步骤 9，它检测到 Left_Sig 的过去值是 0，if 条件不成立 ($\text{Left_Sig} \geq 1$) 它什么都不干。在 T9, T10 的时候，控制程序依然在步骤 1，它正等待乘法器的完成信号。大约在 T11 的时候，控制程序接收到乘法器的完成信号 ($12*9$)

已经操作完成), 它决定关闭乘法器, 并且返回步骤 0。

在 T12 的时候.vt 是步骤 9, 它检测到 Left_Sig 的过去值是 0, if 条件不成立 (Left_Sig >= 1) 它什么都不干。同一个时候, 控制程序在步骤 0, 它检测到 left_Sig 的过去值是 0, if 条件成立 (Left_Sig <= 3) 它“决定”从 FIFO 读取数据, 并且进入下一个步骤。所以在 T12 的未来, 数据 33*10 从 FIFO 读取, 并且控制程序进入步骤 1。

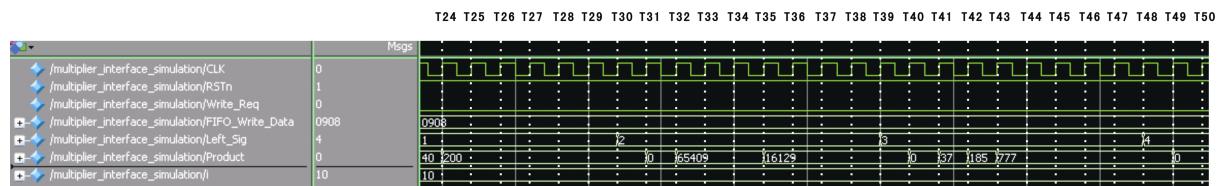
在 T13 的时候.vt 是步骤 9, 它检测到 Left_Sig 的过去值是 1, if 条件不成立 (Left_Sig >= 1) 它“决定”发送数据 9*8。同一个时候, 控制程序在步骤 1, 它“决定”启动乘法器。所以在 T13 的未来, 数据 9*8 会发送在 FIFO_Write_Data 上, 然而乘法器开始启动, 操作信息为 33*10。

在 T14 的时候 .vt 进入步骤 10, 它的工作已经完成了, 所以它会不停的发呆。在同一个时候, 控制程序在步骤 1, 它等待乘法器的完成信号。在 T15~T19 的时候控制程序在步骤 1, 它等待乘法器的完成信号。

大约在 T20 的时候, 控制程序得到乘法器的反馈信号 (33*10 已经操作完成), 所以它“决定”关闭乘法器, 并且进入步骤 0。

在 T21 的时候控制程序在步骤 0, 它检测到 left_Sig 的过去值是 0, if 条件成立 (Left_Sig <= 3) 它“决定”从 FIFO 读取数据, 并且进入下一个步骤。所以在 T21 的未来, 数据 40*5 从 FIFO 读取, 并且控制程序进入步骤 1。

在 T22 的时候, 控制信号在步骤 1, 它“决定”启动乘法器。所以在 T22 的未来, 乘法器开始启动, 操作信息为 40*5。



大约在 T29 的时候, 控制程序得到乘法器的反馈信号 (40*5 已经操作完成), 所以它“决定”关闭乘法器, 并且进入步骤 0。

在 T30 的时候, 控制程序在步骤 0, 它检测到点 left_Sig 的过去值是 1, if 条件成立 (Left_Sig <= 3) 它“决定”从 FIFO 读取数据, 并且进入下一个步骤。所以在 T30 的未来, 数据 127*127 从 FIFO 读取, 并且控制程序进入步骤 1。

在 T31 的时候, 控制信号在步骤 1, 它“决定”启动乘法器。所以在 T31 的未来, 乘法器开始启动操作信息为 127*127。

大约在 T38 的时候, 控制程序得到乘法器的反馈信号 (127*127 已经操作完成), 所以它“决定”关闭乘法器, 并且进入步骤 0。

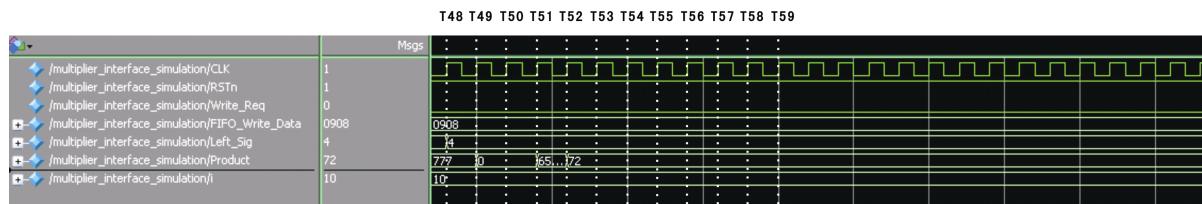
在 T39 的时候控制程序在步骤 0，它检测到在该时间点 left_Sig 的过去值是 2, if 条件成立 ($\text{Left_Sig} \leq 3$) 它“决定”从 FIFO 读取数据，并且进入下一个步骤。所以在 T39 的未来，数据 $37*21$ 从 FIFO 读取，并且控制程序进入步骤 1。

在 T40 的时候，控制信号在步骤 1，它“决定”启动乘法器。所以在 T40 的未来，乘法器开始启动操作信息为 $37*21$ 。

在 T47 的时候，控制程序得到乘法器的反馈信号（ $37*21$ 已经操作完成），所以它“决定”关闭乘法器，并且进入步骤 0。

在 T48 的时候控制程序在步骤 0，它检测到 Left_Sig 的过去值是 3, if 条件成立 ($\text{Left_Sig} \leq 3$) 它“决定”从 FIFO 读取数据，并且进入下一个步骤。所以在 T48 的未来，数据 $9*8$ 从 FIFO 读取，并且控制程序进入步骤 1。

在 T49 的时候，控制信号在步骤 1，它“决定”启动乘法器。所以在 T49 的未来，乘法器开始启动，操作信息为 $9*8$ 。



大约在 T56 的时候，控制程序得到乘法器的反馈信号（ $9*8$ 已经操作完成），所以它“决定”关闭乘法器，并且进入步骤 0。

大约在 T57 候控制程序在步骤 0，可是到目前为止 FIFO 已经为空了，控制程序检测到 Left_Sig 的过去值为 4, if 条件不成立 ($\text{Left_Sig} \leq 3$)，它什么都不干。那么我们可以下结论，该乘法器接口的工作已经完毕（内部的乘法器已经没有信息可以操作了，因为 FIFO 已经为空了）。

实验十九说明：

在这个实验中，我们建立了乘法器接口来测试实验十八的同步 FIFO 模块。可是在仿真结果中笔者没有直接引出 FIFO 的 Write_Req 和 Read_Req 信号，乘法器的 Start_Sig 和 Done_Sig 信号，取而代之的是笔者只是引出 FIFO 的 Left_Sig 信号。

笔者实在无力将全部信号引出，然后在仿真结果中一个一个慢慢解释，如果这样做笔者估计会精尽人亡。相反的，如果读者有好好的理解 4.1~4.3 章中的重点，即使没有笔者的解释，读者也能明白。

在这里，我们来说说仿真中的几个重点吧：

同步 FIFO 模块被调用的时候，最重要的还是使用控制信号 Left_Sig 来进行数据写入和数据读出。有几个比较经典的情况是在 T5~T12 的时候，由于 FIFO 已经为满了 .vt 在步骤 9，根据 Left_Sig 它得知 FIFO 目前的状态。.vt 等待直到 T13 的时候，在 FIFO 不未满的状态下，才“决定”写入数据。

还有一点就是在 T15 和之后的时钟，那时候的 .vt 已经完成工作。反之控制程序还在继续工作着，控制程序在步骤 0~1 之间一直轮替着完成乘法工作 - 读取 FIFO 的数据 - 然后启动乘法器 - 等待乘法器完成。在这期间，没有任何意外发生。.vt 向 FIFO 写入的 5 个数据，都成功被乘法器操作出来。

实验十九结论：

在实验十九证明了同步 FIFO 同样可以应用在接口模块中。

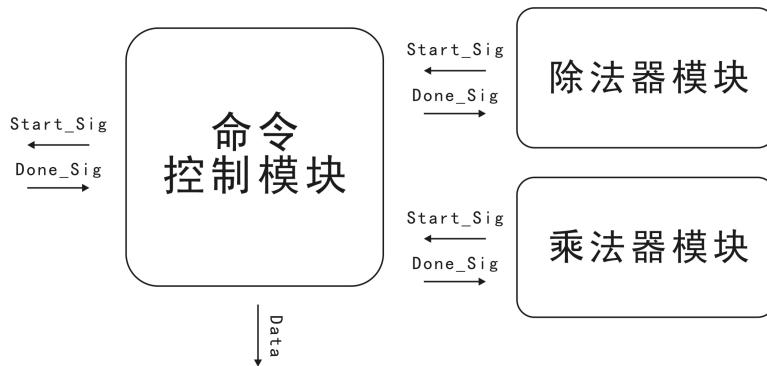
4.5 混种建模的可能性

到目前为止，笔者使用过许多种的建模来完成实验的要求，下面我们来一个简单的回顾。

假设实验的要求是：

A/ B = Q 和 R
Q * R = 答案

从实验的题目中，我们知道我们需要一个除法器执行 A 除以 B 得到结果 Q 和 R，然后再需要一个乘法器完成 Q * R 来得到最终答案。

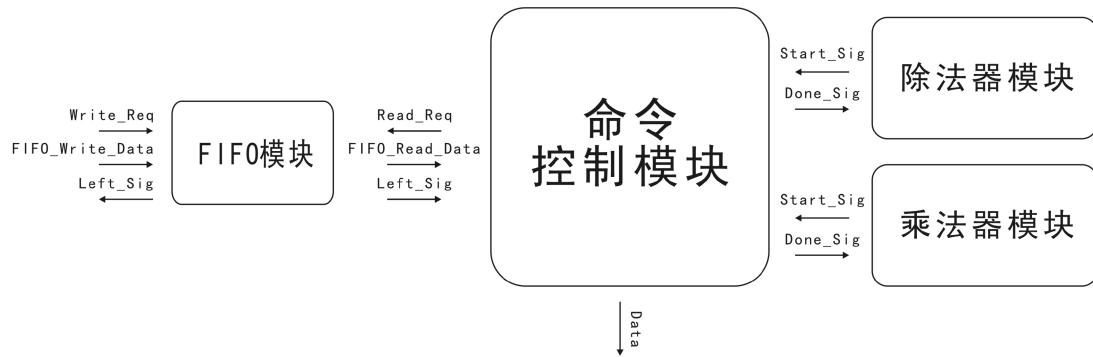


上面是笔者最爱仿顺序操作，它的优点就是容易控制。但是它的缺点是，两次性操作之间有时间间隔。假设除法器和乘法器的都使用 10 个时钟来工作的话，那么每一次的操作都会先启动除法器模块，然后等待它完成操作，并且求得 Q 和 R。得到 Q 和 R 之后，启动乘法器模块，然后等待它完成操作，并且求得最终答案。

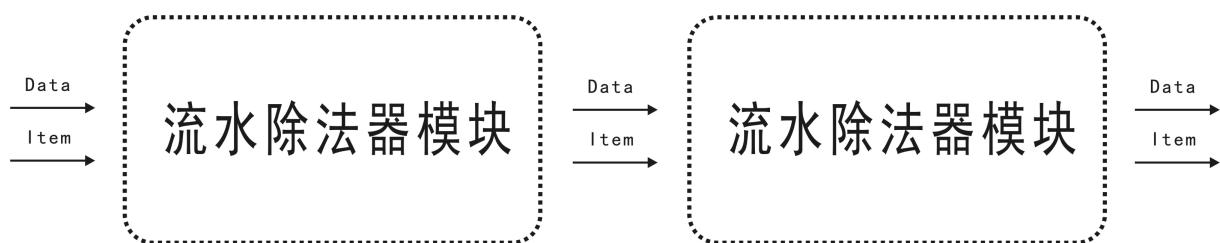
每一次操作中大致的时钟消耗如下：

命令控制模块的沟通时间为 2 个 (Start_Sig 和 Done_Sig 消耗之间为 2 个时钟)，
除法器模块的沟通时间为 2 个，等待除法器模块完成的时间为 10 个，
乘法器模块的沟通时间为 2 个，等待乘法器模块完成的时间为 10 个，
一共使用 $2 + 2 + 10 + 2 + 10 = 26$ 个时钟。

所以说，两次性操作之间的时间间隔是 26 个时钟。尽管仿顺序操作的建模对时钟消耗如此之多，如果实验的速度要求不任性的话，这个缺点可以接受的。

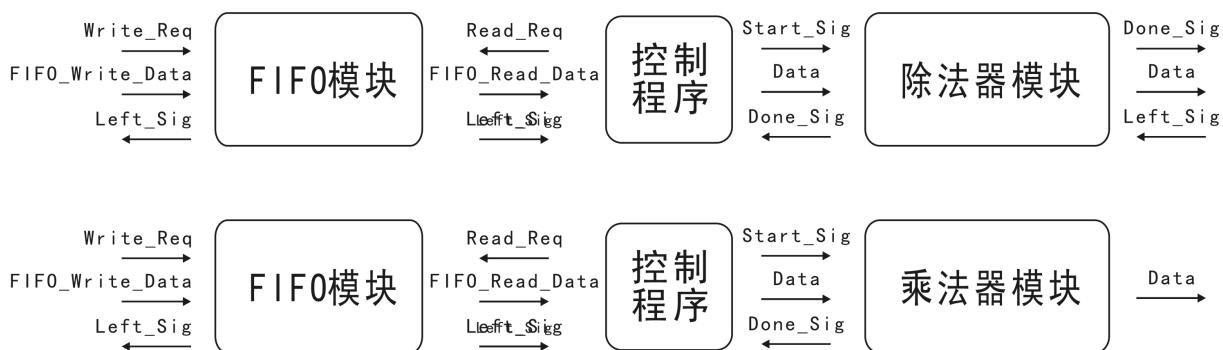


为了解决放顺序操作建模中的缺点（两次性操作之间有时间间隔），笔者可以在命令控制模块的前面加上 FIFO 作为输入缓冲，这也成为了“低级建模”中的“接口建模”。这样作的好处有：每一次调用都可以无需等待内部操作。同样它也有坏处，将模块封装成为“接口”，会使得内部的建模层次和连线关系变成更复杂。



假设上述的办法读者不能接受，那么笔者建议可以使用流水操作建模来完成实验的要求。虽说流水操作的建模，在连线关系上或者层次上都比上述的建模来得更精简。但是读者不要忘了，**流水操作的建模“必须拥有固定的操作步骤”作为前提；此外它还有一个致命的缺点就是潜伏时间**。这些潜伏时间会根据调用频率的不同，变成不可预测，最终使得模块的控制变成困难。

这样也不行，那样也不行，笔者到底要怎样做什么才好呢？当人逼急的时候就会跳墙，笔者就从接口建模捉一点优点，然后又从流水操作建模捉一点优点，然后整合这些优点再建立出新的建模



上图是两个操作模块的接口建模 ... 笔者先建立简单的除法器接口，然后又建立简单的乘法器接口。然后将它们串联起来，就会成为下图：



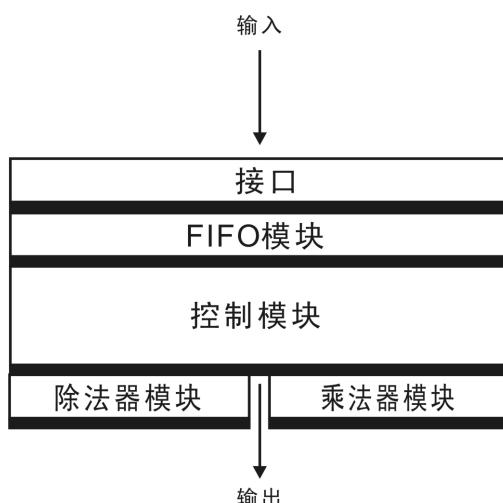
<1>除法器接口的 Done_Sig 驱动乘法器接口 FIFO 的 Write_Req，Data 驱动

FIFO_Write_Data。然而除法器接口右边的 Left_Sig 由乘法器接口的 Left_Sig 驱动。

<3>是存在关联的，需要<3>的判断来决定是否可以写入FIFO 写入需要<1>的使能
在这里，我们使用了流水操作的“外壳”使得连线关系变得简单，然后在里边使用了简
单的接口建模，这使得模块的控制能力提高（真是一石二鸟的好方法）。这样的建模方
法已经是“混种”了，它的优缺点是父类的一半一半。

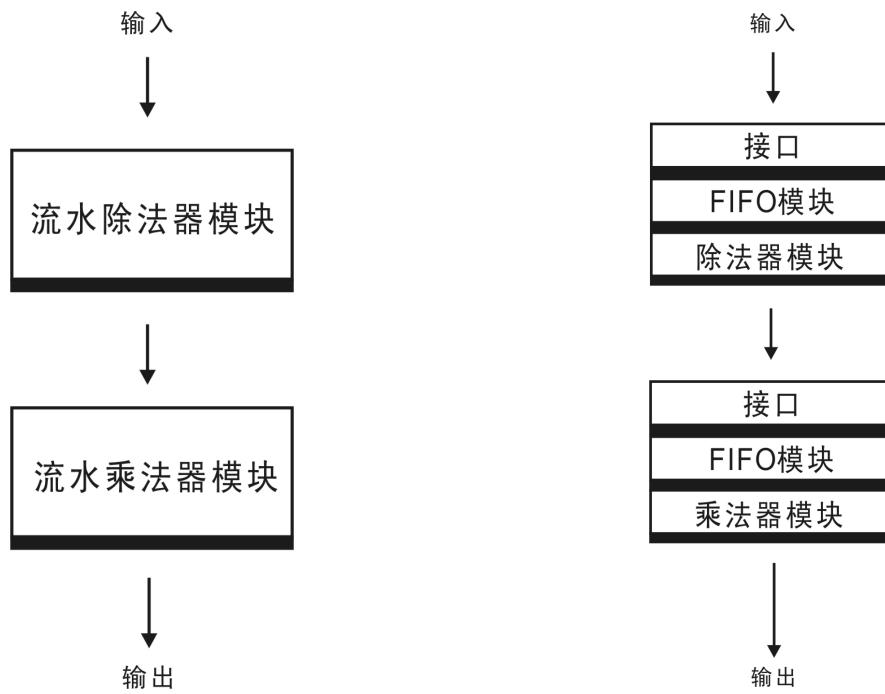
实际上笔者“不承认”这样的建模方法是“另一个新的建模方法”。笔者把它提出来只
是要说明“建模技巧之间混种的可能性”而已。

再来我们从层次的角度去分析各个不同的建模：



上图是（低级建模）接口建模的层次图，接口建模的基本思路就基于仿顺序操作，越高级的建模，它的层次就越高。

注意看中间的控制模块，它凸起来的高度都比其他模块来得多。这也是没办法的事，因为控制模块几乎要联系所有模块，结果使它又凸又肥，这也意味着控制模块的代码量会比其他模块来得多。所以呀，只要控制模块有什么差错，该接口模块就不能正常工作了。还有一点就是，如果层次越多，这也表示每一个层次之间的“沟通时间”（延迟时间）就越多。



上面的左图是流水操作的建模，每一个模块的高度都根据模块的步骤次数。虽说流水操作建模的连线关系很最整洁而且，模块之间的沟通时间也是最短(几乎是1个时钟而已)。但是流水操作有诸多的缺点，如潜伏时间无法预测，建模的难度高等，这使得它黯淡无光。上面的右图是混种的建模。很显明，该建模拥有父类的优缺点的一半一半，说好不见得好到那里去，说坏却不能挑剔什么。

在这里，到底要使用那一种建模来完成实验的要求？如果是笔者的话，笔者还是偏爱低级建模的**接口建模**，因为它最友善，毕竟用久了，多少都有感情。但是实际的情况还是要根据实验的要求而定。

实验二十：混种建模

继续上述的实验要求：

- $A / B = Q$ 和 R ，并且 $Q * R = \text{答案}$ 。

然后再追加几个要求：

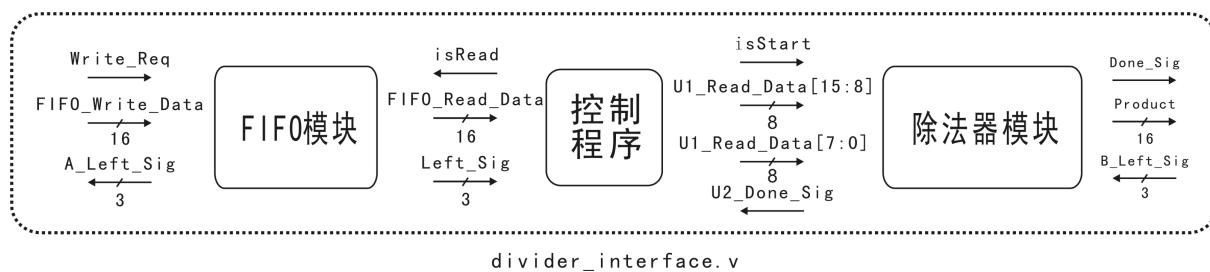
- (一) 除法器使用实验八的传统除法器，乘法器使用实验一的传统乘法器。
- (二) 模块的操作效率上尽可能提升，而且简化模块之间的连线关系。

根据第2项追加要求，读者们可能会立即反应流水操作的建模是首选，但是不能忽略第1项的追加要求，就是除法器和乘法器均的对象是传统的除法器和乘法器。我们知道**传统除法器和乘法器都有一个特性**，就是根据操作数据的不同就有不同操作步骤和时钟消

耗。然而流水操作必须基于“固定的操作步骤，固定的时钟消耗”，显然流水操作的建模是不适合的。

然后再根据第1项的追加要求，低级建模的接口建模肯定是首选，但是考虑了第2项追加要求得话，显得接口建模的执行效率有点低了。所以在这里我们必须使用混种的建模来完成实验的要求，无论是第1项还是第2项追加要求，混种的建模都可以胜任。

divider_interface.v



```

1. module divider_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [15:0] FIFO_Write_Data,
8.     output [2:0] A_Left_Sig,
9.
10.    output Done_Sig,
11.    output [15:0] Product,
12.    input [2:0] B_Left_Sig
13.
14. );
15.
16. ****
17.
18. reg isRead;
19. wire [15:0] U1_Read_Data;
20. wire [2:0] U1_Left_Sig;
21.
22. fifo_module_2 U1( CLK, RSTn, Write_Req, FIFO_Write_Data, isRead, U1_Read_Data, U1_Left_Sig );
23.
24. assign A_Left_Sig = U1_Left_Sig;
25.
26. ****
27.

```

```
28.      reg isStart;
29.      wire U2_Done_Sig;
30.      wire [7:0]U2_Quotient;
31.      wire [7:0]U2_Reminder;
32.
33.      divider_module U2( CLK, RSTn, isStart, U1_Read_Data[15:8], U1_Read_Data[7:0], U2_Done_Sig, U2_Quotient, U2_Reminder );
34.
35.      *****/
36.
37.      reg [1:0]i;
38.      reg isDone;
39.
40.      always @ ( posedge CLK or negedge RSTn )
41.          if( !RSTn )
42.              begin
43.                  i <= 2'd0;
44.                  isRead <= 1'b0;
45.                  isStart <= 1'b0;
46.                  isDone <= 1'b0;
47.              end
48.          else
49.              case( i )
50.
51.                  0:
52.                      if( U1_Left_Sig <= 3 ) begin isRead <= 1'b1; i <= i + 1'b1; end
53.                      else isRead <= 1'b0;
54.
55.                  1:
56.                      if( U2_Done_Sig ) begin isStart <= 1'b0; i <= i + 1'b1; end
57.                      else begin isStart <= 1'b1; isRead <= 1'b0; end
58.
59.                  2:
60.                      if( B_Left_Sig >= 1 ) begin isDone <= 1'b1; i <= i + 1'b1; end
61.                      else isDone <= 1'b0;
62.
63.                  3:
64.                      begin isDone <= 1'b0; i <= 2'd0; end
65.
66.
67.              endcase
68.
69.      *****/
70.
71.      assign Done_Sig = isDone;
72.      assign Product = { U2_Quotient , U2_Reminder };
```

```

73.
74.      ****
75.
76. endmodule

```

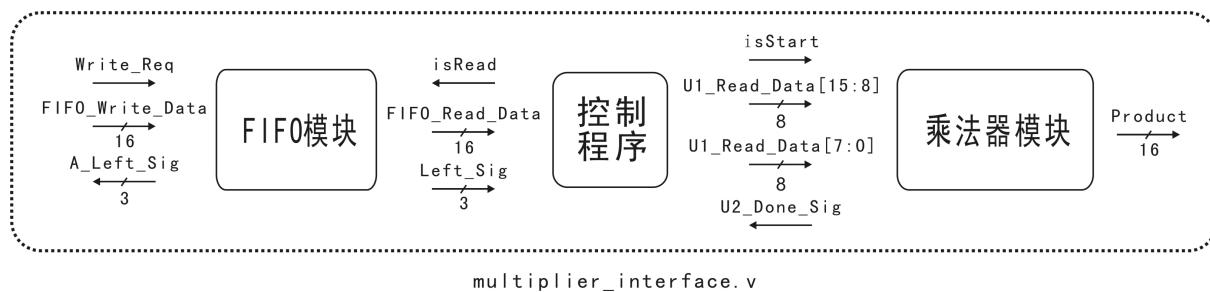
第 18~26 行是同步 FIFO 的实例化，其中也包括同步 FIFO 的连线关系。在 24 行 A_Left_Sig 的输出口是由 U1-同步 FIFO 的 U1_Left_Sig 来驱动。第 28~33 行是由实验八传统除法器实例化而成的 U2。该除法器的输入驱动（被除数和除数）是由 U1 同步 FIFO 的 U1_Read_Data 信号（[15..8]代表被除数，[7..0]代表除数。）。

第 37~67 行是控制程序。在步骤 0（51~53 行）控制程序先检查 FIFO 是否为不为空，如果是的话就拉高 isRead，并且从 FIFO 读取数据，然后步骤 i 递增；否则的话拉低 isRead。在步骤 1（55~57 行）控制程序启动除法器，拉低 isRead（57 行），并且等待除法器反馈完成信号（56 行）。

步骤 2（59~61 行），控制程序判断从外部（右边）的 FIFO 是否为不为满，如果是的话就拉高 isDone，并且将除法器完成操作的数据写入外部的 FIFO（60 行），步骤 i 递增；否则的话拉低 isDone。在步骤 3（63~64 行）控制程序拉低 isDone，并且清理步骤 i，以示步骤从新开始。

在 72 行 Product 的输出，是由除法器 U2 的操作结果 U2_Quotient 和 U2_Reminder 联合驱动（33 行）。

multiply_interface.v



```

1. module multiply_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [15:0]FIFO_Write_Data,
8.     output [2:0]A_Left_Sig,
9.
10.    output [15:0]Product
11.

```

```
12. );
13.
14. ****
15.
16. reg isRead;
17.
18. wire [15:0]U1_Read_Data;
19. wire [2:0]U1_Left_Sig;
20.
21. fifo_module_2 U1( CLK, RSTn, Write_Req, FIFO_Write_Data, isRead, U1_Read_Data, U1_Left_Sig );
22.
23. assign A_Left_Sig = U1_Left_Sig;
24.
25. ****
26.
27. reg isStart;
28.
29. wire U2_Done_Sig;
30.
31. multiplier_module U2( CLK, RSTn, isStart, U1_Read_Data[15:8], U1_Read_Data[7:0], U2_Done_Sig, Product );
32.
33. ****
34.
35. reg i;
36.
37. always @ ( posedge CLK or negedge RSTn )
38. if( !RSTn )
39. begin
40.     i <= 1'd0;
41.     isRead <= 1'b0;
42.     isStart <= 1'b0;
43. end
44. else
45. case( i )
46.
47.     0:
48.         if( U1_Left_Sig <= 3 ) begin isRead <= 1'b1; i <= i + 1'b1; end
49.         else isRead <= 1'b0;
50.
51.     1:
52.         if( U2_Done_Sig ) begin isStart <= 1'b0; i <= i - 1'b1; end
53.         else begin isStart <= 1'b1; isRead <= 1'b0; end
54.
55.
56.     endcase
```

```

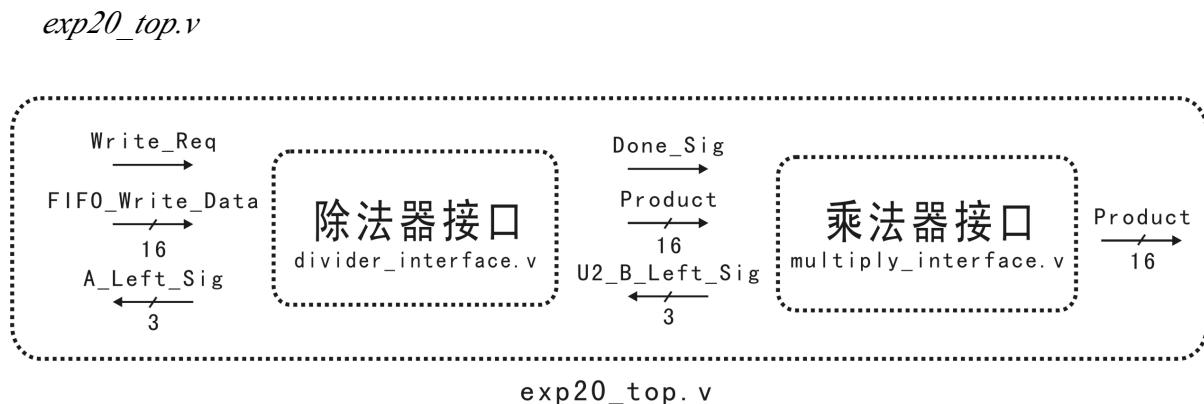
57.
58. ****
59.
60.
61. endmodule

```

第 16~23 行是 U1 同步 FIFO 的实例化。在 23 行的 A_Left_Sig 是由同步 FIFO 的 U1_Left_Sig 来驱动。第 27~31 行是 U2 乘法器的实例化（实验一的传统乘法器），然而在 21 行，乘法器的输入驱动（被乘数和乘数）则由 U1 同步 FIFO 的 U1_Read_Data 信号（[15..8]为被乘数，[7..0]为乘数）。

第 35~56 行是控制程序。在步骤 0 (47~49 行) 先判断 U1 的 FIFO 是否为不为空，如果是的话就拉高 isRead，从 FIFO 读取数据，步骤 i 递增；否则的话就拉低 isRead 并且等待。在步骤 1 (51~53 行) 控制程序启动乘法器和拉低 isRead，并且等待乘法器反馈完成信号。当乘法器反馈完成信号，关闭乘法器，并且返回步骤 0。

注意：乘法器接口的 Product 直接由 U2 乘法器的 product 驱动（31 行）。



exp20_top.v 组合模块是用来组合除法器接口和乘法器接口。

```

1. module exp20_top
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req,
7.     input [15:0]FIFO_Write_Data,
8.     output [2:0]Left_Sig,
9.
10.    output [15:0]Product,
11.
12.    ****

```

```
13.
14.     output SQ_U1_Done_Sig,
15.     output [15:0]SQ_U1_Product,
16.     output [2:0]SQ_U2_A_Left_Sig
17.
18.     *****/
19.
20. );
21.
22. *****/
23.
24.     wire U1_Done_Sig;
25.     wire [15:0]U1_Product;
26.     wire [2:0]U1_B_Left_Sig;
27.     wire [2:0]U2_A_Left_Sig;
28.
29.     divider_interface U1
30. (
31.         .CLK( CLK ),
32.         .RSTn( RSTn ),
33.         .Write_Req( Write_Req ),           // input - from top
34.         .FIFO_Write_Data( FIFO_Write_Data ), // input - from top
35.         .A_Left_Sig( Left_Sig ),          // ouput - to top
36.         .Done_Sig( U1_Done_Sig ),         // output - to U2
37.         .Product( U1_Product ),          // output - to U2
38.         .B_Left_Sig( U2_A_Left_Sig )    // input - form U2
39. );
40.
41. *****/
42.
43.     multiply_interface U2
44. (
45.         .CLK( CLK ),
46.         .RSTn( RSTn ),
47.         .Write_Req( U1_Done_Sig ),        // input - from U1
48.         .FIFO_Write_Data( U1_Product ),   // input - from U1
49.         .A_Left_Sig( U2_A_Left_Sig ),    // output - to U1
50.         .Product( Product )            // output - to top
51. );
52.
53. *****/
54.
55.     assign SQ_U1_Done_Sig = U1_Done_Sig;
56.     assign SQ_U1_Product = U1_Product;
```

```

57.      assign SQ_U2_A_Left_Sig = U2_A_Left_Sig;
58.
59.      /*****
60.
61.
62. endmodule

```

第 3~10 行是组合模块的输入输出口, 14~16 行是仿真输出。24~39 行是 divider_interface.v 的实例化, 43~51 行是 multiply_interface.v 的实例化。连线关系基本上和图形一样, 笔者就不罗嗦了。但是有一点请注意, 在 55~57 行的仿真输出, 笔者将 U1 的完成信号 (55 行) 引出, 因为它充当 U2 的 Write_Req, 故很重要。56 行的 SQ_U1_Product 同样也被笔者引出, 它充当 U2 的 FIFO_Write_Data。57 行的 SQ_U2_A_Left 充当 U2 的 Left_Sig, 它也一样被笔者引出。

exp20_top.vt

```

1. `timescale 1 ps/ 1 ps
2. module exp20_top_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg Write_Req;
8.     reg [15:0]FIFO_Write_Data;
9.
10.    wire [2:0]Left_Sig;
11.    wire [15:0]Product;
12.
13.    wire SQ_U1_Done_Sig;
14.    wire [15:0]SQ_U1_Product;
15.    wire [2:0]SQ_U2_A_Left_Sig;
16.
17.    *****/
18.
19.    exp20_top U-Top
20.    (
21.        .CLK(CLK),
22.        .RSTn(RSTn),
23.        .Write_Req(Write_Req),
24.        .FIFO_Write_Data(FIFO_Write_Data),
25.        .Product(Product),
26.        .Left_Sig(Left_Sig),
27.        .SQ_U1_Done_Sig( SQ_U1_Done_Sig ),
28.        .SQ_U1_Product( SQ_U1_Product ),

```

Verilog HDL 那些事儿 - 时序篇

```
29.      .SQ_U2_A_Left_Sig( SQ_U2_A_Left_Sig )
30. );
31.
32.      ****
33.
34.      initial
35.      begin
36.          RSTn = 0; #10; RSTn = 1;
37.          CLK = 0; forever #10 CLK = ~CLK;
38.      end
39.
40.      ****
41.
42.      reg [3:0]i;
43.
44.      always @ ( posedge CLK or negedge RSTn )
45.          if( !RSTn )
46.              begin
47.                  i <= 4'd0;
48.                  Write_Req <= 1'b0;
49.                  FIFO_Write_Data <= 16'd0;
50.              end
51.          else
52.              case( i ) // A / B = R & Q, R * Q = Answer
53.
54.                  0: // Q = 12, R = 7, anwer = 84
55.                  if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd127 , 8'd10 }; i <= i + 1'b1; end
56.                  else Write_Req <= 1'b0;
57.
58.                  1: // Q = 8, R = 5, anwer = 40
59.                  if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd69 , 8'd8 }; i <= i + 1'b1; end
60.                  else Write_Req <= 1'b0;
61.
62.                  2: // Q = 1, R = 38, anwer = 38
63.                  if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd98 , 8'd60 }; i <= i + 1'b1; end
64.                  else Write_Req <= 1'b0;
65.
66.                  3: // Q = 9, R = 10, anwer = 90
67.                  if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd127 , 8'd13 }; i <= i + 1'b1; end
68.                  else Write_Req <= 1'b0;
69.
70.                  4,5,6,7,8:
71.                  begin Write_Req <= 1'b0; i <= i + 1'b1; end
72.
73.                  9: // Q = 4, R = 10, anwer = 40
```

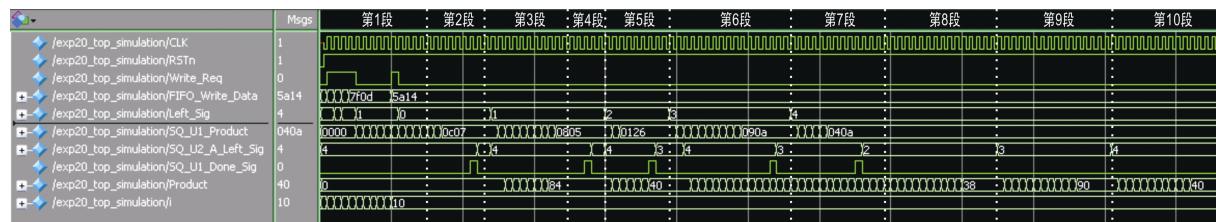
```

74.           if( Left_Sig >= 1 ) begin Write_Req <= 1'b1; FIFO_Write_Data <= { 8'd90 , 8'd20 }; i <= i + 1'b1; end
75.           else Write_Req <= 1'b0;
76.
77.           10:
78.           begin Write_Req <= 1'b0; i <= 4'd10; end
79.
80.       endcase
81.
82.   *****/
83.
84. endmodule

```

第 52~80 行是仿真的操作。步骤 0~3 分别写入 4 个不同的数据（54~68 行），第 70~71 行停止了 5 个步骤(时钟)。然后在步骤 9 又写入另一组数据(73~75 行)。在步骤 10(77~78 行)，拉低 Write_Req 并且永远的徘徊。

仿真结果：



在这里笔者不想再一个时钟一个时钟的解释了，这样笔者会升天的 ...

我们先看第一段，.vt 往 .v 写入 5 个数据，当 .vt 写完 5 个数据后，U1 还是没有完成任一个数据的操作 (SQ_U1_Product)。在第二段，U1 已经求得第一个数据的结果，U1 判断 U2 的 FIFO 状态 (SQ_U2_A_Left_Sig)，发现不为满，然后 U1 就拉高 Done_Sig (SQ_U1_Done_Sig)，并且将求出的结果写入 U2 的 FIFO。

在第三段，U2 已经完成第一个数据的操作（注意 Product）。在同样的阶段，U1 已经完成第二个数据的操作 (SQ_U1_Product)。在第四段，U1 它判断 U2 的 FIFO 的状态 (SQ_U2_A_Left_Sig)，发现不为满，拉高 Done_Sig (SQ_U1_Done_Sig)，将第二个已经完成的数据写入 U2 的 FIFO。

在第五段，U2 已经完成第二个数据的操作（注意 Product），在同一个时间 U1 完成第三个数据的操作 (SQ_U1_Product)，然后它判断 U2 的 FIFO 的状态 (SQ_U2_A_Left_Sig)，发现不为满，拉高 Done_Sig (SQ_U1_Done_Sig) 将第三个以完成的数据写入 U2 的 FIFO（注意 SQ_U1_A_Left_Sig）。

在第六段，U1 已经完成第四个数据的操作 (SQ_U1_Product)，然后它判断 U2 的 FIFO

的状态 (SQ_U2_A_Left_Sig), 发现不为满, 拉高 Done_Sig (SQ_U1_Done_Sig) 将第四个已经完成的数据写入 U2 的 FIFO。在同一个时候 U2 还没有完成第三个数据的操作。

在第七段, U1 已经完成第五个数据的操作 (SQ_U1_Product), 然后它判断 U2 的 FIFO 的状态 (SQ_U2_A_Left_Sig), 发现不为满, 拉高 Done_Sig (SQ_U1_Done_Sig) 将第五个已经完成的数据写入 U2 的 FIFO。在同一个时候 U2 还没有完成第三个数据的操作。

在第八段, U2 完成第三个数据的操作 (注意 Product)。这时候的 U1 已经完成工作, 并且休息中。

在第九段, U2 完成第四个数据的操作 (注意 Product)。在第十段, U2 完成第五个数据的操作 (注意 Product)。

实验二十说明:

从仿真结果上, 我们可以得到以下的信息:

.vt 文件独立于 U1, U1 独立于 U2, 为什么这样说呢? 在第一段的时候 .vt 已经将 5 个数据写入 U1 后, 它的工作就结束了, 但是 U1 和 U2 还在工作。然后在第八段 U1 已经完成 5 个数据的操作, 反之 U2 还在工作。直到第十段 U2 还完成操作。

在这里同步 FIFO 作为了输入缓冲的作用。如果以接口建模的角度来说, FIFO 独立化了每一个模块。在第一段至第十段, 在每一个时钟中每一个模块都在工作, 这也使得时钟的利用率大大的提升 (符合实验的第二项追加要求。)

实验二十结论:

在实验二十中, 我们看到了混种的建模, 达成了实验的挑剔要求。当然, 笔者还是不承认混种建模是一种新的建模技巧, 混种建模的存在原本就是为了实现实验的各种不合理要求 (各种设计要求)。在实验二十中所使用的混种建模, 它拥有流水操作建模的外壳, 接口建模的内部。如果实验二十的要求, 再一次更动, 混种建模的形态会为了符合实验要求, 再一次更动和变形。

总结

这一章的中心好似都是围绕着同步 FIFO 而展开。从了解 Start_Sig 与 Done_Sig 的协调性到掌握同步 FFIO，笔者都只在讲述一个重点，那就是：“模块的沟通”。笔者非常看重“模块的沟通”，因为笔者明白到单文件主义（单模块）是满足不了实验的要求，如果采取多模块的办法，“模块的沟通”就是我们要面对的问题。

稍微回忆一下，在“低级建模”中，模块之间的协调操作，笔者都是以“步骤”的形式来实现。实际上“步骤”就是时钟（时序）的显性指示，因为考虑到初学者对“时钟”的模糊，所以才没有深入。最好的例子就是 Start_Sig 和 Done_Sig 充当模块沟通的控制信号，读者只要懂得使用它们用不着去理解它们（不要关心时序上的关系）。

但是当读者逐渐深入 Verilog HDL 语言，读者会发现这样是完全不够的。就好比实验十七到实验十八同步 FIFO 的改进，单单掌握步骤是应付不了。这一章笔记的最让笔者感到兴奋的是：当笔者了解模块之间沟通的简单规则，混种建模的实现就有可能（这可发现使笔者兴奋了足足一个晚上）。要实现混种建模的前提条件是：掌握各种建模技巧；强化了解步骤与时钟的关系性等。

混种建模有什么好处，估计笔者不用说读者都会理解？

假设有一个实验要求，某个建模技巧无法充分发挥，那么我们可以往设计里面添加其他的建模技巧，使得实验的要求达成。用一句话说就是，建模的弹跳性更强。类似的例子有实验二十 - 乘法器和除法器披着流水操作的外壳，里边却是简化的接口建模。

嗯，话说长了，笔者差不多要结束上半部分了。这上半部分的笔记真的花了笔者很长的时间，消耗了笔者许多的精力，对健康有很大的打击。这一章笔记重点内容就是“模块如何沟通”，或者可以把它看为“从波形图读取信息”的入门。

下半部分：综合和仿真

第五章：仿真前的故事

5.1 我眼中的仿真

关于仿真这东西给笔者很纠结，笔者把仿真看成是学习 Verilog HDL 语言之后的功课。网上常有的一套学习方法就是一边学习用 Verilog HDL 语言建立模块，一边使用 modelsim 观察输出，这样的学习方法很有道理，而且初期也有很大的帮助 … 在这里，笔者说一些老实话，这样的学习方法对初学的朋友是有极限的。如果读者反问笔者为什么呀，其中又要扯出很久很久以前的故事 ……

我们知道 Verilog HDL 语言又分为两套，综合部分和验证部分。综合部分就是在我们常常用来建模，然而验证部分是用来验证模块。自然而然我们会把“仿真”和“建模（综合）”看成两个平台的东西。笔者在接触 Verilog HDL 语言一段时间后，有一种奇怪的感觉一直驱着笔者思考有关“建模，综合语言，验证语言，仿真它们之间的关系”。

笔者看到很多的激励文本都是使用验证语言在编辑，笔者一直反问自己“难道仿真就是等于验证？为什么不可以把这两种东西放在同样的平台上呢？”根据笔者的记忆，笔者看过那么多有关 Verilog HDL 的资料，它们仿真等于验证时绝对的。正是如此笔者萌生出这样的想法“是否综合语言也可以用于仿真？用什么思路更容易理解波形图？”，而且笔者更加相信自己的感觉。

先把这个话题暂停一下，我们先来探讨“什么是仿真？”

在仿真的世界里，仿真唯一的工作就是建立一个理想的“虚拟环境”（没有任何物理问题），然后观察模块的输出，同时间也要有“虚拟的输入”以达到刺激的效果。既然仿真只是建立虚拟环境来观察输出而已，而且 Verilog HDL 语言也没有强制一定要使用验证的语言去编辑激励的过程（理解波形图是没有固定的方法）。

当笔者察觉到上述的问题后，笔者尝试使用建模的思想去完成仿真工作。在笔者的眼里，仿真工作就如同“建立模块，然后将模块下载到“虚拟的黑金开发板”上而已。但是这个“虚拟的黑金开发板”是理想的，单调的和充满许多可能性的。

仿真的重点就是“某个模块在虚拟的环境中执行的过程”，其中激励是关键，了解波形图信是目的。（初学者们常常误认激励就是“虚拟的输入”，但是笔者却保留这一点。关于激励的故事，我们往后有机会再谈。）

笔者曾经是初学者过，当笔者一边用综合来建立模块，一边用验证来仿真模块，笔者的头简直就要爆开来。那时候的学习一点也开心不起来，为了应付两种不同性质的 Verilog HDL 语言，结果都是两头不到岸，学习以“徘徊边缘”告终。

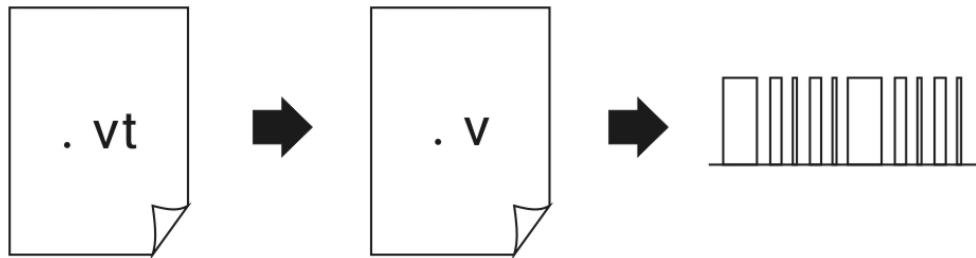
当笔者尝试更换另外一种思想后，重新接触仿真 … 嘿！笔者发现到，一套用在建模的方法，既然也适合用在仿真上。很奇怪的“仿真与建模是两个平台”的概念也随之消失，随着不断深入，笔者也发现到“仿真”和“建模”的亲近关系是非常强，而且 Verilog HDL 的综合语言也可以同时应用在建模和编辑激励文件，此外理解波形图也可以使用建模的思路。

其实一直一来，笔者对仿真始终保留自己的一套方法和想法，笔者也因此纠结了许久。笔者不是故意要和现有的一套方法唱反调，笔者也不是说网上一套的方法是错误。笔者只是希望找到一条更平坦的道路，好让初学者有更多的选择。

我们知道初学 Verilog HDL 语除了面对建模这个大问题以外，还有其它如：“如何区分 Verilog HDL 语言的可综合和可验证？”，“模块是如何沟通？”等问题。如果掌握一套方法，可以应用在许多地方的话，初学者多少也可以更轻松一些。

嗯，笔者也不多话了，上述的内容充其量不过是笔者一厢情愿的想法而已，单单用文字来表达，感觉太抽象了，给人一种摸不到的感觉。如果有耐性和笔者一起探讨“笔者心目中的仿真”的话，读者是很欢迎带着“猜疑”的心情继续浏览笔记。

5.2 激励的故事



在 5.1 一章中，笔者说了激励既不是虚拟输入或者输入。如上图中，初学者都喜欢把激励当成 .vt 文件和 .v 文件之间的箭头。该左边的箭头表示了 .vt 产生输入至 .v 的关系。笔者不是说这样的理解不是不对，只是有所保留而已，笔者对激励有自己另一个观点。

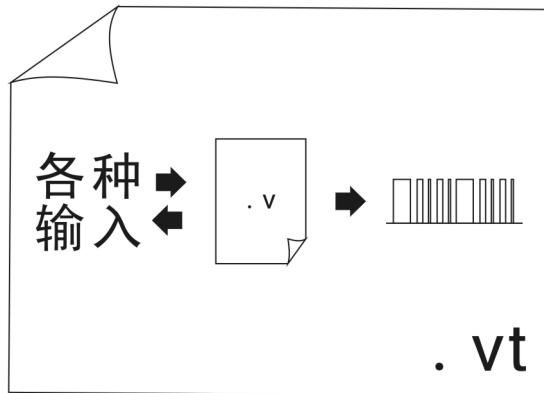
如果把激励放在可以活动的现象中，激励代表了“刺激和反应”。假如把这一观点放映在仿真的身上，那么激励文件表示了“某个模块，有什么刺激（出入），就有什么反应（输出）” - 这一系列过程。因为在仿真的过程中，模块不只是单单接收某个输入，然后产生某个输出，反之仿真中常常会出现“问答”的状况。

举一个例子，在实验一中，我们为了仿真 multiplier_module.v 这个模块，而且我们准备输入 2 组的数据。我们先输入第一组数据，然后等待 multiplier_module.v 的完成反馈，然后再继续输入第二组数据，其中等待和反馈的过程，亦即“问答”。

既然“激励”不是单纯的输入，而是“刺激和反应”，那么什么是“刺激”什么又是“反应”？“刺激”就是输入，在仿真中“输入”分别有“简单输入”“复杂输入”和“条件输入”。简单输入可以是逻辑信号，或者一组数据，复杂输入可以是一组协议，然而条件输入又称为问答输入。反之“反应”在仿真中，就是“输出”或者“反馈”。

假设笔者为了仿真某个模块，笔者就必须编辑好“激励的过程”，“激励的规则”或者“激励的步骤”等 激励文件的创建范围是视情况而定，没有固定的约束。在激励的发生过程中，最重要的就是输出信息了（是地球人都知道），输出的信息（时序图|波形图）反映了该模块是否健康呀？哪里不舒服呀？

读者可能很好奇，笔者为什么那么执着于“定义激励”？概念差一点也不会少了一块肉。笔者如此执着是有原因的，如果笔者的想法是正确的话（没有什么正不正确的啦，见仁见智，每个人都有不同的看法），那么笔者可以更明确的“重新定义”在仿真上的每一个细节。这样做好处可以使学习者更能容易。此外，往后的故事都需要这些“定义”来支持，不然在学习的路上，信心很容易动摇，学习很容易充满疑惑 ...



如果按着笔者的想法，激励文本会是如图上的效果。各种输入驱动着 .v 文件，同时间 .v 文件也反馈和输出信息，然而全部过程称为“激励”。至于“激励”的过程是如何发生，就得看 .vt 文件如何编辑。在这里，读者们能不能接受就见仁见智了… 这毕竟是笔者的一套想法而已，没有任何强迫性。但是笔者不能避免，这样的想法可能会颠覆读者之前对仿真的认识。

5.3 仿真的虚拟环境

《在 Verilog HDL 那些事儿-建模篇》在第一章中笔者说了“FPGA 就是一堆乐高积木，Verilog HDL 就是一双手（工具），用手组合这些乐高积木称为建模，有效的用手段组合这些乐高积木成为建模技巧”。然而这个概念只是针对现实中“FPGA 和 Verilog HDL 语言”之间的关系而已。

在仿真中同样的概念已经不适合了，在这里笔者提出另一种概念，就是“仿真的虚拟环境”，这个“虚拟环境”是理想而且充满许多可能性的。假设一个例子，如果笔者要对某个模块，如串口发送模块执行仿真，那么笔者需要设法建立一个环境去近似现实中我们要测试的对象。

在现实中，这个串口发送模块是针对黑金开发板的串口资源而建模的，那么我们需要模仿黑金开发板去建立这个“虚拟黑金开发板的环境”。这其中，有两个重要的输入就是时钟周期和复位时间。我们知道黑金开发板搭配的时钟频率是 20Mhz，20Mhz 的周期是 50ns，然和一般复位信号的低电平时间大约是 1us~6us。

```
'timescale 1ns / 1ns

module ... ()

.....
initial
begin
    RSTn = 0; # 1000; RSTn = 1; // 1us 的复位时间
    CLK = 0; forever #25 CLK = ~CLK; // 一个周期为 50ns, 半个周期为 25ns
end
.....
endmodule
```

当然，建立虚拟环境不只是需要建立时钟和复位信号那么简单而已，我们还需要在激励文本上，编辑各种的虚拟输入，但是有一点可以肯定的是“时钟和复位是虚拟绝对的”。在这里，笔者只是提出一个简单的概念而已，“仿真的虚拟环境是什么？”一页的空间不肯能把它说清楚，读者还需要从不同的实验中了解，在更多的实例中找到答案。

5.4 综合和仿真

我们先来看看两段不同的 .vt 写法，两种的激励过程同样都是从 FIFO 写入数据。

第一种的.vt 是用验证语言编辑的：

```
.....  
  
reg [7:0]FIFO_Write_Data;  
reg Write_Req;  
  
initial begin  
  
    FIFO_Write_Data = 0; Write_Req = 0; Read_Req = 0;  
    @( posedge RSTn );  
    @( negedge CLK );  
    Write_Req = 1; FIFO_Write_Data = 8'dF0;  
  
    @( negedge CLK );  
    Write_Req = 0;  
    @( negedge CLK );  
    Write_Req = 1;  
    repeat(3) begin  
        FIFO_Write_Data = FIFO_WRite_Data + 8'd02;  
        @( negedge CLK );  
    end  
  
    .....  
.....
```

第二种的 .vt 文件使用建模（综合语言）的方式编辑：

```
reg [7:0]FIFO_Write_Data;  
reg Write_Req;  
reg [3:0]i;  
  
always @ ( posedge CLK or negedge RSTn )  
    if( !RSTn )  
        begin  
            FIFO_Write_Data <= 8'd0;  
            Write_Req <= 1'b0;  
            i <= 4'd0  
        end
```

```
else
  case( i )
    0:
      begin Write_Req <= 1'b1; FIFO_Write_Data <= 8'dF0; i <= i + 1'b1; end

    1:
      begin Write_Req <= 1'b0; i <= i + 1'b1; end

    2, 4, 6:
      begin
        Write_Req <= 1'b1;
        FIFO_Write_Data <= 8'dF0 + 8'd02;
        i <= i + 1'b1;
      end

    3, 5, 7:
      begin Write_Req <= 1'b0; i <= i + 1'b1; end
      .....
```

上面是两种不同的 .vt 文件的写法，两种写法都是向 FIFO 写入几组不同的数据。第一种写法就是网上最常见的用，验证方式去编辑.vt 文件。反之第二种写法是笔者习惯的写法，亦即用“综合语言和建模的思路”编辑的 .vt 文件。

我们以“表达能力”去分析第一种写法的话，估计很多读者都不明白它的意义吧？又或者了解起来很费力气？此外第一种写法“代码和时序之间的关系”表达很模糊。（说实话以笔者的等级笔者也看不清楚。）

第二种的写法读者们是不是觉得很熟悉是吧？它是在低级建模中，建模常用的“仿顺序操作”的写法（这种写法的好处笔者就不在多废话了）。很显然，如果这两种写法相比较的话，第二种写法在许多方面都更胜一筹。

在这里我们可以定下如此的结论：

只要有“时钟信号”和“复位信号”，那么用在建模的一套方法就可以用在仿真身上（在编辑激励文件方面，用建模的思路理解波形图）。如果把这样的想法继续细化的话，我们只要学习 Verilog HDL 语言其中的一部分，亦即“综合语言”，就可以应用在两种不同的地方（建模和仿真）。

世界上是没有免费的饼干，“要把综合语言在编辑激励文件方面应用得好”或者“利用建模的思路去理解波形图”，前提是需要建模技巧的支持。当然，这并不代表掌握建模技巧就是等于掌握仿真，在实际的仿真中不可能那么单纯。我们还需要其他的准备。

总结：

在这一章笔记，笔者只是简单的讲述了、笔者自己本身对仿真一套的看法。笔者曾经也是初学者，笔者深深了解到。打从一开始，如果同时间使用两种不同部分的 Verilog HDL 语言（综合和验证）去完成建模和仿真的话，学习很容易两头不到岸。很多时候在初期，验证语言都不常使用，此外语法也很随便，不利于早期的对 Verilog HDL 语言的学习。

换句话说，我们只要掌握 Verilog HDL 语言的其中一部分，经过“强化”对综合语言的掌握，综合语言也可以使用在两种不同的地方（建模|综合和仿真）。笔者相信这或多或少，可以助初学者更轻松上路，此外对建模的能力可以更上一层楼。笔者很多时候都相信，许多初学者只会看波形图（时序图），则不正真了解波形图上的信息。而且激励文本的编辑也非常矛盾，甚至不知道自己在哪里写错（笑！笔者也是）。

如果把一切看穿到底，仿真的重要性只是其次而已，因为仿真工作只是测试“这个模块如何了？那个模块健康吗？”然后“观察该模块的病征”。真正影响模块本身的就是 Verilog HDL 语言和建模的经过，很好的代码风格和建模技巧可以使得“病症（BUG）更容易避免和指出”。（如果在建模中有强调步骤的存在“病症（BUG）的会更显眼”）

笔者不是说仿真不重要，因为模块都是被动的，即使模块通过编译，模块也不会说“先生，我这里不舒服，麻烦你看一下”之类的话，所以我们需要才需要方法去刺激该模块，然后观察该模块的反应（虐待模块），如果反应在预想范围之内，那么这个模块就是 Okay 的，反之这个模块就“生病”了。模块生病的最多原因都是和代码有直接的关系。

第六章：刺激和激励过程

6.1 精密计数

当笔者看到精密计数，笔者自己也会觉得很好笑，精密计数确实是仿真中最重要的入门课，它和流水灯实验一样经典。就是因为它简单而且经典，所以常常可以使人冒傻和犯错。

我们以一个简单的实验来解释：

实验二十一：仿真定时器

这个实验很简单，我们只要建立两个定时器，一个在 1us 产生定时，则另一个在 3us 产生定时，然后观察这两个定时器在不同状态的输出。

counter_module.v

```
1. module counter_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     output _1US,
7.     output _3US,
8.     output _is1US,
9.     output _is3US,
10.    output [4:0]C1,
11.    output [5:0]C2
12. );
13.
14.     ****
15.
16.    parameter T1US = 5'd20;
17.
18.     ****
19.
20.    reg [4:0]Count_1US;
21.    reg is1US;
22.
```

```

23.    always @ ( posedge CLK or negedge RSTn )
24.        if( !RSTn )
25.            begin Count_1US <= 5'd0; is1US <= 1'b0; end
26.        else if( Count_1US == T1US )
27.            begin Count_1US <= 5'd0; is1US <= 1'b1; end
28.        else
29.            begin Count_1US <= Count_1US + 1'b1; is1US = 1'b0; end
30.
31.    /*****
32.
33.    parameter T3US = 6'd60;
34.
35.    *****/
36.
37.    reg [5:0]Count_3US;
38.    reg is3US;
39.
40.    always @ ( posedge CLK or negedge RSTn )
41.        if( !RSTn )
42.            begin Count_3US <= 6'd0; is3US <= 1'b0; end
43.        else if( Count_3US == T3US )
44.            begin Count_3US <= 6'd0; is3US <= 1'b1; end
45.        else
46.            begin Count_3US <= Count_3US + 1'b1; is3US <= 1'b0; end
47.
48.    *****/
49.
50.    assign _1US = ( Count_1US == T1US ) ? 1'b1 : 1'b0;
51.    assign _3US = ( Count_3US == T3US ) ? 1'b1 : 1'b0;
52.    assign _is1US = is1US;
53.    assign _is3US = is3US;
54.    assign C1 = Count_1US;
55.    assign C2 = Count_3US;
56.
57.    *****/
58.
59. endmodule

```

第 16 行是 1us 的常量声明，第 17 行是 1us 的标志寄存器 is1US。第 23~29 是 1us 的定时器。第 33 行是 3us 的常量声明，第 37 行是 3us 的标志寄存器 is3US。第 40~46 是 3us 的定时器。第 50 行是由组合逻辑驱动 1us 定时信号，第 51 行是由组合逻辑驱动的 3us 定时信号，第 52 行是由 is1US 标志寄存器驱动的 1us 定时信号，第 53 行是由 is3US 标志寄存器驱动的 3us 定时信号。第 54~55 行则是，计数寄存器 Count_1US 和 Count_3US 的相关输出。

counter_module.vt

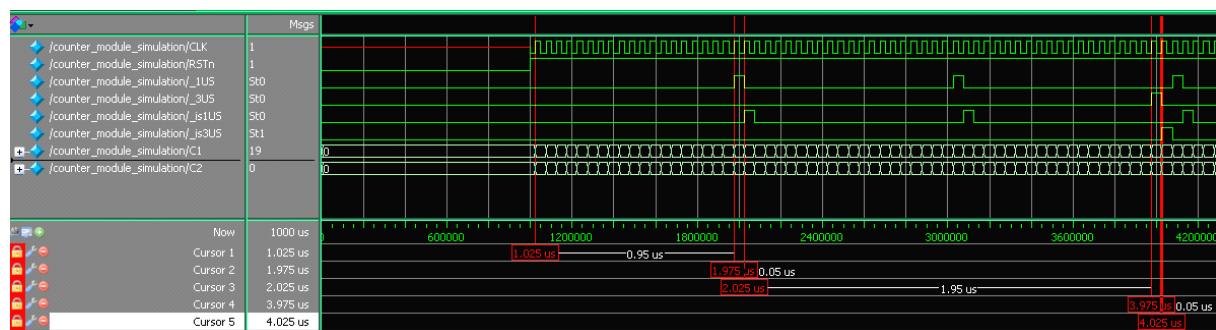
```
1. `timescale 1 ns/ 1 ps
2. module counter_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     wire _1US;
8.     wire _3US;
9.
10.    wire _is1US;
11.    wire _is3US;
12.
13.    wire [4:0]C1;
14.    wire [5:0]C2;
15.
16.    /*****
17.
18.    counter_module i1
19.    (
20.        .CLK(CLK),
21.        .RSTn(RSTn),
22.        ._1US(_1US),
23.        ._3US(_3US),
24.        ._is1US(_is1US ),
25.        ._is3US(_is3US ),
26.        .C1( C1 ),
27.        .C2( C2 )
28.    );
29.
30.    /*****
31.
32.    initial
33.    begin
34.        RSTn = 0; #1000; RSTn = 1;
35.        CLK = 0; forever #25 CLK = ~CLK;
36.    end
37.
38.    /*****
39.
40.
```

41. endmodule

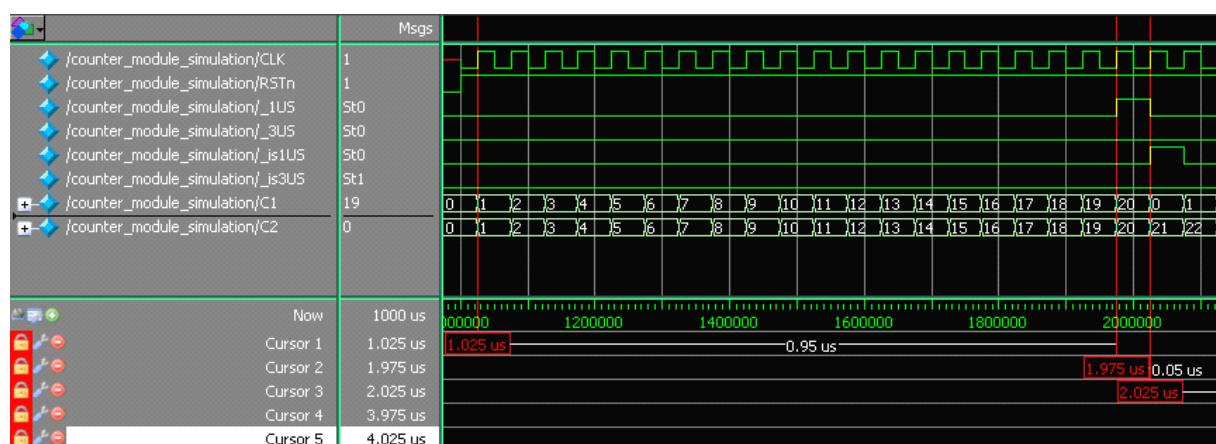
在编辑激励文件的时候，有几个地方笔者非常讲究。在第 4~8 行，笔者尽量会编辑和 .v 的输入输出一致。此外（在 10~20 行）先是实例化要仿真的模块，然后（在 22~26 行）再建立时钟信号和复位信号，余下就是激励过程（这个实验还没有 ... ）。

在第 1 行是时间刻度（时间单位）在这里 1ns 位单位，1ps 为小数（可以无视）。第 4~5 行表示了输入，然而 7~8 行表示了输出。第 10~20 行是测试模块的实例化。第 22~26 行是时钟信号和复位信号，复位信号拉低 1us 的时间（实际情况是 1us~6us）。时钟信号近似了黑金开发板上的时钟频率，亦即 20Mhz。一个时间周期为 50ns，所以半个时钟周期为 25ns。

仿真结果：



在上图仿真的仿真结果是从复位信号拉低开始到 3us 定时信号产生。在这里笔者建立的近似黑金开发板的虚拟环境，其中时间周期为 50ns，复位时间为 1us。在复位信号 RSTn 拉高以后 .v 开始工作。

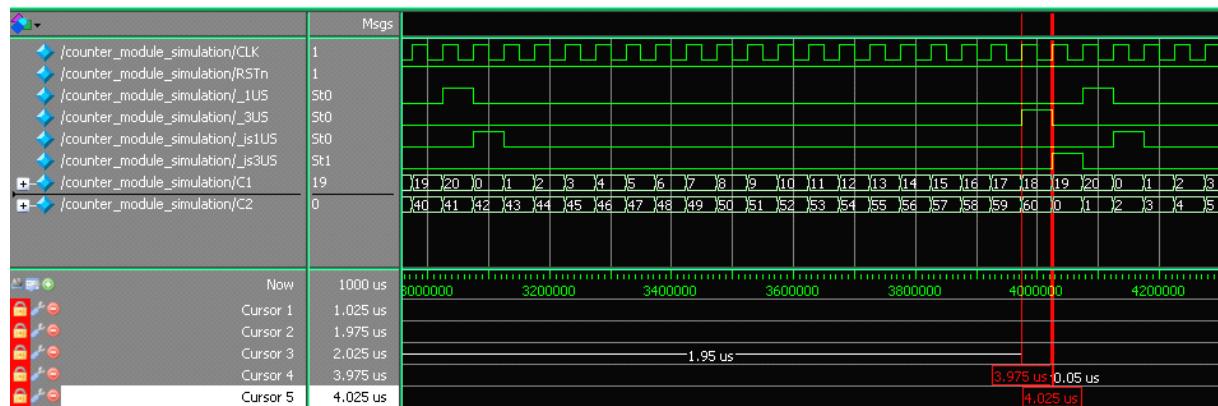


上图是 1us 不同方式的定时结果。

(Cursor 省略为 C) C1~C2 之间表示了由组合逻辑驱动的定时信号，在还未满 1us，亦即在 0.95us 的时候它产生了定时信号。 $0.95\text{us} / 50\text{ns} = 19$ ，如果从 T0 开始算起，亦即在

T18 的时候 .v 它“决定”拉高 _1US 一个时钟（注意：这时候的计数器 C1，计数到 19）。所以在 T18 的未来 _1US 被拉高一个时钟。

反之 C1~C3 之间表示了由寄存器驱动的 1us 定时，在满 1us 产生的时候，它产生定时信号。1us / 50ns = 20，如果从 T0 开始算起，亦即在 T19 的时候 .v 它“决定”拉高 _is1US 一个时钟（注意：这时候的计数器 C1，计数到 20）。所以在 T19 的未来 _is1US 被拉高一个时钟。



上图是 3us 不同方式的定时结果。

(Cursor 省略为 C) C1~C4 之间表示了由组合逻辑驱动的定时信号，在还未满 3us，亦即在 3.975ns - 1.025ns = 2.95ns 的时候它产生了定时信号。2.95us / 50ns = 59。如果从 T0 开始算起，亦即在 T58 的时候 .v 它“决定”拉高 _3US 一个时钟（注意：这时候的计数器 C2，计数到 59）。所以在 T58 的未来 _3US 被拉高一个时钟。

反之 C1~C5 之间表示了由寄存器驱动的 3us 定时，在满 3us 产生的时候，它产生定时信号。3us / 50ns = 60，如果从 T0 开始算起，亦即在 T59 的时候 .v 它“决定”拉高 _is3US 一个时钟（注意：这时候的计数器 C2，计数到 60）。所以在 T59 的未来 _is3US 被拉高一个时钟。

实验二十一说明：

这个实验的目的不是要产生精密度很高的定时，而是要实现精密计数时钟。

在以往的实验，步骤和时钟之间的相差，都是一个又一个的时钟而已，所以我们才可以一个时钟一个时钟的去计数步骤。一旦，某个模块的执行步骤是不固定，又或者时钟消耗高，我们再也不可能一个又一个去计数时钟。相反的，这时候的我们必须取得时钟和时钟相差的时间，然后再“除与”源时钟周期，才能取得真正的“第几个时钟数”。

在实验二十一的仿真中，我们知道 在 T18 _1US “决定”被拉高，在 T19 _is1US “决定”被拉高，在 T58 _3US “决定”被拉高，在 T59 _is3US “决定”被拉高。

此外从仿真结果我们还得到几个信息：

很多朋友常常在建立定时器的时候，假设以 20Mhz 的时钟频率为例，然而要建立 1us 的定时器，那么 1us 的常量声明会是 $20 - 1 = 19$ 。1us 的常量为什么是 19？很多朋友会回答“因为定时器从 0 开始算起”这是一个我们建立定时器的时候的错觉。实际的 1us 常量是 20。（这是很多人常常冒傻的地方，笔者也是 (*^__^*) 嘻嘻……）

笔者假设有一个流水灯的实验：

```
always @ ( posedge CLK )
    if( Count_US == T1US ) Count_US <= 5'd0;
    else Count_US <= Counter_US + 1'b1;

    .....

case( i )
    0, 1, 2, 3, 4, 5, 6, 7:
        if( Count_US == T1US ) begin LED <= ( 4'h01 << i ) ; i <= i + 1'b1; end
    .....
```

如果 T1US 的常量声明为 19，那么流水效果的时间间隔是 0.95us 而不是 1us。反之，如果 T1US 的常量的声明为 20，那么流水灯的时间间隔是 1us。

在实验二十一中，_1US 和 _3US 定时信号时由逻辑组合驱动的。它们比起定时器驱动的 _is1US 和 _is3US 早一个时钟。之所以如此，因为前者是“即时”结果，后者是“时间点”的结果。（如果不明白笔者在说什么，请好好复习“步骤和时钟”）。

接下来，我们来了解“组合逻辑驱动”和“计时器驱动”是如何影响模块的沟通。我们先假设一个流水灯的情况：

在某个试验中，假设 1us 的定时器建立在单个模块 counter.v，流水灯效果建立在另一个单个模块 led.v。

```
module counter
(
    .....
    output _1US,    // 由组合逻辑驱动
    output _is1US   // 有寄存器驱动
);
    parameter T1US = 5'd20; // 时钟频率 20Mhz
    .....
```

```
assign _1US = ( Count_US == T1US ) ? 1'b1 : 1'b0;
assign _is1US = is1US;

endmodule

module led
(
    .....
    input _1US,
    input _is1US
);

case( i )
    0, 1, 2, 3, 4, 5, 6, 7:
        if( _1US ) begin LED_A <= ( 4'h01 << i ) ; i <= i + 1'b1; end
    .....
case( j )
    0, 1, 2, 3, 4, 5, 6, 7:
        if( _is1US ) begin LED_B <= ( 4'h01 << j ) ; j <= j + 1'b1; end
    .....
endmodule
```

如果根据“模块之间沟通”的规则，由寄存器驱动的 1us 定时信号 _is1US，是满 1us 之后就产生定时信号，然而模块的沟通需要至少一个时钟，所以 LED_B 的流水灯效果，实际上是 1.00us + 0.05 us。反之，由组合逻辑驱动的 1us 定时信号 _1US，在满 1us 前一个时钟就产生定时信号了，然而模块的沟通至少需要一个时钟，所以 LED_A 的流水灯效果是 0.95us + 0.05us，亦即准 1us。

事实上在应用中，只有完美的家伙（偏激狂）才有“完美时钟的要求”。普通人少一个时钟还是多一个时钟，也不会有多大的问题。在这里，笔者只是要告诉读者有这样的情况存在而已。（傻帽是没有罪的！）

实验二十一结论：

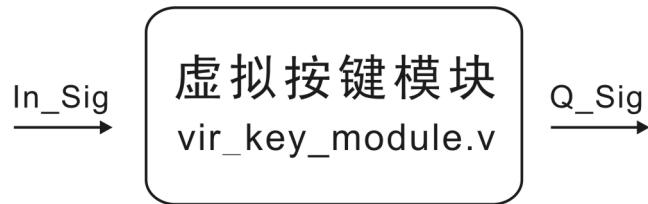
在实验二十一中，我们实现了近似“黑金开发板”虚拟环境，用于仿真模块 counter_module.v。虚拟环境在仿真中是一个非常重要的概念（笔者认为而已），因为我们所仿真的模块，最终要也要下载到某个“现实环境”中，所以虚拟环境越接近现实越好。在这里 counter_module.v 比较简单，笔者只是建立近似“黑金开发板”的“时钟信号”和“复位信号”而已。

此外，激励过程，几乎没有任何活动 呵呵！没办法啦，谁叫这个模块太简单了，但是这个实验的仿真中涉及了太多有关 Verilog HDL 语言的基础知识，尤其是相关“步骤和时钟”的部分。目前这个实验只是暖身而已，往后的仿真工作会更复杂。如果读者还没有掌握好基础，在还没有进入下一章之前，请好好补足基础。

6.2 刺激的各种输入

在前面笔者已经说过，某个模块在激励过程的时候，我们需要各种输入或称虚拟输入，作为激励过程中的刺激。它们分别是“简单输入”“复杂输入”和“问答输入”。接下来，我们会从两种试验中去了解不同的输入。

实验二十二之一：虚拟按键



不知道读者们是否还记得？按键消抖模块它是入门低级建模中最经典的一个实验。如今我们不是要讨论如何去建立这个模块，反之我们要仿真这个模块。在这里可能有读者会问：“按键去抖模块的功能不是用来过滤机械按键产生的抖动吗？在仿真中我们如何产生按键的抖动？”

对！这就是笔者要的问题。当我们要仿真按键去抖模块的时候，我们要建立一个“虚拟按键”，作为按键消抖模块的刺激。其中这个“虚拟按键”有“虚拟的抖动产生”。在这里，我们稍微思考一下，如何建立这个“虚拟按键”呢？笔者的想法很简单，当这个“虚拟”按键的输入产生变化的时候，它就会产生 8ms 的“假抖动”。

vir_key_module.v

```

1. module vir_key_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input In_Sig,
7.     output Q_Sig
8. );
9.
10.    ****
11.
12.    reg F1,F2;
13.

```

```
14.      always @ ( posedge CLK or negedge RSTn )
15.          if( !RSTn )
16.              begin
17.                  F1 <= 1'b1;
18.                  F2 <= 1'b1;
19.              end
20.          else
21.              begin
22.                  F1 <= In_Sig;
23.                  F2 <= F1;
24.              end
25.
26.      *****/
27.
28.      parameter T8MS = 18'd160000;
29.
30.      *****/
31.
32.      reg [17:0]Count_8MS;
33.      reg isCount;
34.
35.      always @ ( posedge CLK or negedge RSTn )
36.          if( !RSTn )
37.              Count_8MS <= 18'd0;
38.          else if( Count_8MS == T8MS && isCount )
39.              Count_8MS <= 18'd0;
40.          else if( isCount )
41.              Count_8MS <= Count_8MS + 1'b1;
42.          else if( !isCount )
43.              Count_8MS <= 18'd0;
44.
45.      *****/
46.
47.      reg [3:0]i;
48.      reg isBounce;
49.
50.      always @ ( posedge CLK or negedge RSTn )
51.          if( !RSTn )
52.              begin
53.                  i <= 4'd0;
54.                  isBounce <= 1'b1;
55.                  isCount <= 1'b0;
56.              end
57.          else
58.              case( i )
```

```
59.
60.          0:
61.          if( F1 != F2 ) i <= i + 1'b1;
62.
63.          1,2,3:
64.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b0; i <= 4'd8; end
65.          else begin isCount <= 1'b1; isBounce <= 1'b0; i <= i + 1'b1; end
66.
67.          4,5,6:
68.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b0; i <= 4'd8; end
69.          else begin isBounce <= 1'b1; i <= i + 1'b1; end
70.
71.          7:
72.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b0; i <= 4'd8; end
73.          else i <= 4'd1;
74.
75.          8:
76.          if( F1 != F2 ) i <= i + 1'b1;
77.
78.          9,10,11:
79.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b1; i <= 4'd0; end
80.          else begin isCount <= 1'b1; isBounce <= 1'b1; i <= i + 1'b1; end
81.
82.          12,13,14:
83.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b1; i <= 4'd0; end
84.          else begin isBounce <= 1'b0; i <= i + 1'b1; end
85.
86.          15:
87.          if( Count_8MS == T8MS ) begin isCount <= 1'b0; isBounce <= 1'b0; i <= 4'd0; end
88.          else i <= 4'd9;
89.
90.
91.          endcase
92.
93.      *****/
94.
95.      assign Q_Sig = isBounce;
96.
97.      *****/
98.
99. endmodule
```

第 6 行是该虚拟按键的触发（输入），第 7 行是该虚拟按键的输出，虚拟按键是低电平触发（低电平有效）。第 12~24 行是电平检测模块，用来检测 In_Sig 的变化。第 32~43

行是 8ms 的计数器，isCount 寄存器用来使能这个计数器（33 行）。在 50~91 行是该模块的核心功能，由于虚拟按键是低电平触发，它在空闲的时候 Q_Sig 总是保持高电平，然而 isBounce 寄存器用来驱动 Q_Sig，所以 isBounce 初始化为 1。

第步骤 0（60~61 行）用来检测 In_Sig 是否有变化。当 In_Sig 产变化就递增 i，以示下一个步骤。步骤 1~3 和步骤 4~6，是执行抖动的操作。在步骤 1~3，使能计数器（isCount 赋值 1）同时间拉低 isBounce（65 行）。步骤 4~6，是拉高 isBounce（69 行）。抖动产生的原理很简单，就是拉低 isBounce，3 个时钟，然后拉高 isBounce，3 个时钟。步骤 7 是指示 i 返回步骤 1（73 行）。在步骤 1~7 之间 if 条件用在判断 8ms 是否已经完成计数？如果“是”，那么不使能 isCount（isCount 赋值逻辑 0），然后拉低 isBounce，最后 i 指示步骤 8（64,68,72 行）。

换句话说，如果在步骤 1~7 之间，8ms 的计数还没有达到的话，步骤 1~7 就会一直重复，isBounce 会不停的拉低又拉高，从而产生出“假抖动”。

在步骤 8（75~76 行），同样也是用来检测 In_Sig 的电平的变化，如果检测到 In_Sig 的电平产生变化，就递增 i 以示下一步步骤。

步骤 9~11 和步骤 12~14，同样也是执行抖动的操作。在步骤 9~11，使能计数器（isCount 赋值 1）同时间拉高 isBounce（80 行）。步骤 12~14，是拉低 isBounce（84 行）。抖动产生的原理很简单，就是拉高 isBounce，3 个时钟，然后拉低 isBounce，3 个时钟。步骤 15 是指示 i 返回步骤 9（88 行）。在步骤 9~14 之间 if 条件是用来判断是否 8ms 已经完成计数，如果 8ms 已经达到，那么不使能 isCount（isCount 赋值逻辑 0），然后拉高 isBounce，最后 i 指示步骤 i 返回 0（79, 83, 87 行）。

也就是说，如果在步骤 9~14 之间，8ms 的计数还没有达到的话，步骤 9~14 就会一直重复，isBounce 会不停的拉高又拉低，从而产生出“假抖动”。

vir_key_module.vt

```

1. `timescale 1 ns/ 1 ps
2. module vir_key_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg In_Sig;
8.     wire Q_Sig;
9.
10.    /*****
11.    vir_key_module U1
12.    (
13.        .CLK(CLK),
14.        .

```

```
15.      .RSTn(RSTn),
16.      .In_Sig(In_Sig),
17.      .Q_Sig(Q_Sig)
18. );
19.
20. ****
21.
22. initial
23. begin
24.     RSTn = 0; #1000; RSTn = 1;
25.     CLK = 0; forever #25 CLK = ~CLK;
26. end
27.
28. ****
29.
30. parameter T1MS = 15'd20000;
31.
32. ****
33.
34. reg [14:0]Count1;
35. reg [9:0]Count_MS;
36. reg [9:0]rTimes;
37. reg isCount;
38.
39. always @ ( posedge CLK or negedge RSTn )
40.     if( !RSTn )
41.         begin
42.             Count1 <= 15'd0;
43.             Count_MS <= 10'd0;
44.         end
45.     else if( isCount && Count_MS == rTimes )
46.         begin
47.             Count1 <= 15'd0;
48.             Count_MS <= 10'd0;
49.         end
50.     else if( isCount && Count1 == T1MS )
51.         begin
52.             Count1 <= 15'd0;
53.             Count_MS <= Count_MS + 1'b1;
54.         end
55.     else if( isCount )
56.         begin
57.             Count1 <= Count1 + 1'b1;
58.         end
59.     else if( !isCount )
```

```

60.          begin
61.              Count1 <= 15'd0;
62.              Count_MS <= 10'd0;
63.          end
64.
65.      *****/
66.
67.      reg [3:0]i;
68.
69.      always @ ( posedge CLK or negedge RSTn )
70.          if( !RSTn )
71.              begin
72.                  In_Sig <= 1'b1;
73.                  isCount <= 1'b0;
74.                  rTimes <= 10'd0;
75.                  i <= 4'd0;
76.              end
77.          else
78.              case( i )
79.
80.                  0:
81.                      if( isCount && Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
82.                      else begin isCount <= 1'b1; rTimes <= 10'd1; end
83.
84.                  1:
85.                      begin In_Sig <= 1'b0; i <= i + 1'b1; end
86.
87.                  2:
88.                      if( isCount && Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
89.                      else begin isCount <= 1'b1; rTimes <= 10'd12; end
90.
91.                  3:
92.                      begin In_Sig <= 1'b1; i <= 4'd3; end
93.
94.              endcase
95.
96.      endmodule

```

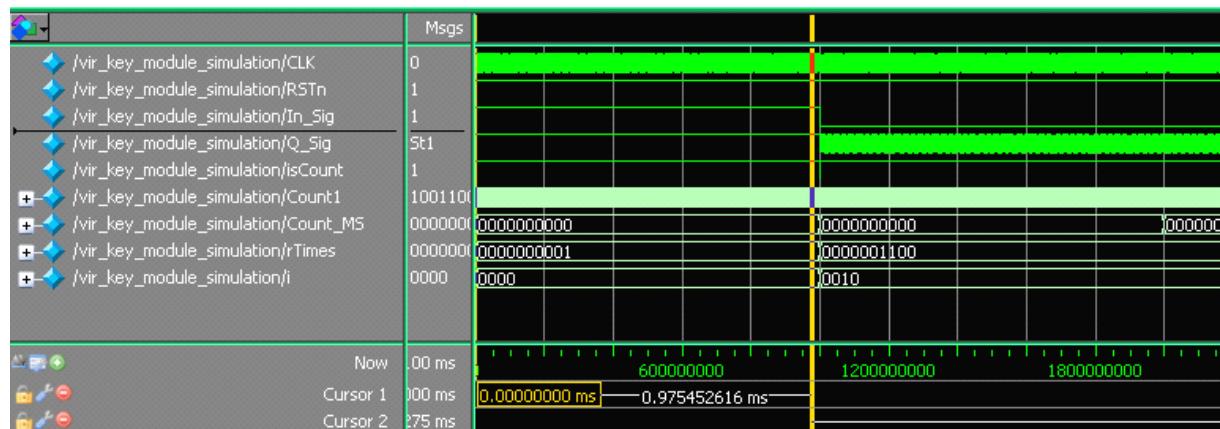
在 23~26 行是近似黑金开发板的时钟信号和复位信号。复位时间为 1us，时钟周期为 50ns。第 30 行是 1ms 的常量声明，39~63 行是 ms 级的计数器。rTimes 寄存器是用来寄存要计数的 ms 时间（36 行）。isCount 寄存器是用来使能该计数器（37 行）。第 64~94 行是 vir_key_module 的激励过程。

我们知道黑金开发板上的按键是低电平有效，然而在空闲的时候它总是处于高电平。所

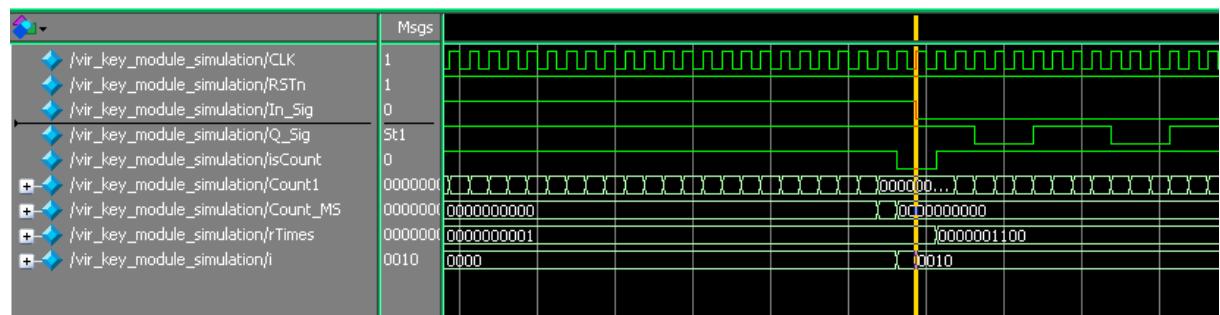
以 In_Sig 必须初始化为逻辑 1 (72 行)。在步骤 0 是 1ms 的延迟 (80~82 行)。步骤 1 是拉低 In_Sig (84~85 行)。步骤 2 是 12ms 的延迟 (87~89 行)。步骤 3 是拉高 In_Sig, 然后停止动作 (91~92 行)。

在激励过程中, 如果用“现实”来说话的话。我们先发呆 1ms, 然后按下按键 12ms, 当 12ms 过后释放按键。就是这么简单。

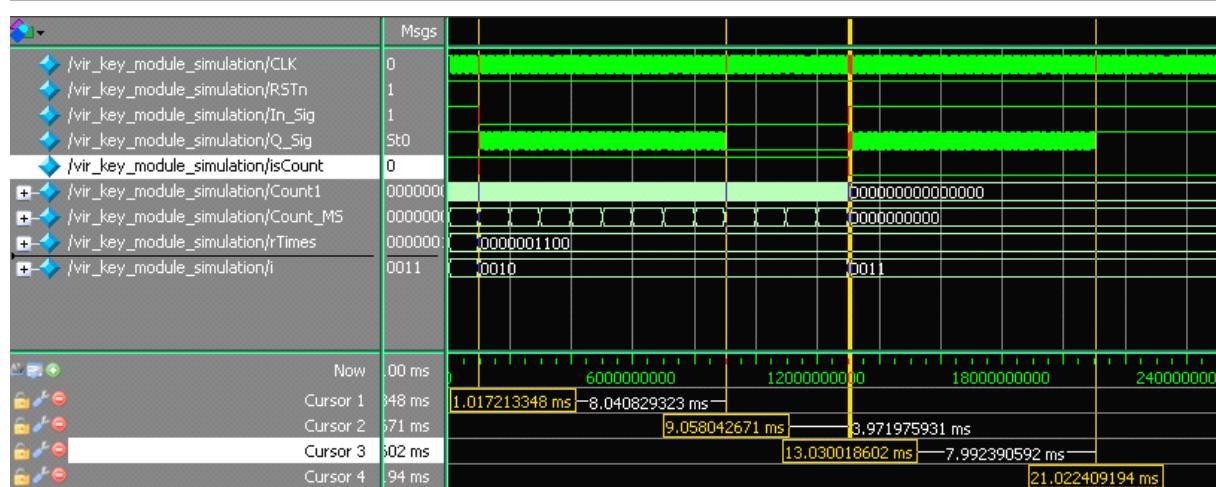
仿真结果:



第一张仿真图说明了在激励过程-步骤 0 中, 我们先发呆了 1ms。注意, 在这个时候 In_Sig 空闲是处于高电平 (Q_Sig 是低电平触发的关系)。



第二张仿真结果说明了, 当我们按下“虚拟按键”(该事件发生在激励过程-步骤 1 中)。注意, In_Sig 被拉低过后不久, Q_Sig 开始不稳定了(抖动开始产生了)。

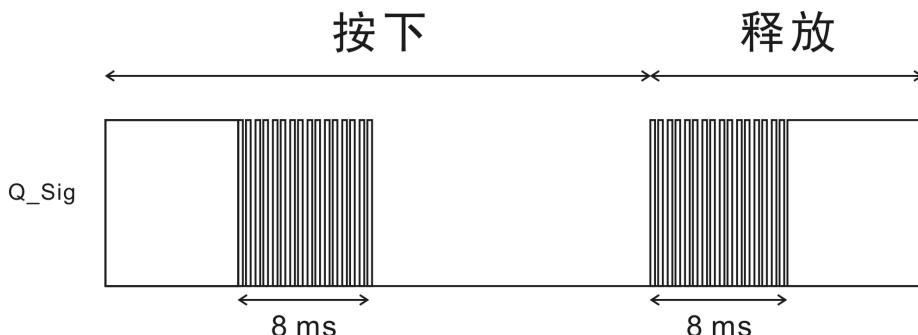


第三张仿真图说明了，在激励过程中-步骤 0~3 发生的事情。我们先发呆 1ms，然后按下“虚拟按键”12ms（步骤 2），然后再释放“虚拟按键”（步骤 3），之后动作就结束了。在仿真结果中，刚开始 In_Sig 和 Q_Sig 都是处于高电平。当“虚拟按键”被按下的时候，抖动开始产生了，抖动的过程大约是 8ms，然后接下来是 4ms 的低电平。当 12ms 过后，我们释放“虚拟按键”，8ms 的抖动又开始产生了。之后 Q_Sig 是处于平静的高电平。

上述的结果告诉我们一个事实，我们在“现实中”先发呆 1ms，然后按下“虚拟按键”12ms，12ms 过后释放“虚拟按键”。所以在仿真结果中（第三张图），才会出现如此的时序图。

按下 12ms 得“虚拟按键”先产生 8ms 的抖动，然后是 4ms 平静的低电平（这是因为我们按下“虚拟按键”12ms 的关系，12ms 减去 8ms 等于 4ms）。当我们释放“虚拟按键”的时候，又产生 8ms 的抖动，然后是永远的高电平（当我们释放“虚拟按键”之后，就结束动作了）。

实验二十二之一说明：



现实的独立按键会产生的抖动大约是 5~10ms，在实验二十二之一中我们建立了近似的虚拟按键的 vir_key_module.v（如上图）。vir_key_module.v 的抖动时间是 8ms，然而抖动频率是：

(50 + 50 + 50 + 50 + 50 + 50) ns = 300 ns （拉高 3 个时钟，拉低 3 个时钟）

$$F = 1 / T$$

$$= 1 / 300\text{ns}$$

$$= 3.33\text{Mhz}$$

（哎！现实中机械按键的抖动频率不肯能那么高 ... 这不过这是“虚拟按键”而已嘛~ 不要太介意，不要太介意，最重要是可以产生一定时间的抖动效果。）

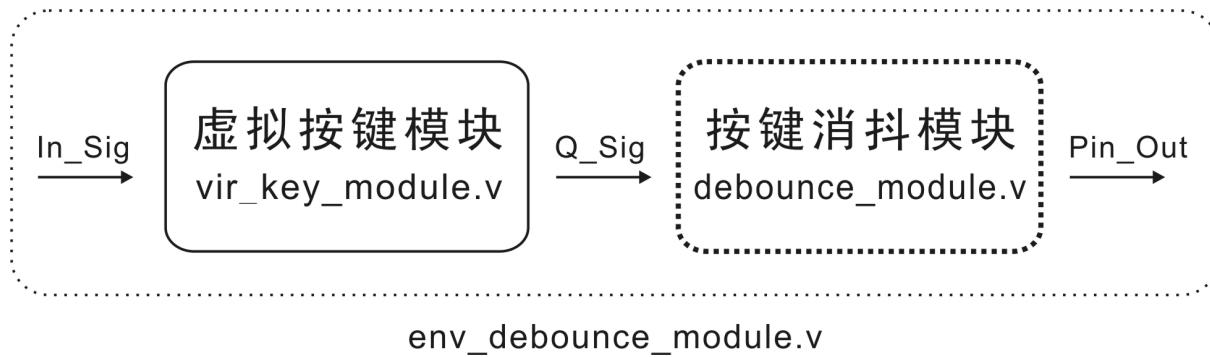
在激励过程中，我们在“假现实”先发呆 1ms，然后再按下“虚拟按键” 12ms，12ms 过后释放“虚拟按键”，继续发呆。所以在仿真结果中出现，Q_Sig 先拉高大约 1ms，然后产生抖动大约 8ms，再然后低电平 4ms，过后再产生抖动 8ms，最后保持高电平到永远。

实验二十二之一结论：

在这一个实验中 vir_key_module.vt 是 vir_key_module.v 的激励文件（激励过程）。其中在 .vt 的 In_Sig 是 .v 文件的简单输入。但是在未来里 vir_key_module.v 将会是 debounce_module.v 的复杂输入。

实验二十二之二：仿真按键消抖模块

这个实验我们要仿真《Verilog HDL 那些事儿-建模篇》中，实验三的按键消抖模块。



上图显示了，虚拟按键模块（实验二十二之一）和按键消抖模块组合在“虚拟环境”`env_debounce_module.v` 中。在这里，`env_debounce_module.v` 虽然是组合模块，但是它已经不是低级建模中定义的组合模块了，它在这里仅是充当按键消抖模块的仿真环境。

我们稍微回忆一下按键消抖模块的功能：

当按键被按下的时候，过滤抖动 10ms 然后拉高 `Pin_Out`。当按键释放的时候过滤抖动 10ms 然后拉低 `Pin_Out`。

env_debounce_module.v

```
1. module env_debounce_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input In_Sig,
7.     output Pin_Out,
8.
9.     output SQ_Q_Sig
10. );
11.
12.     ****
13.
14.     wire Q_Sig;
15.
16.     vir_key_module U1
```

```

17.      (
18.          .CLK( CLK ),
19.          .RSTn( RSTn ),
20.          .In_Sig( In_Sig ),
21.          .Q_Sig( Q_Sig )
22.      );
23.
24.      *****/
25.
26.      debounce_module U2
27.      (
28.          .CLK( CLK ),
29.          .RSTn( RSTn ),
30.          .Pin_In( Q_Sig ),
31.          .Pin_Out( Pin_Out )
32.      );
33.
34.      *****/
35.
36.      assign SQ_Q_Sig = Q_Sig;
37.
38.      *****/
39.
40.
41. endmodule

```

在 16~22 行实例化了虚拟按键模块。在 26~32 行实例化了按键消抖模块。在 36 行将虚拟按键模块的输出引出来。（笔者再强调一下，在这里的组合模块 env_debounce_module.v 再也不是低级建模定义中的组合模块，而是一个仿真用的虚拟环境。）

env_debounce_module.vt

```

1.  `timescale 1 ns/ 1 ps
2.  module env_debounce_module_simulation();
3.
4.      reg CLK;
5.      reg RSTn;
6.
7.      reg In_Sig;
8.      wire Pin_Out;
9.
10.     wire SQ_Q_Sig;
11.

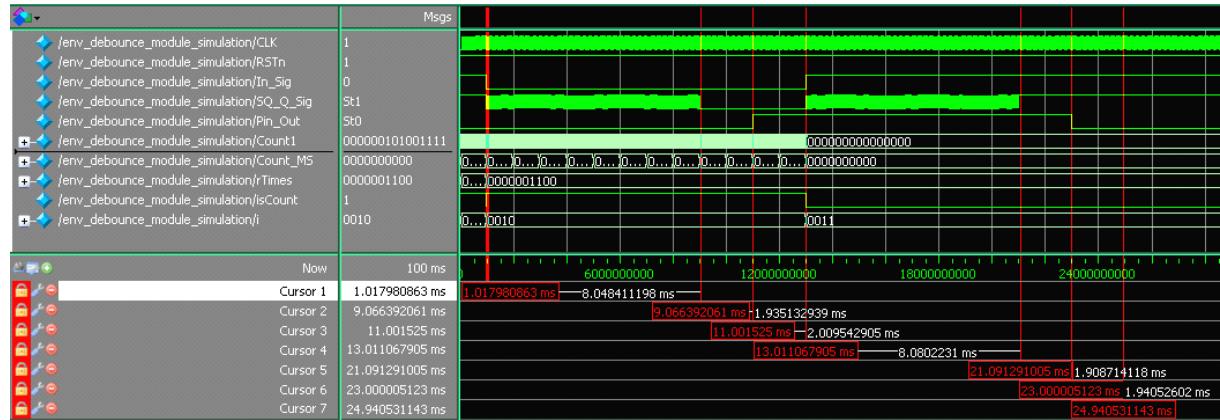
```

```
12.      *****/
13.
14.      env_debounce_module U1
15.      (
16.          .CLK( CLK ),
17.          .RSTn( RSTn ),
18.          .In_Sig( In_Sig ),
19.          .Pin_Out( Pin_Out ),
20.          .SQ_Q_Sig( SQ_Q_Sig )
21.      );
22.
23.      *****/
24.
25.      initial
26.      begin
27.          RSTn = 0; #1000; RSTn = 1;
28.          CLK = 0; forever #25 CLK = ~CLK;
29.      end
30.
31.      *****/
32.
33.      parameter T1MS = 15'd20000;
34.
35.      *****/
36.
37.      reg [14:0]Count1;
38.      reg [9:0]Count_MS;
39.      reg [9:0]rTimes;
40.      reg isCount;
41.
42.      always @ ( posedge CLK or negedge RSTn )
43.          if( !RSTn )
44.              begin
45.                  Count1 <= 15'd0;
46.                  Count_MS <= 10'd0;
47.              end
48.          else if( isCount && Count_MS == rTimes )
49.              begin
50.                  Count1 <= 15'd0;
51.                  Count_MS <= 10'd0;
52.              end
53.          else if( isCount && Count1 == T1MS )
54.              begin
55.                  Count1 <= 15'd0;
56.                  Count_MS <= Count_MS + 1'b1;
```

```
57.         end
58.     else if( isCount )
59.         begin
60.             Count1 <= Count1 + 1'b1;
61.         end
62.     else if( !isCount )
63.         begin
64.             Count1 <= 15'd0;
65.             Count_MS <= 10'd0;
66.         end
67.
68.     /***** */
69.
70.     reg [3:0]i;
71.
72.     always @ ( posedge CLK or negedge RSTn )
73.         if( !RSTn )
74.             begin
75.                 In_Sig <= 1'b1;
76.                 isCount <= 1'b0;
77.                 rTimes <= 10'd0;
78.                 i <= 4'd0;
79.             end
80.         else
81.             case( i )
82.
83.                 0:
84.                     if( isCount && Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
85.                     else begin isCount <= 1'b1; rTimes <= 10'd1; end
86.
87.                 1:
88.                     begin In_Sig <= 1'b0; i <= i + 1'b1; end
89.
90.                 2:
91.                     if( isCount && Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
92.                     else begin isCount <= 1'b1; rTimes <= 10'd12; end
93.
94.                 3:
95.                     begin In_Sig <= 1'b1; i <= 4'd3; end
96.
97.             endcase
98.
99. endmodule
```

在 14~21 行实例化了要仿真的虚拟环境。在 25~29 建立了近似“黑金开发板”的时钟信号和复位信号。第 72~97 行是激励过程，激励过程模仿了人在现实中“按下和释放按键”的情况（具体的过程和上一个实验一模一样）。

仿真结果：

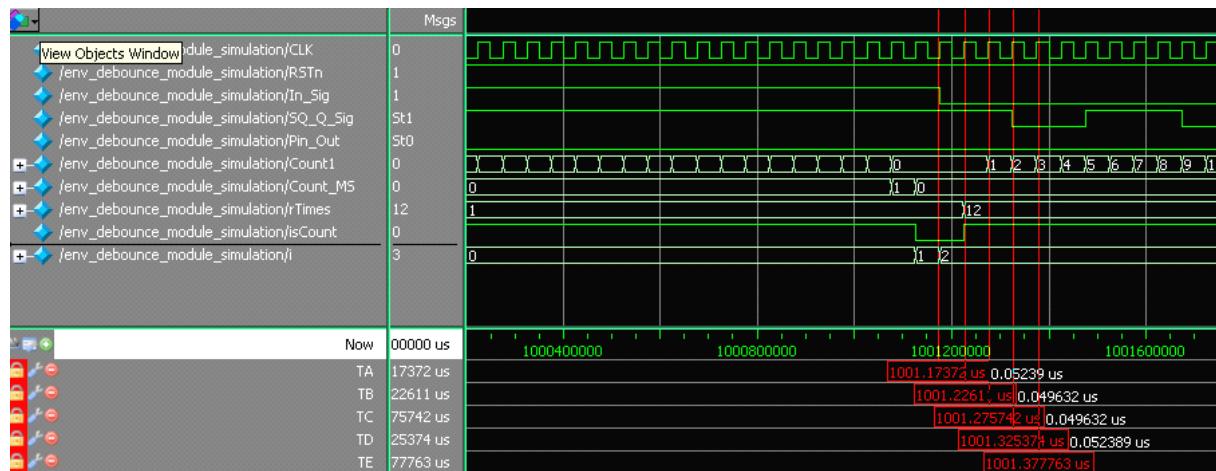


上图仿真结果显示了，按键消抖模块受到虚拟按键模块的刺激后所产生的时序图。
(Cursor 简略为 C) C1~C2，表示了当“虚拟按键”按下之后的情况，“虚拟按键”产生 8ms 的抖动 (SQ_Q_Sig)。当“虚拟按键”按下之际，按键消抖模块检测到 Q_Sig 由高变低，那么按键消抖模块开始过滤抖动 10ms，C1~C3。

我们知道“虚拟按键”只是产生 8ms 的抖动而已，然而按键消抖模块过滤抖动的时间是 10ms，所以在 C2~C3 表示了按键消抖模块余下的过滤时间。当按键消抖模块消抖时候就会拉高输出，亦即 Key_Out 输出高电平 (C3 之后)。

在 C4 的时候，“虚拟按键”被释放了，然后“虚拟按键”再一次产生 8ms 的抖动时间 (SQ_Q_Sig)，亦即 C4~C5。当“虚拟按键”释放之际，按键消抖模块也检测到了 Q_Sig 由低变高，它也开始过滤抖动 10ms 的工作 (C4~C6)。按键消抖模块余下的抖动时间是在 C5~C6。

当按键消抖模块过滤抖动 10ms 之后，就拉低 Key_Out (C6 之后)。

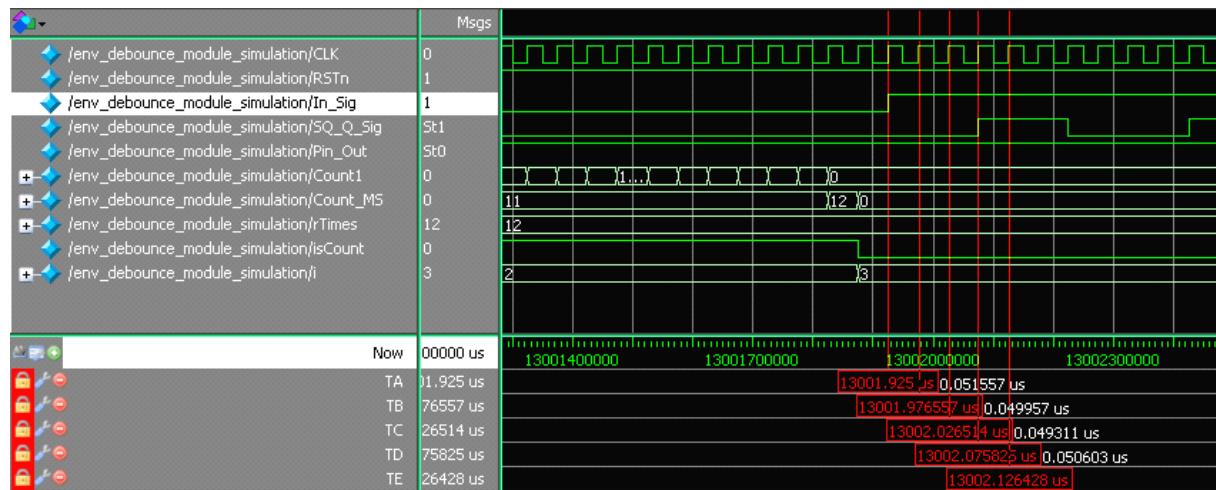


上面的仿真结果“虚拟按键”按下经过的时序图。(TA~TE“虚拟按键”按下的时候),在 TA 的时候,也就是 .vt 在步骤 1 的时候,它“决定”拉低 In_Sig , 亦即按下“虚拟按键”。在 TA 的未来, In_Sig 拉低。

在 TB 的时候, 虚拟按键模块它检查 In_Sig 的变化。虚拟按键模块的内部分别建立 F1 和 F2 寄存器用于检查电平的变化。在 TA 之际 F1 读取 In_Sig 的过去值, 亦即逻辑 1。在 TB 之际, F1 读取 In_Sig 的过去值, 亦即逻辑 0, 然而 F2 读取 F1 的过去值, 亦即逻辑 1。所以在 TB 的时候, 虚拟按键模块检查到 In_Sig 的电平产生变化, 它“决定”产生抖动。所以 Q_Sig 的抖动时发生在 TD 之后。

(这其中, 涉及许多“步骤和时钟”和建模的知识, 笔者就不详谈了, 如果读者不明白笔者在说什么的话, 请好好复习“步骤和时钟”和“建模篇”。)

同样的道理, 按键消抖模块也有电平检测模块。在 TD 的时候按键消抖模块的 F1 读取到 Q_Sig (SQ_Q_Sig) 的过去值, 亦即逻辑 0。在 TE 的时候, F1 读取到 (SQ_Q_Sig) 的过去值, 亦即逻辑 1, F2 读取 F1 的值, 亦即逻辑 0。按键消抖模块检测到虚拟按键的输出产生变化, 便开始执行消抖 10ms 的工作。



在上图的仿真结果是“虚拟按键”被释放的时候所产生的时序图。在 TA 的时候 .vt 在步骤 3，它决定拉高 In_Sig，亦即释放“虚拟按键”。所以在 TA 的未来，In_Sig 拉高电平。

在 TB 的时候，虚拟按键模块检测到 In_Sig 的电平变化。在 TA 的时候，虚拟按键模块的 F1 读取到 In_Sig 的过去值是逻辑 0。然后在 TB 的时候，F1 读取到 In_Sig 的过去值是逻辑 1，F2 读取 F1 亦即是逻辑 0。在同一个时候，虚拟按键模块检测到 In_Sig 的电平产生变化，所以在 TD 的未来，它决定产生 8ms 的抖动。

在 TD 的时候，按键消抖模块的 F1 读取到虚拟按键模块 Q_Sig (SQ_Q_Sig) 的过去值，是逻辑 0。然后在 TE 的时候，由于虚拟按键开始产生抖动 (Q_Sig 产生变化)，F1 读取到 Q_Sig (SQ_Q_Sig) 的过去值是逻辑 1，F2 读取 F1 是逻辑 0，按键消抖模块检查到 Q_Sig (SQ_Q_Sig) 产生变化，便“决定”执行“过滤消抖 10ms”。所以在 TE 之后的（大约 2~3 个时钟），按键消抖模块就开始执行消抖工作。

实验二十二之二说明：

在这个实验的仿真中，In_Sig 充当虚拟按键模块的“简单输入”，虚拟按键模块的输出 Q_Sig 充当按键消抖模块的复杂输入。为什么 Q_Sig 信号称为复杂输入呢？虚拟按键模块所产生的假抖动块是为了刺激按键消抖模块，然而这个“假抖动”是模拟“现实按键”的输出带抖动，故才称为复杂输入。

实验二十二之二结论：

在这个实验的仿真中，笔者建立了“虚拟按键”充当按键消抖模块的“虚拟输入”或者“复杂输入”。

在仿真的虚拟环境里，笔者使用了建模的方法，把虚拟按键模块和按键消抖模块组合在同一个环境里。然后笔者使用了综合的办法编辑激励过程。在这里，笔者没有涉及任何和“验证”有关的语法（时钟信号和复位信号除外）全部的激励过程都是综合语言一手包办。

读者可能经过这个实验之后，对仿真的认识会更上一层楼。此外，笔者还有一点需要强调的是 … 在很久以前（初学 Verilog HDL 的时候），笔者以为仿真只是观察波形而已。但是日子久了，技术越来越成熟了，笔者明白到仿真不只是单纯的观察输出波形，编辑“刺激”也是重要的工作。类似实验二十二之二的情况，要刺激按键消抖模块的话，如果没有“虚拟按键模块”笔者也不知道要仿真什么鸟儿？

在此，笔者很建议读者不要小瞧仿真，它绝对不是网上所说的那样那样单纯。它甚至比 Verilog HDL 的建模来得更深不可测。如果读者没有补足好基础的话，在仿真的路上，读者会常常跌倒的。

实验二十三：PS2 模块仿真

在上一个试验中，我们建立了一个虚拟按键模块来充当按键消抖模块的虚拟输入。这一章，我们做同样的东西，但是我们不是再建立另一个“虚拟输入”的模块，而是直接在激励文件上建立“虚拟输入”。

在这里，我们稍微回顾一下 PS2 模块的大致功能：

PS2 数据一帧有 11 位，PS2_Data 数据读取都是在 PS2_CLK 的下降沿有效。PS2 模块主要是由电平检测模块和 PS2 解码模块组合而成，大致功能有：前者用来检测 PS2 时钟的下降沿；后者用来过滤处理一帧 11 位的数据。PS2 解码模块是通码和断码的 0xf0 通吃，但是不吃断码之后的通码。当完成一帧 11 位的数据解码以后，就产生一个完成信号。

ps2_module.vt

```
1. `timescale 1 ns/ 1 ps
2. module ps2_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg PS2_CLK_Pin_In;
8.     reg PS2_Data_Pin_In;
9.
10.    wire [7:0]PS2_Data;
11.    wire PS2_Done_Sig;
12.
13.    ****
14.
15.    ps2_module U1
16.    (
17.        .CLK(CLK),
18.        .RSTn(RSTn),
19.        .PS2_CLK_Pin_In(PS2_CLK_Pin_In),
20.        .PS2_Data_Pin_In(PS2_Data_Pin_In),
21.        .PS2_Data(PS2_Data),
22.        .PS2_Done_Sig(PS2_Done_Sig)
23.    );
24.
25.    ****
26.
```

```
27.      initial
28.      begin
29.          RSTn = 0; #1000; RSTn = 1;
30.          CLK = 0; forever #25 CLK = ~CLK;
31.      end
32.
33.      /*****
34.
35.      parameter T50US = 10'd1000;
36.
37.      *****/
38.
39.      reg [9:0]Count1;
40.      reg isCount;
41.
42.      always @ ( posedge CLK or negedge RSTn )
43.          if( !RSTn )
44.              Count1 <= 10'd0;
45.          else if( isCount && Count1 == T50US )
46.              Count1 <= 10'd0;
47.          else if( isCount )
48.              Count1 <= Count1 + 1'b1;
49.          else if( !isCount )
50.              Count1 <= 10'd0;
51.
52.      *****/
53.
54.      reg [5:0]i;
55.      reg [5:0]Go;
56.      reg [10:0]rData;
57.
58.      always @ ( posedge CLK or negedge RSTn )
59.          if( !RSTn )
60.              begin
61.                  PS2_CLK_Pin_In <= 1'b1;
62.                  PS2_Data_Pin_In <= 1'b0;
63.                  i <= 6'd23;
64.                  Go <= 6'd0;
65.                  rData <= 11'd0;
66.              end
67.          else
68.              case( i )
69.
70.                  // Step(i) 0~21 is PS2 Send Function
71.
```

```

72.          0,2,4,6,8,10,12,14,16,18,20;
73.          if( Count1 == T50US ) begin isCount <= 1'b0; i <= i + 1'b1; end
74.          else begin isCount <= 1'b1; PS2_CLK_Pin_In <= 1'b1; PS2_Data_Pin_In <= rData[ i >> 1 ]; end
75.
76.          1,3,5,7,9,11,13,15,17,19,21;
77.          if( Count1 == T50US ) begin isCount <= 1'b0; i <= i + 1'b1; end
78.          else begin isCount <= 1'b1; PS2_CLK_Pin_In <= 1'b0; end
79.
80.          22: // Return to next action
81.          begin PS2_CLK_Pin_In <= 1'b1; i <= Go; end
82.
83.          23:
84.          begin   rData <= { 2'b11, 8'h1C, 1'b0 }; Go <= i + 1'b1; i <= 6'd0; end
85.
86.          24:
87.          begin   rData <= { 2'b11, 8'hF0, 1'b0 }; Go <= i + 1'b1; i <= 6'd0; end
88.
89.          25:
90.          begin   rData <= { 2'b11, 8'h1C, 1'b0 }; Go <= i + 1'b1; i <= 6'd0; end
91.
92.          26:
93.          i <= 6'd26;
94.
95.          endcase
96.
97.
98. endmodule

```

上面激励文件的激励过程完全是模仿现实中“按键-A 被按下后，然后被释放的全过程”。第 15~23 行实例化了要仿真的 PS2 模块。第 27~31 行建立了近似“黑金开发板”的环境，亦即时钟信号和复位信号，复位信号拉低 1us，然而时钟的周期是 50ns。第 35 行，定义了 50us 的常量。第 39~50 行是 50us 的定时器。

我们知道拥有 PS2 接口的设备都是慢速设备，而且 PS2 的时钟频率大约是 10KHz，亦即一个时钟周期为 100us，半个时钟周期为 50us。第 35 行，声明了 50us 的常量（以 20Mhz 为准）。第 54~95 行是模仿按键 A 被按下后又被释放的情况。在 55 行声明了寄存器 Go 是用来返回步骤（仿函数需要用到）。第 56 行声明了寄存器 rData，是用来暂存 1 帧 11 位的数据。（[0]开始位，[1~8]数据位，[9]校验位，[10]停止位。）

在步骤 0~22 行是发送 PS2 数据的仿函数。步骤 0~21（偶数-72~74 行），是拉高 PS2 时钟和设置（更新）数据数据的操作。步骤 0~21（奇数-76~78 行），是拉低 PS2 时钟（用从机干锁存|读取数据）。步骤 0~21 的每一个操作，都需要等待 50us 的延迟。步骤 22 是恢复 PS2 时钟为高电平然后返回操作 Go 指向的步骤（80~81 行）。（注：PS2 的传输是从低位开始，注意在 74 行 rData 的位寻址。）

在这里稍微注意一下，在 63 行 i 初始化为 23，这也是步骤 i 不是从 0 开始而是从 23 开始。在步骤 23 (83~84 行)，rData 寄存 { 2'b11 , 8'h1C, 1'b0 }，亦即按键 A 的通码，然后寄存器 Go 赋值于 i+1 (表示下一个步骤)；然后 i 赋予步骤 0，以示进入 PS2 发送数据仿函数。（[0]开始位-逻辑 0，[9]校验位-如果没有特别需求可以随便，笔者就填入逻辑 1，[10]停止位-逻辑 1。）

在这里先假设一个情况：

当进入步骤 23，rData 寄存{ 2'b11 , 8'h1C, 1'b0 }，Go 寄存 i + 1，亦即 24，然后 i 指向仿函数，亦即步骤 0。在步骤 0，rData 会从低至高被发送出去。

发送过程大致如下：

位	10	9	8	7	6	5	4	3	2	1	0
.vt 文件 - 激励过程	1	1	0	0	0	1	1	1	0	0	0
.v 文件 - PS2 解码	无视	无视	0	0	0	1	1	1	0	0	无视

PS2 模块吃通码

当发完 1 帧 11 位的数据后，会进入步骤 22。步骤 22 会恢复 PS2 的时钟成为高电平（PS2 时钟在空闲的时候总是处于高电平）。

步骤 24~25 是模仿按键 A 的释放的操作，亦即先发送 8'hF0，然后再发送 8'h1C。

位	10	9	8	7	6	5	4	3	2	1	0
.vt 文件 - 激励过程	1	1	1	1	1	1	0	0	0	0	0
.v 文件 - PS2 解码	无视	无视	1	1	1	1	0	0	0	0	无视

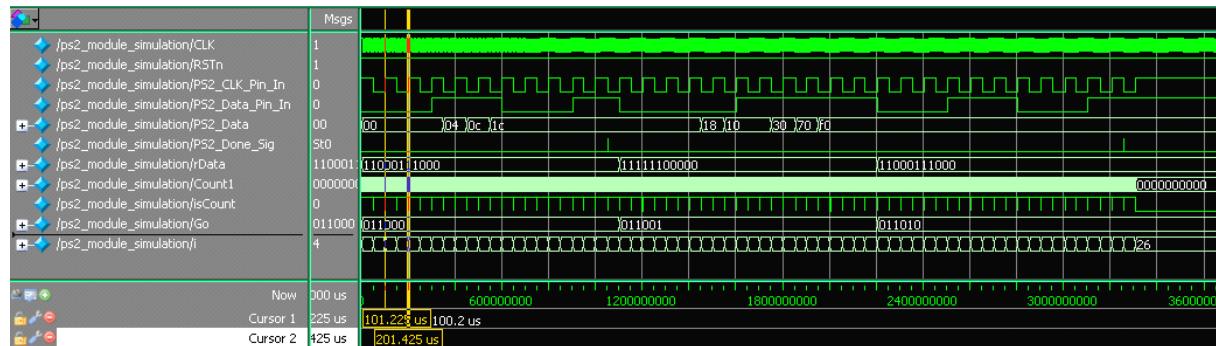
PS2 模块吃断码 0xf0

位	10	9	8	7	6	5	4	3	2	1	0
.vt 文件 - 激励过程	1	1	0	0	0	1	1	1	0	0	0
.v 文件 - PS2 解码	无视										

PS2 模块不吃断码以后的通码

步骤 26 是停止操作。

仿真结果:



上图的 C1~C2 表示了 PS2 的时钟周期，亦即 10kHz 的时钟频率。读者应该了解笔者设计的 PS2 模块，通码和断码的 0xf0 都是通吃，但是不吃断码以后的通码。在上图仿真结果中，当“虚拟按键 A”被按下的时候，亦即在 .vt 的步骤 23，（注意 PS2_Done_Sig 的第一个高脉冲之前）数据 { 2'b11, 8'h1C, 1'b0} 开始被 PS2 模块解码，最后数据 8'h1C 被保留并输出（PS2_Data），然后产生一个完成信号。

当“虚拟按键 A”被释放的时候，亦即在 .vt 的步骤 24~25。.vt 先发送数据 { 2'b11, 8'hF0, 1'b0 }，它成功被 PS2 模块解码，8'hF0 保留并被输出（PS2_Data）。然后数据 { 2'b11, 8'h1C, 1'b0 } 会被.vt 发送，但是这一段数据是断码之后的通码，PS2 模块无视，并且产生一个完成信号。所以在 PS2_Data 的输出上，依然是 8'hF0。

实验二十三说明:

在实验二十三中，仿真工作不像实验二十二那样，建立一个虚拟按键模块，然后在仿真的虚拟环境 env_debounce_module.v 执行仿真。然而这个实验的仿真，刺激是直接在激励文件编辑“模仿按键-A 按下后又释放”。将“简单输入”有步骤的编辑成为“复杂输入”。

实验二十三结论:

实验二十二和实验二十三相比，一个是将复杂输入建立在一个模块里（虚拟按键），一个是在激励过程中有多个不同的步骤的简单输入，建立起复杂输入。有很多时候有一些模块会有正负的关系，亦即一方为发送，另一方为接受。在仿真工作中，一方可以直接刺激另一方，从而免去重新编辑“虚拟输入”的麻烦。

6.3 模块相互刺激

在众多的模块中，有时候后会出现正负的关系，亦即模块的一方是发送，而模块的另一方是接收，又或者有从机与主机的关系。其中最为亮相的就是串口模块。我们知道串口模块是以一对的形式出现，亦即串口发送模块，和串口接收模块。在激励过程中串口发送模块由简单输入刺激，然而串口接收模块由串口发送模块的输出刺激，即达到复杂输入的效果。

在这里，笔者就借用《Verilog HDL 那些事儿-建模篇》中实验十的串口发送|接收模块吧。

实验二十四之一：仿真串口发送模块

我们稍微回忆一下串口发送模块，大致的功能：



上图是串口发送模块的图形。在允许不了解内部结构的情况下，我们只要准备数据在 TX_Data，然后拉高 TX_En_Sig。直到 TX_Done_Sig 产生完成信号之前，数据会以串口传输的模式，一帧 11 位发送。（注：这个模块是 9600kbps 的波特率。）

tx_module.vt

```

1. `timescale 1 ns/ 1 ps
2. module tx_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg TX_En_Sig;
8.     reg [7:0]TX_Data;
9.
10.    wire TX_Done_Sig;
11.    wire TX_Pin_Out;
12.
13.    /*****

```

```

14.
15.      tx_module U1
16.      (
17.          .CLK(CLK),
18.          .RSTn(RSTn),
19.          .TX_Data(TX_Data),
20.          .TX_Done_Sig(TX_Done_Sig),
21.          .TX_En_Sig(TX_En_Sig),
22.          .TX_Pin_Out(TX_Pin_Out)
23.      );
24.
25.  *****/
26.
27. initial
28. begin
29.     RSTn = 0; #1000; RSTn = 1;
30.     CLK = 0; forever #25 CLK = ~CLK;
31. end
32.
33. *****/
34.
35. reg [3:0]i;
36.
37. always @ ( posedge CLK or negedge RSTn )
38.     if( !RSTn )
39.         begin
40.             TX_En_Sig <= 1'b0;
41.             i <= 4'd0;
42.             TX_Data <= 8'd0;
43.         end
44.     else
45.         case( i )
46.
47.             0:
48.                 if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
49.                 else begin TX_En_Sig <= 1'b1; TX_Data <= 8'h2E; end
50.
51.             1:
52.                 if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
53.                 else begin TX_En_Sig <= 1'b1; TX_Data <= 8'h3f; end
54.
55.             2:
56.                 if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end
57.                 else begin TX_En_Sig <= 1'b1; TX_Data <= 8'hdd; end

```

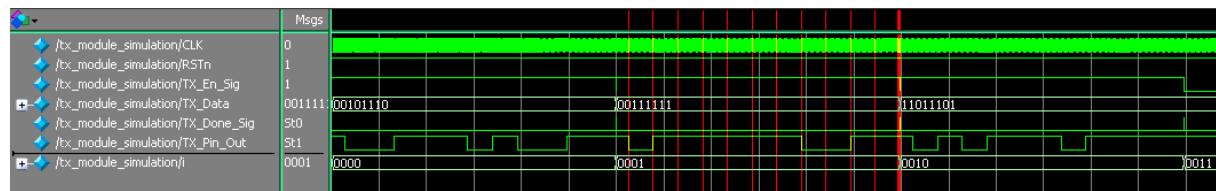
```

58.
59.          3:
60.          begin i <= 4'd3; end
61.
62.
63.      endcase
64.
65.      /*****
66.
67.
68. endmodule

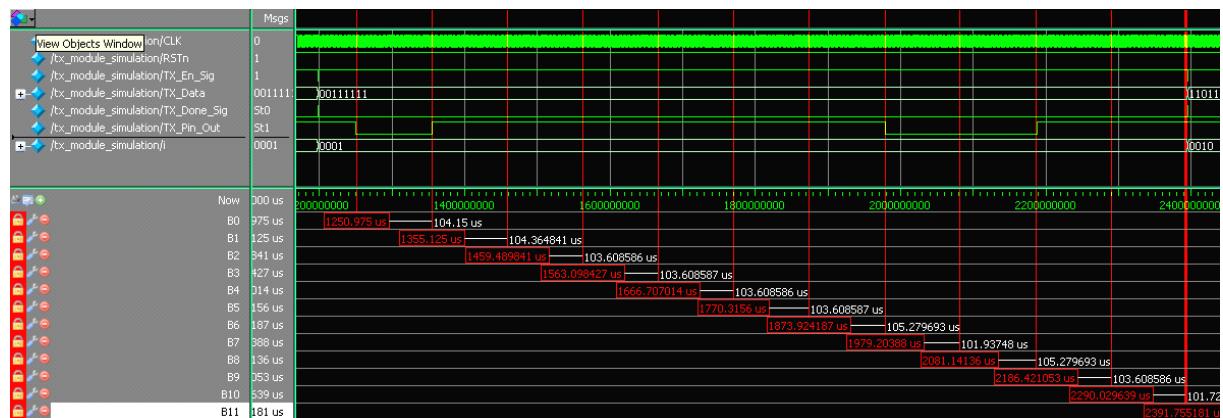
```

第 37~61 行是激励过程，其中问答输入甚多。在步骤 0 的时候（47~49 行），将数据 8'h2E 发送至串口发送模块后使能串口发送模块（49 行），然后等待串口发送模块反馈完成信号（48 行）。同样的动作也有 ... 步骤 1（51~53 行）是发送数据 8'h3f，然而步骤 2（55~57 行）是发送数据 8'hdd。步骤 3（59~60 行）是停止动作。

仿真结果：



上图是发送 3 组不同数据的仿真结果。在第一个完成信号产生之后，串口发送模块已经输出 0x2e。在第二个完成信号的产生之后，串口发送模块已经输出 0x3f。在第三个完成信号的产生之后，串口发送模块已经输出 0xdd。



再来我们放大第二个数据 0x3f，发送的过程。我们知道串口发送模块，发送数据的格式是一帧 11 位。[0]开始位-逻辑 0，[1:8]数据位，[9]校验位- 没有需要可以随便填，这里笔者填逻辑 1，[10]停止位-逻辑 1。

位	10	9	8	7	6	5	4	3	2	1	0
.vt 文件 - 激励过程			0	0	1	1	1	1	1	1	
位	0	1	2	3	4	5	6	7	8	9	10
.v 文件 - 串口发送	0	1	1	1	1	1	1	0	0	1	1

串口发送模块发送数据 0x3f 的过程大致如上。我们又知道串口发送模块配置的波特率是 9600kbps，所以一个数据逗留的时间是 大约 104us。在仿真结果中，在 B0~B1 是数据[0]，B1~B2 是数据[1]…… B10~B11 是数据[10]。Bx~Bx 之间的时间大约是 104us。

实验二十四之一说明:

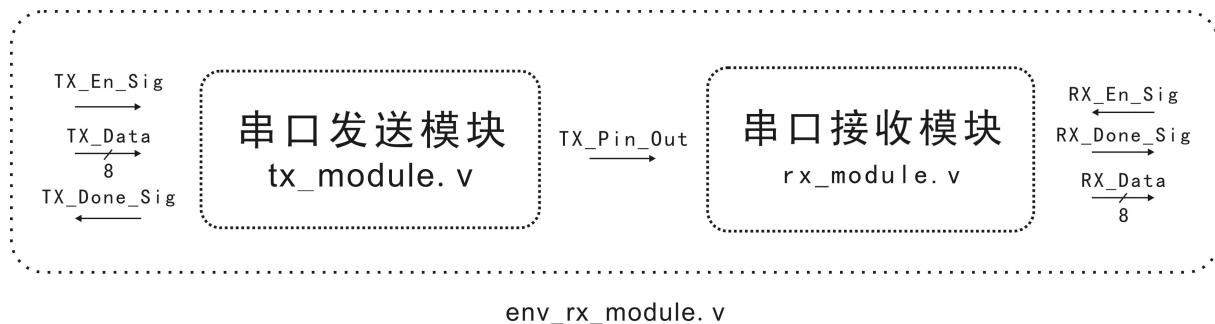
实验二十四之一的仿真，串口发送模块在激励过程中，以问答的方式刺激该模块。

实验二十四之一结论:

该实验比较简单，我们只是要观察串口发送模块的输出而已。在激励过程中，用问答输入刺激串口发送模块。

实验二十四之二：仿真串口接收模块

实验二十四之一我们仿真了串口发送模块，感觉 okay 后，现在我们要用串口发送模块作为串口接收模块的刺激，亦即串口接收模块的输入（复杂输入）。



上图是仿真虚拟环境 `env_rx_module.v` 它组合了串口发送模块和串口接收模块。`env_rx_module.v` 拥有 `TX_En_Sig`, `TX_Data`, `TX_Done_Sig`, `RX_En_Sig`, `RX_Done_Sig`, `RX_Data` 等信号。在激励的过程中，我们需要对这些信号控制。

在这里我们稍微回忆一下串口接收模块的功能：

当 `RX_En_Sig` 拉高的时候，串口接收模块开始准备接收数据了。当一帧 11 位数据发送至串口接收模块，并且被串口接收模块过滤。最后经过过滤的数据会输出至 `RX_Data`，然后产生一个完成信号至 `RX_Done_Sig`。（在这里数据格式四 1 帧 11 位，并且波特率为 9600kbps。）

`env_rx_module.v`

```

1. module env_rx_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input TX_En_Sig,
7.     output TX_Done_Sig,
8.     input [7:0]TX_Data,
9.
10.    input RX_En_Sig,
11.    output RX_Done_Sig,
12.    output [7:0]RX_Data,
13.
14.    output SQ_TX_Pin_Out

```

```
15.  
16. );  
17.  
18. *****/  
19.  
20. wire TX_Pin_Out;  
21.  
22. tx_module U1  
23. (  
24.     .CLK( CLK ),  
25.     .RSTn( RSTn ),  
26.     .TX_En_Sig( TX_En_Sig ),  
27.     .TX_Data( TX_Data ),  
28.     .TX_Done_Sig( TX_Done_Sig ),  
29.     .TX_Pin_Out( TX_Pin_Out )  
30. );  
31.  
32. *****/  
33.  
34. rx_module U2  
35. (  
36.     .CLK( CLK ),  
37.     .RSTn( RSTn ),  
38.     .RX_Pin_In( TX_Pin_Out ),  
39.     .RX_En_Sig( RX_En_Sig ),  
40.     .RX_Data( RX_Data ),  
41.     .RX_Done_Sig( RX_Done_Sig )  
42. );  
43.  
44. *****/  
45.  
46. assign SQ_TX_Pin_Out = TX_Pin_Out;  
47.  
48. *****/  
49.  
50.  
51. endmodule
```

笔者特意将串口发送模块 U1 的 TX_Pin_Out 引出来（14 行。46 行）。

env_rx_module.v

```
1. `timescale 1 ns/ 1 ps
2. module env_rx_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg TX_En_Sig;
8.     reg [7:0]TX_Data;
9.     wire TX_Done_Sig;
10.
11.    reg RX_En_Sig;
12.    wire [7:0]RX_Data;
13.    wire RX_Done_Sig;
14.
15.    wire SQ_TX_Pin_Out;
16.
17.    /*****
18.
19.    env_rx_module U1
20.    (
21.        .CLK(CLK),
22.        .RSTn(RSTn),
23.        .TX_En_Sig(TX_En_Sig),
24.        .TX_Data(TX_Data),
25.        .TX_Done_Sig(TX_Done_Sig),
26.        .RX_En_Sig(RX_En_Sig),
27.        .RX_Data(RX_Data),
28.        .RX_Done_Sig(RX_Done_Sig),
29.        .SQ_TX_Pin_Out(SQ_TX_Pin_Out)
30.
31.    );
32.
33.    *****/
34.
35.    initial
36.    begin
37.        RSTn = 0; #1000; RSTn = 1;
38.        CLK = 0; forever #25 CLK = ~CLK;
39.    end
40.
41.    *****/
```

```
42.  
43.    reg [3:0]i;  
44.  
45.    always @ ( posedge CLK or negedge RSTn )  
46.        if( !RSTn )  
47.            begin  
48.                i <= 4'd0;  
49.                TX_En_Sig <= 1'b0;  
50.                TX_Data <= 8'd0;  
51.            end  
52.        else  
53.            case( i )  
54.  
55.                0,1,2:  
56.                    i <= i + 1'b1;  
57.  
58.                3:  
59.                    if( TX_Done_Sig ) begin TX_En_Sig <= 1'b0; i <= i + 1'b1; end  
60.                    else begin TX_En_Sig <= 1'b1; TX_Data <= 8'h3f; end  
61.  
62.                4:  
63.                    i <= 4'd4;  
64.  
65.            endcase  
66.  
67.    /*****  
68.  
69.    reg [3:0]j;  
70.  
71.    always @ ( posedge CLK or negedge RSTn )  
72.        if( !RSTn )  
73.            begin  
74.                j <= 4'd0;  
75.                RX_En_Sig <= 1'b0;  
76.            end  
77.        else  
78.            case( j )  
79.  
80.                0:  
81.                    if( RX_Done_Sig ) begin RX_En_Sig <= 1'b0; j <= j + 1'b1; end  
82.                    else RX_En_Sig <= 1'b1;  
83.  
84.                1:  
85.                    j <= 4'd1;
```

```

86.
87.           endcase
88.
89. endmodule

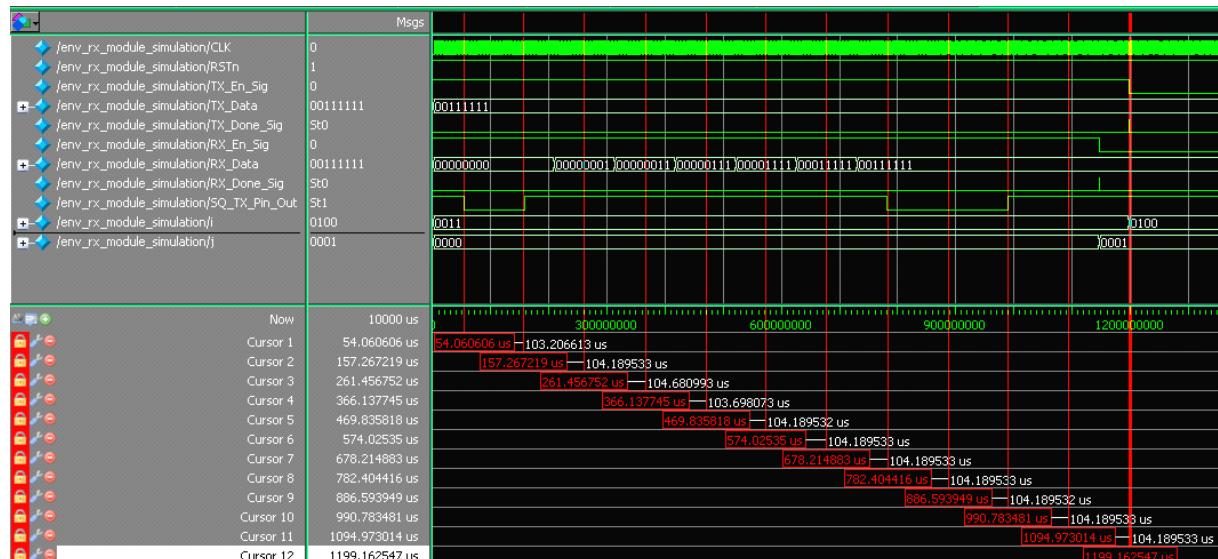
```

在 43~65 行的激励过程是针对串口发送模块。第 69~87 行的激励过程是针对串口接收模块。在一开始的时候，在步骤 j - 0 (80~82 行)，使能串口接收模块 (82 行) 并且让它进入就绪状态。

在步骤 i，是 0~2 的时候，它延迟 3 个时钟，给串口接收模块有充足的就绪时间(55~56 行)。在步骤 i 是 3 (58~60 行) 的时候，它使能串口发送模块，并且将数据 8'h3f 发送出去 (60 行)。直到发送数据完成为止，它才不使能串口发送模块，并且进入步骤 4 (59 行)。当步骤 i 是 4 的时候，它已经结束操作了。

当步骤 i 进入 3 的时候，串口发送模块已经开始发送数据 8'h3f，此时的串口接收模块已经就绪好接收数据 (82 行)。直到串口接收模块读完一帧数据，它就会不使能串口接收模块 (81 行)，然后进入步骤 j 的 1。步骤 j 的 1 是串口接收模块已经完工 (84~85 行)。

仿真模块：



哇啊啊，好壮观呀。在上图仿真结果显示了“串口发送模块作为串口接收模块的刺激”的激励过程。(Cursor 省略为 C) 在 C1~C2 之间是第一帧数据的传送，C2~C3 是第二位数据的传送，其他的以此类推，然和 C1~C12 是一帧数据 11 位的传送过程。注意，每个 Cx~Cx 之间的时间大约是 104us，亦即 9600kbps 的波特率。

在 C11~C12 之间，是串口模块之间最后一位数据的传输，亦即停止位。当串口接收模块接收到最后一个数据，然后产生完成信号，数据 8'h3f (8'b00111111) 输出至 RX_Data。

实验二十四之二说明:

在这个实验的仿真中，我们仿真了串口接收模块，然而串口发送模块作为该模块的复杂输入。

实验二十四之二结论:

实验二十四不同与实验二十三或者实验二十二，它既不是建立一个新的模块用于复杂输入，也不是直接编辑激励过程来近似复杂输入，而是用自身和自己有正负关系的模块作为复杂输入。实际上，除了串口模块意外，还有很多类似的情况，如：SPI 从机和主机等

6.4 麻烦的 IO 口仿真

当仿真工作在执行的时候，最怕莫过于遇见 IO 了，哎！IO 真的是一个“麻烦友”（粤语）。在建模（综合）的时候，要处理 IO 的引脚有两件工作要点，就是“如何从 IO 读取输出和如何驱动 IO”。

```
.....
inout SIO; // IO 端口

.....
reg isOut; // 输出使能

.....
always @ ( posedge CLK )
    .....
        case( i )
            .....
            5: begin rData <= SIO; i <= i + 1'b1; end      // 从 IO 读取数据
            6: begin isOut <= 1'b1; rQ <= 1'b1; i <= i + 1'b1; end // 从 IO 输出数据
            .....
.....
assign SIO = isOut ? rQ : 1'bz; // 关系
```

上面一段代码是在建模中会常常见到的 IO 被调用的过程。IO 在读取的时候比较简单，它和输入口一样，直接调用即可。反之 IO 在输出的时候比较麻烦，它必须有输出使能。IO 在仿真环境中也会出现类似的情况。

```
.....
reg treg_SIO; // IO 驱动用

wire SIO; // IO 输出用

assign SIO = treg_SIO; // 关系
```

上面的代码是 IO 在仿真中出现的样子。

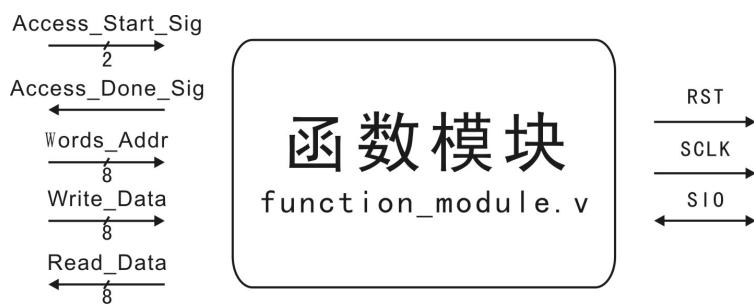
```
.....
case( i )
    .....
        5: begin treg_SIO <= 1'b1; i <= i + 1'b1; end // 驱动
        6: begin rData <= SIO; i <= i + 1'b1; end      // 读取
    .....
```

如果笔者要调用这个 IO 口，激励过程会是如上（比起在建模中调用，真是简单不少了）。

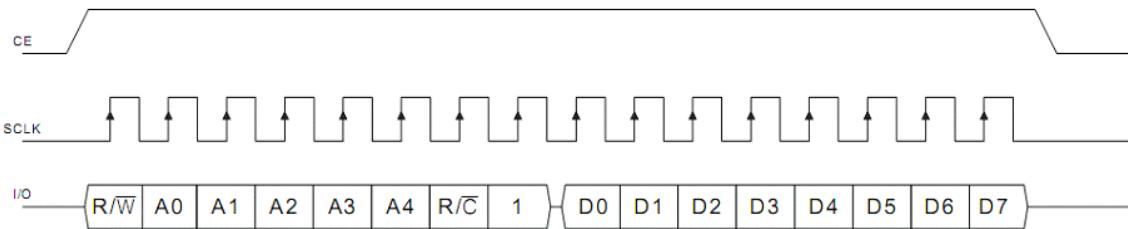
在这一章中，我们要作的工作就是仿真带有 IO 口的模块，其中《Verilog HDL 那些事儿 - 建模篇》实验十三（DS1302 驱动实验）中的 function_module.v 会是仿真对象。

实验二十五：仿真带有 IO 的模块

在这里我们稍微回忆一下 function_module.v 的大致功能：

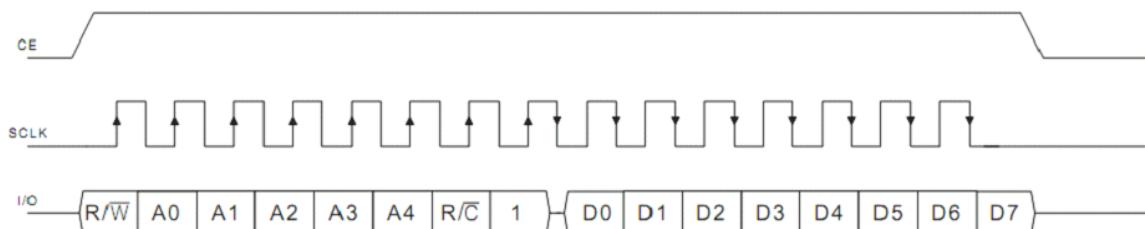


function_module.v 是 ds1302_module.v 的函数模块。它包含了 ds1302_module.v 最基本的读操作和写操作。它可以支持 2 个命令，如果命令是 2'b10 就是写操作，如果命令式 2'b01 就是读操作。function_module.v 可以的输入和输出口基本上和图形一样，其中 SIO 就是 IO。



上图是 function_module.v 在写操作的时序图（DS1302 芯片的写时序）。

function_module.v 的写操作很简单，就是一直输出数据而已。输出数据对仿真的来说没有什么难度。



上图是 function_module.v 的读操作时序图 (DS1302 芯片的读时序)。第一个字节 function_module.v 是输出操作，然而第二个字节的读操作，在仿真中是个问题。

在这里，笔者稍微说一个故事：

在很久很久以前，笔者曾经在 Ourdev 上发过这样一个问题：“IO 的输入数据，是从哪里来的”。某一个大大这样回答笔者：“你的时钟信号时如何产生，那么 IO 的输入数据就如何产生” 大大们说的话永远都很深奥，好在笔者的悟性不差。

在上述的故事中某大大的意思是说，在激励过程中，如果一方要从 IO 读取数据。那么在另一方的就要驱动 IO。用一个简单示例来说的话：

```
case( i )
```

```
...
1:
begin rData <= SIO; ...
...
```

```
case(j)
```

```
0: // 非即时结果，使用时间点概念
begin treg_SIO <= 1'b1; ... ....
```

// 或者

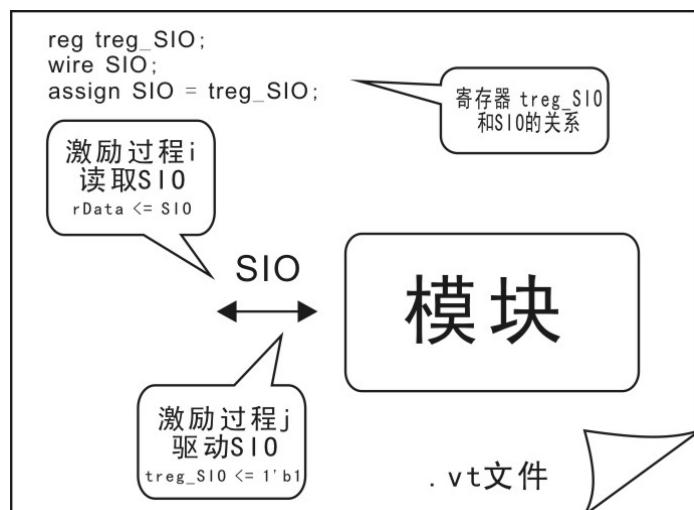
```
case( j )
```

```
.....
1: // 即时结果
begin treg_SIO = 1'b1; ...
```

一方在激励过程中，从 IO 中读取数据

一方在激励过程中，驱动 IO 口

上面的代码表示了，当激励过程 i 是 1 的时候，它从某个模块的 IO 读取数据。然后在同一个时间，另一方的激励过程 j ... 如果按照“时间点”的概念，那么它必须早一个时钟准备好要驱动的数据，亦即 j 是 0 的时候；反之，如果无视时间点的概念，在 j 是 1 的时候可以以“阻塞式赋值”驱动 IO（即时结果）并且和激励过程 i 同步。

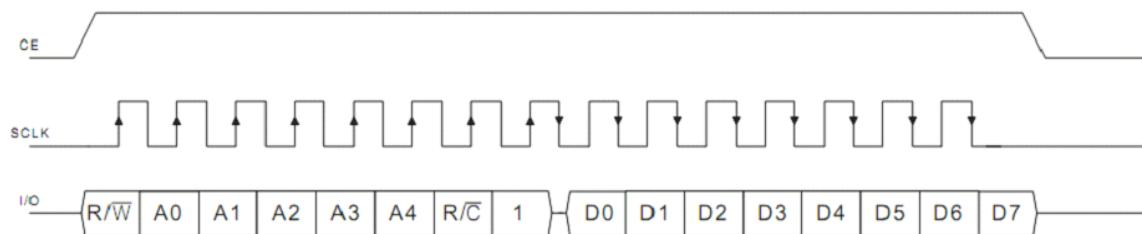


如果用具体的形象来表示的话，结果会是如同上图那样。在激励文件中，我们先声明 treg_SIO 和 SIO 这个 IO 的关系。然后在激励过程 i 对模块的 SIO 读取数据，在同一个时间，在激励过程 j 对 treg_SIO 驱动数据（输入数据）。

基本上 IO 的调用在仿真是非常容易。但是唯一的问题是“一个激励过程在什么时候要读取 IO，那么另一个激励过程又该在什么时候驱动 IO？”也就是说，IO 读取和 IO 驱动，两方的激励过程的同步性必须照顾得很好。

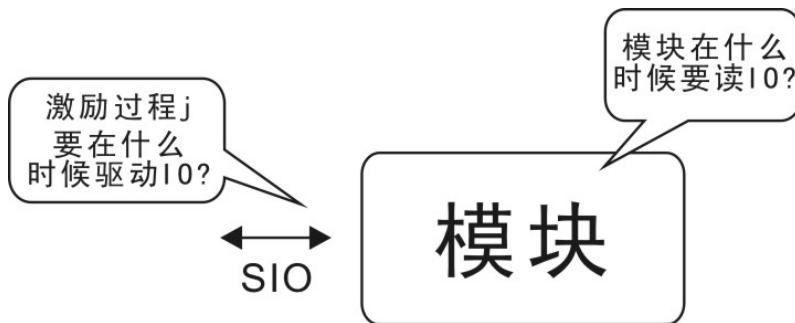
问题来了！如果我们还要配合 function_module.v 的步骤（DS1302 芯片读取时序）去执行激励过程，亦即在仿真上我们要按 fucntion_module.v 步骤的要求来驱动 IO。使得 IO 读取和驱动可以同步执行，我们该如何是好？

（这也是众多参考书中一直避开的问题。）



我们再来看一看上面这一张时序图，它是 function_module.v 的读操作时序图（DS1302 读操作的时序图）。我们可以无视第一个字节的写数据，然而第二字节的读数据就是问题点。因为我们不知道“该模块在什么时候要读 IO 口？”。

这个问题和上面例子中两个激励过程不一样，在上面例子中 IO 读取要求是在模块外部发生。



然而现在我们面对的问题是“模块在什么时候要读取 IO”和“激励过程要在什么时候驱动 IO”。换句话说，IO 读取要求是在模块内部发生。要解决这个问题，就必须和建模技巧扯上关系了。（读者稍微让脑袋冷静一下吧 … 后面的故事还很长。）

function_module.v

```
1. module function_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Words_Addr,
6.     Write_Data,
7.     Read_Data,
8.     Done_Sig,
9.     RST,
10.    SCLK,
11.    SIO,
12.
13.    SQ_i
14. );
15.
16.    input CLK;
17.    input RSTn;
18.    input [1:0]Start_Sig;
19.    input [7:0]Words_Addr;
20.    input [7:0]Write_Data;
21.    output [7:0]Read_Data;
22.    output Done_Sig;
23.    output RST;
24.    output SCLK;
25.    inout SIO;
26.
27.    output [5:0]SQ_i;
28.
29.    ****
30.
31.    parameter T0P5US = 4'd9;
32.
33.    ****
34.
35.    reg [3:0]Count1;
36.
37.    always @ ( posedge CLK or negedge RSTn )
38.        if( !RSTn )
39.            Count1 <= 4'd0;
40.        else if( Count1 == T0P5US )
41.            Count1 <= 4'd0;
42.        else if( Start_Sig[0] == 1'b1 || Start_Sig[1] == 1'b1 )
43.            Count1 <= Count1 + 1'b1;
```

```
44.           else
45.             Count1 <= 4'd0;
46.
47.           ****
48.
49.           reg [5:0]i;
50.           reg [7:0]rData;
51.           reg rSCLK;
52.           reg rRST;
53.           reg rSIO;
54.           reg isOut;
55.           reg isDone;
56.
57.           always @ ( posedge CLK or negedge RSTn )
58.             if( !RSTn )
59.               begin
60.                 i <= 6'd0;
61.                 rData <= 8'd0;
62.                 rSCLK <= 1'b0;
63.                 rRST <= 1'b0;
64.                 rSIO <= 1'b0;
65.                 isOut <= 1'b0;
66.                 isDone <= 1'b0;
67.               end
68.             else if( Start_Sig[1] )
69.               case( i )
70.
71.               0 :
72.                 begin rSCLK <= 1'b0; rData <= Words_Addr; rRST <= 1'b1; isOut <= 1'b1; i <= i + 1'b1; end
73.
74.               1, 3, 5, 7, 9, 11, 13, 15 :
75.                 if( Count1 == T0P5US ) i <= i + 1'b1;
76.                 else begin rSIO <= rData[ (i >> 1) ]; rSCLK <= 1'b0; end
77.
78.               2, 4, 6, 8, 10, 12, 14, 16 :
79.                 if( Count1 == T0P5US ) i <= i + 1'b1;
80.                 else begin rSCLK <= 1'b1; end
81.
82.               17 :
83.                 begin rData <= Write_Data; i <= i + 1'b1; end
84.
85.               18, 20, 22, 24, 26, 28, 30, 32 :
86.                 if( Count1 == T0P5US ) i <= i + 1'b1;
87.                 else begin rSIO <= rData[ (i >> 1) - 9 ]; rSCLK <= 1'b0; end
88.
```

```
89.          19, 21, 23, 25, 27, 29, 31, 33 :
90.          if( Count1 == T0P5US ) i <= i + 1'b1;
91.          else begin rSCLK <= 1'b1; end
92.
93.          34 :
94.          begin rRST <= 1'b0; i <= i + 1'b1; end
95.
96.          35 :
97.          begin isDone <= 1'b1; i <= i + 1'b1; end
98.
99.          36 :
100.         begin isDone <= 1'b0; i <= 6'd0; end
101.
102.         endcase
103.        else if( Start_Sig[0] )
104.          case( i )
105.
106.            0 :
107.            begin rSCLK <= 1'b0; rData <= Words_Addr; rRST <= 1'b1; isOut <= 1'b1; i <= i + 1'b1; end
108.
109.            1, 3, 5, 7, 9, 11, 13, 15 :
110.            if( Count1 == T0P5US ) i <= i + 1'b1;
111.            else begin rSIO <= rData[ (i >> 1) ]; rSCLK <= 1'b0; end
112.
113.            2, 4, 6, 8, 10, 12, 14, 16 :
114.            if( Count1 == T0P5US ) i <= i + 1'b1;
115.            else begin rSCLK <= 1'b1; end
116.
117.            17 :
118.            begin isOut <= 1'b0; i <= i + 1'b1; end
119.
120.            18, 20, 22, 24, 26, 28, 30, 32 :
121.            if( Count1 == T0P5US ) i <= i + 1'b1;
122.            else begin rSCLK <= 1'b1; end
123.
124.            19, 21, 23, 25, 27, 29, 31, 33 :
125.            if( Count1 == T0P5US ) begin i <= i + 1'b1; end
126.            else begin rSCLK <= 1'b0; rData[ (i >> 1) - 9 ] <= SIO; end
127.
128.            34 :
129.            begin rRST <= 1'b0; isOut <= 1'b1; i <= i + 1'b1; end
130.
131.            35 :
132.            begin isDone <= 1'b1; i <= i + 1'b1; end
133.
```

```
134.          36 :
135.          begin isDone <= 1'b0; i <= 6'd0; end
136.
137.          endcase
138.
139.          /*****
140.
141.          assign Read_Data = rData;
142.          assign Done_Sig = isDone;
143.
144.          assign RST = rRST;
145.          assign SCLK = rSCLK;
146.          assign SIO = isOut ? rSIO : 1'bz;
147.
148.          *****/
149.
150.          assign SQ_i = i;
151.
152.          *****/
153.
154. endmodule
```

function_module.v 的具体功能笔者就不重复了，在《Verilog HDL 那些事儿-建模篇》第 4.3 篇中的实验十三，哪里有仔细的介绍。在这里，关键是第 13 行，27 行和 150 行，将步骤 i 引出的过程。在 103~137 行是 function_module.v 命令为 2'b01 的读操作。步骤 i 从 0~16 是第一个字节的写操作，我们可以无视。重点就是在于从步骤 17 开始，亦即步骤 18~33 的读操作。

如果我们要知道“该模块要在什么时候读取 IO？”那么答案就是在步骤 18~33。如果我们又要知道“该模块在什么时候要读取哪一个位数据？”那么步骤 18（第零位），20（第一位），22（第二位），24（第三位），26（第四位），28（第五位），30（第六位），32（第七位）等都是我们的答案。（DS1302 的传输时从 LSB，亦即最低位开始）

为什么是步骤 18，20，22，24，26，28，30，32 而不是步骤 19，21，23，25，27，29，31，33 呢？我们知道 DS1302 在读操作的时候，第二个字节的读数据是 SCLK “下降沿有效”，然而数据的“设置”（更新）是发生在 SCLK “上升沿”，步骤 18，20，22，24，26，28，30，32 等都是上升沿的操作。

在仿真的时候，我们只要将 function_module.v 的步骤 i 引出来以后，我们即可知道“该模块在什么时候要读取 IO”。

function_module.vt

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

```
2. module function_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     reg [1:0] Start_Sig;
8.     reg [7:0] Words_Addr;
9.     reg [7:0] Write_Data;
10.
11.    wire Done_Sig;
12.    wire [7:0]Read_Data;
13.
14.    wire RST;
15.    wire SCLK;
16.
17.    wire SIO;
18.    reg treg_SIO;
19.    assign SIO = treg_SIO;
20.
21.    wire [5:0]SQ_i;
22.
23.    ****
24.
25.    function_module U1
26.    (
27.        .CLK(CLK),
28.        .RSTn(RSTn),
29.        .Start_Sig(Start_Sig),
30.        .Words_Addr(Words_Addr),
31.        .Write_Data(Write_Data),
32.        .Done_Sig(Done_Sig),
33.        .Read_Data(Read_Data),
34.        .RST(RST),
35.        .SCLK(SCLK),
36.        .SIO(SIO),
37.        .SQ_i(SQ_i)
38.
39.    );
40.
41.    ****
42.
43.    initial
44.    begin
45.        RSTn = 0; #1000; RSTn = 1;
46.        CLK = 0; forever #25 CLK = ~CLK;
```

```
47.      end
48.
49.      ****
50.
51.      reg [3:0]i;
52.
53.      always @ ( posedge CLK or negedge RSTn )
54.          if( !RSTn )
55.              begin
56.                  i <= 4'd0;
57.                  Start_Sig <= 2'd0;
58.                  Words_Addr <= 8'd0;
59.                  Write_Data <= 8'd0;
60.              end
61.          else
62.              case( i )
63.
64.                  0:
65.                      if( Done_Sig ) begin Start_Sig <= 2'b00; i <= i + 1'b1; end
66.                      else begin Start_Sig <= 2'b10; Words_Addr <= 8'hf0; Write_Data <= 8'hf2; end
67.
68.                  1:
69.                      if( Done_Sig ) begin Start_Sig <= 2'b00; i <= i + 1'b1; end
70.                      else begin Start_Sig <= 2'b01; Words_Addr <= 8'hf0; end
71.
72.                  2:
73.                      i <= 4'd2;
74.
75.              endcase
76.
77.      ****
78.
79.      reg [7:0] DS1302_Data;
80.
81.      always @ ( posedge CLK or negedge RSTn )
82.          if( !RSTn )
83.              begin
84.                  treg_SIO <= 1'b0;
85.                  DS1302_Data <= 8'h33;
86.              end
87.          else if( Start_Sig == 2'b01 )
88.              case( SQ_i )
89.
90.                  18: treg_SIO <= DS1302_Data[0];
91.                  20: treg_SIO <= DS1302_Data[1];
```

```

92.          22: treg_SIO <= DS1302_Data[2];
93.          24: treg_SIO <= DS1302_Data[3];
94.          26: treg_SIO <= DS1302_Data[4];
95.          28: treg_SIO <= DS1302_Data[5];
96.          30: treg_SIO <= DS1302_Data[6];
97.          32: treg_SIO <= DS1302_Data[7];
98.
99.      endcase
100.
101. ****
102.
103.
104.
105.
106. endmodule

```

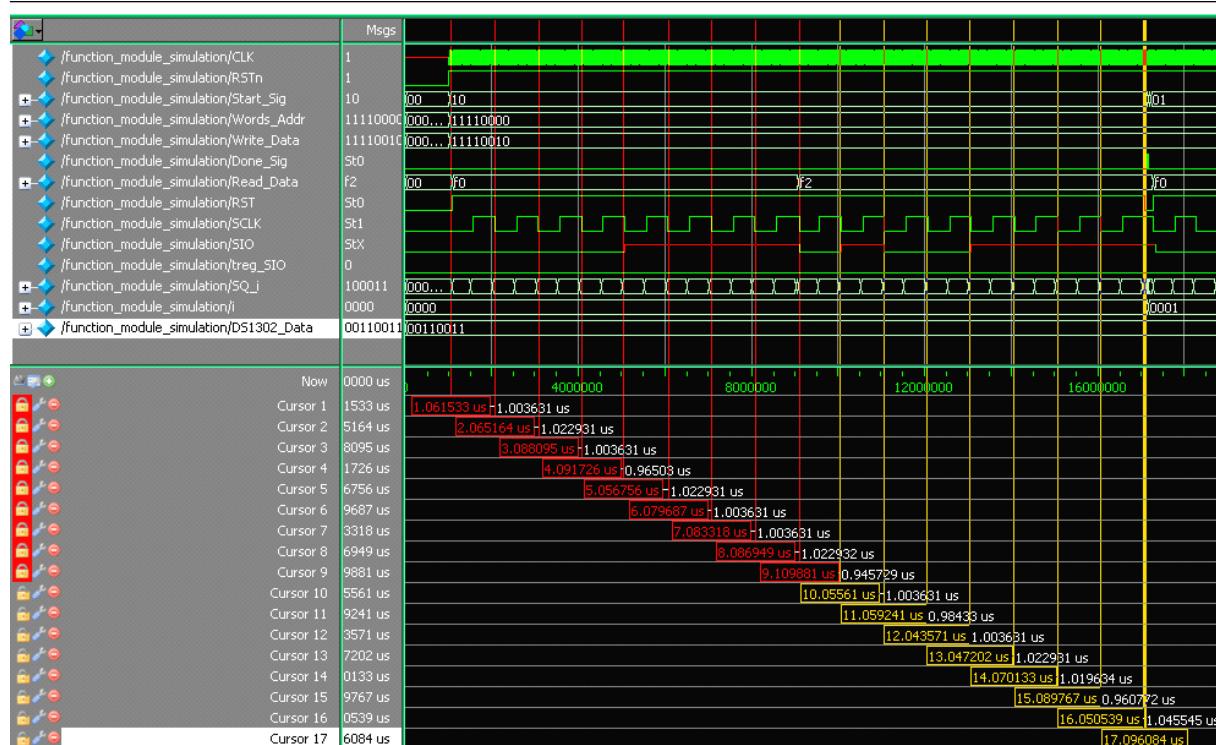
第 17~19 行是定义了 SIO 和 treg_SIO 之间的关系。在 51~75 行是激励过程 i，它主要是在步骤 0(64~66 行)的时候命令 function_module.v 写操作，写入的数据是 Words_Addr 8'hf0 和 Write_Data 8'hf2（这点不重要）。在步骤 1 的时候（68~70 行）是命令 function_module.v 读操作（这才是重点）。

在 79~99 行是激励过程 j，它主要是用来驱动 IO。第 79 行是寄存器 DS1302_Data，它用来寄存了要给 IO 驱动的数据，该数据是 8'h33。在 87 行表示了，该激励过程在 function_module.v 被命令为读操作的时候（2'b01）才有效。第 88~99 行是根据 function_module.v 引出来的步骤 i，对 IO 驱动的操作。

第 90 行，SQ_i 是 18，既是读操作的第一个上升沿，驱动 IO 的数据为寄存器第[0]位值。
 第 91 行，SQ_i 是 20，既是读操作的第二个上升沿，驱动 IO 的数据为寄存器第[1]位值。
 第 92 行，SQ_i 是 22，既是读操作的第三个上升沿，驱动 IO 的数据为寄存器第[2]位值。
 第 93 行，SQ_i 是 24，既是读操作的第四个上升沿，驱动 IO 的数据为寄存器第[3]位值。
 第 94 行，SQ_i 是 26，既是读操作的第五个上升沿，驱动 IO 的数据为寄存器第[4]位值。
 第 95 行，SQ_i 是 28，既是读操作的第六个上升沿，驱动 IO 的数据为寄存器第[5]位值。
 第 96 行，SQ_i 是 30，既是读操作的第七个上升沿，驱动 IO 的数据为寄存器第[6]位值。
 第 97 行，SQ_i 是 32，既是读操作的第八个上升沿，驱动 IO 的数据为寄存器第[7]位值。

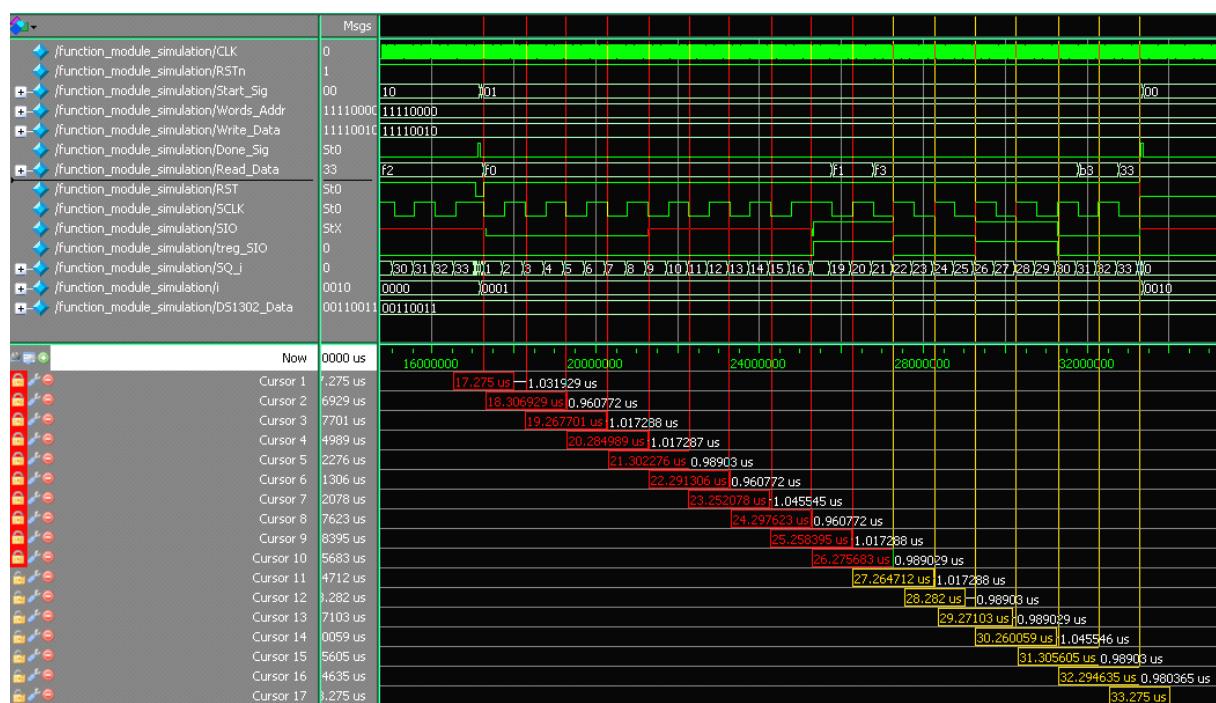
仿真结果：

在这里读者要注意一点，由于笔者的 Modelsim 不给力的关系，在 IO 的输出上，逻辑 1 是呈现红线（一般上红线是悬空的意思），事实上它不是悬空，而是逻辑 1。但是这个问题不会给仿真结果带来影响，只是在视觉上比较别扭一点。



上图仿真结果是激励过程 i, 是步骤 0 对 function_module.v 执行的写操作。写操作的命令是 2'b10 (Start_Sig), Words_Addr 的数据时 8'hf0, Write_Data 的数据时 8'hf2。function_module.v 在这个时候仿真只是对 IO (SIO) 输出数据而已。

在仿真结果中, (Cursor 省略为 C) C1~C9 表示了 Words_Addr 的数据 8'hf0 在 SIO 上的输出。C1~C2 之间是 Words_Addr 的第[0]位值在 SIO 的数据, 其余的以此类推。C9~C17 是 Wrte_Data 的数据 8'hf2 在 SIO 上的输出。C9~C10 是 Write_Data 的第[0]值在 SIO 的数据, 其它的也是如此。(注意 DS1302 的传输是低位开始高位结束。)



上图仿真结果是激励过程 i, 步骤 1 对 function_module.v 执行的读操作。读操作的命令是 2'b01 (Start_Sig), 在同一个时间激励过程 j 被使能。C1~C9 是第一个字节发送数据 8'hf0 在 SIO 上的输出 (写操作)。(重点在这之后)

C9~C17 是 function_module.v 从 SIO 上读取数据的时候, 在另一方面激励过程 j 要按照 SCLK 时钟对 SIO 驱动。其中 SQ_i 是 function_module.v 模块内部操作的显示指示。激励过程 j 就是根据 SQ_i 对 treg_SIO 赋值 (SIO 驱动)。(注: DS1302_Data 为 8'h33。)

在 C9~C10 是 function_module.v 模块要求从 SIO 读取第[0]位数据, 在 SQ_i 为 18 的时候, 亦即是读操作 SCLK 时钟第一个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[0]值。然后在 SQ_i 为 19 的时候, 该模块从 SIO 读取第[0]位数据。

在 C10~C11 是 function_module.v 模块要求从 SIO 读取第[1]位数据, 在 SQ_i 为 20 的时候, 亦即是读操作 SCLK 时钟第二个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[1]值。然后在 SQ_i 为 21 的时候, 该模块从 SIO 读取第[1]位数据。

在 C11~C12 是 function_module.v 模块要求从 SIO 读取第[2]位数据, 在 SQ_i 为 22 的时候, 亦即是读操作 SCLK 时钟第三个上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[2]值。然后在 SQ_i 为 23 的时候, 该模块从 SIO 读取第[0]位数据。

在 C12~C13 是 function_module.v 模块要求从 SIO 读取第[3]位数据, 在 SQ_i 为 24 的时候, 亦即是读操作 SCLK 时钟第四个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[3]值。然后在 SQ_i 为 25 的时候, 该模块从 SIO 读取第[3]位数据。

在 C13~C14 是 function_module.v 模块要求从 SIO 读取第[4]位数据, 在 SQ_i 为 26 的时候, 亦即是读操作 SCLK 时钟第五个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[4]值。然后在 SQ_i 为 27 的时候, 该模块从 SIO 读取第[4]位数据。

在 C14~C15 是 function_module.v 模块要求从 SIO 读取第[5]位数据, 在 SQ_i 为 28 的时候, 亦即是读操作 SCLK 时钟第六个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[5]值。然后在 SQ_i 为 29 的时候, 该模块从 SIO 读取第[5]位数据。

在 C15~C16 是 function_module.v 模块要求从 SIO 读取第[6]位数据, 在 SQ_i 为 30 的时候, 亦即是读操作 SCLK 时钟第七个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[6]值。然后在 SQ_i 为 31 的时候, 该模块从 SIO 读取第[6]位数据。

在 C16~C17 是 function_module.v 模块要求从 SIO 读取第[7]位数据, 在 SQ_i 为 32 的时候, 亦即是读操作 SCLK 时钟第八个的上升沿, treg_SIO 驱动源为 DS1302_Data 寄存器的[7]值。然后在 SQ_i 为 33 的时候, 该模块从 SIO 读取第[7]位数据。

当 function_module.v 完成读操作的时候, 它就产生一个完成信号, 然后将从 SIO 读取到的结果输出在 Read_Data 上。DS1302_Data 寄存器的值是 8'h33, 所以 Read_Data 的值也是 8'h33。

实验二十五说明:

IO 在仿真上的调用，如果仿真对象是很单纯的话，仿真就没有什么困难。相反的，如果被仿真对像 function_module.v 那样，那么激励过程一定要准确的掌握该模块“在什么时候读取 IO”。

在该实验的激励过程中“刺激|输入”尽是问答输入（条件输入）。

实验二十五结论:

实际上，在实验二十五的仿真中，IO 的调用如此的简单，这全部“好处”都是托低级建模的福。当我们使用低级建模的去完成建模，很多时候模块都带有步骤 i。在仿真的时候，往往步骤 i 都会对仿真的工作起到帮助。如果没有步骤 i 的帮助，笔者真的不知道要如何了解“模块在什么时候要从 IO 读取数据”，也不知道“激励过程要在什么时候对 IO 驱动”……

总结：

在这一章中笔者提出各种基于综合（建模）去完成仿真工作；如何编辑激励过程；如何产生预期的“刺激”（输入）等，都是这一章笔记的重点。在前一章中，笔记已经说过激励过程就是仿真对象“刺激和反应”的经过。其中“刺激”既是“输入”，“反应”既是“输出”。“输入”，有简单输入，复杂输入和问答输入（条件输入）。

简单的虚拟输入可以是逻辑信号或者一组数据。

反之复杂输入，最简单的例子是实验二十二。在实际情况中，按键消抖模块是为了过滤按键的抖动才存在着。假如把这一观点带入“虚拟的环境”的话，我们要建立近似“按键抖动”的“虚拟输入”。当然我们可以选择在激励过程建立“按键抖动”，又或者直接建立一个“虚拟按键模块”来实现“按键抖动”。此外，如果仿真对象是有“正负”关系的话，其中一方可以充当刺激。

问答输入，亦即条件输入。简单的例子会是“Start_Sig 和 Done_Sig”之间的作用。复杂的话，会是如同实验二十五那样，利用不同的激励过程，一方激励过程对 function_module.v 刺激，另一方激励过程对 IO 刺激。

在这一章中，所用仿真的建立都是近似“黑金开发板”这个虚拟环境，比较有象征性是“时钟信号”和“复位信号”。

最后，笔者还是需要再强调一次，笔记中全部的内容，都是笔者一厢情愿的想法而已，没有任何强迫性。笔者也没有说过在激励文件不可以使用 Verilog HDL 的验证语言。只是笔者个人在想法上，想用综合语言用去完成两种不同工作，从而更了解 Verilog HDL 语言。至于笔者的想法对不对，合不合适，读者就见仁见智了。

第七章：反应和调试过程

7.1 输出中珍贵的信息

在这一章中，我们要讨论的内容恰好是和前一章 180 度的不同。在前一章中笔者说了，刺激在激励过程中怎样又怎样，反之这一章，笔者要说的东西是反应在激励过程中，怎样又怎样。

“反应”在“激励”中是“输出”的意思，故“输出”不只是在仿真软件上看到的波形图而已，然而“输出”在仿真中也代表“模块的反应”。我们把“模块的反应”摊开先不说，毕竟在前面的实验，笔者都有形或无形的提及过。还是先把焦点围绕着“波形图”的身上。因为只要把波形图看懂了，自然而然读者什么都会明白。

当某个模块受到“刺激”时，它都会把“反应”吐在波形图上。实际上在波形图里是充满许多有用的信息，可是要把这些信息看懂并且和 Verilog HDL 语言扯上关系，说实话这是一件苦差事，而且比起建模更困难。

在现实中就像普通人看病例书一样，完全都看不懂，但是医生却看得懂。模块是病人，波形图是病例书，我们是医生。我们要干的事情就只有一件，就是当起“模块的医生”看看模块“哪里不舒服”和“吃错什么了”等，从波形图中分析模块的问题。

在这里，笔者就以《Verilog HDL 那些事儿-建模篇》中实验九之一的 vga 模块里的 sync_module.v 作为实例。选择它的原因很简单，因为 sync_module.v 用不着编辑复杂的激励过程，只要有时钟信号和复位信号，我们就能观察它的输出。接下来，我们要从波形图中了解 sync_module.v 的信息，并且优化它。（老实说，笔者这样作就像是自己抽自己的脸。）

实验二十六：优化 vga 的同步模块

同步模块是 VGA 模块中最重要的模块，它负责驱动，此外它也反馈出当前显示的 x 地址，y 地址和显示有效信号（Ready_Sig）。在某一个角度来看 sync_module.v 是负责着“显示标准”的工作。在这里我们以 800 x 600 x 60Hz 为例。

800 x 600 x 60Hz	a 段	b 段	c 段	d 段	e 段-总共 n 个列像素
HSYNC Signal 列像素	128	88	800	40	1056
800 x 600 x 60Hz	o 段	p 段	q 段	r 段	s 段-总共 n 个行像素
VSYNC Signal 行像素	4	23	600	1	628

上表是以 800 x 600 x 60Hz 显示标准的时序分段（具体的解释请参考建模篇）。我们知道在 VGA 的时序中，最小的单位是“一个列像素”。如果以 800 x 600 x 60Hz 为显示标准，那么这个“一个列像素”的时间单位是 25ns，亦即 40Mhz 为驱动频率。在仿真的时候我们不可能以一个时钟一个时钟去计算，所以笔者将它们转换为时间，单位为 us。

我们稍微回忆一下，假设以 800 x 600 x 60Hz 为显示标准：

$$\text{一个列像素} = \text{显示主频的周期} = 1 / 40\text{Mhz} = 25\text{ns}$$

$$\text{一个行像素} = 1056 \text{ 个列像素} = 1056 * 25 = 26.4\mu\text{s}$$

800 x 600 x 60Hz	a 段	b 段	c 段	d 段	e 段-总共 n 个列像素
HSYNC Signal 列像素	3.2us	2.2us	20us	1us	26.4us
800 x 600 x 60Hz	o 段	p 段	q 段	r 段	s 段-总共 n 个行像素
VSYNC Signal 行像素	105.6us	607.2us	15840us	26.4us	16579.2us

sync_module.v (before)

```

1. module sync_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig, Ready_Sig,
5.     Column_Addr_Sig, Row_Addr_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Ready_Sig;
13.    output [10:0]Column_Addr_Sig;
14.    output [10:0]Row_Addr_Sig;
15.
16.    *****/
17.
18.    reg [10:0]Count_H;
19.
20.    always @ ( posedge CLK or negedge RSTn )
21.        if( !RSTn )
22.            Count_H <= 11'd0;
23.        else if( Count_H == 11'd1056 )

```

```

24.          Count_H <= 11'd0;
25.      else
26.          Count_H <= Count_H + 1'b1;
27.
28.      *****/
29.
30.      reg [10:0]Count_V;
31.
32.      always @ ( posedge CLK or negedge RSTn )
33.          if( !RSTn )
34.              Count_V <= 11'd0;
35.          else if( Count_V == 11'd628 )
36.              Count_V <= 11'd0;
37.          else if( Count_H == 11'd1056 )
38.              Count_V <= Count_V + 1'b1;
39.
40.      *****/
41.
42.      reg isReady;
43.
44.      always @ ( posedge CLK or negedge RSTn )
45.          if( !RSTn )
46.              isReady <= 1'b0;
47.          else if( ( Count_H > 11'd216 && Count_H < 11'd1017 ) &&
48.                  ( Count_V > 11'd27 && Count_V < 11'd627 ) )
49.              isReady <= 1'b1;
50.          else
51.              isReady <= 1'b0;
52.
53.      *****/
54.
55.      assign VSYNC_Sig = ( Count_V <= 11'd4 ) ? 1'b0 : 1'b1;
56.      assign HSYNC_Sig = ( Count_H <= 11'd128 ) ? 1'b0 : 1'b1;
57.      assign Ready_Sig = isReady;
58.
59.
60.      *****/
61.
62.      assign Column_Addr_Sig = isReady ? Count_H - 11'd217 : 11'd0; // Count from 0
63.      assign Row_Addr_Sig = isReady ? Count_V - 11'd28 : 11'd0; // Count from 0
64.
65.      *****/
66.
67. endmodule

```

上面是优化之前的 sync_module.v，读者先刷新刷新自己的大脑吧。

sync_module.vt

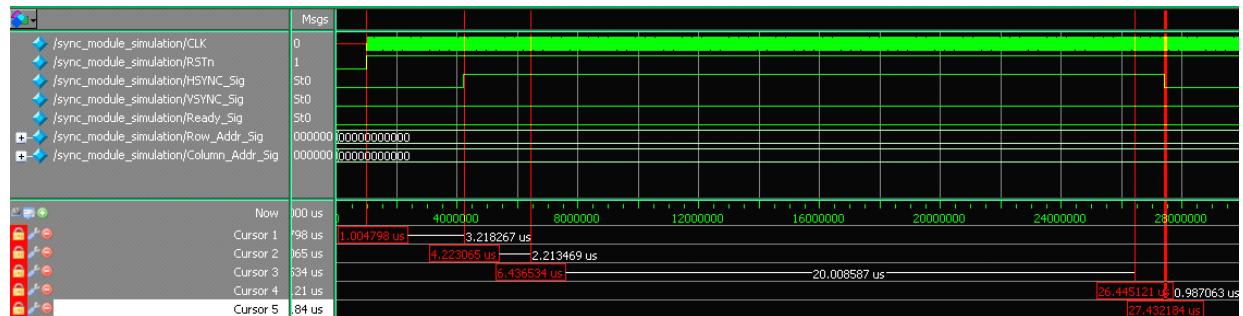
```
1. `timescale 1 ps/ 1 ps
2. module sync_module_simulation();
3.
4.     reg CLK;
5.     reg RSTn;
6.
7.     wire HSYNC_Sig;
8.     wire VSYNC_Sig;
9.     wire Ready_Sig;
10.    wire [10:0]Row_Addr_Sig;
11.    wire [10:0]Column_Addr_Sig;
12.
13.   *****/
14.
15.   sync_module U1
16.   (
17.       .CLK(CLK),
18.       .RSTn(RSTn),
19.       .HSYNC_Sig(HSYNC_Sig),
20.       .VSYNC_Sig(VSYNC_Sig),
21.       .Ready_Sig(Ready_Sig),
22.       .Row_Addr_Sig(Row_Addr_Sig),
23.       .Column_Addr_Sig(Column_Addr_Sig)
24.   );
25.
26.   *****/
27.
28.   initial
29.   begin
30.       RSTn = 0; #1000000; RSTn = 1;
31.       CLK = 0; forever #12500 CLK = ~CLK;
32.   end
33.
34.   *****/
35.
36. endmodule
```

上面是激励文件，比较简单，除了时钟信号和复位信号以外，几乎没有复杂的激励过程。其中需要注意的是第 1 行的时间刻度，笔者将它设置为 ps。因为以 800 x 600 x 60Hz 为

显示标准，时钟的周期是 25ns。如果时间刻度为 ns，无法取得完整的半周期时间（12.5 小数必须进位或者割舍）。

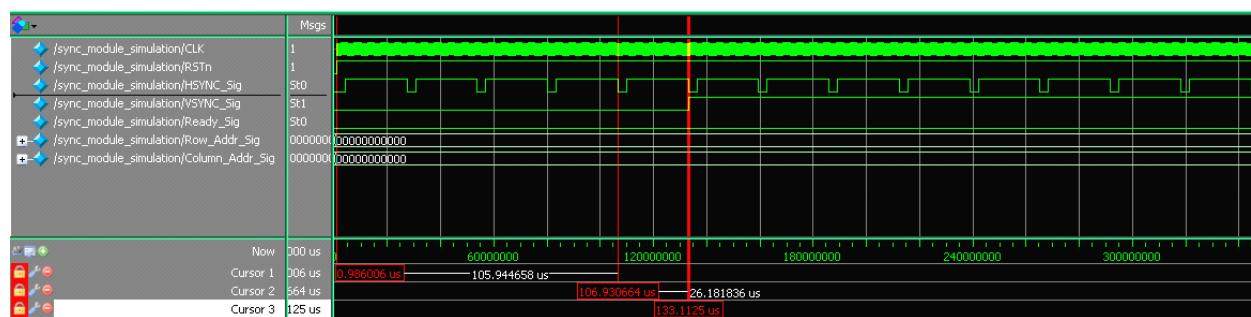
第 30 行，笔者设置了 1000000ps 的复位，亦即 1us 的复位。然而在 31 行，笔者设置了 12500ps 时钟的半周期时间，亦即 12.5ns，完整周期是 $12500\text{ps} \times 2 = 25000\text{ps} = 25\text{ns}$ 。

优化过程：

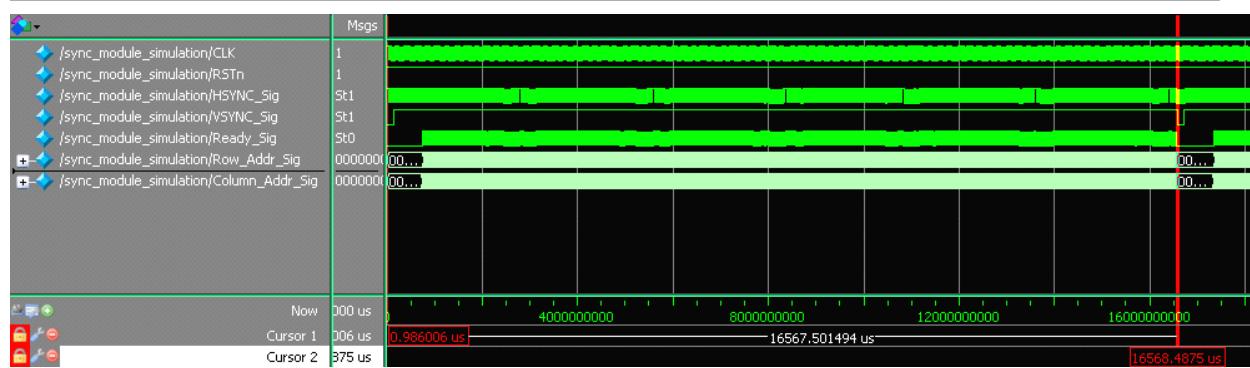


上面的仿真图是一个列像素的时序图(波形图),(Cursor 省略为 C) C1~C2 是 HSYNC_Sig 的 a 段，时间大约是 3.2us。C2~C3 是 b 段，时间大约是 2.2us。C3~C4 是 d 段，时间大约是 20us。C4~C5 是 e 段，时间大约是 1us。1056 列像素的总时间在 C1~C5，时间大约是 26.4 us (C5 减掉 C1 的时间)。

粗略看来 HYSNC_Sig 的驱动很完美，没有什么东西好修改的。



上图是 VSYNC_Sig 的 o 段，我们知道 o 段所需要的时间是 $4 * 1056 * 25\text{ns}$ (C1~C2)，亦即 105.6us。从波形图上，我们看到 VSYNC_Sig 拉高之前，有 5 个 $1056 * 25\text{ns}$ (C1~C3) 的列像素。这 有点问题，理论上是 4 个而已呀，怎么多了一个(C2~C3)? 在这里一定出了什么问题 (我们先放下这个问题)。



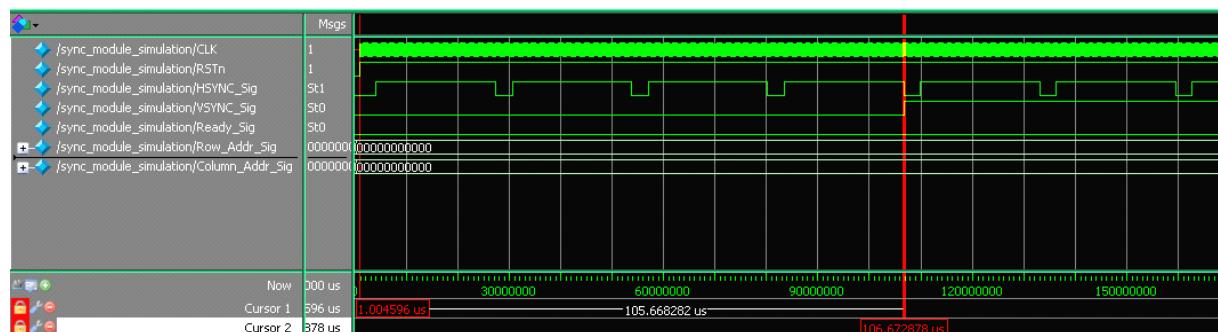
上图是一个行像素的时序图，亦即 $628 * 1056 * 25\text{ns}$ (C1~C2)，时间大约是 16579.2us。这也显得，VSYNC_Sig 的在 o 段出现的问题，却没有对“一个行像素”造成影响。

代码到底是在哪里写错了？

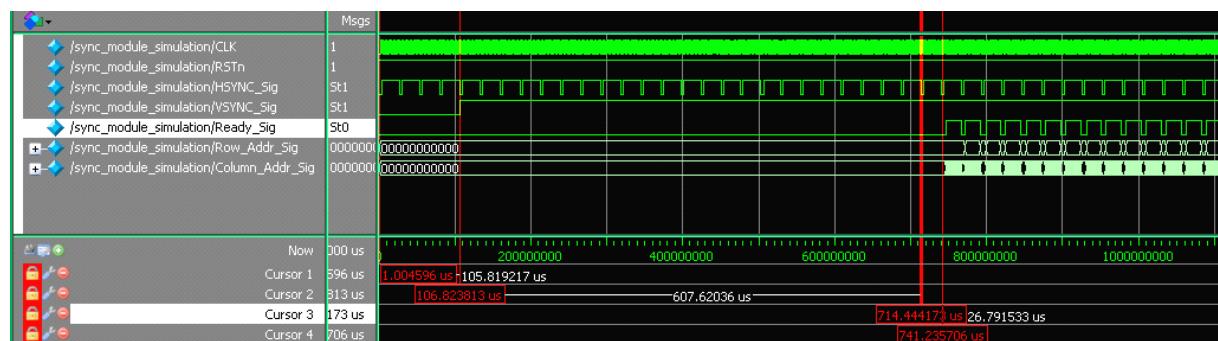
```
55. assign VSYNC_Sig = ( Count_V <= 11'd4 ) ? 1'b0 : 1'b1;
```

```
55. assign VSYNC_Sig = ( Count_V <= 11'd3 ) ? 1'b0 : 1'b1;
```

笔者把 sync_module.v 的第 55 行做了一点修改 Count_V $\leq 11'\text{d}4$ 变成 Count_V ≤ 3 。然后再重新仿真。



上图是经修改 sync_module.v 第 55 行代码后再一次仿真的结果。焦点依然是 VSYNC_Sig 的 o 段。从上图中我们可以看到 VSYNC_Sig 拉高之前，o 段有 $4 * 1056 * 25\text{ns}$ (C1~C2)，时间大约是 105.6us。嗯，这个结果符合要求了。我们继续往下调试。

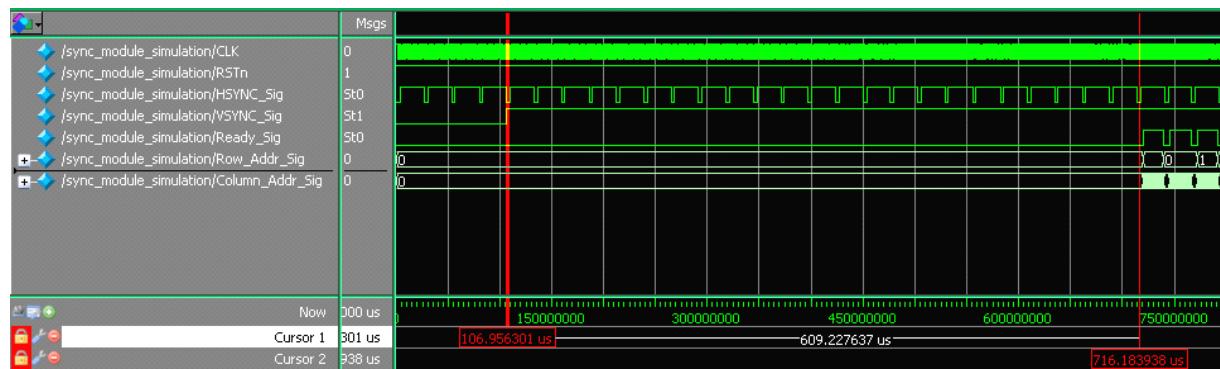


上图的仿真结果是 VSYNC_Sig 的 p 段。原理上 p 段是 23 个 $1056 * 25\text{ns}$, 时间大约是 $607.2\mu\text{s}$ (C2~C3)。在 p 段之后, Ready_Sig 应该是拉高才对呀 怎么 Ready_Sig 拉高延迟了一个 $1056 * 25\text{ns}$ (C3~C4)? 哪里又发生问题了?

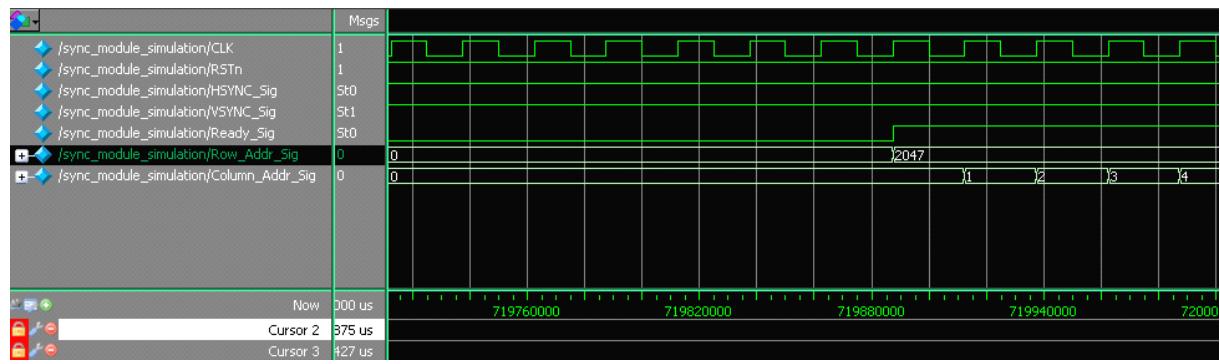
```
47.     else if( ( Count_H > 11'd216 && Count_H < 11'd1017 ) &&
48.             ( Count_V > 11'd27 && Count_V < 11'd627 ) )
```

```
47.     else if( ( Count_H >= 11'd216 && Count_H < 11'd1017 ) &&
48.             ( Count_V >= 11'd27 && Count_V < 11'd627 ) )
```

凶手原来是 47~48 行的 $\text{Count_H} > 11'\text{d}216$ 和 $\text{Count_V} > 11'\text{d}27$, 笔者把它修改为 $\text{Count_H} >= 11'\text{d}216$, $\text{Count_V} >= 11'\text{d}27$ 。然后笔者再一次启动仿真。



上图是经过修改第 47~48 行后, 重新仿真的结果。该焦点依然是 VSYNC_Sig 的 p 段。上图中它显示了 p 段 (C1~C2), 时间大约是 $609\mu\text{s}$ 。嗯! 这才“听话嘛”。VSYNC_Sig 的 p 段校正以后, 我们继续往下看。

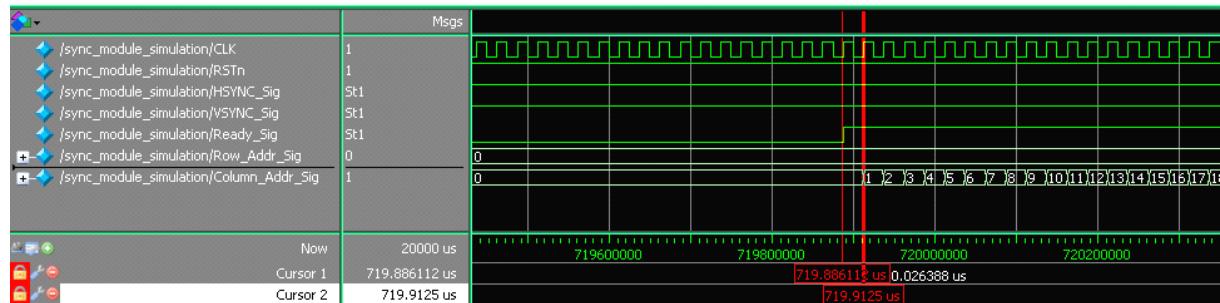


上图是 VSYNC_Sig p 段之后的 q 段。Column_Addr_Sig 是从 0 开始计数, 奇怪 Row_Addr_Sig 也应该从 0 开始计数的吗? 它既然从 2047 开始计数。原来问题发生在 63 行。

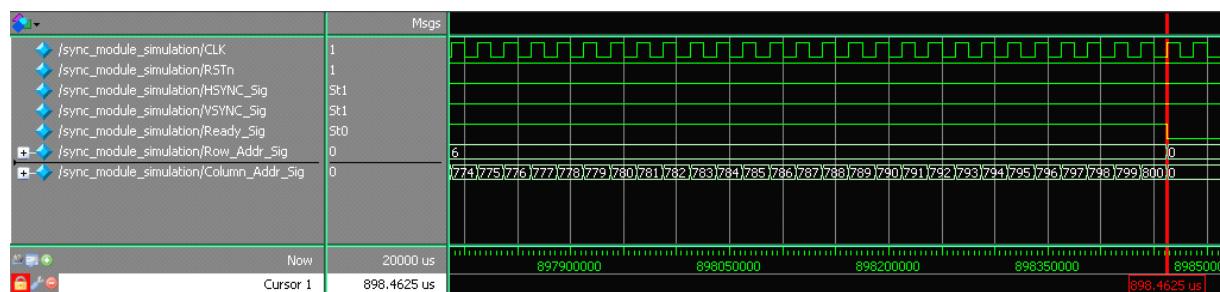
```
63.     assign Row_Addr_Sig = isReady ? Count_V - 11'd28 : 11'd0;
```

```
63.     assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0;
```

在这里，笔者将 Count_V - 11'd28 改成 Count_V - 11'd27。然后再一次仿真。



经过修改 63 行以后，VSYNC_Sig 的 p 段之后的 q 段，Row_Addr_Sig 已经从 0 开始计数了。我们继续往下看（读者还是先休息一会儿吧！）



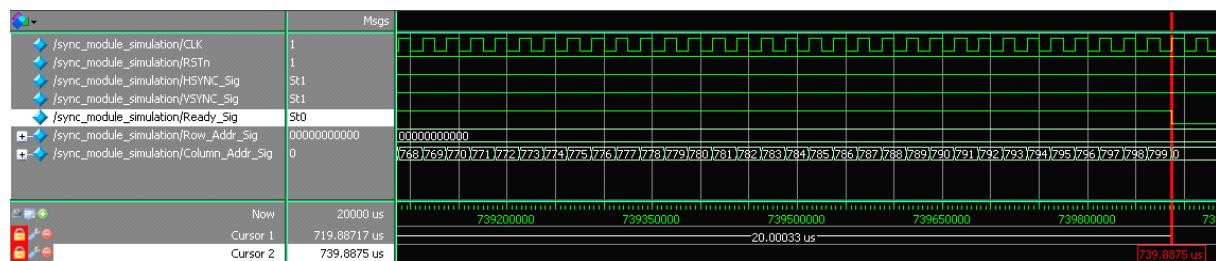
上图是 VSYNC_Sig q 段的第一个行像素。我们知道在 HSYNC_Sig 的 C 段，它包含了 800 个列像素(0~799)，它表示是当前 x 地址。但是在上面的仿真图中，Column_Addr_Sig (x 地址)反映出计数的最终结果是 800，而不是 799 (0~799 而不是 0~800)？它多计数 1 个列像素了（计数到 800）。

然后再稍微注意一下 Ready_Sig，它应该是 Column_Addr_Sig 计数到 799 之后拉低才合理吗？

```
47.         else if( ( Count_H >= 11'd216 && Count_H < 11'd1017 ) &&
48.             ( Count_V >= 11'd27 && Count_V < 11'd627 ) )
```

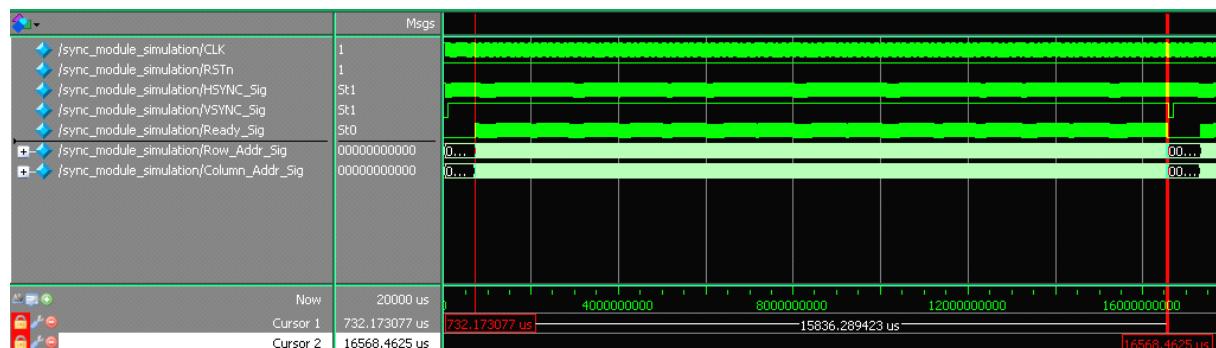
```
47.         else if( ( Count_H >= 11'd216 && Count_H < 11'd1016 ) &&
48.             ( Count_V >= 11'd27 && Count_V < 11'd627 ) )
```

看来问题是发生在第 47 行的 Count_H < 11'd1017，笔者将它修改为 Count_H < 11'd1016，然后再一次启动仿真。

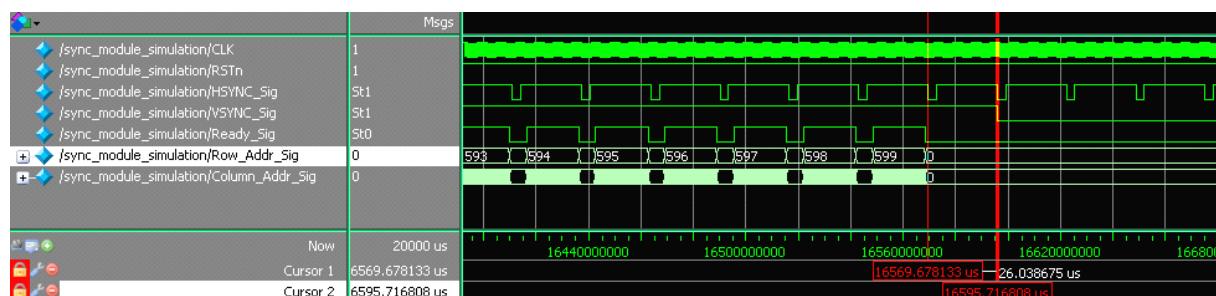


上图是经过修改 47 行的仿真结果，当 Column_Addr_Sig 计数到 799 的时候，它停止计数，并且 Ready_Sig 拉低。原理上，只要 VSYNC_Sig 的 q 段第一个行像素调式成功，接下来的 599 个行像素也是一样的结果。

到目前为止，我们已经完成 HYSNC_Sig 的 a~d 段和 VSYNC_Sig 的 o~q 段。剩下的只有 VSYNC_Sig 的 r 段。



上图是经过修改 47 行的仿真结果。其中 C1~C2 是 VSYNC_Sig 的 q 段，时间大约是 15840us。结果符合要求。



上图是 VSYNC_Sig q 段之后的 r 段。根据原理 r 段占 1 个行像素，亦即 $1056 * 25ns$ ，时间大约是 26.4us。当 VSYNC_Sig q 段之后，这也表示一帧 800 x 600 的扫描已经完成。（注意，Row_Addr_Sig 计数到 599 之后，Ready_Sig 就拉低）

在这里，我们可以说 sync_module.v 的调试和优化已经完成。

sync_module.v (after)

```
1. module sync_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig, Ready_Sig,
5.     Column_Addr_Sig, Row_Addr_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Ready_Sig;
13.    output [10:0]Column_Addr_Sig;
14.    output [10:0]Row_Addr_Sig;
15.
16.    ****
17.
18.    reg [10:0]Count_H;
19.
20.    always @ ( posedge CLK or negedge RSTn )
21.        if( !RSTn )
22.            Count_H <= 11'd0;
23.        else if( Count_H == 11'd1056 )
24.            Count_H <= 11'd0;
25.        else
26.            Count_H <= Count_H + 1'b1;
27.
28.    ****
29.
30.    reg [10:0]Count_V;
31.
32.    always @ ( posedge CLK or negedge RSTn )
33.        if( !RSTn )
34.            Count_V <= 11'd0;
35.        else if( Count_V == 11'd628 )
36.            Count_V <= 11'd0;
37.        else if( Count_H == 11'd1056 )
38.            Count_V <= Count_V + 1'b1;
39.
40.    ****
41.
```

```
42.      reg isReady;
43.
44.      always @ ( posedge CLK or negedge RSTn )
45.          if( !RSTn )
46.              isReady <= 1'b0;
47.          else if( ( Count_H >= 11'd216 && Count_H < 11'd1016 ) &&           // ( * )
48.                  ( Count_V >= 11'd27 && Count_V < 11'd627    ))    // ( * )
49.              isReady <= 1'b1;
50.          else
51.              isReady <= 1'b0;
52.
53.      ****
54.
55.      assign VSYNC_Sig = ( Count_V <= 11'd3 ) ? 1'b0 : 1'b1;    // ( * )
56.      assign HSYNC_Sig = ( Count_H <= 11'd128 ) ? 1'b0 : 1'b1;
57.      assign Ready_Sig = isReady;
58.
59.
60.      ****
61.
62.      assign Column_Addr_Sig = isReady ? Count_H - 11'd217 : 11'd0;
63.      assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0;    // ( * )
64.
65.      ****
66.
67. endmodule
```

上面是修改过后的 .v 文件，在修改的地方添加了 (*), 亦即, 47, 48, 55, 63 行。

实验二十六说明:

实验二十六中，笔者通过仿真的输出（波形图），一步一步对 sync_module.v 调式，直到最终 sync_module.v 的输出达到可接受的范围。

实验二十六结论:

有时候笔者觉得很讽刺的是，既然 Verilog HDL 语言既然有验证的部分的存在，验证语言顶多只是写写激励文件而已，反而要了解波形图，还是需要用到建模时的一套思想。笔者之所以故意选 sync_module.v 作为仿真对象就想表达一点“仿真最重要的不是学会编辑什么激励文件，反之看懂波形图和代码之间关系更重要”。

7.2 迟了一步的数据

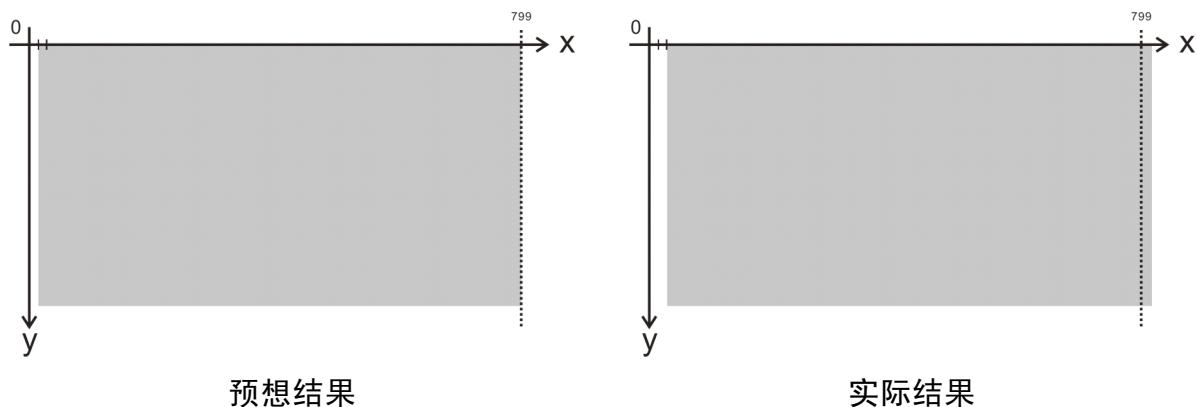
在前一章节中，我们优化了 sync_module.v 模块。在这一章节之中，我们将实验二十六的 sync_module.v 模块与《Verilog HDL 那些事儿-建模篇》中实验九之一的 vga 模块里的 vga_control_module.v，组合成为简单的 vga 模块并且执行仿真，然后进一步优化。

实验二十七：vga 模块仿真。

我们稍微回忆一下《Verilog HDL 那些事儿-建模篇》中实验九之一的 vga 模块里的 vga_control_module.v 的功能：

vga_control_module.v 在 vga 模块中，它是负责图像显示的工作。然而在《Verilog HDL 那些事儿-建模篇》中实验九之一里，它是负责显示 高为 10(y0~y9)，长为 799(x1~x799) 的矩形。

不知道读者是否记得，在《Verilog HDL 那些事儿-建模篇》中实验九之一里，sync_module.v 和 vga_control_module.v 使用了同样的时钟频率。其实这一举动使得 10 x 799 的矩形，向右偏移一个 x 地址。换句话说，该矩形的原显示位置是从 x1 开始，向右偏移一个 x 地址之后，使得该矩形从 x2 开始显示。



上面左图时我们预想的结果，右图是实际的结果。在右图中，超过虚线的部分表示显示失败。那么以上所发生的问题，如何从输出（波形图）中了解到呢？我们先来刷新刷新对 vga_control_module.v 内容的认识。

vga_control_module.v

1. module vga_control_module
 2. (
-

```

3.     CLK, RSTn,
4.     Ready_Sig, Column_Addr_Sig, Row_Addr_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.     input CLK;
8.     input RSTn;
9.     input Ready_Sig;
10.    input [10:0]Column_Addr_Sig;
11.    input [10:0]Row_Addr_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.   *****/
17.
18.   reg isRectangle;
19.
20.   always @ ( posedge CLK or negedge RSTn )
21.     if( !RSTn )
22.       isRectangle <= 1'b0;
23.     else if( Column_Addr_Sig > 11'd0 && Row_Addr_Sig < 11'd100 )
24.       isRectangle <= 1'b1;
25.     else
26.       isRectangle <= 1'b0;
27.
28.   *****/
29.
30.   assign Red_Sig = Ready_Sig && isRectangle ? 1'b1 : 1'b0;
31.   assign Green_Sig = Ready_Sig && isRectangle ? 1'b1 : 1'b0;
32.   assign Blue_Sig = Ready_Sig && isRectangle ? 1'b1 : 1'b0;
33.
34.   *****/
35.
36.
37. endmodule

```

在上面的内容中，第 23 行的 if 条件决定了该矩形的面积和显示位置。其中 Coum_Addr_Sig > 11'd0 表示矩形从地址 x1 开始显示，长度为 1~799，Row_Addr_Sig < 11'd100 表示矩形从 y0 开始显示，长度为 0~99。

请注意矩形的表示是以 isRectangle 标志寄存器来指示（注意：在“流水操和建模”和“模块的沟通”章节中，笔者都说过寄存器至少要一个时钟的时间读取数据。这是矩形偏移的主要原因。）

env_vga_module.v

```
1. module env_vga_module
2. (
3.     CLK_Sync,
4.     CLK_Control,
5.     RSTn,
6.     VSYNC_Sig, HSYNC_Sig,
7.     Red_Sig, Green_Sig, Blue_Sig,
8.     SQ_Ready, SQ_Column, SQ_Row
9. );
10.
11.    input CLK_Sync;
12.    input CLK_Control;
13.    input RSTn;
14.    output VSYNC_Sig;
15.    output HSYNC_Sig;
16.    output Red_Sig;
17.    output Green_Sig;
18.    output Blue_Sig;
19.    output SQ_Ready;
20.    output [10:0]SQ_Column;
21.    output [10:0]SQ_Row;
22.
23.    ****
24.
25.    wire [10:0]Column_Addr_Sig;
26.    wire [10:0]Row_Addr_Sig;
27.    wire Ready_Sig;
28.
29.    sync_module U1
30.    (
31.        .CLK( CLK_Sync ),
32.        .RSTn( RSTn ),
33.        .VSYNC_Sig( VSYNC_Sig ),           // output - to U2
34.        .HSYNC_Sig( HSYNC_Sig ),          // output - to U2
35.        .Column_Addr_Sig( Column_Addr_Sig ), // output - to U2
36.        .Row_Addr_Sig( Row_Addr_Sig ),      // output - to U2
37.        .Ready_Sig( Ready_Sig )           // output - to U2
38.    );
39.
40.    ****
41.
```

```

42.      vga_control_module U2
43.      (
44.          .CLK( CLK_Control ),
45.          .RSTn( RSTn ),
46.          .Ready_Sig( Ready_Sig ),           // input - from U1
47.          .Column_Addr_Sig( Column_Addr_Sig ), // input - from U1
48.          .Row_Addr_Sig( Row_Addr_Sig ),     // input - from U1
49.          .Red_Sig( Red_Sig ),             // output - to top
50.          .Green_Sig( Green_Sig ),         // output - to top
51.          .Blue_Sig( Blue_Sig )           // output - to top
52.      );
53.
54.      ****
55.
56.      assign SQ_Ready = Ready_Sig;
57.      assign SQ_Column = Column_Addr_Sig;
58.      assign SQ_Row = Row_Addr_Sig;
59.
60.      ****
61.
62. endmodule

```

`env_vga_module.v` 是我们要仿真的环境，与《Verilog HDL 那些事儿-建模篇》中实验九之一里的 `vga_module.v` 有所不同的是，笔者拿掉 `pll_module.v`，并且将所有引脚都引出来。其中最重要的是第 3~4 行的 `CLK_Sync` 和 `CLK_Control` 的输入信号。前者是 U1 的 `.CLK` 的输入信号（31 行），后者是 U2 的 `.CLK` 的输入信号（44 行）。

（不知道是不是笔者人品的关系，每当笔者在做仿真的时候，凡是有 ip 出现的地方，仿真工作会非常的不顺利 ... 所以笔者在作仿真的时候都故意把 ip 拿掉，或者用其他办法来替代 ip。）

env_vga_module.v (40Mhz/40Mhz)

```

1. `timescale 1 ps/ 1 ps
2. module env_vga_module_simulation();
3.
4.     reg CLK_40Mhz;
5.     reg CLK_80Mhz;
6.     reg RSTn;
7.
8.     wire VSYNC_Sig;
9.     wire HSYNC_Sig;
10.    wire Red_Sig;

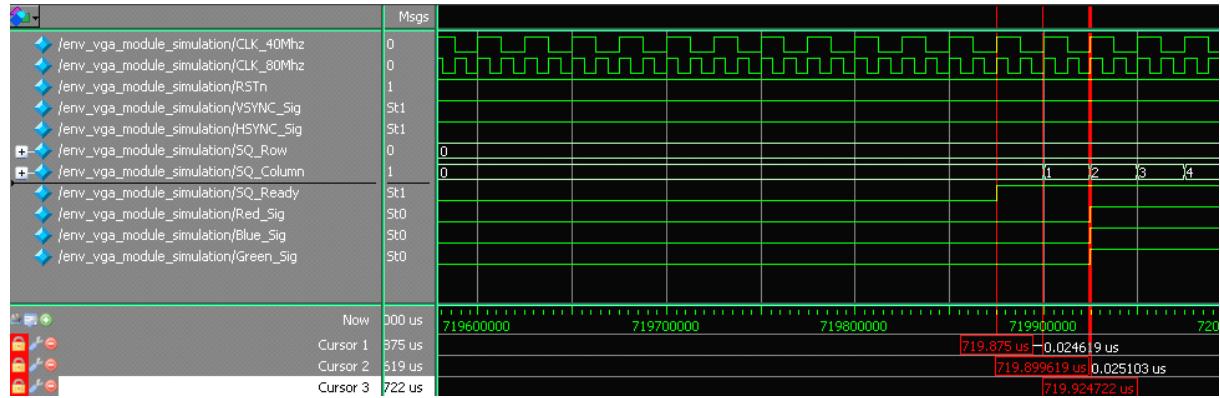
```

```
11.    wire Blue_Sig;
12.    wire Green_Sig;
13.    wire SQ_Ready;
14.    wire [10:0]SQ_Row;
15.    wire [10:0]SQ_Column;
16.
17.
18. env_vga_module U1
19. (
20.     .CLK_Sync(CLK_40Mhz),
21.     .CLK_Control(CLK_40Mhz),
22.     .RSTn(RSTn),
23.     .VSYNC_Sig(VSYNC_Sig),
24.     .HSYNC_Sig(HSYNC_Sig),
25.     .Red_Sig(Red_Sig),
26.     .Green_Sig(Green_Sig),
27.     .Blue_Sig(Blue_Sig),
28.     .SQ_Ready( SQ_Ready ),
29.     .SQ_Row( SQ_Row ),
30.     .SQ_Column( SQ_Column )
31. );
32.
33. initial
34. begin
35.     RSTn = 0; #1000000; RSTn = 1;
36. end
37.
38. initial
39. begin
40.     CLK_40Mhz = 1 ; forever #12500 CLK_40Mhz = ~CLK_40Mhz;
41. end
42.
43. initial
44. begin
45.     CLK_80Mhz = 1; forever #6250 CLK_80Mhz = ~CLK_80Mhz;
46. end
47.
48.
49.
50. endmodule
```

这个是 env_vga_module.v 的激励文本，依然也很简单。在 33~35 行复位信号的刺激，第 38~40 行是 40Mhz 时钟信号的刺激，第 43~46 行是 80Mhz 时钟信号的刺激。在这里先无视那个 CLK_80Mhz。在 20~21 行 U1 和 U2 亦即 sync_module.v 和

vga_control_module.v 使用同样的时钟信号，亦即 CLK_40Mhz。

仿真结果 (40MHz/40MHz):



上面的仿真结果是 vga_control_module.v 开始显示矩形的时候。SQ_Row , SQ_Column 和 SQ_Ready 它们分别对应 sync_module.v 的输出 Row_Addr_Sig , Column_Addr_Sig 和 Ready_Sig。在 (Cursor 省略为 C) C1 的未来，SQ_Ready 被拉高，这也证明已经进入有效的显示区域，这时候的 SQ_Row 和 SQ_Column 也开始反馈当前的显示地址 y 和 x。

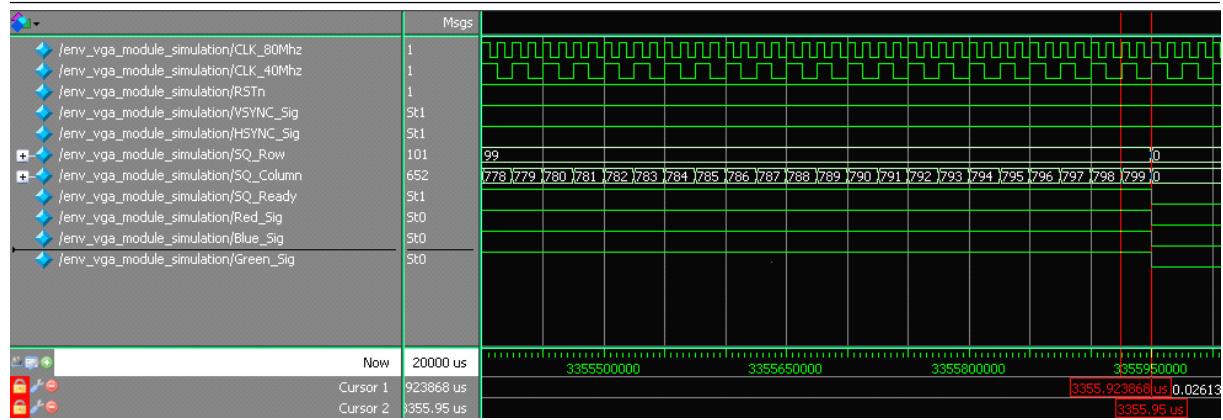
由于 vga_control_module.v 和 sync_module.v 使用同样的时钟，在 C1 这个时候 vga_control_module.v 检测 SQ_Row , SQ_Column 和 SQ_Ready 的过去值，都是逻辑 0。所以它没有“决定”什么。

在 C2 的时候，SQ_Row 不变反而 SQ_Column 已经“决定”递增为 1。在这个时候 vga_control_module.v 检测 SQ_Column 的过去值，还是是逻辑 0。它还是一样什么也没有“决定”。所以在 C2 的未来，SQ_Column 显示为 1。

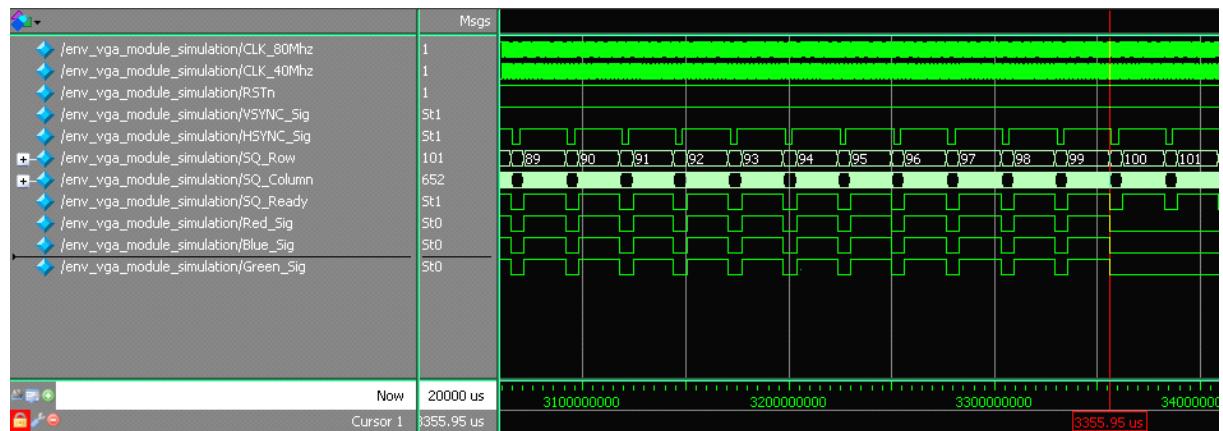
在 C3 的时候，SQ_Column 已经“决定”递增为 2。在同一个时候 vga_control_modul.v 检测到 SQ_Column 的过去值是 1。if 条件成立，它“决定”使能 isRectangle。所以在 C3 的未来 Red/Green/Blue_Sig 被拉高。(这也使得：当前 x 为 2, y 为 0 的时候矩形才开始显示。)

上述的内容，如果简单归纳的话：

当 sync_module.v (U1) 告诉 vga_control_module.v (U2) 当前的 x 地址是 0, y 地址是 0 的时候，U2 在发呆。当 U1 告诉 U2 当前的 x 地址是 1, y 地址是 0 的时候，U2 依然在发呆。当 U1 告诉 U2 当前的 x 地址是 2, y 地址是 0 的时候，这时候的 U2 才反应过来，然后 U2 才拉高 Red/Green/Blue_Sig，事实上 U2 已经慢了一个时钟拉高 Red/Green/Blue_Sig 。最终结果使得 vga_control_module.v 显示的矩形是从 x2 开始而不是 x1 开始。



上图的仿真结果表示了，在该矩形的长度结束显示在 SQ_Column 的 799，因为在 C2 之前 Red/Green/Blue_Sig 信号都是被拉高。换句话说，矩形从 x2 开始显示到 x 799 结束显示，长度为 798，亦即有一个长度显示失败。



上图的仿真结果表示了该矩形的高度是 0~99。因为在 C1 之前 Red/Green/Blue_Sig 都被拉高。

结果我们可以如此总结：

矩形的高度显示没有问题，但是矩形的长度显示就有问题。它向右偏移了一个像素。具体的原因是 vga_control_module.v 慢了一个拍子（时钟）。

在这里读者可能会问：“到底有什么办法可以解决这个问题呢？”一个常见的办法就是，初学者们都会把 sync_module.v 和 vga_module.v 集成在一个文件里（亦即单文件主义），然后矩形的生成使用组合逻辑来驱动，我们知道组合逻辑得到的结果是“即时结果”，所以它能避免数据延迟一个时钟的问题。

如果从笔者的观点出发，这个办法虽然很简单和有效，但是这个办法非常极限。有一种更有效的办法，也是这一章节的重点，那就是“vga_control_module.v 使用更高的时钟频率”。

env_vga_module.v (40Mhz/80Mhz)

```
1. `timescale 1 ps/ 1 ps
2. module env_vga_module_simulation();
3.
4.     reg CLK_40Mhz;
5.     reg CLK_80Mhz;
6.     reg RSTn;
7.
8.     wire VSYNC_Sig;
9.     wire HSYNC_Sig;
10.    wire Red_Sig;
11.    wire Blue_Sig;
12.    wire Green_Sig;
13.    wire SQ_Ready;
14.    wire [10:0]SQ_Row;
15.    wire [10:0]SQ_Column;
16.
17.
18.    env_vga_module U1
19.    (
20.        .CLK_Sync(CLK_40Mhz),
21.        .CLK_Control(CLK_80Mhz),
22.        .RSTn(RSTn),
23.        .VSYNC_Sig(VSYNC_Sig),
24.        .HSYNC_Sig(HSYNC_Sig),
25.        .Red_Sig(Red_Sig),
26.        .Green_Sig(Green_Sig),
27.        .Blue_Sig(Blue_Sig),
28.        .SQ_Ready( SQ_Ready ),
29.        .SQ_Row( SQ_Row ),
30.        .SQ_Column( SQ_Column )
31.    );
32.
33.    initial
34.    begin
35.        RSTn = 0; #1000000; RSTn = 1;
36.    end
37.
38.    initial
39.    begin
40.        CLK_40Mhz = 1      ; forever #12500 CLK_40Mhz = ~CLK_40Mhz;
41.    end
```

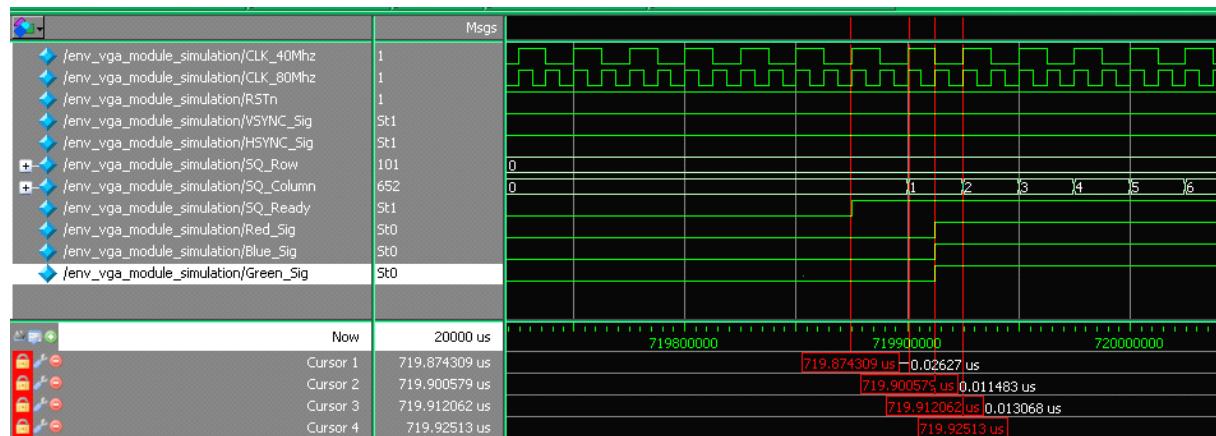
```

42.
43.     initial
44.     begin
45.         CLK_80Mhz = 1; forever #6250 CLK_80Mhz = ~CLK_80Mhz;
46.     end
47.
48.
49.
50. endmodule

```

上面是 env_vga_module.v 的激励文件。其中在 21 行 vga_control_module.v 是由 CLK_80Mhz 驱动，亦即比 sync_module.v 更高一倍的时钟。

仿真结果 ($40\text{MHz}/80\text{MHz}$):



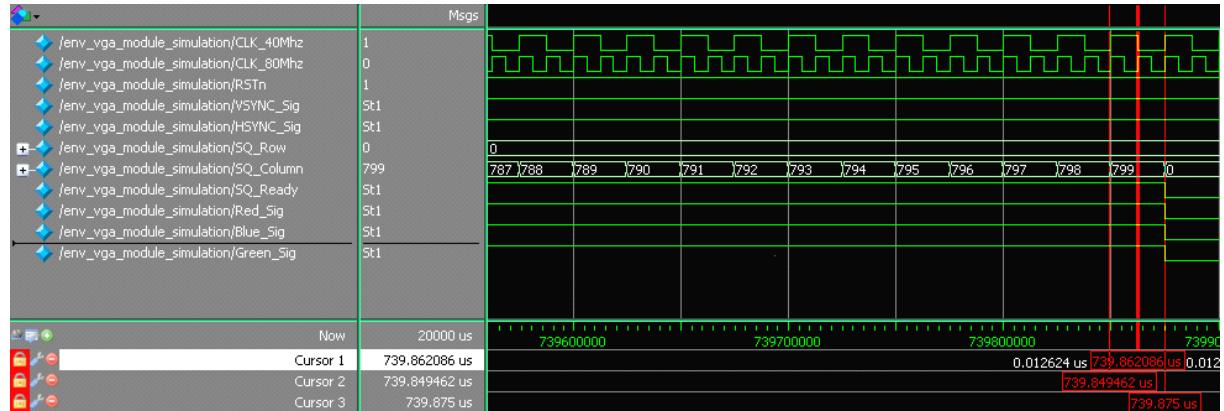
在 C1 的未来 SQ_Ready 被拉高，SQ_Row 和 SQ_Column 均反馈为 0，亦即当前显示地址 x 和 y 都是 0。在 C1~C2 之间由于 SQ_Column 为 0，所以 vga_control_module 无视。

在 C2 的未来 sync_module.v 反馈了 SQ_Column 为 1。我们知道现在 vga_control_module.v 是由 80Mhz 的时钟频率驱动着。C2~C3 之间是 sync_module.v 的前半周期，然而 vga_control_module.v 在 C2 这个时候检测 SQ_Column 的过去值是 0，它“决定”什么都不干。

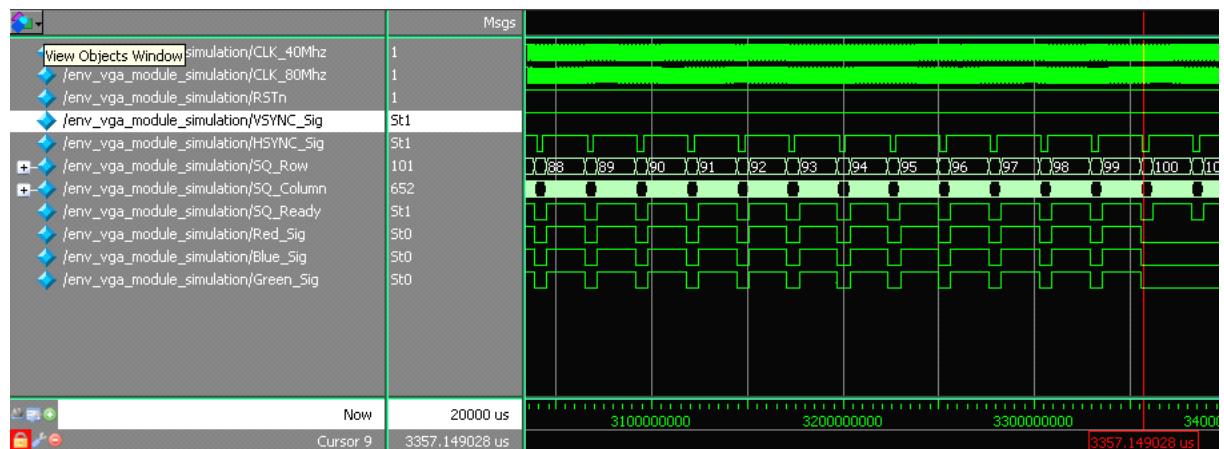
在 C3 的时候，亦即 sync_module.v 的后半周期，但是在这个时候也是 vga_control_module.v 另一个新周期（上升沿）。vga_control_module.v 在 C3 检测了 SQ_Column 的过去值是 1，if 条件成立，所以在 C3 的未来它决定拉高 Red/Green/Blue_Sig（开始显示矩形）。

在 C2, C3 和 C4 之间波形图告诉了我们，由于 vga_control_module.v 的执行频率高于 sync_module.v 一倍。每当 sync_module.v 更新输出，vga_control_module.v 在

sync_module.v 的后半周期就可以检测到 sync_module.v 更新的值，然后“决定”干些什么。由此一来，我们可以避免数据延迟的问题。换句话说，就是 sync_module.v 在反馈当前的 x 地址为 1 的时候，Reg/Green/Blue_Sig 都被拉高。



上图的仿真结果，vga_control_module.v 显示的矩形长度最长为 799。



上图仿真结果告诉我们，矩形的高度达到 99。

在这里我们可以如此总结：

由于 vga_control_module.v 的执行频率比 sync_module.v 的执行频率高一倍，亦即 2 倍。sync_module.v 的输出一旦更新，vga_control_module.v 就会在 sync_module.v 的后半周期检测到，并且做出“决定”。实际上这办法还不足完美，vga_control_moduel.v 只用 sync_module.v 的后半周期去更新和保留 Red/Green/Blue_Sig 的输出。

如果 vga_control_module.v 可以使用 sync_module 的 3/4 时间去更新和保留 Red/Green/Blue_Sig 输出的话，Red/Green/Blue_Sig 的输出会更稳定，亦即 vga_control_module.v 需要比 sync_module.v 高出 4 倍的执行频率。

实验二十七说明:

之所以会发生矩形移位显示是因为 sync_module.v 输出的数据 vga_control_module.v 迟了一个时钟读取，具体的原因是 sync_module.v 和 vga_control_module.v 使用了相同的时钟频率。当 vga_control_module.v 提高了 2 倍于 sync_module.v 的执行频率以后，矩形移位（数据延迟）的问题就解决了。

实验二十七结论:

这一章节的实验，笔者只是简单的提高 vga_control_module.v 的时钟频率就可以解决数据延迟的问题，其中笔者没有修改过 vga_control_module.v。解决问题的过程也只是在波形图（输出）了解到一些存有问题的信息。透过这些带信息从而知道 sync_module.v 和 vga_control_module.v 之间的沟通出现问题（数据延迟）了。

在这里，笔者稍微强调一下。实验本身的目的不是调式模块，而是如何从波形图中了解模块的沟通，从而发现存在问题的信息。

7.3 即时结果和非即时结果

关于阻塞赋值“=”和非阻塞赋值“<=”值，虽然它们的资料都是漫天飞了，但是还有一些小细节笔者需要详细探讨。

如果用笔者的话去定义非阻塞赋值“<=”，非阻塞赋值如果在 always 区域里，它有“时间点”的概念，求得的结果都是在该时间点的未来（非即时结果）。如果非阻塞赋值不在 always 区域里，它就和“小于等于”运算符没有什么区别。阻塞赋值“=”，无论它是否有没有在 always 区域里，它都是不遵守“时间点”的概念，它求得的结果都是“即时结果”。

初学者常常都会误会“<=”是 always 区域里专用的赋值运算符，其实“=”也可以使用在 always 区域里。但是过多的“=”出现会使得 RTL 级的设计乱了套。如果要在 RTL 级设计里区运用好“<=”和“=”，前提条件就是要掌握好基础。

接下来我们用要做的工作是，在仿真的过程中观察“<=”和“=”对仿真对象（模块）带来的反应。

实验二十八：即时结果的需要

在这一个实验中笔者引用《Verilog HDL 那些事儿-建模篇》实验十九演示中一段控制程序。

gm_control_module.v (before)

```
1. module gm_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     output [3:0]SQ_i,
7.     output [4:0]SQ_X,
8.     output [6:0]SQ_Y,
9.     output [6:0]SQ_Addr,
10.    output SQ_isCount,
11.    output SQ_isWrite
12.
13. );
14.
15.     *****/
```

```
16.
17.     parameter T1US = 7'd80;
18.
19.     /*****/
20.
21.     reg [6:0]C1;
22.     reg [9:0]rTimes;
23.     reg isCount;
24.
25.     always @ ( posedge CLK or negedge RSTn )
26.         if( !RSTn )
27.             C1 <= 7'd0;
28.         else if( C1 == T1US )
29.             C1 <= 7'd0;
30.         else if( isCount )
31.             C1 <= C1 + 1'b1;
32.         else
33.             C1 <= 7'd0;
34.
35.     /*****
36.
37.     reg [9:0]CUS;
38.
39.     always @ ( posedge CLK or negedge RSTn )
40.         if( !RSTn )
41.             CUS <= 10'd0;
42.         else if( CUS == rTimes )
43.             CUS <= 10'd0;
44.         else if( C1 == T1US )
45.             CUS <= CUS + 1'b1;
46.
47.     /*****
48.
49.     reg [3:0]i;
50.     reg [6:0]Y;
51.
52.     always @ ( posedge CLK or negedge RSTn )
53.         if( !RSTn )
54.             Y <= 7'd0;
55.         else
56.             case( i )
57.
58.                 0 : Y <= 7'd0;
59.                 2 : Y <= 7'd16;
```

```
60.          4 : Y <= 7'd32;
61.          6 : Y <= 7'd48;
62.          8 : Y <= 7'd64;
63.         10: Y <= 7'd80;
64.
65.      endcase
66.
67.  *****/
68.
69. reg [6:0]rAddr;
70. reg [4:0]X;
71. reg isWrite;
72.
73. always @ ( posedge CLK or negedge RSTn )
74.   if( !RSTn )
75.     begin
76.       i <= 4'd0;
77.       rAddr <= 7'd0;
78.       X <= 5'd0;
79.       isWrite <= 1'b0;
80.       isCount <= 1'b0;
81.       rTimes <= 10'd100;
82.     end
83.   else
84.     case ( i )
85.
86.       0, 2, 4, 6, 8, 10:
87.       if( X == 16 ) begin rAddr <= 7'd0, X <= 5'd0; isWrite <= 1'b0; i <= i + 1'b1; end
88.       else begin rAddr <= Y + X; X = X + 1'b1; isWrite <= 1'b1;end
89.
90.       1, 3, 5, 7, 9, 11:
91.       if( CUS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
92.       else begin rTimes <= 10'd250; isCount <= 1'b1; end
93.
94.       12:
95.       i <= 4'd0;
96.
97.     endcase
98.
99. *****/
100.
101. assign SQ_i = i;
102. assign SQ_X = X;
103. assign SQ_Y = Y;
```

```
104.     assign SQ_Addr = rAddr;
105.     assign SQ_isCount = isCount;
106.     assign SQ_isWrite = isWrite;
107.
108.     *****/
109.
110. endmodule
```

上面的 .v 文件是引用《Verilog HDL 那些事儿 - 建模篇》实验十九演示 vga_interface_demo.v 中的一段控制程序（笔者稍微修改了）。控制程序的功能很简单，就是每隔 250us，向 vga 接口的 RAM 写入 16 words x 16 bits 的数据。

第 6~11 行，笔者将 i, X, Y, rAddr, isCount, isWrite 全部引出来。该文件不再使用 40Mhz 的时钟频率，而是使用 80Mhz 的时钟频率（没有特别的理由）。此外，笔者还将 ms 级的延迟修改为 us 级的延迟。第 17 行是 1us 的常量声明。

第 21~47 行分别是 1us 定时器和 us 级计数器。第 49~65 行是根据不同的步骤 i 更新偏移量 Y。（注意：此时 Y 的赋值是使用非阻塞赋值）

第 69~97 行是控制程序的核心部分，当步骤 i 是偶数的时候，Y 更新偏移量，然后拉高 isWrite，并且输出 Y 的 16 个递增地址。当步骤 i 是奇数的时候就延迟 250us。

gm_control_module.vt

```
1. `timescale 1 ps/ 1 ps
2. module gm_control_module_simulation();
3.
4.     reg CLK_80Mhz;
5.     reg RSTn;
6.
7.     wire [3:0]SQ_i;
8.     wire [4:0]SQ_X;
9.     wire [6:0]SQ_Y;
10.    wire [6:0]SQ_Addr;
11.    wire SQ_isCount;
12.    wire SQ_isWrite;
13.
14.    *****/
15.
16.    gm_control_module U1
17.    (
18.        .CLK(CLK_80Mhz),
19.        .RSTn(RSTn),
```

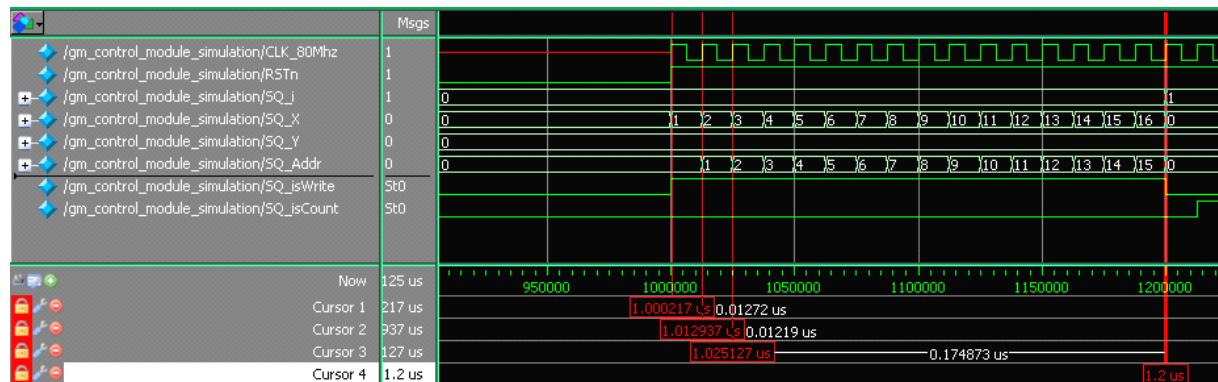
```

20.      .SQ_Addr(SQ_Addr),
21.      .SQ_X(SQ_X),
22.      .SQ_Y(SQ_Y),
23.      .SQ_i(SQ_i),
24.      .SQ_isCount(SQ_isCount),
25.      .SQ_isWrite(SQ_isWrite)
26. );
27.
28. ****
29.
30. initial
31. begin
32.     RSTn = 0; #1000000; RSTn = 1;
33.     CLK_80Mhz = 1; forever #6250 CLK_80Mhz = ~CLK_80Mhz;
34. end
35.
36. endmodule

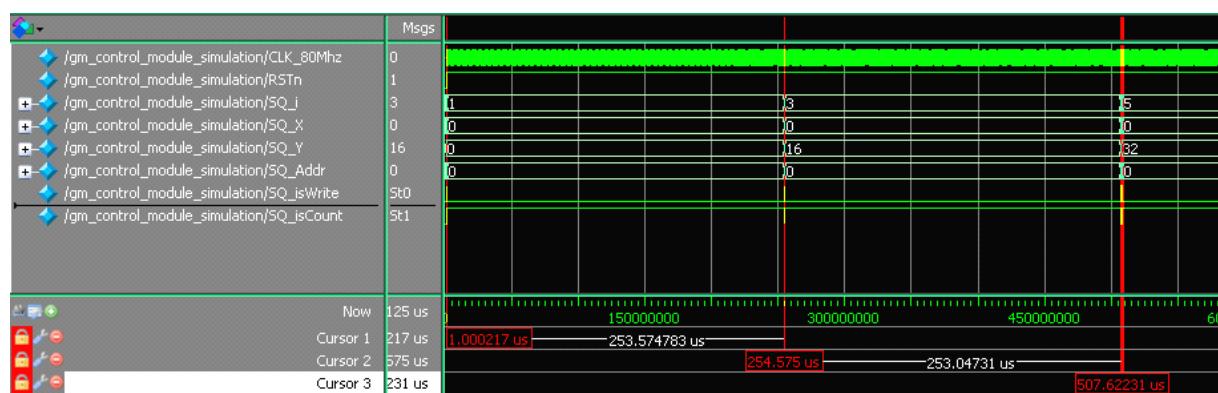
```

上面是 gm_control_module.v 的激励文件, 其中在第 1 行时间的刻度设置为 ps。在 30~34 行是复位信号 1us 和时钟信号 80Mhz 的刺激。

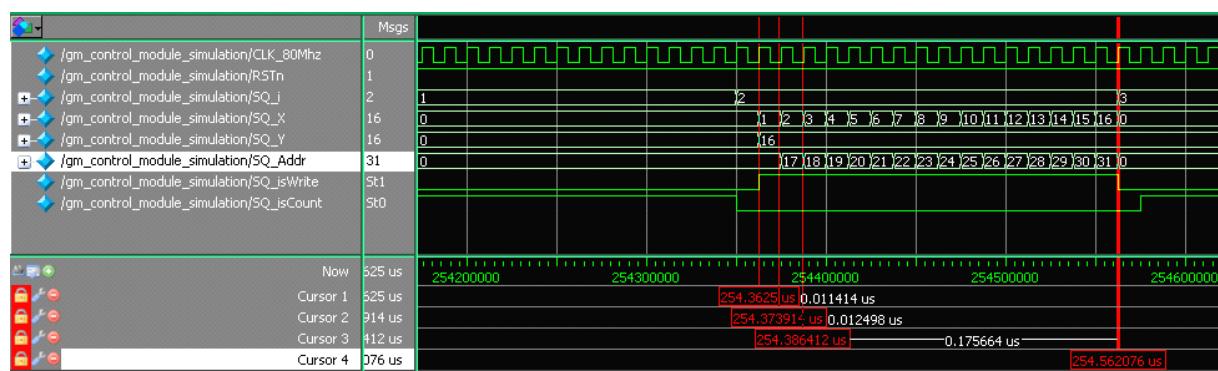
仿真结果 (*Before*) :



上图的仿真结果是 .v 文件在步骤 i 等于 0 的时候, 亦即第一次更新 Y 偏移量。(Cursor 省略为 C) C1~C4 是 SQ_Y 输出的 16 个递增地址。在 C1 的过去, 由于初始化的关系 Y 的值是 0, 所以不影响 16 个递增地址的输出, 亦即 SQ_Addr 的输出是从 0~15。



上图仿真结果表示了，每一次 Y 偏移量更新都需要延迟 250us。

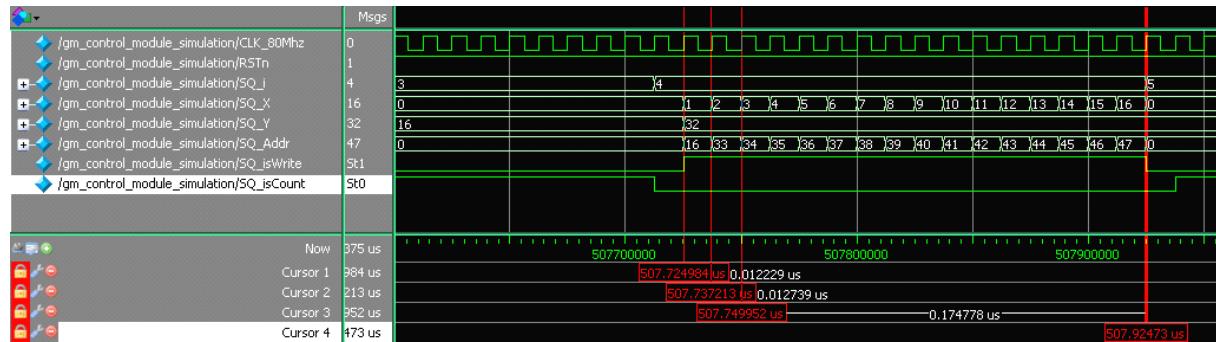


上图的仿真结果是当 .v 文件在步骤 2 的时候。C1~C4 表示了第二次 Y 偏移量输出和更新。在 C1 的时候 .v 文件检测到 SQ_i 的过去值是 2，所以它“决定”更新 Y 偏移量,也就是说 Y 偏移量的更新发生在 C1 的未来。

咦？SQ_Addr 的第一个输出应该是从 16 开始才对嘛（注意 SQ_Addr 在 C1 的未来）？为什么会是 0？

由于 Y 偏移量的更新是遵守“时间点”的概念（使用非阻塞赋值的关系），在 C1 的时候 SQ_Addr 取得的 Y 是 SQ_Y 在 C1 的过去值，亦即 0。在 C1 的未来，SQ_Addr 的输出是 $rAddr \leq Y + X$ ，（X 同样是取得 SQ_X 在 C1 的过去值，亦即 0） $rAddr \leq 0 + 0$ ，结果是 0。

在 C2 的时候，SQ_Addr 取得是 SQ_Y 在 C2 的过去值，亦即 16，同样 SQ_Addr 取得的 X 是 SQ_X 在 C2 的过去值，亦即 1。所以 C2 的未来，SQ_Addr 的输出是 $rAddr \leq 16 + 1$ ，结果是 17。接下来 SQ_Addr 的 14 个递增输出都是正确。



上图仿真结果是 .v 文件在步骤 4 的时候，亦即第三次的 Y 偏移量更新。同样的问题也发生在这里。C1~C4 表示 SQ_Addr 输出的 16 个递增地址。在 C1 的时候，由于 SQ_Addr 取得的 Y 是 SQ_Y 在 C1 的过去值，亦即 16（第二次更新 Y 偏移量后的残留结果）。在同一个时间 SQ_Addr 也取得 SQ_X 在 C1 的过去值，亦即 0。所以在 C1 的未来 SQ_Addr 的输出是 $rAddr \leq 16 + 0$ ，结果是 16 而不是 32。

在 C2 的时候，Y 已经更新完毕，SQ_Addr 取得的 Y 是 SQ_Y 在 C2 的过去值，亦即 32。在同一个时候 SQ_Addr 取得的 X 是 SQ_X 在 C2 的过去值，亦即 1。所以在 C2 的未来，SQ_Addr 的输出是 $, rAddr \leq 32 + 1$ ，结果是 33。

SQ_Addr 其后的 14 个递增输出地址都是正常。

接下来第 4~6 次 Y 偏移量的更新，SQ_Addr 第一个的输出地址都会不正常。读者可能要问“到底是发生了什么事？”原因是 Y 在更新偏移量的时候，它使用非阻塞赋值的关系，非阻塞赋值是遵守“时间点”的概念，取得的结果是“非即时结果”。所以 SQ_Addr 在读取第一个 SQ_Y 的值的时候都慢了一个时钟。

要解决这个问题很简单，就是在 .v 文件里将 Y 所使用的非阻塞赋值“ \leq ”改为阻塞赋值“=”即可。

gm_control_module.v (after)

```

1. module gm_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     output [3:0]SQ_i,
7.     output [4:0]SQ_X,
8.     output [6:0]SQ_Y,
9.     output [6:0]SQ_Addr,
```

```
10.      output SQ_isCount,
11.      output SQ_isWrite
12.
13. );
14.
15.      *****/
16.
17. parameter T1US = 7'd80;
18.
19.      *****/
20.
21. reg [6:0]C1;
22. reg [9:0]rTimes;
23. reg isCount;
24.
25. always @ ( posedge CLK or negedge RSTn )
26.     if( !RSTn )
27.         C1 <= 7'd0;
28.     else if( C1 == T1US )
29.         C1 <= 7'd0;
30.     else if( isCount )
31.         C1 <= C1 + 1'b1;
32.     else
33.         C1 <= 7'd0;
34.
35.      *****/
36.
37. reg [9:0]CUS;
38.
39. always @ ( posedge CLK or negedge RSTn )
40.     if( !RSTn )
41.         CUS <= 10'd0;
42.     else if( CUS == rTimes )
43.         CUS <= 10'd0;
44.     else if( C1 == T1US )
45.         CUS <= CUS + 1'b1;
46.
47.      *****/
48.
49. reg [3:0]i;
50. reg [6:0]Y;
51.
52. always @ ( posedge CLK or negedge RSTn )
53.     if( !RSTn )
```

```

54.           Y <= 7'd0;
55.       else
56.           case( i )
57.
58.               0 : Y = 7'd0;
59.               2 : Y = 7'd16;
60.               4 : Y = 7'd32;
61.               6 : Y = 7'd48;
62.               8 : Y = 7'd64;
63.               10: Y = 7'd80;
64.
65.       endcase
66.
67. /******
68.
69. reg [6:0]rAddr;
70. reg [4:0]X;
71. reg isWrite;
72.
73. always @ ( posedge CLK or negedge RSTn )
74.     if( !RSTn )
75.         begin
76.             i <= 4'd0;
77.             rAddr <= 7'd0;
78.             X <= 5'd0;
79.             isWrite <= 1'b0;
80.             isCount <= 1'b0;
81.             rTimes <= 10'd100;
82.         end
83.     else
84.         case ( i )
85.
86.             0, 2, 4, 6, 8, 10:
87.                 if( X == 16 ) begin rAddr <= 7'd0; X <= 5'd0; isWrite <= 1'b0; i <= i + 1'b1; end
88.                 else begin rAddr <= Y + X; X = X + 1'b1; isWrite <= 1'b1;end
89.
90.             1, 3, 5, 7, 9, 11:
91.                 if( CUS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
92.                 else begin rTimes <= 10'd250; isCount <= 1'b1; end
93.
94.             12:
95.                 i <= 4'd0;
96.
97.         endcase

```

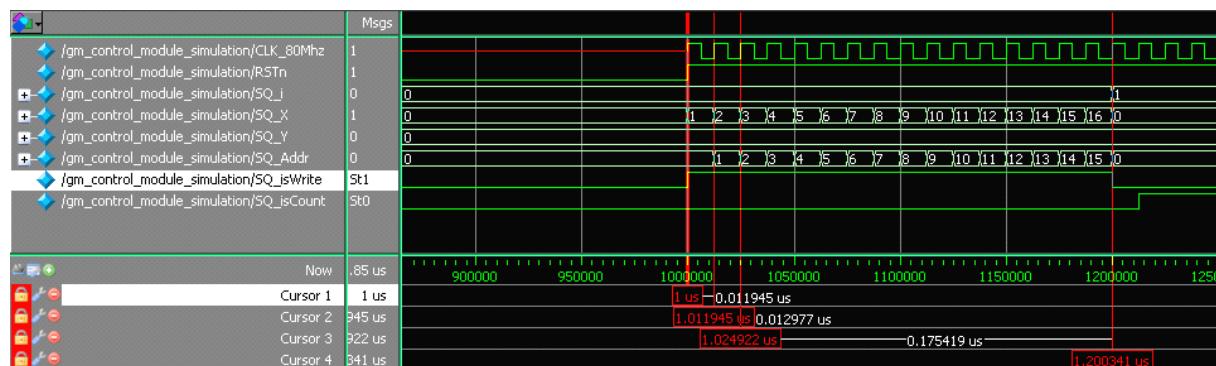
```
98.  
99.      ****  
100.  
101.     assign SQ_i = i;  
102.     assign SQ_X = X;  
103.     assign SQ_Y = Y;  
104.     assign SQ_Addr = rAddr;  
105.     assign SQ_isCount = isCount;  
106.     assign SQ_isWrite = isWrite;  
107.  
108.      ****  
109.  
110. endmodule
```

上面的 .v 文件是经过修改的 gm_control_module.v。在 58~63 行中，将“`<=`”都改为“`=`”。其余的仍然和之前的一样。

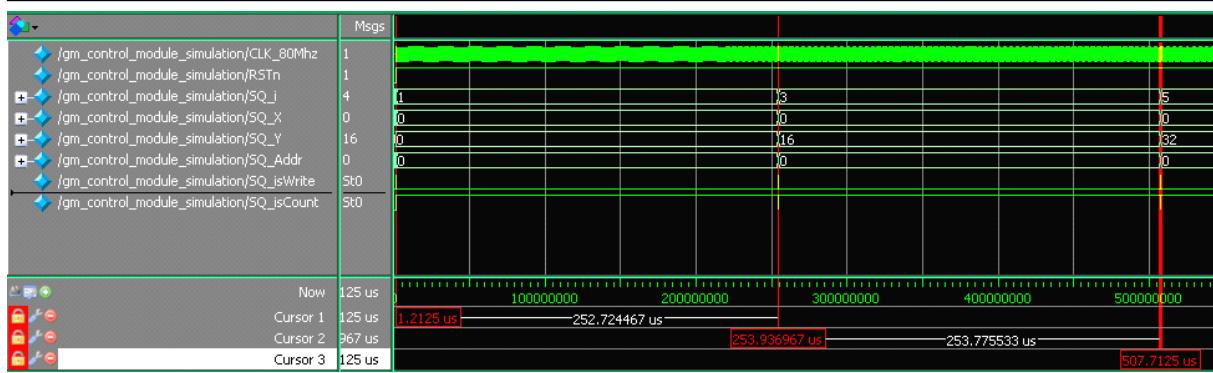
gm_control_module.vt

使用同样的激励文本。

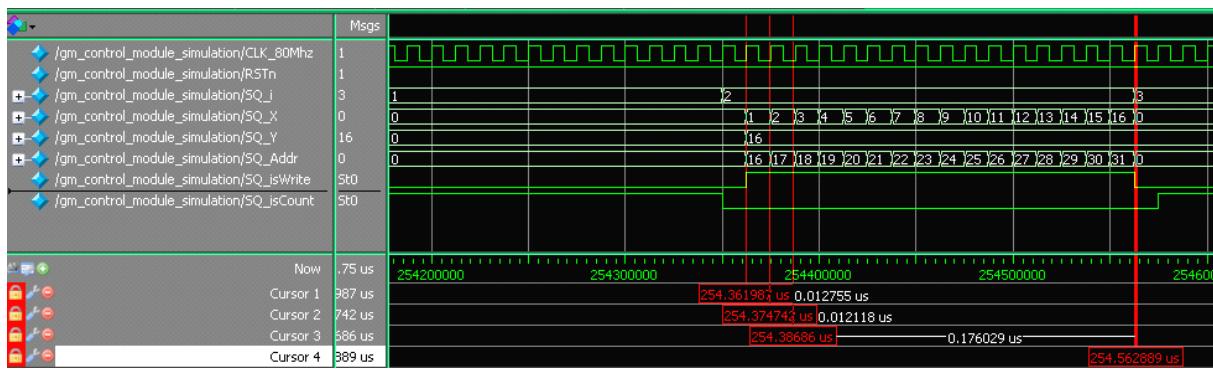
仿真结果 (*After*):



上图的仿真结果是 .v 文件在步骤 0 的时候。C1~C4 是 SQ_Addr 输出的 16 个递增地址。无论 Y 有没有取得“既是结果”，都不会影响 SQ_Addr 的输出。因为在初始化的时候，Y 已经被初始化为 0。

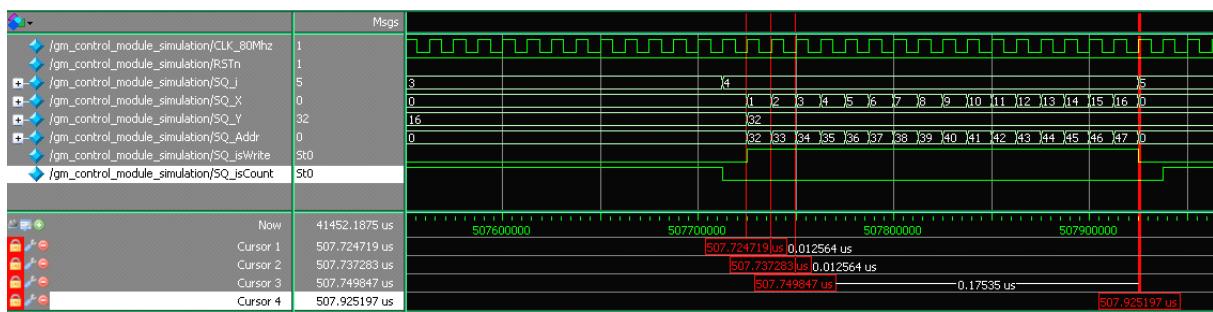


上图仿真结果表示，每一次 Y 更新时候的延迟。时间大约是 250us。



上图的仿真结果是 .v 文件第二次更新 Y 偏移量的时候。C1~C4 是 SQ_Addr 输出的十六个递增地址。在 C1 的时候 .v 文件检测到 SQ_i 的过去值是 2, Y “即时”更新为 16。在同一个时候，SQ_Addr 取得的 Y 再也不是 SQ_Y 的过去值而是 Y 的“即时结果”，亦即 16。反之 SQ_Addr 取得的 X 是 SQ_X 的过去值，亦即 0。所以在 C1 的未来 SQ_Addr 的输出是 $rAddr \leq 16 + 0$ ，结果是 16。

SQ_Addr 在接下来的 15 个递增输出地址都正常。



上图的仿真结果是 .v 文件第三次更新 Y 偏移量。C1~C4 是 SQ_Addr 输出的十六个递增地址。在 C1 的时候 .v 文件检测到 SQ_i 的过去值是 4, Y “即时”更新为 32。在同一个时候，SQ_Addr 取得的 Y 再也不是 SQ_Y 的过去值而是 Y 的“即时结果”，亦即 32。反之 SQ_Addr 取得的 X 是 SQ_X 的过去值，亦即 0。所以在 C1 的未来 SQ_Addr 的输出是 $rAddr \leq 32 + 0$ ，结果是 32。

SQ_Addr 在接下来的 15 个递增输出地址都正常。

实验二十八说明:

在实验二十八中，由于 Y 偏移量更新是使用非阻塞赋值 “`<=`” 的关系，亦即遵守“时间点”的概念。结果使得 SQ_Addr 的第一个输出地址带来错误（应该说是慢了一步）。当我们把非阻塞赋值 “`<=`” 更改为阻塞赋值 “`=`”，SQ_Addr 取得的 Y 再也不是在某个时钟 SQ_Y 的过去值，而是即时结果。从而解决 SQ_Addr 的第一个输出地址的错误。

同样的问题也发生在：

《Verilog HDL 那些事儿-建模篇》实验二十一演示 lcd_interface_demo.v 的 49~50 行。

```
41.    reg [1:0]Z;
42.
43.    always @ ( posedge CLK or negedge RSTn )
44.        if( !RSTn )
45.            Z <= 2'd0;
46.        else
47.            case ( i )
48.
49.                0: Z <= 2'd0;
50.                2: Z <= 2'd1;
51.
52.            endcase
```

《Verilog HDL 那些事儿-建模篇》实验二十四 page_control_module.v 的 44~55 行。

```
42.    case( Menu_Sig )
43.
44.        12'b100_000_000_000: Z <= 3'd0;
45.        12'b010_000_000_000: Z <= 3'd1;
46.        12'b001_000_000_000: Z <= 3'd2;
47.        12'b100_100_000_000: Z <= 3'd3;
48.        12'b100_010_000_000: Z <= 3'd4;
49.        12'b100_001_000_000: Z <= 3'd5;
50.        12'b010_000_100_000: Z <= 3'd3;
51.        12'b010_000_010_000: Z <= 3'd4;
52.        12'b010_000_001_000: Z <= 3'd5;
53.        12'b001_000_000_100: Z <= 3'd3;
54.        12'b001_000_000_010: Z <= 3'd4;
```

```
55.           12'b001_000_000_001: Z <= 3'd5;  
56.  
57.      endcase
```

等其他还有类似的代码，这些代码都有相似的地方，都是在更新偏移量的时候使用非阻塞赋值“`<=`”。在早期的时候，考虑到初学者对 Verilog HDL 源的不熟悉，笔者没有深入讨论，此外为了避免图像显示失误，笔者将图像的第一列和最后一列都空白处理（这做法是最简单处理手段）。

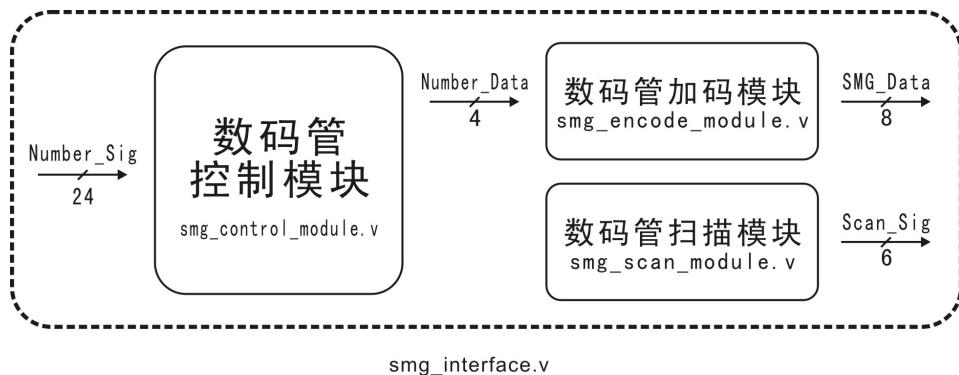
实验二十八结论：

在这一个试验中，仿真的过程和激励文本的编辑都很简单，此外笔者也没有建立什么仿真的虚拟环境。该实验的重点就是挑战 Verilog HDL 语言的基本功，看读者如何从波形图中（输出中）了解到“非阻塞赋值”对模块的沟通（输出）造成的影响。

7.4 波形图在我的脑海中

当读者把 Verilog HDL 语言掌握到某种程度的时候，用不着故意仿真，一种默认的波形图自然而然会呈现在脑海中，但是这个默认波形图的清晰度会因人而异。读者千万不要把笔者当着神棍在说荒谬的玄学，笔者所说的东西都是事实，当读者深入 Verilog HDL 语言到某个程度的时候，这种感觉就会越来越真实。

我们就以一个简单的例子来说明：



上图是《Verilog HDL 那些事儿-建模篇》实验十五中的数码管接口的图形。笔者稍微帮助读者回忆一些数码管接口的大致功能。`smg_scan_module.v` 会每隔 1ms 拉低数码管，`smg_control_module.v` 会每隔 1ms 依 `Number_Sig` 每个“四位”，将数据往 `smg_encode_module.v`。`smg_encode_module.v` 是加码模块，它会将 `Number_Data` 加码为数码管码，然而加码需要使用一个时钟。

现在我们开始来分析了：

```

20. always @ ( posedge CLK or negedge RSTn )
21.     if( !RSTn )
22.         begin
23.             rSMG <= 8'b1111_1111;
24.         end
25.     else
26.         case( Number_Data )
27.
28.             4'd0 : rSMG <= _0;
29.             4'd1 : rSMG <= _1;
30.             4'd2 : rSMG <= _2;
31.             4'd3 : rSMG <= _3;
32.             4'd4 : rSMG <= _4;
33.             4'd5 : rSMG <= _5;
34.             4'd6 : rSMG <= _6;

```

```
35.          4'd7 : rSMG <= _7;
36.          4'd8 : rSMG <= _8;
37.          4'd9 : rSMG <= _9;
38.
39.      endcase
```

上面是数码管加码模块的部分代码，其中最关键的部分是第 20~39 行，它表示了数码管加码模块在加码数据的时候，由于该模块是用寄存器来寄存加码后的数据，所以它至少需要一个时钟的消耗。

假设笔者使用的时钟频率是 20Mhz，那么 1ms 的定时常量是：

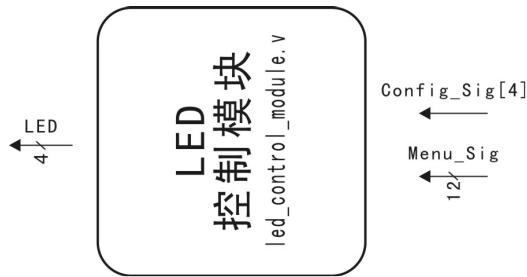
$$1\text{ms} / (1/20\text{Mhz}) = 1\text{E-3} / 50\text{E-9} = 20000$$

当一组数据驱动着 Numbr_Sig，在第一个 1ms，Number[23:20] 会被送去加码，并且第一位数码管会被使能。我们知道数码管接口是使用动态扫描，每一次的使能间隔是 1ms。这也就是说每一位数码管在使能中使用大约 20000 个时钟。

在第一个 1ms 的时候，由于加码过程用掉了 1 个时钟，所以停留在 SMG_Data 的时钟大约是 19999 而已。反之在 Scan_Sig 停留的时钟大约是 20000，比起 SMG_Data 多了一个时钟。

数码管的动态显示效果，不会因为 SMG_Data 的输出少了一个时钟而影响实际的效果。（人类的肉眼是很迟钝），所以很多时候我们用不着特意去仿真这个数码管接口，有没有仿真都不会影响实际的效果。

笔者再来举个例子：



上图是 GUI 系统中的 led_control_module.v，它的大致功能如下：

该控制模块会根据不同的 Menu_Sig 产生不同的 LED 效果：每当 Menu_Sig 更动都会影响 led_control_module.v 更新内部相关的寄存器，但是 Config_Sig[4] 必须接受一个高脉冲，LED 的效果才能更新。

下边是该模块的部分相关代码，读者们稍微刷新自己的脑袋：

```
45.      reg [11:0]F1;    // Filter 1 of Menu_Sig
46.      reg [11:0]F2;    // Filter 2 of Menu_Sig
47.
48.      always @ ( posedge CLK or negedge RSTn )
49.          if( !RSTn )
50.              begin
51.                  F1 <= 12'd0;
52.                  F2 <= 12'd0;
53.              end
54.          else
55.              begin
56.                  F1 <= Menu_Sig;
57.                  F2 <= F1;
58.              end
59.
60.      /*************************************************************************/
61.
62.      reg [2:0]Mode;
63.      reg [8:0]Delay;
64.
65.      always @ ( posedge CLK or negedge RSTn )
66.          if( !RSTn )
67.              begin
68.                  Mode <= 3'b100;
69.                  Delay <= 9'd400;
70.              end
71.          else if( F1 != F2 )
72.              case( Menu_Sig )
73.
74.                  12'b100_000_000_000: begin Mode <= 3'b100; Delay <= 9'd400; end
75.                  12'b010_000_000_000: begin Mode <= 3'b010; Delay <= 9'd400; end
76.                  12'b001_000_000_000: begin Mode <= 3'b001; Delay <= 9'd400; end
77.                  12'b100_100_000_000: begin Mode <= 3'b100; Delay <= 9'd400; end
78.                  12'b100_010_000_000: begin Mode <= 3'b100; Delay <= 9'd200; end
79.                  12'b100_001_000_000: begin Mode <= 3'b100; Delay <= 9'd100; end
80.                  12'b010_000_100_000: begin Mode <= 3'b010; Delay <= 9'd400; end
81.                  12'b010_000_010_000: begin Mode <= 3'b010; Delay <= 9'd200; end
82.                  12'b010_000_001_000: begin Mode <= 3'b010; Delay <= 9'd100; end
83.                  12'b001_000_000_100: begin Mode <= 3'b001; Delay <= 9'd400; end
84.                  12'b001_000_000_010: begin Mode <= 3'b001; Delay <= 9'd200; end
85.                  12'b001_000_000_001: begin Mode <= 3'b001; Delay <= 9'd100; end
86.
87.              endcase
```

```

88.
89.      **** // "Enter key"
90.
91.      reg [2:0]LED_Mode;
92.      reg [8:0]rTimes;
93.
94.      always @ ( posedge CLK or negedge RSTn )
95.          if( !RSTn )
96.              begin
97.                  LED_Mode <= 3'b100;
98.                  rTimes <= 9'd400;
99.              end
100.         else if( Config_Sig ) // if press down Enter key...
101.             begin
102.                 LED_Mode <= Mode;
103.                 rTimes <= Delay;
104.             end

```

第 71~81 行是一个关键的部分，当 Menu_Sig 产生变化的时候 Mode 和 Delay 寄存器的更新需要使用一个时钟的时间（使用非阻塞赋值的关系），在 Mode 和 Delay 寄存器更新之前和更新之后的这一个时钟里，如果刚刚好 Config_Sig[4] 接收一个高脉冲，那么这个高脉冲就会无效而被抵消。

因为在 Mode 和 Delay 寄存器更新之前和更新之后这一个时钟里，Mode 和 Delay 还没来得及更新，Config_Sig[4] 接收高脉冲的结果使得 LED 显示效果是 Mode 和 Delay 更新之前的数据，而不是更新之后的数据。

现实中，触发 Menu_Sig 和 Config_Sig[4] 是人为的去按下按键所造成的。除非我们人们有闪电侠的速度，按下影响 Menu_Sig 的按键，然后又按下影响 Config_Sig[4] 的按键，之间的时间间隔不到一个时钟，就可以引发上述的事情。

事实上这个事情是不可能发生的，因为人们按下影响 Menu_Sig 的按键，然后又按下影响 Config_Sig[4] 的按键，最快的时间也需要 100ms 以上。这个 100ms 已经足够 Mode 和 Delay 数据去更新数据。有没有仿真都不会影响实际的效果。

在这里，笔者举了两个例子是要告诉读者，在很多时候我们可以用不着刻意去仿真。当我们一边建模，心理的波形图就会一直变化，这一张心理的波形图会是我们最好的“显示”和“参考”。最重要的是，如果我们在建模的时候，心理有一张波形图的话，建模的思路会清晰许多，而且在建模的过程也不会那么枉然，心理比较有底。

其实，笔者真的很理解许多初学者如果一边建模没有一边仿真，心理是很没有底子的。因为我们无法观察到自己所建立的模块“哪里不舒服”或者“健康如何”等。如果用笔

者的话说,这种依赖于仿真的心情实际上是反映出自己对 Verilog HDL 语言掌握的信心。笔者不是说仿真不重要,在很多时候我们还是需要仿真用于细化模块。但是我们不可能,每当添加一行代码就仿真一次嘛

总结：

这一章笔记所强调是“反应和调试过程”，用另一句话来说，重点就是在仿真中如何看懂波形图（输出），然后对模块调试和优化。

其实这一章笔记的大部分内容，笔者已经有形或无形的讨论过，尤其是“模块的沟通”这一环。这一章节笔记的内容，笔者只不过在“模块的沟通”基础上，更深入讨论而已。其中还添加模块优化和各种调试的方法。归根究底，如何看懂波形图中（输出）的信息才是这一章笔记的重点。

笔者在初期的时候，曾经很盲目的依赖仿真，如果波形图上没有出现预期的结果，笔者的心理就很没有底。现在回想起来，那种依赖仿真的心情是自己对自己掌握的 Verilog HDL 语言没有信心吧？又或者说自己的程度还不到家？说实话，初期的笔者到底能不能看懂波形图上的信息，笔者到现在还是很怀疑自己 … 不知道读者有没有同样的感觉。

就如笔者说所的那样，仿真的重点就是如何看懂波形图，在笔者的心目中，笔者把“波形图”定义为“模块的沟通和执行”的显性记录，这个记录是最清洁的。要如何编辑激励文件，如何产生刺激（各种输入）的重要都是其次。

在这里，读者可能会有这样的错觉吗：在这一章笔记的所有试验的调试过程（优化过程）看似非常简单，又或者说模块的病症很直接，所有有问题的代码，都能很快被找到。其实是不然的，这一切一切都是低级建模带来的好处。如果有读过《Verilog HDL 那些事儿-建模篇》，读者们一定会非常清楚笔者在那一本笔记中最强调的就是“模块的表达能力”和“代码风格”等这些都被许多初学者忽略的细节。这一些小细节会为后期的工作带来许多好处。

仿真工作充其量也是设计者用各种“残忍的方法”去“虐待模块”使得“模块吐出所有反应”，然后根据模块的反应来判断是不是达到预期的效果。但是，最实在也是最影响模块的，就是模块本身所包含的代码。之所以，许多初学者非常依赖仿真就是它们对 Verilog HDL 语言的掌握不好，或者没有信心。

如果我们能够把 Verilog HDL 语言掌握好，基本上仿真很简单（仿真不过是不同环境的建模而已）。在这一章笔记里，笔者从波形图中取得信息，所使用的“思想”都是和建模的时候同一个“思想”。如果读者建模掌握不错，而且对上半部分内容又足够理解的话 … 读者估计会和笔者一样，可以非常轻松从波形图中理解信息。

至于激励过程（激励文件）的“刺激”读者要选择“验证”好？还是“综合”好？这是见仁见智，没有强迫性的。如果读者是有建模技巧的底子的话，笔者还是推荐读者用综合语言来完成仿真（编辑激励文件）。因为这样作会帮助读者更能了解 Verilog HDL 语言。

结束语

终于引来这一本笔记的结束了！

这一本笔记是另于《Verilog HDL 那些事儿 - 建模篇》的深入讨论。上半部分我们加强对“步骤和时钟”的认识，后半部分我们加强对“综合和仿真”的认识。这一切的一切都是为了细化我们的模块。在前面笔者已经说过，建模过后的模块是很粗糙，虽然可以使用但是效果能不能完全发挥，需要细化后才能知道（细化的另一个意思是深入分析 Verilog HDL 语言）。

“早期的低级建模已经为模块埋下细化的种子”这一句话的意思就是说：当我们用低级建模的技巧去描述（建立）某个模块的形状之际，低级建模所要求使用者遵守的尊则，实际上已经为后期的工作做了许多准备。所以细化等后期的工作才显得那么简单和直接。但是建模技巧也不是无敌的，最终设计者还得靠自己对 Verilog HDL 语言的掌握程度，才能完全细化模块。

这本笔记除了焦距在“细化”模块以外，笔记还特地把读者们带入 Verilog HDL 语言更深入的世界，用更多不一样的角度去认识 Verilog HDL 语言。就好比笔记第三章的“流水操作和建模”，被初学者视为不容易入门的“流水操作”，我们可以巧妙的借助“步骤”的力量，从仿顺序操作的向流水操作转换，就可以很简单的完成流水操作的设计。当然笔记第三章所强调的是“使用建模来提高流水操作的表达能力”和“步骤和时钟之间的关系”。

在笔记的第四章，其中“混种建模的可能性”是笔者最感兴趣的地方。这一内容告诉我们“混种建模”是多花样的变形建模，它非常弹性从而可以解决许多挑剔的设计要求。很遗憾的是，这已经是笔记可以讨论的范围之外，故笔者才没有继续深入。毕竟第四章的重点是“模块的沟通”而不是“建模”…（啊啊！笔者不是故意要泼冷水的！）

最后一点要笔者比较顾忌的是“综合和建模”这一部分的内容。因为笔者完全颠覆初学者一般对仿真的认识。笔者心目中的仿真是以“建模的思想”为准，“建模”和“仿真”应用同样的“思想”。所以内容都不怎么提及“验证语言”，激励文件都是以“综合语言”为准。虽然笔者在每一章笔记的总结一直强调“这是笔者的一厢情愿的想法而已，没有任何强迫性…”，但是笔者还是衷心的建议读者多理解这几章笔记的内容，涵义深远，多理解对读者更有益处。（笔者希望读者可以了解笔者的苦心）。

接下来是大家最关心的问题了…读完这本笔记以后，接下的路要如何继续？笔者虽然不能决定什么，但是笔者可以建议：

- (1) 学习 NIOS II;
 - (2) 学习静态时序分析;
 - (3) 学习验证;
 - (4) 继续研究 Verilog HDL 语言;
-

如果读者问笔者接下来要走的路？笔者选择还是第四点，但是笔者没有之前那么疯狂了，笔者还有其他东西要学习，没有太多的时间专注在同一种东西。

基本上《Verilog HDL 那些事儿 - 建模篇》和《Verilog HDL 那些事儿 - 时序篇》已经把 Verilog HDL 语言覆盖得很广了。笔者也好，读者也好，如果已经把这两本笔记看完，并且消化和吸收，那么就可以用不着过于焦距在 Verilog HDL 语言了。这两本笔记的内容足够读者应付许多 Verilog HDL 语言的难题了，但是笔者不敢肯定这两本笔记可以解决所有难题。世上没有最全的资料，更何况是笔记呢？解决难题的王道是不停的更新自己。笔记可以做到的事情，就是让学习者们轻松上路和指出一些小细节而已。

在这里，笔者再一次感谢黑金动力社区，不然的话就没有《Verilog HDL 那些事儿 - 时序篇》的出现。原本笔者只是计划《Verilog HDL 那些事儿 - 建模篇》的附录而已，多亏了社区的建议，笔者才有完成《Verilog HDL 那些事儿 - 时序篇》的念头。

在结束之前，笔者要说声抱歉。笔记的第七章暴露了《Verilog HDL 那些事儿 - 建模篇》代码的不成熟。但是请读者不要忘了《Verilog HDL 那些事儿 - 建模篇》这本笔记的真正的初衷是什么？读者又在这本笔记里面学习到了什么重要的内容？很多时候笔者都有许多苦衷和苦心

如果读者对笔者的笔记有兴趣的话，笔者很建议去笔者的博客下载。如果读者想对笔者说什么的话，可以到社区里留言。

笔者博客：<http://blog.ednchina.com/akuei2>

黑金动力社区：<http://www.oshcn.com>
