

黑金动力社区系列教程

# Verilog HDL 那些事儿

**建模篇(for DB4CE15)**

黑金动力社区荣誉出品

## 书语

学习 Verilog HDL 和 FPGA 之间 ,始终会出现一组群体 ,他们都是徘徊在学习的边缘。在他们的心中一直回响着这样的一个问题 :“我在学什么 ,为什么不管我怎么学 ,我都没有实感 ... ” 没错这就是初学 Verilog HDL + FPGA 的心声。

在众多的 Verilog HDL 参考书 ,隐隐约约会会出现这样的一个“建模”。建模在 Verilog HDL 的世界里是一个重要的基础 ,笔者始终无法明白 ,为什么参考书们怎么都不甘情愿的好好描述它们。“建模”顾名思义就是“模块建立”的省略。FPGA 的逻辑资源 ,好比乐高的积木 ,要组合乐高就是需要工具 ,那 Verilog HDL 就是 FPGA 建模的工具。

Verilog HDL 作为“建模”的一个工具 ,但是没有技巧的使用它们是无法很好的发挥到它。读者们 ,曾经何时有没有为建模的规划而头疼过 ?读者们 ,曾经何时有没有为天书般的源码 ,想把头去撞墙 ?到最后的最后 ,读者们 ,曾经何时有没有冲动想用一把火把全部东西都烧掉。这些心情笔者也拥有过 ,而且笔者也干过 ,这一切的一切都只是一个原因 :

“没有建模的技巧 ... ”

网络上常说学习 Verilog HDL 就是要明白什么是 RTL 级代码 ,多参考别人写的代码 ,但是前提是“你能不能看懂别人在写什么 ,别人在设计什么 ,别人在做什么”。有一句学

习 Verilog HDL 的名言 “参考别人的代码有如半死不活的受折磨”，当你看懂别人在写什么的时候，估计在那之际你已经形成一具行尸走肉，这一切的一切都是：

## “没有建模技巧 ... ”

在这里笔者没有攻击他人的意思，笔者始终觉得一个好的设计不仅是自己看得懂，而且还要别人看得懂，设计的表达能力要直接，代码要整齐，建模有结构。

笔者一直觉得可恨，为什么建模技巧作为 Verilog HDL 的基本功，它甚至比时序分析，功能仿真来得更重要，但是却没有被重视。建模技巧的潜能是难以估计，笔者一直深信拥有建模技巧的建模，Verilog HDL 语言绝对不会亚于其他高级语言，甚至还可以超越它们。关于这一点，这一本笔记已经可以证明。

很多初学 Verilog HDL + FPGA 的朋友会成为徘徊在边缘的一群，主要原因就是他们没有掌握好建模技巧，而形成他们继续前进的一大阻碍。

在这里笔者将自己养成的建模技巧，故笔者称为“低级建模”这一建模技巧。笔者经过一段时间使用后，编辑成为一本笔记。好让许更多初学的朋友越过这一段学习的大障碍。

## 前言

说实话，在这本笔记还没有开始之前，笔者正是初到社会之际。那时候的笔者正好是徘徊在学习 Verilog HDL + FPGA 的边缘，每天早上七时工作到晚上七时，失去大量的学习热情的同时，再加上无法突破学习的障碍。笔者真的很想很想什么东西都不干，甚至放下笔者所喜爱的学习。

在学习 Verilog HDL + FPGA 的时候，笔者有尝试过把头撞墙，表演“狮子”。最后笔者一股劲将所有之前所学的资料，都一把火（Shift + Delete）通通烧掉。好似将所有有关 Verilog HDL + FPGA 的学习回归到零，那种心情笔者到现在还记忆犹新。

在工厂工作一段时间后，不知为何学习的冲动一只从心里深处涌出！笔者向自己说“好想再一次接触 Verilog HDL + FPGA 呀，就这样放弃我真的很不甘心”。到底是偶然还是冥冥之中，笔者在 ourdev 上看见了 FPGA 黑金开发板……（笔者真的很感谢黑金动力社区的 AVIC 大大，很感谢他提供了一个给笔者编写这一本笔记的平台）

笔者告诉自己，不会再犯同样的错误，要找出“障碍的原因”。于是笔者开始测试许多不同的实验，最后笔者发现到一个关键的东西，那就是“建模”。笔者为了证实自己的想法是对的，就开始针对“建模”写了一本关于 Verilog HDL 建模技巧的思路篇笔记。之后，想法越来越多，“建模技巧”也越来越成熟。

大约是 2010 年七月末 FPGA 黑金开发板开始发售了，笔者很意外的接到 AVIC 大大为 FPGA 黑金开发板写教程的要求。那时候的心情笔者真的很惊奇，因为笔者重来没有干过这样的事情。笔者写学习笔记的目的是为了更好的反馈自己，想不到会成为另一种形式的开端 ...

初头笔者一直犹豫着和担心着什么，但是笔者明白到如果某件事情发生在自己的身上就一定有它的原因，此外笔者一直还有为建模技巧写一本实例篇的打算。就这样长达 3 个月学习笔记开始编写了 ...

“低级建模”作为笔者的“建模技巧”( 建模习惯 )，笔者真的很用心去记录每一个内容。虽然文中可能会出现许多“奇怪的字眼和用词”，关于这一点笔者希望读者们不要太认真，笔者的语言能力有限，用字不好。如果文中有得罪的地方，真的很抱歉，笔者一向都是直言直语，单刀直入那种人。

akuei2 24-10-2010 上

笔者的博客：

<http://blog.ednchina.com/akuei2>

黑金动力社区：

<http://www.heijing.com>

这本笔记的适合群：

一、初学者。

二、希望从另一个角度去了解 Verilog HDL 的人们。

这一本笔记确实有点不适合入门者，怎么说呢？因为这本笔记笔者按着自己的习惯编辑，则笔者不是按着典型的方向去编辑。入门者应该先从一些权威的参考书去了解“什么是 Verilog HDL 语言”，对 Verilog HDL 语言有一定的了解后，再来看这本笔记。此外，稍微要知道 Quartus II 是什么（版本随便）。

## 目录：

书语 .....	二
前言 .....	四
目录： .....	七
第一章：我眼中的 FPGA 和 VERILOG HDL .....	十五
第二章：低级建模 - 基础知识 .....	十六
2.1 顺序操作和并行操作 .....	十六
实验一：永远的流水灯。 .....	十七
led0_module.v .....	十八
led1_module.v .....	十九
led2_module.v .....	二十
led3_module.v .....	二十一
top_module.v .....	二十二
实验一说明： .....	二十四
实验一结论： .....	二十四
2.2 倾向并行操作 .....	二十五
实验二：闪耀灯和流水灯 .....	二十七
flash_module.v .....	二十八
run_module.v .....	二十九
mix_module.v .....	三十
实验二说明： .....	三十二
实验二结论： .....	三十二
2.3 VERILOG HDL 不是“编程”是“建模” .....	三十三
2.4 听听低级建模的故事 .....	三十四
2.5 低级建模的资源 .....	三十五
实验三：消抖模块之一 .....	三十六
detect_module.v .....	三十七
delay_module.v .....	三十九
debounce_module.v .....	四十二
实验三说明： .....	四十三
实验三结论： .....	四十三
实验四：消抖模块之二 .....	四十四
delay_module.v .....	四十四

---

实验四说明:	四十五
2.6 控制模块的尴尬.....	四十六
实验五: SOS 信号之一.....	四十七
sos_module.v.....	四十七
control_module.v.....	五十
sos_generator_module.v.....	五十一
实验五说明: .....	五十二
实验五结论: .....	五十二
实验六: SOS 信号之二.....	五十三
inter_control_module.v .....	五十三
exp06_top.v.....	五十四
实验六说明: .....	五十五
实验六结论: .....	五十六
总结 .....	五十七
第三章 : 低级建模 - 基础建模.....	五十八
3.1 实验七: 数码管电路驱动.....	五十八
number_mod_module.v.....	五十九
smg_encoder_module.v.....	六十
smg_scan_module.v.....	六十三
column_scan_module.v.....	六十三
row_scan_module.v .....	六十五
smg_scan_module.v.....	六十七
exp07_top.v.....	六十九
实验七说明: .....	七十一
实验七结论: .....	七十一
3.2 实验八: PS2 解码 .....	七十二
PS2 的简单认识.....	七十二
对编码键盘 “键盘码” 的简单认识 .....	七十二
detect_module.v.....	七十五
ps2_decode_module.....	七十六
ps2_module .....	七十八
实验八说明 :.....	七十九
实验八结论 :.....	八十一
实验八演示: .....	八十一

---

---

cmd_control_module.v.....	八十一
exp08_demo.v .....	八十二
实验八演示说明: .....	八十四
实验八演示结论: .....	八十四
实验九: VGA 驱动 .....	八十五
实验九之一: 驱动概念 .....	八十五
sync_module.v.....	八十九
vga_control_module.v.....	九十一
vga_module.v .....	九十二
实验九之一说明: .....	九十四
实验九之一结论: .....	九十四
实验九之二: 向下兼容概念.....	九十五
sync_module.v.....	九十六
vga_control_module.v.....	九十八
vga_module.v .....	九十八
实验九之二说明: .....	一〇〇
实验九之二结论: .....	一〇〇
实验九之三: 点阵概念 .....	一〇一
sync_module.v.....	一〇三
vga_control_module.v.....	一〇五
vga_module.v .....	一〇七
实验九之三说明: .....	一一〇
实验九之三结论: .....	一一〇
实验九之四: 图层概念 .....	一一二
sync_module.v.....	一一五
vga_control_module.v.....	一一六
vga_module.v .....	一一八
实验九之四说明: .....	一二一
实验九之四结论: .....	一二一
实验九之五: 帧的概念 .....	一二二
sync_module.v.....	一二五
vga_control_module.v.....	一二六
vga_module.v .....	一三〇
实验九之五说明: .....	一三二

---

---

实验九之五结论:	一三二
实验九说明:	一三三
实验九结论:	一三三
3.4 实验十: 串口模块	一三四
实验十之一: 串口接收模块	一三五
detect_module.v	一三六
rx_bps_module.v	一三七
rx_control_module.v	一三八
rx_module.v	一四〇
实验十之一说明:	一四二
实验十之一结论:	一四三
实验十之一演示:	一四四
control_module.v	一四四
rx_module_demo.v	一四五
实验十之一演示说明:	一四七
实验十之一演示结论:	一四七
实验十之二: 串口发送模块	一四八
tx_bps_module.v	一四九
tx_control_module.v	一五〇
tx_module.v	一五二
实验十之二说明:	一五三
实验十之二结论:	一五三
实验十之二演示:	一五四
control_module.v	一五四
tx_module_demo.v	一五六
实验十之二演示说明:	一五七
实验十之二演示结论:	一五七
实验十说明:	一五八
实验十结论:	一五八
总结:	一五九
第四章 : 低级建模 - 仿顺序操作	一六〇
4.1 基本思路	一六〇
实验十一 : SOS 信号之三	一六一
s_module.v	一六一

---

---

o_module.v.....	一六四
sos_control_module .....	一六六
两仪性和多义性的问题: .....	一六八
sos_module.v.....	一六九
实验十一说明: .....	一七一
实验十一结论: .....	一七二
实验十一演示: .....	一七三
sos_module_demo.v.....	一七三
4.2 实验十二: 12864 (ST7565P) 液晶驱动 .....	一七五
SPI 发送模块 .....	一八一
spi_write_module.v.....	一八三
初始化模块.....	一八六
initial_control_module.v .....	一八六
initial_module.v.....	一八九
绘图模块 .....	一九一
draw_control_module.v.....	一九一
draw_module.v .....	一九五
液晶模块 .....	一九七
lcd_control_module.v .....	一九八
lcd_module.v.....	一九九
实验十二说明: .....	二〇一
实验十二结论: .....	二〇二
4.3 命令式的仿顺序操作 .....	二〇三
function_module.v .....	二〇五
cmd_control_module.v.....	二〇七
实验十三: DS1302 实时时钟驱动 .....	二一〇
ds1302_module.v.....	二一四
function_module.v .....	二一五
cmd_control_module.v.....	二二〇
ds1302_module.v.....	二二五
实验十三说明: .....	二二七
实验十三结论: .....	二二七
实验十三演示: .....	二二八
exp13_demo.v .....	二二八

---

---

总结:	二三一
第五章: 低级建模-封装 (接口建模)	二三二
5.1 实验十四 - 独立按键封装	二三二
key_interface.v	二三三
实验十四演示:	二三六
optional_pwm_module.v	二三八
exp14_demo.v	二四一
实验十四演示说明:	二四二
实验十四演示结论:	二四三
5.2 实验十五: 数码管封装	二四四
smg_control_module.v	二四六
smg_encode_module.v	二四七
smg_scan_module.v	二四九
smg_interface.v	二五一
实验十五说明:	二五二
实验十五结论:	二五二
实验十五演示:	二五三
demo_control_module.v	二五三
smg_interface_demo.v	二五五
实验十五演示说明:	二五六
实验十五演示结论:	二五七
5.3 实验十六: 蜂鸣器封装	二五八
beep_function_module.v	二六一
beep_control_module.v	二六四
beep_interface.v	二六六
实验十六说明:	二六八
实验十六结论:	二六八
实验十六演示:	二六九
beep_interface_demo.v	二六九
实验十六演示说明:	二七一
实验十六演示结论:	二七二
5.4 实验十七: PS2 封装	二七三
detect_module.v	二七三
ps2_decode_module.v	二七四

---

---

ps2_control_module.v.....	二七六
ps2_interface.v .....	二七七
实验十七说明: .....	二七九
实验十七结论: .....	二七九
实验十七演示: .....	二八〇
ps2_interface_demo.v .....	二八〇
实验十七演示说明: .....	二八二
实验十七演示结论: .....	二八二
5.5 实验十八: 串口发送 接收 封装.....	二八三
串口发送接口: .....	二八三
tx_top_control_module.v .....	二八三
tx_interface.v.....	二八五
串口接收接口: .....	二八七
rx_top_control_module.v .....	二八七
rx_interface.v.....	二八九
实验十八演示: .....	二九二
inter_control_module.v .....	二九二
rx_tx_interface_demo.v.....	二九四
实验十八演示说明: .....	二九五
实验十八演示结论: .....	二九六
5.6 实验十九: VGA 封装 .....	二九七
sync_module.v.....	二九八
vga_control_module.v.....	二九九
ram_module.v.....	三〇一
vga_interface.v .....	三〇四
实验十九说明: .....	三〇五
实验十九结论: .....	三〇六
实验十九演示: 小绿人请加油 .....	三〇八
vga_interface_demo.v .....	三〇八
实验十九演示说明: .....	三一二
实验十九演示结论: .....	三一二
5.7 实验二十: LCD (12864) 封装 .....	三一四
lcd_ram_module.v .....	三一五
spi_write_module.v.....	三一六

---

---

lcd_control_module.v .....	三一七
lcd_interface.v .....	三二二
实验二十说明: .....	三二三
实验二十结论: .....	三二四
实验二十演示: .....	三二五
lcd_interface_demo.v.....	三二五
实验二十演示说明: .....	三二九
实验二十结论说明: .....	三二九
5.8 实验二十一： RTC 接口.....	三三〇
rtc_control_module.v .....	三三二
rtc_interface.v.....	三四七
实验二十一说明: .....	三四八
实验二十一结论: .....	三四八
总结: .....	三四九
第六章 : 低级建模 - 系统建模.....	三五一
6.1 实验二十二： SOS 系统.....	三五一
sos_system.v.....	三五一
实验二十二说明: .....	三五三
实验二十二将结论: .....	三五三
6.2 实验二十三： RTC 系统.....	三五五
rtc_sytem.v.....	三五五
实验二十三说明: .....	三五七
实验二十三结论: .....	三五七
6.3 实验二十四： GUI 系统.....	三五八
menu_module.v .....	三六二
page_control_module.v .....	三七一
led_control_module.v .....	三七四
gui_system.v.....	三七八
实验二十四说明: .....	三八〇
实验二十四结论: .....	三八一
总结: .....	三八二
结束语 .....	三八三

---

# 第一章：我眼中的 FPGA 和 Verilog HDL

当接触一门新知识的时候，如果在心中没有任何形状(概念)，掌握的感觉都是遥不可及的。当然，FPGA 也好 Verilog HDL 也好，也是一回事儿。今天笔者就将这个秘密告诉读者！在新手之间有一个很普遍的问题 “FPGA 和 Verilog HDL 是什么？”。在我的心里，FPGA 宛如 “**一堆乐高积木**” 和 Verilog HDL 是自己的手（工具），自己可以随心所愿的要怎么拆就怎么拆。

这句话所包含的意义又是什么呢？

**第一 “形状”。**很多的新手都发问 FPGA 到底是什么？对于笔者的看法，FPGA 就是**一堆乐高积木**而已。但是这堆乐高积木又能做什么。我相信很多朋友接触 FPGA 之前，都有接触过单片机。当学习 FPGA 的时候，我们会不知不觉把 FPGA 当成 “**控制器**” 的形状。这样的想法并非完全正确也非完全错误。无论是 “接口”，“控制器”，“IC”，从最简单的到最复杂的，都是 FPGA 都可以涉及的范围，都是 FPGA 可以实现的 “**形状**”。FPGA 就是一堆乐高积木，只要方法得到，手段有效，就没有拆不出来的 “**形状**”

**第二 “学习的形状”。**我们应该用什么的形状来学习 FPGA 呢？笔者的答案都很肯定，就是什么都涉及，但是不求最困难的，只求最相似和最简单的。如:一个接口的设计，可以是一个数码管驱动程式。在笔者的众多学习之中，都是从 “**控制器**” 的这个形状入门开始，当掌握资源的驱动方法后，尝试建模，然后学习接口封装，最后甚至可以建立起属于自己的系统。

**第三 “组合 FPGA 的一双手”。**Verilog HDL 语言就如**“组合乐高积木的一双手”**（工具），没有了 “这双手” 我们就无法使 FPGA 组装起来。有一句话笔者一直都在强调，硬件描述语言是基本功，必须很好的掌握。

**第四 “建模技巧”。**建模技巧可以看成是 **“组合乐高积木的手段”**，这也是这本笔记要讨论的东西。事实上是诸多新手面对最大的瓶颈。建模这东西原本就没有具体的规则，笔者也是按照自己建模习惯，作为一个基础，希望可以帮助到更多的新手越过这段瓶颈。

**第五 “要掌握到何种的地步才算足够？”**作为曾经是新手的笔者，对于这问题笔者表示压力很大。但是经过一轮的思考，发现自己真的是笨级了，这些 “杞人忧天” 的问题，还是顺其自然的好。最重要就是学习的心态，学习是一件快乐的事情，为什么要搞得如此忧郁呢？在笔者看来，最起码的，也是最重要的就是 “**掌握基本功**”。

事实上这本笔记仅是讨论 “**Verilog HDL 基本功的一部分**”，因为 FPGA 涉及的知识实在是太广泛了，如时序分析等相关的知识。所以呀，我希望大家保持正向的 “**学习心态**”，好好的阅读这本笔记。这本笔记不会涉及太多专业的知识，而是非常焦距在建模的基础之上。“学习如同玩具，要一点一滴的去享受，才能体会到乐趣，如果只是单纯的为了目的或者是匆忙来去，无论是多么有趣的玩具在你的手中，你永远也不会体验到当中的乐趣”。学习应该时时刻刻保持正向的心态。

## 第二章：低级建模 - 基础知识

### 2.1 顺序操作和并行操作

顺序操作和并行操作，是新手们很容易混乱的一个重点。但是为了将低级建模发挥到极限，这一点必须好好的理解。Verilog HDL 语言，虽然不同与其他高级语言，有优秀结构性，但是作为硬件描述语言的它，最大的优势的是它支持并行操作。

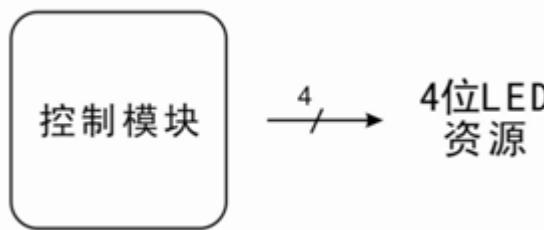
顺序操作有如“步骤”概念，如果上一个行为没有完成，下一个行为就没有执行的意义。而并行操作最为不同的是，两个行为都是独立执行，互不影响。那么，我们从一个典型的实验“流水灯实验”，在具体上来理解它们的不同之处。

下图是两种以不同操作方式建立的“流水灯实验”。

- 1)点亮第一个 LED，延迟一段时间。
- 2)点亮第二个 LED，延迟一段时间。
- 3)点亮第三个 LED，延迟一段时间。
- 4)点亮第四个 LED，延迟一段时间。
- 5)重复第一个步骤。

从上面看来，我们明白“[流水灯效果的产生](#)”主要是以“[顺序的方式](#)”执行 5 个步骤。这可能是人类自然的思维方式吧，人类真的是奇怪的动物，虽然人类的大脑是并行操作的，但是人类的思维方式比较偏向“顺序操作”。为什么呢？

如果引用现实中的实例，如果四个 LED 失去了“[指挥者](#)”，那么它们就罢工了！因为它们失去“[执行发号](#)”的第二方，这样的情况就如同上面内容如果没有了“1”，“2”，“3”，“4”，“5”的数目字，那么读者又如何看懂“[流水灯如何产生呢](#)”？

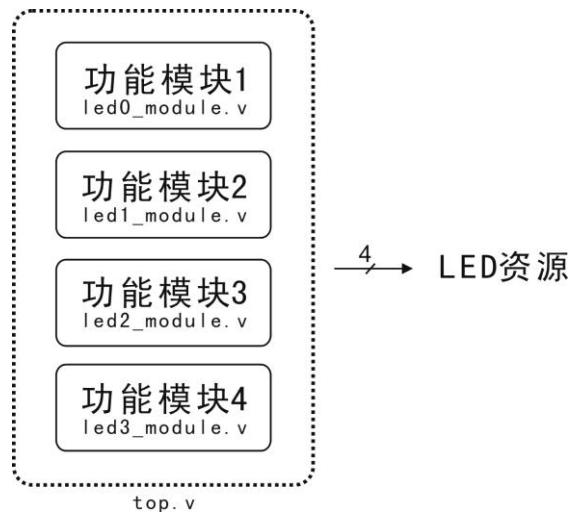


换一句话说，“[顺序操作](#)”的代表往往都有一个“[指挥者](#)”或者名为“[控制器](#)”东西的存在，执行着“[工作的次序（步骤）](#)”。

笔者相信很多学习 FPGA 的朋友都有学过单片机。学习单片机的时候，可能是 C 语言或者汇编语言的关系，所以很多朋友在不知不觉中的情况习惯了“[顺序操作](#)”这样的概念。新手们常常忽略了，FPGA 其实是并存着“[顺序操作](#)”和“[并行操作](#)”的操作概念。如果打从一开始就忽略了它们，往后的日子很难避免遇见瓶颈。

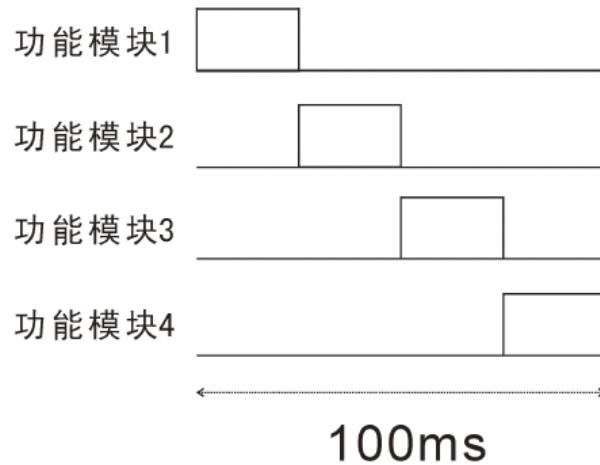
那么换成是“[并行操作](#)”的流水灯是如何的呢？结果我们从实验中理解...

## 实验一：永远的流水灯。



在这一个实验，我们要以上图作为基础，建立一个并行操作的流水灯模块。扫描频配置定为 100 Hz，而每一个功能模块在特定的时间内，将输出拉高。

示意图如下：



从上图我们可以看到，功能模块 1 在时间的第一个 1/4 拉高输出，功能模块 2 在时间的第二个 1/4 拉高输出，其余的两个功能模块也是以此类推。所以在一个固定的时间周期内（10ms），每一个功能模块所占的时间都是 2.5ms。

*led0\_module.v*

```
1. module led0_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output LED_Out;
9.
10.    /*****
11. //DB4CE15 开发板使用的晶振为 50MHz, 50M*0.01=5_000_000
12. parameter T100MS = 23'd5_000_000;
13. *****/
14.
15. reg [22:0]Count1;
16.
17. always @ ( posedge CLK or negedge RSTn )
18.     if( !RSTn )
19.         Count1 <= 23'd0;
20.     else if( Count1 == T100MS )
21.         Count1 <= 23'd0;
22.     else
23.         Count1 <= Count1 + 1'b1;
24.
25. *****/
26.
27. reg rLED_Out;
28.
29. always @ ( posedge CLK or negedge RSTn )
30.     if( !RSTn )
31.         rLED_Out <= 1'b0;
32.     else if( Count1 >= 23'd0 && Count1 < 23'd1_250_000)
33.         rLED_Out <= 1'b1;
34.     else
35.         rLED_Out <= 1'b0;
36.
37. *****/
38.
39. assign LED_Out = rLED_Out;
40.
```

## Verilog HDL 那些事儿 – 建模篇

```
41.      *****/
42.
43.
44. endmodule
```

*led1\_module.v*

```
1. module led1_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output LED_Out;
9.
10.    *****/
11. //DB4CE15 开发板使用的晶振为 50MHz, 50M*0.01=5_000_000
12. parameter T100MS = 23'd5_000_000;
13.
14.    *****/
15.
16. reg [22:0]Count1;
17.
18. always @ ( posedge CLK or negedge RSTn )
19.     if( !RSTn )
20.         Count1 <= 23'd0;
21.     else if( Count1 == T100MS )
22.         Count1 <= 23'd0;
23.     else
24.         Count1 <= Count1 + 1'b1;
25.
26.    *****/
27.
28. reg rLED_Out;
29.
30. always @ ( posedge CLK or negedge RSTn )
31.     if( !RSTn )
32.         rLED_Out <= 1'b0;
33.     else if( Count1 >= 23'd1_250_000 && Count1 < 23'd2_500_000 )
34.         rLED_Out <= 1'b1;
35.     else
36.         rLED_Out <= 1'b0;
```

```
37.  
38.    ****  
39.  
40.    assign LED_Out = rLED_Out;  
41.  
42.    ****  
43.  
44.  
45. endmodule
```

*led2\_module.v*

```
1. module led2_module  
2. (  
3.     CLK, RSTn, LED_Out  
4. );  
5.  
6.     input CLK;  
7.     input RSTn;  
8.     output LED_Out;  
9.  
10.    ****  
11.    //DB4CE15 开发板使用的晶振为 50MHz, 50M*0.01=5_000_000  
12.    parameter T100MS = 23'd5_000_000;  
13.  
14.    ****  
15.  
16.    reg [22:0]Count1;  
17.  
18.    always @ ( posedge CLK or negedge RSTn )  
19.        if( !RSTn )  
20.            Count1 <= 23'd0;  
21.        else if( Count1 == T100MS )  
22.            Count1 <= 23'd0;  
23.        else  
24.            Count1 <= Count1 + 1'b1;  
25.  
26.    ****  
27.  
28.    reg rLED_Out;  
29.  
30.    always @ ( posedge CLK or negedge RSTn )  
31.        if( !RSTn )
```

## Verilog HDL 那些事儿 – 建模篇

```
32.          rLED_Out <= 1'b0;
33.      else if( Count1 >= 23'd2_500_000 && Count1 < 23'd3_750_000 )
34.          rLED_Out <= 1'b1;
35.      else
36.          rLED_Out <= 1'b0;
37.
38.      /*****
39.
40.      assign LED_Out = rLED_Out;
41.
42.      *****/
43.
44.
45. endmodule
```

*led3\_module.v*

```
1. module led3_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output LED_Out;
9.
10.    *****/
11.
12.    parameter T100MS = 23'd5_000_000;
13.
14.    *****/
15.
16.    reg [22:0]Count1;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            Count1 <= 23'd0;
21.        else if( Count1 == T100MS )
22.            Count1 <= 23'd0;
23.        else
24.            Count1 <= Count1 + 1'b1;
25.
26.    *****/
```

```

27.
28.     reg rLED_Out;
29.
30.     always @ ( posedge CLK or negedge RSTn )
31.         if( !RSTn )
32.             rLED_Out <= 1'b0;
33.             lse if( Count1 >= 23'd3_750_000 && Count1 < 23'd5_000_000 )
34.                 rLED_Out <= 1'b1;
35.             else
36.                 rLED_Out <= 1'b0;
37.
38.     /*************************************************************************/
39.
40.     assign LED_Out = rLED_Out;
41.
42.     /*************************************************************************/
43.
44.
45. endmodule

```

*top\_module.v*

```

1. module top_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output [3:0]LED_Out;
9.
10.    /************************************************************************/
11.
12.    wire LED0_Out;
13.
14.    led0_module U1
15.    (
16.        .CLK( CLK ),
17.        .RSTn( RSTn ),
18.        .LED_Out( LED0_Out )
19.    );
20.
21.    /************************************************************************/

```

```
22.  
23.     wire LED1_Out;  
24.  
25.     led1_module U2  
26.     (  
27.         .CLK( CLK ),  
28.         .RSTn( RSTn ),  
29.         .LED_Out( LED1_Out )  
30.     );  
31.  
32.     /*****  
33.  
34.     wire LED2_Out;  
35.  
36.     led2_module U3  
37.     (  
38.         .CLK( CLK ),  
39.         .RSTn( RSTn ),  
40.         .LED_Out( LED2_Out )  
41.     );  
42.  
43.     /*****  
44.  
45.     wire LED3_Out;  
46.  
47.     led3_module U4  
48.     (  
49.         .CLK( CLK ),  
50.         .RSTn( RSTn ),  
51.         .LED_Out( LED3_Out )  
52.     );  
53.  
54.     /*****  
55.  
56.     assign LED_Out = { LED3_Out, LED2_Out, LED1_Out, LED0_Out};  
57.  
58.     /*****  
59.  
60. endmodule
```

### 实验一说明:

led0\_module.v, led1\_module.v, led2\_module.v, led3\_module.v 的源码都是大同小异。第 12 行是 100ms 计数器的常量声明。第 16~24 行是一个计数器，计数器计数范围为 100ms。而在 28~36 行是决定 4 个功能模块的不同之处。

led0\_module.v 功能模块是在时间的第一个 1/4 拉高输出。

led1\_module.v 功能模块是在时间的第二个 1/4 拉高输出。

led2\_module.v 功能模块是在时间的第三个 1/4 拉高输出。

led3\_module.v 功能模块是在时间的第四个 1/4 拉高输出。

而 top.v 是顶层模块，用来组织这四个功能模块。

### 实验一结论:

从实验的结果看来，利用“并行操作”实现的流水灯，但是在肉眼中和“顺序操作”没有任何两样。但是从“理解上”就有很大的差别。

led0\_module.v 功能模块是在时间的第一个 1/4 拉高输出。

led1\_module.v 功能模块是在时间的第二个 1/4 拉高输出。

led2\_module.v 功能模块是在时间的第三个 1/4 拉高输出。

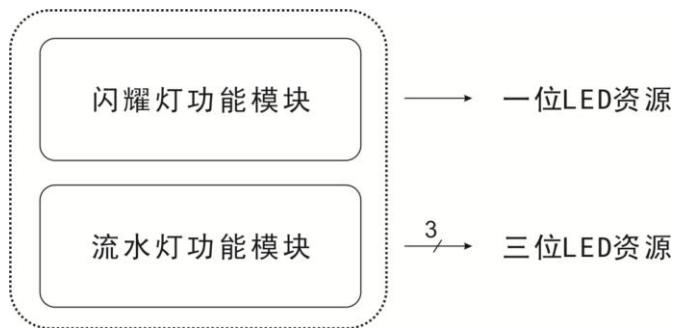
led3\_module.v 功能模块是在时间的第四个 1/4 拉高输出。

上面的内容表示了，4 个功能模块是独立操作的。由于输出在时间上不同，在肉眼中才会看到流水灯的效果。用现实的角度去思考的话，宛如四个局内人，无不关系，各自只是按照自己的节奏完成自己的工作。在局外人的眼中，他们如同有默契般，**不需要“指挥者”**也能完成任务。

在这里说一点局外话。“**并行操作**”的思维对于初学者来说，一开始的时候会觉得很不习惯。但是不要忘了“**FPGA+Verilog HDL**”必须使用“**并行操作**”的思维去理解。平常我们在设计程序的时候，在无意识中使用了“**顺序操作**”的思维，如果“**并行操作**”的思维也可以无意识般的使用，设计范围是可以更“**广泛**”起来。

## 2.2 倾向并行操作

在 2.1 章理解了“顺序操作”和“并行操作”的区别之后，这一章我们要讨论并且习惯“并行操作”的思维。



上图是一个组合模块，里边包含了两个功能模块。一是对闪耀灯控制的功能模块，二是对流水灯控制的功能模块。假设我要利用“顺序操作”实现如图的功能模块。实际上是怎样的一种效果呢？

我们以“C 语言”作为“顺序操作”的代表，来分析一下过程。.

```

While ( 1 )          //大循环
{
    Flash_Funct();   //闪耀功能模块
    Run_Funct();     //流水灯功能模块
}

```

如果按照上面的代码，程序一开始就会先执行闪耀功能模块（Flash\_Funct()），然后再执行流水灯功能模块（Run\_Funct()）。但是在现实中，由于控制器的扫描速度很快快，使得人类的肉眼看到错觉“**两个模块并行执行着**”。事实上，无论控制器的扫描速度怎么快，也是会存在着“**上一步延迟**”。

亦即 Run\_Funct() 要执行前，必须等待 Flash\_Funct() 执行完毕，Flash\_Funct() 要重复执行，必须等待 Run\_Funct() 执行完毕。除此之外，在“**理解上**”，人类的思维会惯性的以“**顺序**”的方式去理解上面的代码。

1. 执行 Flash\_Funct();
2. 执行 Run\_Funct();
3. 重复步骤 1.

如此一来我们明白了一个道理，控制器的快速扫描使得“**顺序操作**”在效果上，被误认为“**并行操作**”。

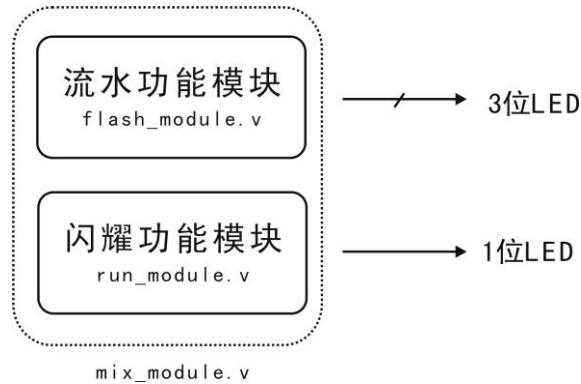
```
module top_module ( . . . . . );
    flash_module U1 //功能模块 1 实例化
    (
        . . . . .
    );
    run_module U2 //功能模块 2 实例化
    (
        . . . . .
    );
endmodule
```

上面内容是以 Verilog HDL 语言实例化的过程，操作的概念完全属于“并行操作”。实例 U1 和 U2 都有各自的功能，各自好不相互。我们只要确保实例化的格式正确而已。

在前面笔者已经说过“并行操作”的功能模块功能都是独立执行。在“[设计上](#)”，flash\_module 和 run\_module 的建模工作都是“[分开执行](#)”，这正好符合“[理解上](#)”，该两个模块都是“[分开执行](#)”。

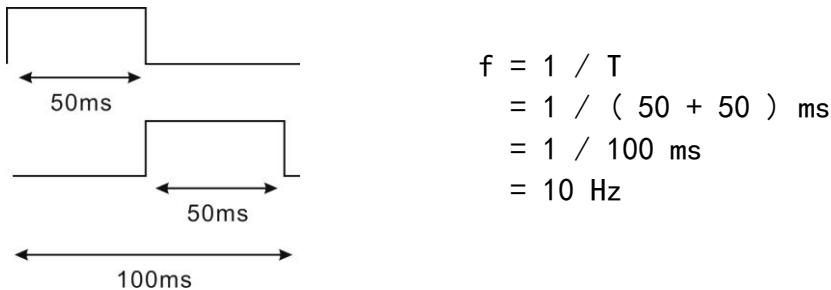
换一句话说，上面的内容是真正意义上的“[并行操作](#)”。如果“[顺序操作](#)”要实现“近似并行操作”，原理上它可以依靠“[高频扫描](#)”或者“[任务调度程序](#)”来达到近似并行操作。但是“[顺序操作](#)”无论怎么“[依靠](#)”都不是真正意义上的“[并行操作](#)”。

## 实验二：闪耀灯和流水灯

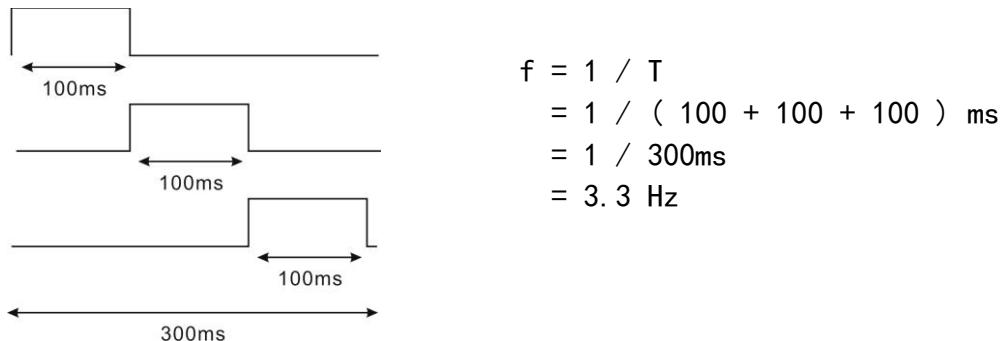


功能模块	执行频率	资源占用
闪耀灯功能模块 flash_module.v	10hz (100ms)	一位 LED
流水灯功能模块 run_module.v	3.3Hz (300ms)	三位 LED

实验二主要是建立如上图的功能模块。乘着这个实验，笔者来解释一下一个长久以来美丽的误会就是“[扫描频率和闪耀频率？](#)”



闪耀频率是指一个 LED 开和关的周期时间。实验二中的 `flash_module` 所制定的输出如上。



扫描频率是指一次完成流水灯动作所的时间周期。实验二中的 run\_module 所制定的扫描输出如上。

回忆以前，记得笔者在入门单片机的时候，在第一个实验就是流水灯，笔者过分在乎肉眼中所看到的效果，而忘记了这些重要的信息。估计很多人也是吧？但是经过前面 2.1 章和 2.2 章的实验创作，笔者我明白“小小的细节，大大的道理”的道理。原来流水灯实验，不仅是经典，而且包含了很多的知识。有时候做人太注重着外面的世界，不小心之间忘记了心里的世界。

*flash\_module.v*

```
1. module flash_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output LED_Out;
9.
10.    /*****
11.    //DB4CE15 使用的晶振为 50MHz, 50M*0.05-1=2_499_999
12.    parameter T50MS = 22'd2_499_999;
13.
14.    *****/
15.
16.    reg [21:0]Count1;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            Count1 <= 22'd0;
21.        else if( Count1 == T50MS )
22.            Count1 <= 22'd0;
23.        else
24.            Count1 <= Count1 + 1'b1;
25.
26.    *****/
27.
28.    reg rLED_Out;
29.
30.    always @ ( posedge CLK or negedge RSTn )
31.        if( !RSTn )
```

## Verilog HDL 那些事儿 – 建模篇

```
32.          rLED_Out <= 1'b0;
33.      else if( Count1 == T50MS )
34.          rLED_Out <= ~rLED_Out;
35.
36.      /*****/
37.
38.      assign LED_Out = rLED_Out;
39.
40. endmodule
```

*run\_module.v*

```
1. module run_module
2. (
3.     CLK, RSTn, LED_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output [2:0]LED_Out;
9.
10.    /*****
11.    //DB4CE15 使用的晶振为 50MHz, 50M*0.001-1=49_999,从 0 开始, 需要减 1
12.    parameter T1MS = 16'd49_999;
13.
14.    *****/
15.
16.    reg [15:0]Count1;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            Count1 <= 16'd0;
21.        else if( Count1 == T1MS )
22.            Count1 <= 16'd0;
23.        else
24.            Count1 <= Count1 + 1'b1;
25.
26.    /*****
27.
28.    reg [9:0]Count_MS;
29.
30.    always @ ( posedge CLK or negedge RSTn )
31.        if( !RSTn )
```

```

32.          Count_MS <= 10'd0;
33.      else if( Count_MS == 10'd100 )
34.          Count_MS <= 10'd0;
35.      else if( Count1 == T1MS )
36.          Count_MS <= Count_MS + 1'b1;
37.
38.      /*****
39.
40.      reg [2:0]rLED_Out;
41.
42.      always @ ( posedge CLK or negedge RSTn )
43.          if( !RSTn )
44.              rLED_Out <= 3'b001;
45.          else if( Count_MS == 10'd100 )
46.              begin
47.
48.                  if( rLED_Out == 3'b000 )
49.                      rLED_Out <= 3'b001;
50.                  else
51.                      rLED_Out <= { rLED_Out[1:0], 1'b0 };
52.              end
53.
54.      *****/
55.
56.      assign LED_Out = rLED_Out;
57.
58.      *****/
59.
60.
61. endmodule

```

*mix\_module.v*

```

1. module mix_module
2. (
3.     CLK, RSTn, Flash_LED, Run_LED
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output Flash_LED;
9.     output [2:0]Run_LED;
10.

```

```
11.      *****/
12.
13.      wire Flash_LED;
14.
15.      flash_module U1
16.      (
17.          .CLK( CLK ),
18.          .RSTn( RSTn ),
19.          .LED_Out( Flash_LED )
20.      );
21.
22.      *****/
23.
24.      wire [2:0]Run_LED;
25.
26.      run_module U2
27.      (
28.          .CLK( CLK ),
29.          .RSTn( RSTn ),
30.          .LED_Out( Run_LED )
31.      );
32.
33.      *****/
34.
35.      assign Flash_LED = Flash_LED;
36.      assign Run_LED = Run_LED;
37.
38.      *****/
39.
40. endmodule
```

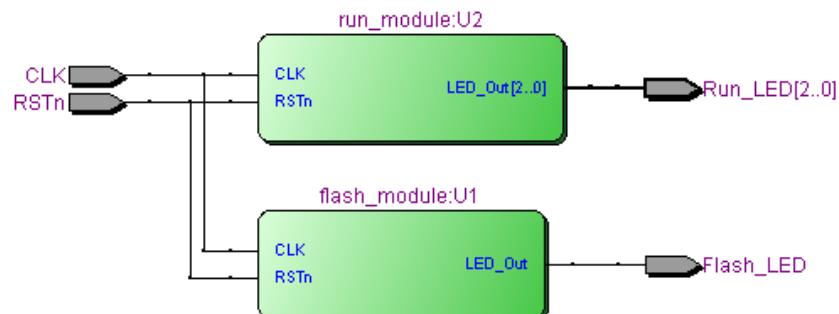
## 实验二说明:

flash\_module.v 是一个 10hz, 50% 占空比输出的功能模块, 说得简单点就是“定时开关”。

run\_module.v 是一个 3.3hz 扫描频率的流水灯, 采用“控制模块”形式编写。16~24 行是 1ms 的计数器。28~36 行是基于 count1 的 100ms 的计数器。40~52 行是 3Bit 的移位操作, 45 行表示每间隔 100 微妙发生一次移位操作。48 行, 判断移位末期。而 51 行是移位的代码。

mix\_module.v 是组合模块, 代码很容易明白。自己看着办吧。

## 实验二完成框图:



## 实验二结论:

无论是实验二结果, 还是理解上的思维, 或者设计上的方向, 两个模块是“并行操作”。所以呀, 思维倾向“并行操作”对, VerilogHDL 语言的理解是非常重要。

## 2.3 Verilog HDL 不是“编程”是“建模”

在初次接触 Verilog HDL 语言的时候，网上很多朋友都说它和 C 语言很相似？但是对于这个问题，笔者真的头疼很久！如果说 Verilog HDL 与 C 语言不同，但是是什么地方不同，笔者却有说不出的感觉。

单片机是一种已经完成的硬件，但是单片机没有“[灵魂](#)”，c 语言给单片机“[灌输灵魂](#)”，单片机才能舞动起来。c 语言就是扮演这样的一个角色。不同的是，Verilog HDL 语言是一种富有“[形状](#)”的语言，它可以描述单片机的任何部分，如 UART 资源，定时器资源等。

“[建模](#)”这一词是指，使用“[硬件描述语言](#)”去[建立某个资源模块](#)。如果说 c 语言可以使用“[编程](#)”一词，那么 Verilog HDL 语言使用“[建模](#)”这一词更合适不过了。从理解上，C 语言，我们可以使用“[代码](#)”和“[编程](#)”的形式去理解，如：函数的概念，参数的概念，局部全局的概念等。但是“[代码的形式](#)”却完全不适合 Verilog HDL 语言。既然“[代码的形式](#)”不适合 Verilog HDL 语言，那么 Verilog HDL 语言又该用“[什么](#)”去看待？

如果以“[代码](#)”的形式去理解 Verilog HDL 语言的设计，到底有多大问题？从网上的众多求救贴中，就可以理解这一点。笔者不是完全否定这样的理解形式，毕竟“[代码](#)”是语言的基础，而且建模的完整性，都需要“[代码级](#)”的微调。

前面笔者已经说过 Verilog HDL 语言是一种富有“[形状](#)”的语言。“[建模](#)”一词使得 Verilog HDL 语言的更有形象和更有具体感。

如果以笔者的角度去了解“[建模](#)”……

笔者每次在设计之前，都喜欢将一个一个模块画在纸上，然后再使用连线的方式去完成一个设计。从这一点，我们可以知道笔者是从“[图形](#)”去完成设计的准备。换一句话说，笔者是以“[形状](#)”的形式作为设计的理解基础。

笔者只想告诉一个事实而已。如果着手以“[建模](#)”去理解 Verilog HDL 语言，以“[形状](#)”去完成 Verilog HDL 语言的设计。在感觉上 Verilog HDL + FPGA 是“[可所触及](#)”，是一种“[实实在在](#)”的感觉，不相等于“[编程](#)”时的那种“[抽象感](#)”。

“低级建模”中的“[建模](#)”，正如上面的内容所说。那么“[低级](#)”呢？“[低级](#)”这一词原本是指，[最基本的也是最简单的](#)。“低级建模”不过是笔者的建模习惯而已，或者说是笔者的“[建模风格](#)”而已。在笔者的心中“[低级建模](#)”有“[万丈高楼从地起](#)”的意思。接下来的章节都是“[低级建模](#)”的故事。一开始的“接触”，可能使读者们会出现不适的感觉，情况如同初鸭下水。但是有一点可以肯定的是，当读者了解“[低级建模](#)”以后，Verilog HDL 的设计会变得有趣起来。

## 2.4 听听低级建模的故事

经过两章的洗礼，这一章就放松放松吧 ... 听听笔者讲故事。

笔者大约初学 FPGA 有两个月左右后，笔者就步入学习 FPGA 的瓶颈期。那时候，笔者虽然很好掌握 Verilog HDL 语言的基础，并且很熟悉 RTL 级代码，可是笔者始终有一种“[不可触及](#)”的感觉。我到底缺少什么呢？

带着这个问题，笔者浑浑噩噩的继续学习一段时间。期间笔者写了“[SMG 接口设计](#)”和“[低级建模 · 仿顺序操作 思路篇](#)”这两本笔记。“[低级建模](#)”的影子慢慢的笔者我脑海中隐现出来。同期间很不幸的，现实中要我陷入巨大改变。就这样一个思路就进入一个死胡同。

2010 年 6 月，笔者进入某工厂工作。现实总是残忍，同期笔者的经济和时间陷入危机，笔者不得不把所有的精神和时间往工作里抛去。在工厂工作，那种环境真的不是人所为，短短的一个月笔者被抽去许多学习的动力，兴趣，和感情。

眼下，笔者察觉到自己的问题。笔者请了一天假期，回家好好的思考思考。“想着想着，不知不觉我睡着了 .....

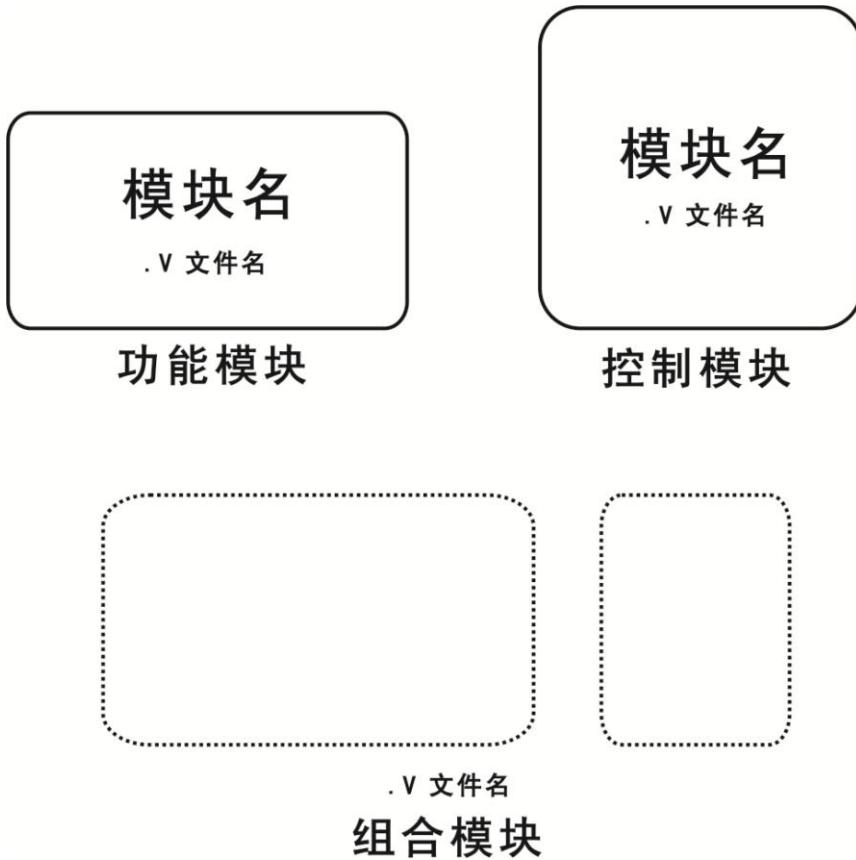
在梦中，“低级建模”不断挣扎，一股又一股的刺痛使笔者注意了现实中被疏忽的要事。在工作中。[一项巨型的项目，需要几个部门共同维护，而且每一个部门都有几个个小组相互支撑，其中每一个小组中又有多少员工互相支持（低级建模的基本思路）。](#)

( ⊙ o ⊙ )啊（惊醒）！云之间，失去了一个月的热情，立即回到笔者的体内。

2010 年 7 月，笔者放弃了加班的高薪，笔者用更多的时间去建立“低级建模”的基础。呵呵，笔者再一次遇见现实的残忍，我没有资源的支持，但是学习的路上总是柳暗花明，在这里黑金动力社区的出现，这个问题很巧妙的被解决了 .....

## 2.5 低级建模的资源

低级建模有讲求资源的分类，目的是以“[图形](#)”的方式来提高建模的解读性（表达能力）。

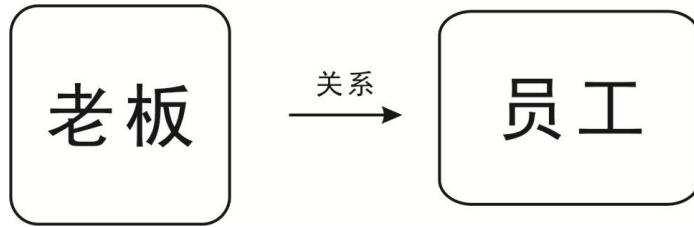


图上是低级建模最基本的建模图像，估计大家在实验一和实验二已经眼熟过。功能模块（低级功能模块）是一个水平的长方形，而控制模块（低级控制模块）是矩形。组合模块，可以是任意的形状（随意正方形）。注意功能模块和控制模块都包含“[模块名](#)”和“[.v 文件名](#)”，相反组合模块只含“[.v 文件名](#)”。

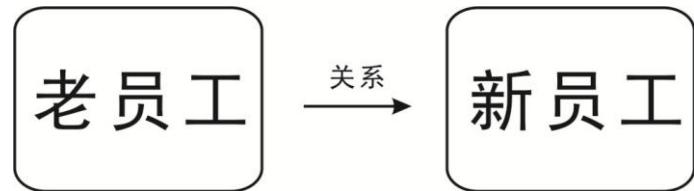
每一个“低级建模资源”的任务如自身命名一样。功能模块的例子有“[flash\\_module.v](#)”，控制模块的例子有“[run\\_module.v](#)”，和“[mix\\_module.v](#)”是组合模块。

一个完整的建模都是由图上的基本资源“[组合再组合](#)”。当然低级建模有一个问题，就是建模量很大，但是这个问题是见仁见智。除此之外，低级建模有一个[必须遵守的准则](#)就是“[一个功能模块（控制模块 仅有一个功能）](#)”。为什么这个准则那么重要呢？

因为低级建模是利用“[图形](#)”来表达一个完整的建模，其中模块与模块之间是以“[连线](#)”来表达关系。过多的功能在一个模块里面，会破坏“[建模的和谐](#)”，此外还使得“[图形](#)”的绘制更复杂。



假设一个例子：一个老板对一个员工命令。老板（控制模块），和员工（功能模块）。如果从另一种角度去思考，“老板发号”，“员工干活”。“连线”表示了他们是“宾主关系”。

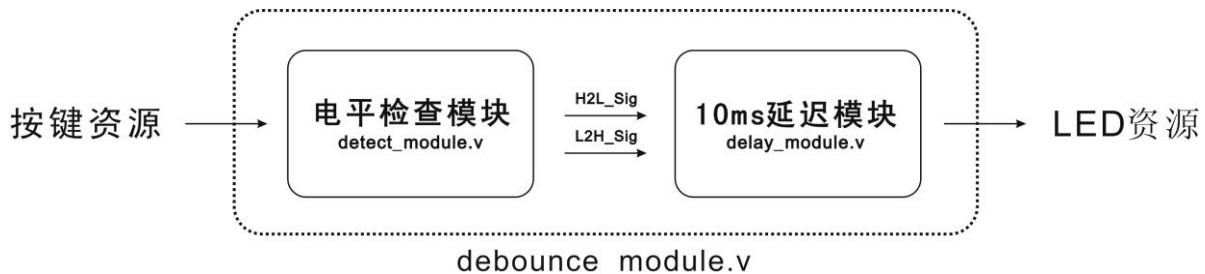


又或者另一个例子：一个老员工对一个新手员工发号。在现实中，老员工是来打工的（功能模块），新手也是来打工的（功能模块）。但是老员工有时也需要新手来支持，如上图。那么从“连接”上推断，结果即可“一目了然”，亦即它们是“老手和新手的关系”

上面两个例子，表示了笔者非常强调“**一个功能模块（控制模块）仅有一个功能**”的原因。这个准则会大大的提升模块的解读性（表达能力），除此之外也使得建模更容易更多样化。

单单两个例子是无法说明白“低级建模”的好处，我们以一个简单实验，从设计中理解。

### 实验三：消抖模块之一



图上是一个简单的按键消抖模块。设计的方法主要是由“电平检查模块”和“10ms 延迟模块”组合合成。

设计的思路如下：

- 1) 一但检测到按键资源按下（高电平到低电平变化），“电平检查模块”就会拉高

H2L\_Sig 电平，然后拉低。

- 2) “10ms 延迟模块”，检测到 H2L\_Sig 高电平，就会利用 10ms 过滤 H2L\_Sig，拉高输出。
- 3) 当按键被释放“电平检测模块”，会拉高 L2H\_Sig 电平，然后拉低。
- 4) “10ms 延迟模块”，检查到 L2H\_Sig 就会利用 10ms 过滤 H2L\_Sig，然后拉低输出。

*detect\_module.v*

```
1. module detect_module
2. (
3.     CLK, RSTn, Pin_In, H2L_Sig, L2H_Sig
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     input Pin_In;
9.     output H2L_Sig;
10.    output L2H_Sig;
11.
12.    /*****
13.    //DB4CE15 开发板使用的晶振为 50MHz, 50M*0.0001-1=4_999
14.    parameter T100US = 11'd4999;
15.
16.    *****/
17.
18.    reg [10:0]Count1;
19.    reg isEn;
20.
21.    always @ ( posedge CLK or negedge RSTn )
22.        if( !RSTn )
23.            begin
24.                Count1 <= 11'd0;
25.                isEn <= 1'b0;
26.            end
27.        else if( Count1 == T100US )
28.            isEn <= 1'b1;
29.        else
30.            Count1 <= Count1 + 1'b1;
31.
```

```

32.      *****/
33.
34.      reg H2L_F1;
35.      reg H2L_F2;
36.      reg L2H_F1;
37.      reg L2H_F2;
38.
39.      always @ ( posedge CLK or negedge RSTn )
40.          if( !RSTn )
41.              begin
42.                  H2L_F1 <= 1'b1;
43.                  H2L_F2 <= 1'b1;
44.                  L2H_F1 <= 1'b0;
45.                  L2H_F2 <= 1'b0;
46.              end
47.          else
48.              begin
49.                  H2L_F1 <= Pin_In;
50.                  H2L_F2 <= H2L_F1;
51.                  L2H_F1 <= Pin_In;
52.                  L2H_F2 <= L2H_F1;
53.              end
54.
55.      *****/
56.
57.
58.      assign H2L_Sig = isEn ? ( H2L_F2 & !H2L_F1 ) : 1'b0;
59.      assign L2H_Sig = isEn ? ( !L2H_F2 & L2H_F1 ) : 1'b0;
60.
61.
62.      *****/
63.
64. endmodule

```

detect\_module.v 是电平检测的功能模块。14 行定义了 100us 的常量，而第 18~30 行是用于延迟 100us。**因为电平检测模块是非常敏感，在复位的一瞬间，电平容易处于不稳定的状态，我们需要延迟 100us。** isEn = 1 寄存器是表示 100us 的延迟已经完成 (28 行)。

第 34~37 行，声明了四个寄存器。H2L\_F1, H2L\_F2，是针对检测电平由高变低。相反的 L2H\_F1, L2H\_F2，则是针对检测电平由低变高。在 41~46 行，对各个寄存器初始化了，由于 H2L\_Fx 是为了检测由高变低的电平，所以初始化为逻辑 1。L2H\_Fx 是为了检测由低变高的电平，初值被设置为逻辑 0。

//初始化

```
H2L_F1 <= 1'b1;  
H2L_F2 <= 1'b1;  
  
//每一个时间的操作  
H2L_F1 <= Pin_In;  
H2L_F2 <= H2L_F1;  
  
//每一个时间的布尔运算输出  
Pin_Out = H2L_F2 & !H2L_F1
```

上面代码是用来检测电平由高变低。H2L\_F1 和 H2L\_F2 的初值都是逻辑 1。假设第一个时间 Pin\_In 为低电平，H2L\_F1 就会被赋值为逻辑 0，而 H2L\_F1 则是被赋值为 H2L\_F1 上一个时间的值（也就是 H2L\_F1 的初值）。

我们知道在第一个时间，H2L\_F1 为逻辑 0，H2L\_F2 位逻辑 1。由于对 H2L\_F1 的取反操作，H2L\_F1 与 H2L\_F2 “求与” 运算，所以这个表达式的输出是逻辑 1。

再假设第二个时间 Pin\_In 保持为低电平，H2L\_F1 同样会被赋值为逻辑 0，而 H2L\_F2 则是被赋值为 H2L\_F1 上一个时间的值，亦即逻辑 0（第一个时间的值）。再经过布尔表达式的运算，在第二个时间，H2L\_F1 是逻辑 0，H2L\_F2 是逻辑 0，所以输出的结果是逻辑 0.

时间	H2L_F1	H2L_F2	Pin_Out = ( !H2L_F1 ) & H2L_F2
Initial	1	1	0
T1	0	1	1
T2	0	0	0

第 48~53 行，正是执行如上的操作，无论是检测电平由高变低或者由低变高，思路基本都是一样。而最后的 58~59 行，是关于“电平检测”的布尔表达式。但是有一点不同的是，Pin\_Out 的输出，是发生在 100us 之后，因为 100us 之前被 isEn 寄存器所限制（原因之前已经说了）。换一句话说，电平检测模块的有效是发生在 100us 的延迟之后。

*delay\_module.v*

```
1. module delay_module  
2. (  
3.     CLK, RSTn, H2L_Sig, L2H_Sig, Pin_Out  
4. );  
5.  
6.     input CLK;  
7.     input RSTn;  
8.     input H2L_Sig;
```

```
9.      input L2H_Sig;
10.     output Pin_Out;
11.
12.     /*************************************************************************/
13.
14.     parameter T1MS = 16'd49_999;
15.
16.     /*************************************************************************/
17.
18.     reg [15:0]Count1;
19.
20.     always @ ( posedge CLK or negedge RSTn )
21.         if( !RSTn )
22.             Count1 <= 16'd0;
23.         else if( isCount && Count1 == T1MS )
24.             Count1 <= 16'd0;
25.         else if( isCount )
26.             Count1 <= Count1 + 1'b1;
27.         else if( !isCount )
28.             Count1 <= 16'd0;
29.
30.     /*************************************************************************/
31.
32.     reg [3:0]Count_MS;
33.
34.     always @ ( posedge CLK or negedge RSTn )
35.         if( !RSTn )
36.             Count_MS <= 4'd0;
37.         else if( isCount && Count1 == T1MS )
38.             Count_MS <= Count_MS + 1'b1;
39.         else if( !isCount )
40.             Count_MS <= 4'd0;
41.
42.     /*************************************************************************/
43.
44.     reg isCount;
45.     reg rPin_Out;
46.     reg [1:0]i;
47.
48.     always @ ( posedge CLK or negedge RSTn )
49.         if( !RSTn )
50.             begin
51.                 isCount <= 1'b0;
52.                 rPin_Out <= 1'b0;
```

```
53.           i <= 2'd0;
54.       end
55.   else
56.       case ( i )
57.
58.           2'd0 :
59.               if( H2L_Sig ) i <= 2'd1;
60.               else if( L2H_Sig ) i <= 2'd2;
61.
62.           2'd1 :
63.               if( Count_MS == 4'd10 ) begin isCount <= 1'b0; rPin_Out <= 1'b1; i <= 2'd0; end
64.               else isCount <= 1'b1;
65.
66.           2'd2 :
67.               if( Count_MS == 4'd10 ) begin isCount <= 1'b0; rPin_Out <= 1'b0; i <= 2'd0; end
68.               else isCount <= 1'b1;
69.
70.
71.       endcase
72.
73.   /*****
74.
75.   assign Pin_Out = rPin_Out;
76.
77.   *****/
78.
79.
80. endmodule
```

delay\_module.v 是 10ms 延迟的功能模块。模块采用“[仿顺序操作](#)”的写法（第四章）。第 16~42 行是“[延迟](#)”的写法，第 18~28 行是 1ms 的定时器，而第 32~40 行是计数器。无论是定时器或者计数器都是由 isCount 标志寄存器使能。

第 44~71 行是“[仿顺序操作](#)”（第四章），第 46 行 i 寄存器用来控制执行的步骤。开始的时候 i 会根据“[H2L\\_Sig](#)”或者“[L2H\\_Sig](#)”进入不同的步骤（58~60 行）。从 62~64，或者 66~68 行，都是延迟 10ms 的操作，不同的地方就只有 rPin\_Out 寄存器的赋值（63 行与 67 行）。

我们先简单的了解一下地 44~71 行的设计思路：

- 1) 如果 H2L\_Sig 信号有反应，就进入步骤 1，
- 2) 在进入步骤 1 之后，由于达不到 if 条件，isCount 使能。定时器，计数器呀开始执行。
- 3) 在经过 10ms 之后，isCount 不使能，定时器，计数器停止执行。rPin\_Out 为逻辑 1 。返回步骤 0。

又或者：

- 1) 如果 L2H\_Sig 信号有反应，就进入步骤 2，
- 2) 在进入步骤 2 之后，由于达不到 if 条件，isCount 使能。定时器，计数器呀开始执行。
- 3) 在经过 10ms 之后，isCount 不使能，定时器，计数器停止执行。rPin\_Out 为逻辑 0 。返回步骤 0。

*debounce\_module.v*

```

1. module debounce_module
2. (
3.     CLK, RSTn, Pin_In, Pin_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     input Pin_In;
9.     output Pin_Out;
10.
11.    *****/
12.
13.    wire H2L_Sig;
14.    wire L2H_Sig;
15.
16.    detect_module U1
17.    (
18.        .CLK( CLK ),
19.        .RSTn( RSTn ),
20.        .Pin_In( Pin_In ),      // input - from top
21.        .H2L_Sig( H2L_Sig ),  // output - to U2
22.        .L2H_Sig( L2H_Sig )  // output - to U2
23.    );
24.
25.    *****/
26.
27.    delay_module U2
28.    (
29.        .CLK( CLK ),
30.        .RSTn( RSTn ),
31.        .H2L_Sig( H2L_Sig ),  // input - from U1
32.        .L2H_Sig( L2H_Sig ), // input - from U1

```

## Verilog HDL 那些事儿 – 建模篇

```
33.           .Pin_Out( Pin_Out )      // output - to top
34.       );
35.
36.       /*****
37.
38. endmodule
```

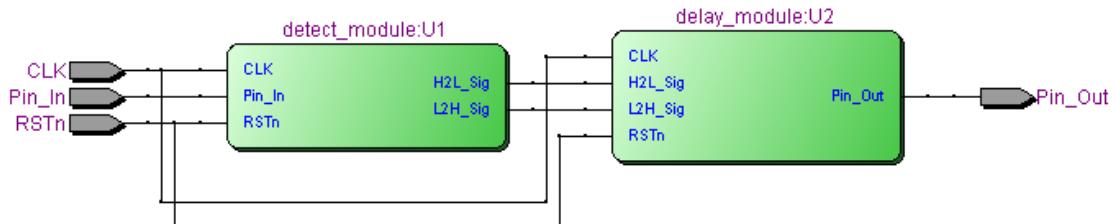
Debounce\_module.v 是组合模块。模块的组合上上图。信号连线的地方已经很清楚的注释。

实验三说明:

可能很多朋友会对 detect\_module.v 的延迟 100us 产生疑问？如果说简单一点就是：

在初始化的时候电平可能会持续 1~10us 的不稳定状态（笔者估计），延迟 100us 的目的是为了过滤这个不稳定状态。当 100us 之后 isEn 寄存器就会被设置，之后的事就不再跟它有关系了。

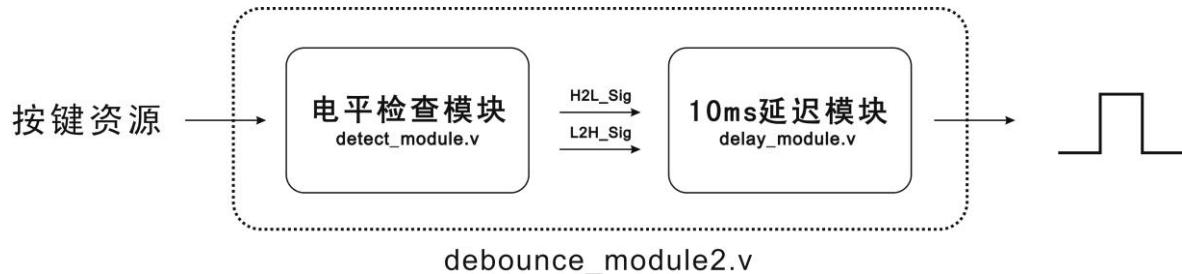
完成的扩展图：



实验三结论:

实验三是“**功能模块对功能模块**”的组合建模。建模方法遵守了“**一个模块一个功能**”的准则，再加上以“**图形**”的方式表现和使用“**连线**”表达关系。当然“**低级建模**”的好处不止在这里而已，随着建模的工程度和层次增加，就会越发的凸显“**低级建模**”的优势。

## 实验四：消抖模块之二



实验四和实验三的区别就是在于输出。实验三的 `debounce_modulve.v` 当检测到由高变低的电平变化时就拉高输出，然而当检查到由低变高的电平变化时拉低输出。反之实验四的 `debounce_module.v` 当检测到由高变低的电平变化时，产生一个时钟的高脉冲。当检测到由低变高的电平变化时，只有消抖动作，输出没有任何影响。

`delay_module.v`

```

56.      case ( i )
57.
58.          3'd0 :
59.              if( H2L_Sig ) i <= 2'd1;
60.              else if( L2H_Sig ) i <= 2'd3;
61.
62.          3'd1 :
63.              if( Count_MS == 4'd10 ) begin isCount <= 1'b0; rPin_Out <= 1'b1;i <= 2'd2;end
64.              else isCount <= 1'b1;
65.
66.          3'd2 :
67.              begin rPin_Out <= 1'b0; i <= 2'd0; end
68.
69.          3'd3 :
70.              if( Count_MS == 4'd10 ) begin isCount <= 1'b0; i <= 2'd0; end
71.              else isCount <= 1'b1;
72.
73.
74.      endcase

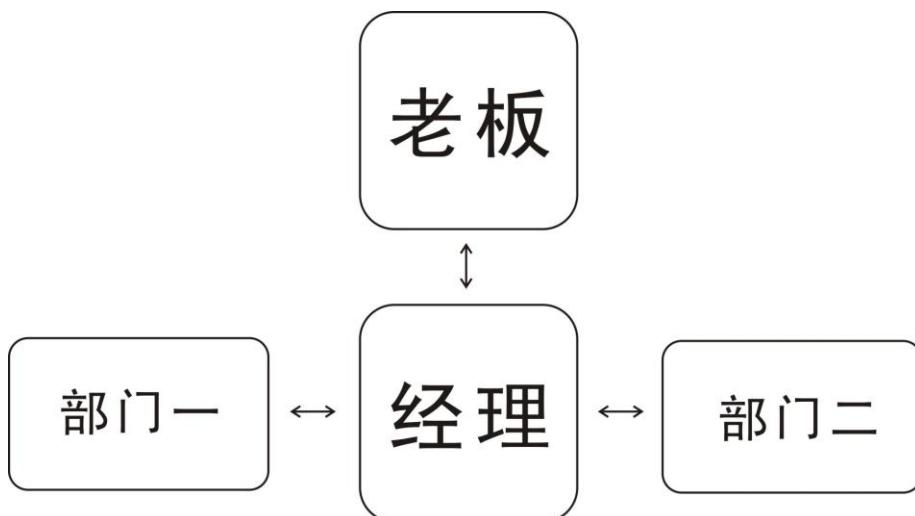
```

实验四说明:

大部分的源码和实验三一样，不同的地方就在于 delay\_module.v 的 56~74 行。从上面的代码我们可以看到，当检测到由高变低的电平变化的时候（第 59 行），步骤 i 就进入，步骤 1。经延迟 10ms 过后，拉高输出，然后进入步骤 2。步骤 2 会拉低输出，然后回到步骤 0。同样的，在步骤 0 当检测到由低变高的电平变化时，就会进入步骤 3，经延迟 10ms 过后，就会返回步骤 0。

## 2.6 控制模块的尴尬

在“低级建模”中，为了“管理”和“协调”多个功能模块，“控制模块”的角色是举足轻重，不可忽视。



如上图，经理的存在是为了协调两个部门的工作，而老板的存在是管理系统的运作。在这里无论是老板或者是经理，它们都是执行着“**控制**”的工作。从上图我们可以知道，所谓的“**控制**”有两点：

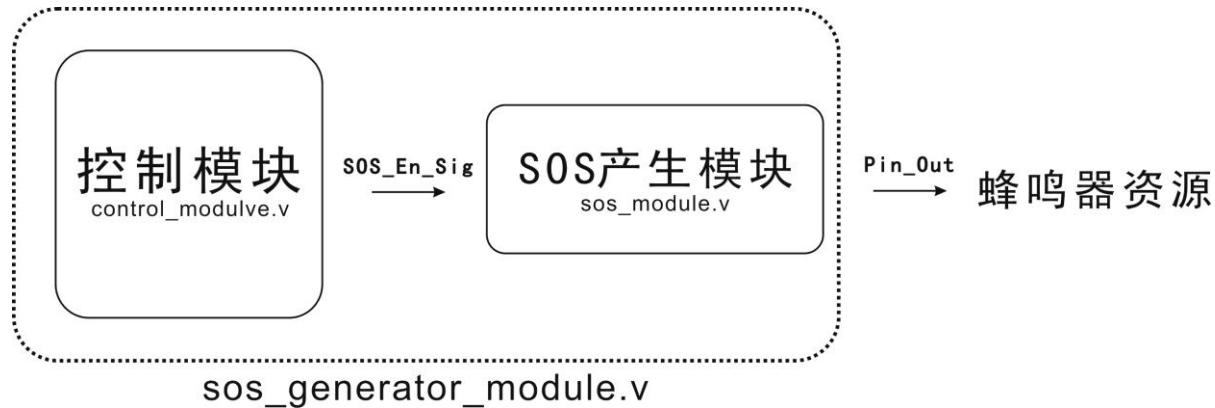
一是，**协调工作**。二是，**管理运作**。

“低级建模”是一个“多模块”的建模方法，模块和模块之前难免会出现“**沟通不好**”或者“**缺少管理**”的状况，这是“控制模块”存在的价值。虽然从宏观上“**控制建模**”有“**多指一举**”的行为，但是在微观上，却提升了不少设计的可能性。

在实验三中，我们实现了“**功能模块对功能模块**”的实验。接下来的实验五和六则是“**控制模块对功能模块**”。

好了不多话了，一句老话：从实验中理解。

## 实验五：SOS 信号之一



sos\_module.v 如字面上的意思，是产生“SOS 信号”的“功能模块”。但是看简单一点，就是有次序的控制输出莫斯密码的“点”，“画”和“间隔”。而 control\_modulve.v 是一个简单的定时触发器，每一段时间都会使能 sos\_module.v。

在这里稍微认识一下“摩斯密码”的基本三个元素：

元素	标志	从音节表示
.	点	短音节
-	画	长音节
''	空格（间隔）	无音节

英文字母“S”是“...”，亦即三个短音节之中夹杂间隔。而英文字母“O”是“—”，亦即三个长音节之中夹杂间隔。所以 SOS 形成了“...—...—...”。

你知道吗？有一个美丽的误会说道，铁达尼号是世界上第一艘发送 SOS 求救信号的船。虽然事实不是如此，但是实际上铁达尼号却是发送了 SOS 信号，才有幸存者（Rose 是其中之一）。笔者本身就对摩斯密码非常的偏爱，以前做过的几个实验也是基于摩斯密码。

*sos\_module.v*

```

1. module sos_module
2. (
3.     CLK, RSTn, Pin_Out, SOS_En_Sig
4. );

```

```
5.      input CLK;
6.      input RSTn;
7.      input SOS_En_Sig;
8.      output Pin_Out;
9.
10.
11.     /***** */
12.     //DB4CE15 开发板使用的晶振为 50MHz, 50M*0.001-1=49_999
13.     parameter T1MS = 16'd49_999;
14.
15.     /***** */
16.
17.     reg [15:0]Count1;
18.
19.     always @ ( posedge CLK or negedge RSTn )
20.         if( !RSTn )
21.             Count1 <= 16'd0;
22.         else if( isCount && Count1 == T1MS )
23.             Count1 <= 16'd0;
24.         else if( isCount )
25.             Count1 <= Count1 + 1'b1;
26.         else if( !isCount )
27.             Count1 <= 16'd0;
28.
29.     /***** */
30.
31.     reg [9:0]Count_MS;
32.
33.     always @ ( posedge CLK or negedge RSTn )
34.         if( !RSTn )
35.             Count_MS <= 10'd0;
36.         else if( isCount && Count1 == T1MS )
37.             Count_MS <= Count_MS + 1'b1;
38.         else if( !isCount )
39.             Count_MS <= 10'd0;
40.
41.     /***** */
42.
43.     reg isCount;
44.     reg rPin_Out;
45.     reg [4:0]i;
46.
47.     always @ ( posedge CLK or negedge RSTn )
48.         if( !RSTn )
```

## Verilog HDL 那些事儿 – 建模篇

```
49.          begin
50.              isCount <= 1'b0;
51.              rPin_Out <= 1'b0;
52.              i <= 5'd0;
53.          end
54.      else
55.          case( i )
56.
57.              5'd0 :
58.                  if( SOS_En_Sig ) i <= 5'd1;
59.
60.                  5'd1, 5'd3, 5'd5,      // short
61.                  5'd13, 5'd15, 5'd17 :
62.                      if( Count_MS == 10'd100 ) begin isCount <= 1'b0; rPin_Out <= 1'b0; i <= i + 1'b1; end
63.                      else begin isCount <= 1'b1; rPin_Out <= 1'b1; end
64.
65.                  5'd7, 5'd9, 5'd11 :      // long
66.                      if( Count_MS == 10'd300)begin isCount <= 1'b0; rPin_Out <= 1'b0; i <= i + 1'b1; end
67.                      else begin isCount <= 1'b1; rPin_Out <= 1'b1; end
68.
69.                  5'd2, 5'd4, 5'd6,      // interval
70.                  5'd8, 5'd10, 5'd12,
71.                  5'd14, 5'd16, 5'd18 :
72.                      if( Count_MS == 10'd50 ) begin isCount <= 1'b0; i <= i + 1'b1; end
73.                      else isCount <= 1'b1;
74.
75.                  5'd19 :
76.                      begin rPin_Out <= 1'b0; i <= 5'd0; end    // end
77.
78.          endcase
79.
80.      /*****
81.
82.      assign Pin_Out = rPin_Out;
83.
84.      *****/
85.
86.
87.  endmodule
```

sos\_module.v 的写法与 delay\_module.v 很相似，第 13~39 行是延时器的符合写法。第 45 行定义了 i 寄存器。第 55~78 行是“[仿顺序操作](#)”的写法，i 表示了执行步骤。

第 60~63 行，定义了莫斯密码的短音，换句话说是 100ms 的拉高输出。

第 65~67 行，定义了摩斯密码的长音，亦即 300ms 的拉高输出。

第 69~73 行，定义是空音节，也就是 50ms 的拉低输出。

最后，第 75~76 行是撤销操作，主要是初始化 i 寄存器和 rPin\_Out 寄存器。

**(很抱歉，为了最大限度讲解“控制模块”，实验中多次出现“仿顺序操作”的写法，详细的介绍会在第四章进行。目前先看着办吧。)**

*control\_module.v*

```

1. module control_module
2. (
3.     CLK, RSTn, SOS_En_Sig
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output SOS_En_Sig;
9.
10.    /*****
11.
12.    parameter T3S = 28'd149_999_999;
13.
14.    *****/
15.
16.    reg isEn;
17.    reg [27:0]Count1;
18.
19.    always @ ( posedge CLK or negedge RSTn )
20.        if( !RSTn )
21.            begin
22.                isEn <= 1'b0;
23.                Count1 <= 28'd0;
24.            end
25.        else if( Count1 == T3S )
26.            begin
27.                isEn <= 1'b1;
28.                Count1 <= 28'd0;
29.            end
30.        else
31.            begin
32.                isEn <= 1'b0;
33.                Count1 <= Count1 + 1'b1;

```

```
34.         end
35.
36.     /*************************************************************************/
37.
38.     assign SOS_En_Sig = isEn;
39.
40.     /*************************************************************************/
41.
42. endmodule
```

control\_module.v 是一个简单的触发器。每隔三秒钟就输出一个始终的高电平。

*sos\_generator\_module.v*

```
1. module sos_generator_module
2. (
3.     CLK, RSTn, Pin_Out
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     output Pin_Out;
9.
10.    /************************************************************************/
11.
12.    wire SOS_En_Sig;
13.
14.    control_module U1
15.    (
16.        .CLK( CLK ),
17.        .RSTn( RSTn ),
18.        .SOS_En_Sig( SOS_En_Sig )
19.    );
20.
21.    wire Pin_Out_Wire;
22.
23.    sos_module U2
24.    (
25.        .CLK( CLK ),
26.        .RSTn( RSTn ),
27.        .SOS_En_Sig( SOS_En_Sig ),
28.        .Pin_Out( Pin_Out_Wire )
29.    );
```

```

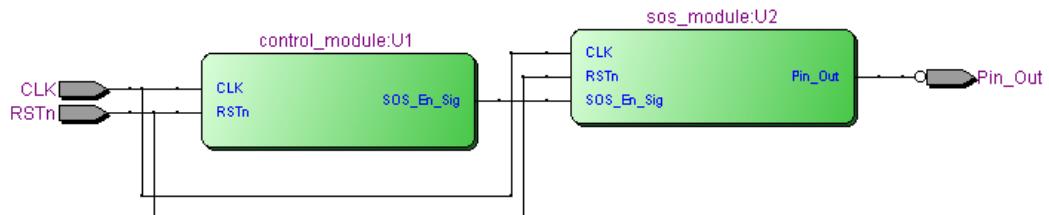
30.
31. *****/
32.
33. assign Pin_Out = !Pin_Out_Wire;
34.
35. *****/
36.
37. endmodule

```

sos\_generator\_module.v 是 sos\_module.v 和 control\_module.v 的组合模块。而第 33 行，是一个取反的操作，因为在实际的硬件资源中，蜂鸣器的触发是**低电平有效**。

**实验五说明：**

完成后的扩展图：



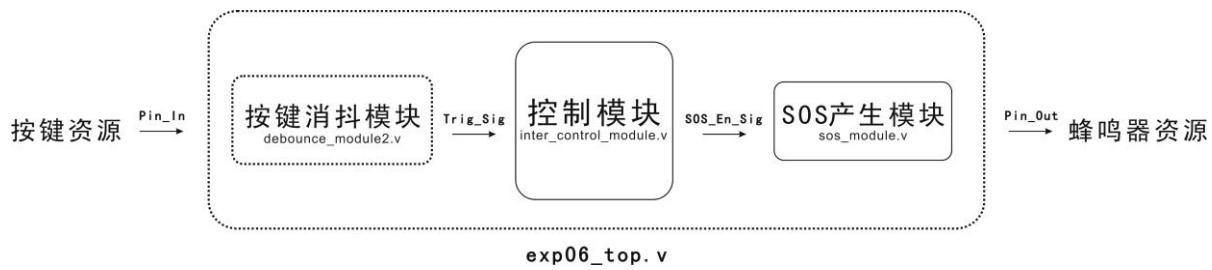
**实验五结论：**

在实验五中，control\_module.v 每隔 3 秒就给 sos\_module.v 一个“**执行**”的触发信号。如果反映至现实，就像老板每隔一段时间，给员工发一个命令。

实验五的设计比较傻瓜，但是主要是解释“**控制模块**”的“**发号**”角色。

## 实验六：SOS 信号之二

实验五中的“控制模块”是站在最顶层执行发号的角色。除此之外“控制模块”还有“[协调](#)”的角色，这也就是实验六的初衷。



图上是，实验六要进行的建模。按键消抖模块是来至实验四的 `debounce_module2.v`，而 `sos_module.v` 是来至实验五。

实验六的内容是要建立一个扮演“[中介](#)”或者称为“[协调](#)”的“[控制模块](#)”。我们可以把 `debounce_module2.v` 看成是一个输入，而 `sos_module.v` 是一个输出。其中 `inter_control_module.v` 负责着两个模块之间的“[沟通](#)”。

设计思路如下：

当按键资源按下的时候 `debounce_module2.v` 会过滤抖动，然后产生一个时钟的高脉冲（`Trig_Sig`）信号。当 `inter_control_module.v` 接收到这个高脉冲信号，它会“[转发](#)”并且产生一个时钟的高脉冲（`SOS_En_Sig`）信号。就这样 `inter_control_module.v` 间接触发 `sos_module.v`。

*inter\_control\_module.v*

```

1. module inter_control_module
2. (
3.     CLK, RSTn, Trig_Sig, SOS_En_Sig
4. );
5.
6.     input CLK;
7.     input RSTn;
8.     input Trig_Sig;
9.     output SOS_En_Sig;
10.

```

```

11.   *****/
12.
13.   reg i;
14.   reg isEn;
15.
16.   always @ ( posedge CLK or negedge RSTn )
17.       if( !RSTn )
18.           begin
19.               i <= 1'd0;
20.               isEn <= 1'b0;
21.           end
22.       else
23.           case( i )
24.
25.               2'd0 :
26.                   if( Trig_Sig ) begin isEn <= 1'b1; i <= 1'd1; end
27.
28.               2'd1 :
29.                   begin isEn <= 1'b0; i <= 1'd0; end
30.
31.           endcase
32.
33.   *****/
34.
35.   assign SOS_En_Sig = isEn;
36.
37.   *****/
38.
39. endmodule

```

*exp06\_top.v*

```

1. module exp06_top
2. (
3.     CLK, RSTn,
4.     Pin_In, Pin_Out
5. );
6.
7.     input CLK;
8.     input RSTn;
9.     input Pin_In;
10.    output Pin_Out;

```

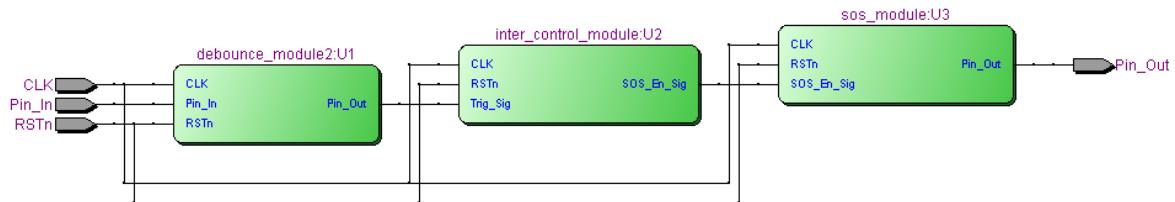
```
11.
12.      *****/
13.
14.      wire Trig_Sig;
15.
16.      debounce_module2 U1
17.      (
18.          .CLK( CLK ),
19.          .RSTn( RSTn ),
20.          .Pin_In( Pin_In ),    // input - from top
21.          .Pin_Out( Trig_Sig ) // output - to U2
22.      );
23.
24.      *****/
25.
26.      wire SOS_En_Sig;
27.
28.      inter_control_module U2
29.      (
30.          .CLK( CLK ),
31.          .RSTn( RSTn ),
32.          .Trig_Sig( Trig_Sig ),        // input - from U1
33.          .SOS_En_Sig( SOS_En_Sig )  // output - to U2
34.      );
35.
36.      *****/
37.
38.      sos_module U3
39.      (
40.          .CLK( CLK ),
41.          .RSTn( RSTn ),
42.          .SOS_En_Sig( SOS_En_Sig ), // input - from U2
43.          .Pin_Out( Pin_Out )       // ouput - to top
44.      );
45.
46.      *****/
47.
48. endmodule
```

实验六说明:

inter\_control\_module.v 在实验六中它是扮演一个“**间接**”或者“**协调**”的角色。第 25~29

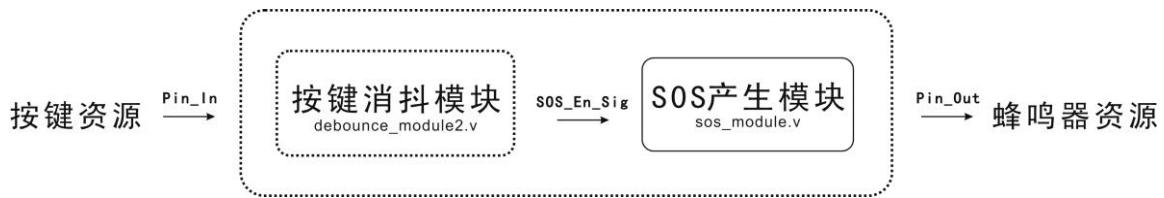
行就是证明了这一点。当有 Trig\_Sig 触发（第 26 行），SOS\_En\_Sig 会产生一个时间的高脉冲。

完成的扩展图：



实验六结论：

虽然从设计的角度上，我们可以再将设计简化到如下图：



在小功能中上图的设计很理智。因为 `debounce_module2.v` 与 `sos_module.v` 之间节省了至少二~三个时间周期的转换。但是在多功能的设计上，如此的设计会产生诸多问题，如“扩展性差”等问题。

在实验六中“控制模块”提供了一个“**扩展**”的空间。假设一个例子，如果我要从实验六的基础上，再添加几个 `debounce_module.v`，那么我们仅要修改组合模块(`exp06_top.v`)和 `inter_control_module.v` 即可。

## 总结

从实验五和实验六我们明白到“控制模块”在“低级建模”中扮演的角色。如前面所说的，“低级建模”是“[多模块](#)”的建模。如果模块种类只有“功能模块”，我们很难有效的将两个“功能模块”“[链接和沟通](#)”起来。其外，还有一点就是“控制模块”有“[发号](#)”的角色，如实验五中的 control\_module.v，每隔 3 秒就会命令 sos\_module.v 工作一次。

在这里读者可能会产生这样的疑问。如果自己是设计者，那么自己要如何规划自己的模块是功能模块还是控制模块。实际上是功能模块也好还是控制模块也好，都是没有强制性的，这完全是取决于设计者。

假设自己觉得“这个模块改为控制模块能更好表达自己的设计...”那么该模块就设计成为控制模块。如果按笔者的习惯，笔者很喜欢将位于中心或者“[控制执行次序](#)”的模块设计成为控制模块，然而周围的都是功能模块，

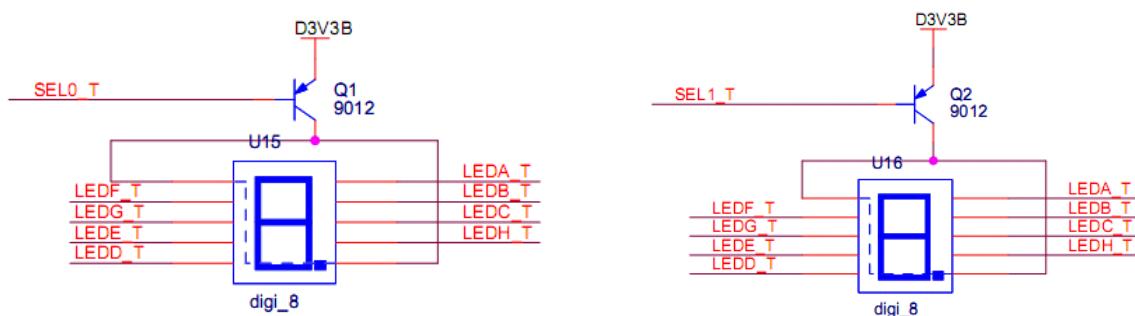
组合模块是一个比较特别的模块，随着建模的层次的不同，它都有不同的意义。

当读者读到这里，笔者已经把“低级建模”的基础，详细的描述一番，并且还使用各个实验来演示。说实话，第二章是很宝贵的一章，因为第二章所涉及的不单单只是“低级建模”的基础知识，而且还包含许多初学者容易疏忽的几个重点。从另一个角度去看，即使读者不打算学习“低级建模”的思路，笔者也相信第二章有读者忽略的重点。

## 第三章：低级建模 - 基础建模

### 3.1 实验七：数码管电路驱动

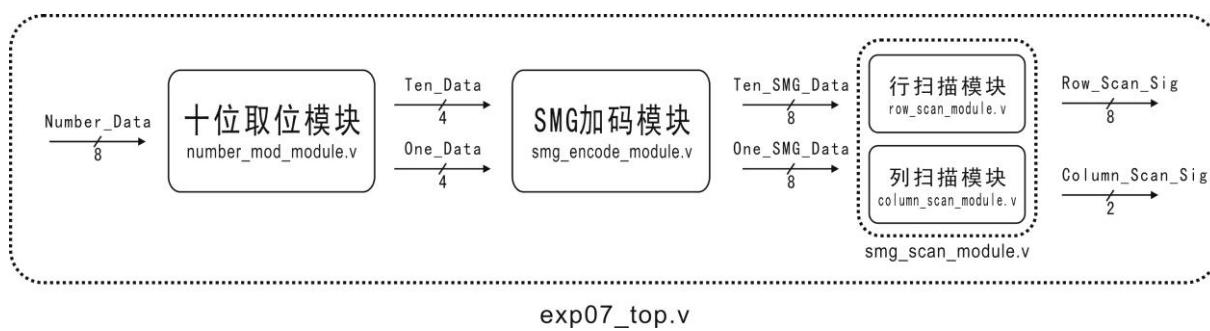
学习过单片机的朋友，多多少少都对数码管有些认识吧？关于这些基础知识，笔者就不再重复了，因为网上已经有一大堆这样的资料。笔者也曾经学过不同驱动方法，但是从黑金开发板的电路来看，笔者觉得设计者真的很 GOOD JOB。因为黑金开发板所承载的电路很基础，所以它才有学习的价值。



数码管是共阳，而是用 PNP 管来反响驱动并且控制列扫描（SEL0\_T 和 SEL1\_T）。而且所有的数码管的“段选信号”（LEDA .. LEDH）都共用同样的引脚。结论来说，数码管都信号都是“**低电平有效**”。

实验七的实验目的，是设计现实最大为 99 数字在 2 个数码管资源上。采取的现实方法是同步动态扫描。估计“**动态扫描**”应该学习过吧，“**同步动态扫描**”是指“**行信号**”和“**列信号**”同步扫描。换句话说，是真正意义上的并行操作。

我们先看看实验七要建立的模块：



实验七的组合模块 exp07\_top.v 是由四个功能模块组成。每一个功能模块的功能都如同本身的命名一样。程序的设计思路如下：

“十位取位模块”接收最大为 99 的数目。然后由该模块进行取位的操作，将十位和各位划分，然后输出独自的信号线 Ten\_Data 和 One\_Data。中间是“SMG 加码模块”，将由二进制组成的数目“转换”成 SMG 码。Ten\_Data 和 One\_Data 被转换后，再经 Ten\_SMG\_Data 和 One\_SMG\_Data 输出。最后由“扫描模块” smg\_can\_module.v，执行“同步动态扫描”点亮数码管。

*number\_mod\_module.v*



“number\_mod\_module.v”的设计很简单，就是利用数学运算符“%”和“/”分别取得十位和各位。因为是十位取位的关系，所以最大的输入数是 00~99 而已。

```
1. module number_mod_module
2. (
3.     CLK, RSTn,
4.     Number_Data,
5.     Ten_Data, One_Data
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input [7:0]Number_Data;
11.    output [3:0]Ten_Data;
12.    output [3:0]One_Data;
13.
14.    /*****
15.
16.    reg [31:0]rTen;
17.    reg [31:0]rOne;
18.
19.    always @ ( posedge CLK or negedge RSTn )
20.        if( !RSTn )
21.            begin
22.                rTen <= 32'd0;
```

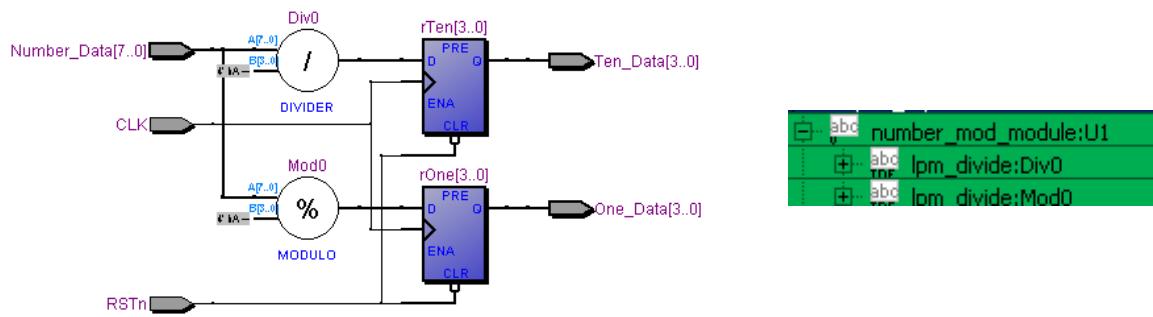
```

23.          rOne <= 32'd0;
24.      end
25.  else
26.  begin
27.      rTen <= Number_Data / 10;
28.      rOne <= Number_Data % 10;
29.  end
30.
31.  /*************************************************************************/
32.
33. assign Ten_Data = rTen[3:0];
34. assign One_Data = rOne[3:0];
35.
36.  /*************************************************************************/
37.
38. endmodule

```

上面的源码比较简单，但是有几点读者必须注意。第 16~17 行声明了 32 位的寄存器，在 27~28 行该寄存器用来寄存取位结果的十位和个位。但是为什么是 32 位呢？

笔者记得在 quartus II 9.0 版本中，“除法器”可以自己定义。但是在更高级的版本中，这些选项就消失了。尤其是说消失，还不如说更“默认化”了。默认下“除法器”和“求余器”是 32 位输出。但是经过“编译”过后，编译器会“自动优化”最适合的位宽。所以这也是为什么，在 16~17 行的声明是 32 位呢！除此之外“除法器”和“求余器”.v 文件会“自动添加”在原本 .v 文件的层次之下。



编译后的“除法器”和“求余器”

编译过后的.v 文件

*smg\_encoder\_module.v*



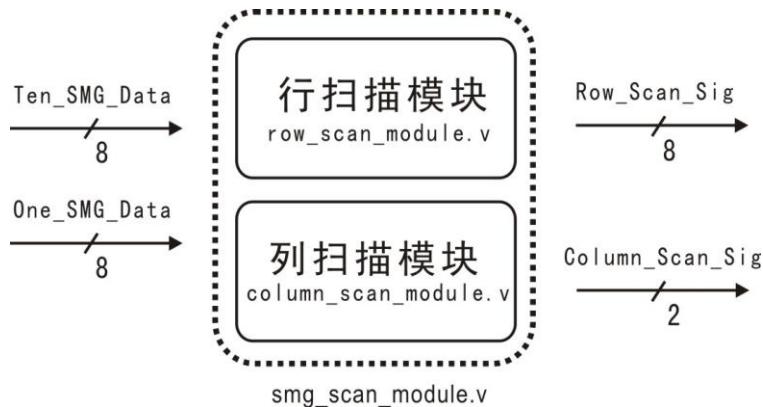
```

1. module smg_encoder_module
2. (
3.     CLK, RSTn,
4.     Ten_Data, One_Data,
5.     Ten_SMG_Data, One_SMG_Data
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input [3:0]Ten_Data;
11.    input [3:0]One_Data;
12.    output [7:0]Ten_SMG_Data;
13.    output [7:0]One_SMG_Data;
14.
15.    /*************************************************************************/
16.
17. parameter _0 = 8'b1100_0000, _1 = 8'b1111_1001, _2 = 8'b1010_0100,
18.           _3 = 8'b1011_0000, _4 = 8'b1001_1001, _5 = 8'b1001_0010,
19.           _6 = 8'b1000_0010, _7 = 8'b1111_1000, _8 = 8'b1000_0000,
20.           _9 = 8'b1001_0000;
21.
22.    /*************************************************************************/
23.
24. reg [7:0]rTen_SMG_Data;
25.
26. always @ ( posedge CLK or negedge RSTn )
27.     if( !RSTn )
28.         begin
29.             rTen_SMG_Data <= 8'b1111_1111;
30.         end
31.     else
32.         case( Ten_Data )
33.
34.             4'd0 : rTen_SMG_Data <= _0;
35.             4'd1 : rTen_SMG_Data <= _1;
36.             4'd2 : rTen_SMG_Data <= _2;
37.             4'd3 : rTen_SMG_Data <= _3;
  
```

```
38.          4'd4 : rTen_SMG_Data <= _4;
39.          4'd5 : rTen_SMG_Data <= _5;
40.          4'd6 : rTen_SMG_Data <= _6;
41.          4'd7 : rTen_SMG_Data <= _7;
42.          4'd8 : rTen_SMG_Data <= _8;
43.          4'd9 : rTen_SMG_Data <= _9;
44.
45.      endcase
46.
47.
48.  *****/
49.
50. reg [7:0]rOne_SMG_Data;
51.
52. always @ ( posedge CLK or negedge RSTn )
53.   if( !RSTn )
54.     begin
55.       rOne_SMG_Data <= 8'b1111_1111;
56.     end
57.   else
58.     case( One_Data )
59.
60.       4'd0 : rOne_SMG_Data <= _0;
61.       4'd1 : rOne_SMG_Data <= _1;
62.       4'd2 : rOne_SMG_Data <= _2;
63.       4'd3 : rOne_SMG_Data <= _3;
64.       4'd4 : rOne_SMG_Data <= _4;
65.       4'd5 : rOne_SMG_Data <= _5;
66.       4'd6 : rOne_SMG_Data <= _6;
67.       4'd7 : rOne_SMG_Data <= _7;
68.       4'd8 : rOne_SMG_Data <= _8;
69.       4'd9 : rOne_SMG_Data <= _9;
70.
71.     endcase
72.
73.  *****/
74.
75. assign Ten_SMG_Data = rTen_SMG_Data;
76. assign One_SMG_Data = rOne_SMG_Data;
77.
78. *****/
79.
80. endmodule
```

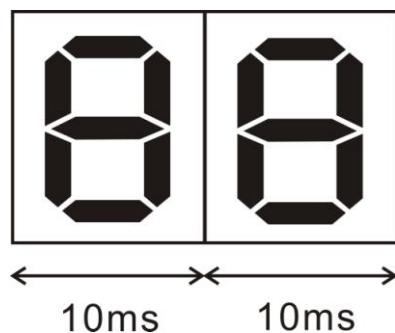
内容依然也很简单，第 17~20 行声明了 SMG 码的常量。第 24~45 行是针对“[十位数](#)”的加码操作（32 行）。第 50~71 行也是同样的操作，但是是针对“[个位数](#)”（58 行）。第 75~76 行是输出。

*smg\_scan\_module.v*



`smg_scan_module.v` 无疑是 `row_scan_module.v` 和 `column_scan_module.v` 的组合模块。无论是“[行扫描模块](#)”还是“[列扫描模块](#)”，扫描频率都制定位 [50hz](#)。

*column\_scan\_module.v*



因为实验七的要求是两个数码管资源，假设各个数码管点亮时间是 [10ms](#)，两个数码管所占用的时间自然是 [20ms](#)。换句话说，就是完成一次扫描占用 [20ms](#) 的周期时间。`column_scan_module.v` 是负责“[列扫描](#)”，亦即每隔 [10ms](#) 就使能（点亮）不同的数码管。

```
1. module column_scan_module
2. (
3.     CLK, RSTn, Column_Scan_Sig
4. );
```

```
5.      input CLK;
6.      input RSTn;
7.      output [1:0]Column_Scan_Sig;
8.
9.      /*****
10.     //DB4CE15 外部晶振为 50M, 50M*0.01-1=499_999
11.     parameter T10MS = 19'd499_999;
12.
13.     *****/
14.     *****/
15.
16.     reg [18:0]Count1;
17.
18.     always @ ( posedge CLK or negedge RSTn )
19.         if( !RSTn )
20.             Count1 <= 19'd0;
21.         else if( Count1 == T10MS )
22.             Count1 <= 19'd0;
23.         else
24.             Count1 <= Count1 + 19'b1;
25.
26.     *****/
27.
28.     reg [1:0]t;
29.
30.     always @ ( posedge CLK or negedge RSTn )
31.         if( !RSTn )
32.             t <= 2'd0;
33.         else if( t == 2'd2 )
34.             t <= 2'd0;
35.         else if( Count1 == T10MS )
36.             t <= t + 1'b1;
37.
38.     *****/
39.
40.     reg [1:0]rColumn_Scan;
41.
42.     always @ ( posedge CLK or negedge RSTn )
43.         if( !RSTn )
44.             rColumn_Scan <= 2'b10;
45.         else if( Count1 == T10MS )
46.             case( t )
47.
48.                 2'd0 : rColumn_Scan <= 2'b10;
```

```
49.          2'd1 : rColumn_Scan <= 2'b01;
50.
51.      endcase
52.
53.      *****/
54.
55.      assign Column_Scan_Sig = rColumn_Scan;
56.
57.      *****/
58.
59.
60. endmodule
```

第 12 行声明了 10ms 的常量。第 16~24 行是 10ms 的定时器，而 28~36 行是用来控制扫描的次序。寄存器 t 表示了[当前的扫描数目](#)是 0 还是 1（是第 0 个数码管还是第 1 个数码管）。第 40~61 行是 column\_scan\_module.v 的具体操作。当 t 等于 0 是将第 0 个数码管使能（拉低），当 t 等于 1 是将第 1 个数码管使能（拉低）。

*row\_scan\_module.v*

row\_scan\_module.v 从字面是“[行扫描](#)”的意思。基于 column\_scan\_module.v 我们知道该模块每隔 10ms 使能不同的数码管资源，而 row\_scan\_module.v 的功能也非常相似。row\_scan\_module.v 主要是每隔 10ms，[输出不同的 SMG 码](#)。具体的操作还是看代码：

```
1. module row_scan_module
2. (
3.     CLK, RSTn,
4.     Ten_SMG_Data, One_SMG_Data,
5.     Row_Scan_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input [7:0]Ten_SMG_Data;
11.    input [7:0]One_SMG_Data;
12.    output [7:0]Row_Scan_Sig;
13.
14.    *****/
15.
16.    parameter T10MS = 19'd499_999;
17.
```

```
18.  *****/
19.
20.  reg [18:0]Count1;
21.
22.  always @ ( posedge CLK or negedge RSTn )
23.      if( !RSTn )
24.          Count1 <= 19'd0;
25.      else if( Count1 == T10MS )
26.          Count1 <= 19'd0;
27.      else
28.          Count1 <= Count1 + 19'b1;
29.
30.  *****/
31.
32.  reg [1:0]t;
33.
34.  always @ ( posedge CLK or negedge RSTn )
35.      if( !RSTn )
36.          t <= 2'd0;
37.      else if( t == 2'd2 )
38.          t <= 2'd0;
39.      else if( Count1 == T10MS )
40.          t <= t + 1'b1;
41.
42.  *****/
43.
44.  reg [7:0]rData;
45.
46.  always @ ( posedge CLK or negedge RSTn )
47.      if( !RSTn )
48.          rData <= 8'd0;
49.      else if( Count1 == T10MS )
50.          case( t )
51.
52.              2'd0 : rData <= Ten_SMG_Data;
53.              2'd1 : rData <= One_SMG_Data;
54.
55.          endcase
56.
57.  *****/
58.
59.  assign Row_Scan_Sig = rData;
60.
61.  *****/
```

```
62.  
63. endmodule
```

第 16~40 行完全和 column\_scan\_module.v 一样。在 44~55 行 row\_scan\_module.v 每隔 10ms 的时间输出不同的 SMG 码。在 t 等于 0 的时候输出 “Ten\_SMG\_Data” (第 52 行), 当 t 等于 1 的时候输出 “One\_SMG\_Data” (第 53 行)。

*smg\_scan\_module.v*

```
1. module smg_scan_module  
2. (  
3.     CLK, RSTn,  
4.     Ten_SMG_Data, One_SMG_Data,  
5.     Row_Scan_Sig, Column_Scan_Sig  
6. );  
7.  
8.     input CLK;  
9.     input RSTn;  
10.    input [7:0]Ten_SMG_Data;  
11.    input [7:0]One_SMG_Data;  
12.    output [7:0]Row_Scan_Sig;  
13.    output [1:0]Column_Scan_Sig;  
14.  
15.    /*****  
16.  
17.    row_scan_module U1  
18.    (  
19.        .CLK( CLK ),  
20.        .RSTn( RSTn ),  
21.        .Ten_SMG_Data( Ten_SMG_Data ), // input - from top  
22.        .One_SMG_Data( One_SMG_Data ), // input - from top  
23.        .Row_Scan_Sig( Row_Scan_Sig ) // output - to top  
24.    );  
25.  
26.    column_scan_module U2  
27.    (  
28.        .CLK( CLK ),  
29.        .RSTn( RSTn ),  
30.        .Column_Scan_Sig( Column_Scan_Sig ) // output - to top  
31.    );  
32.
```

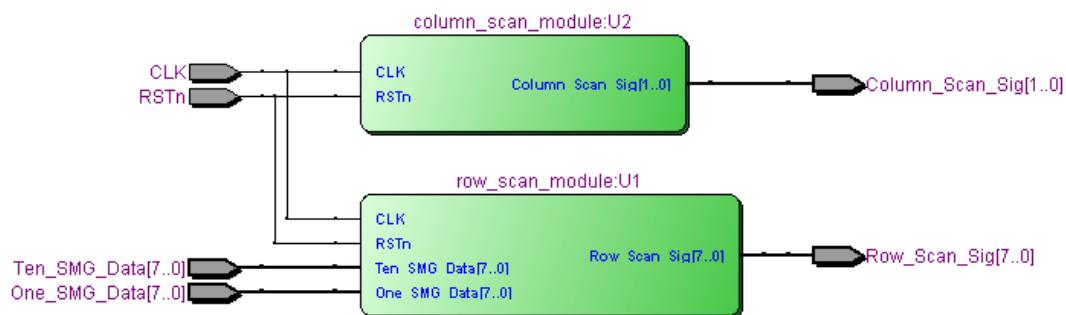
```
33.      ****  
34.  
35. endmodule
```

smg\_scan\_module.v 的组合模块。代码比较简单，自己看着办吧。

总体来说 smg\_scan\_module.v 的功能可以用以下图表来表示：

时间	列扫描	行扫描
初始化	无	无
T0	使能第 0 个数码管资源	十位的 SMG 码
T1	使能第 1 个数码管资源	个位的 SMG 码

完成后的扩展图：



exp07\_top.v

```

1. module exp07_top
2. (
3.     CLK, RSTn,
4.     Number_Data,
5.     Row_Scan_Sig, Column_Scan_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input [7:0]Number_Data;
11.    output [7:0]Row_Scan_Sig;
12.    output [1:0]Column_Scan_Sig;
13.
14.    /*************************************************************************/
15.
16.    wire [3:0]Ten_Data;
17.    wire [3:0]One_Data;
18.
19.    number_mod_module U1

```

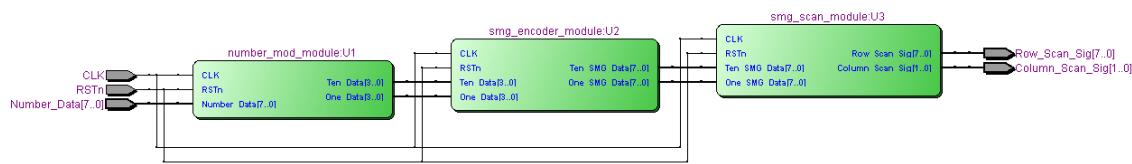
```
20.  (
21.      .CLK( CLK ),
22.      .RSTn( RSTn ),
23.      .Number_Data( Number_Data ), // input - from top
24.      .Ten_Data( Ten_Data ),           // output - to U2
25.      .One_Data( One_Data )           // output - to u2\|U2
26. );
27.
28. *****/
29.
30. wire [7:0]Ten_SMG_Data;
31. wire [7:0]One_SMG_Data;
32.
33. smg_encoder_module U2
34. (
35.     .CLK( CLK ),
36.     .RSTn( RSTn ),
37.     .Ten_Data( Ten_Data ),           // input - from U1
38.     .One_Data( One_Data ),           // input - from U1
39.     .Ten_SMG_Data( Ten_SMG_Data ), // output - to U3
40.     .One_SMG_Data( One_SMG_Data ) // output - to U3
41. );
42.
43. *****/
44.
45. smg_scan_module U3
46. (
47.     .CLK( CLK ),
48.     .RSTn( RSTn ),
49.     .Ten_SMG_Data( Ten_SMG_Data ), // input - from U2
50.     .One_SMG_Data( One_SMG_Data ), // input - from U2
51.     .Row_Scan_Sig( Row_Scan_Sig ), // output - to top
52.     .Column_Scan_Sig( Column_Scan_Sig ) // output - to top
53. );
54.
55. *****/
56.
57. endmodule
```

exp07\_top.v 是组合模块，基本上和图形无两样。

## 实验七说明:

总体来说，我们之所以很自然的自右向左去理解，具体的原因是发生“[图形](#)”的“[连线](#)”上。因为“[图形](#)”将实验七描述成“[从左输入，向右输出](#)”。经我们明白每一个模块的功能之后，再经由“[连线关系](#)”，自然而然很容易明白整个模块的流程和操作。

完成扩展图：



## 实验七结论:

实验七是一个完整且初级的“[低级建模](#)”，而且建模条件完全按照“[低级建模](#)”的准则。虽然“[低级建模](#)”的建模量是很多，对此却使得[设计更灵活](#)。此外“一个模块一个功能”的准则和“[图形](#)”的效果，使整体的表达能力和[解读性大大提升](#)。

实验七的设计单单是针对“[2个数码管资源](#)”，当然读者也可以自行修改将它[扩展到支持“六个数码管资源”与“0~999999”](#)，又或者其他其他的。

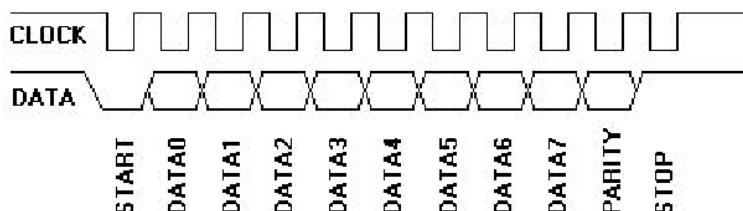
## 3.2 实验八：PS2 解码

PS2 的简单认识

在以前使用单片机对 PS2 进行解码的时候一句话就是苦，因为 PS2 的解码要吗不是中断触发解码，要吗就是查询解码。如果是 CPLD 或者 FPGA 的作为前提的话，PS2 的解码是非常有意义的。

Male 公的	Female 母的	6-pin Mini-DIN (PS/2):	6 脚 Mini-DIN(PS/2)
		1 - Data 2 - Not Implemented 3 - Ground 4 - +5v 5 - Clock	1—数据 2—未实现，保留 3—电源地 4—电源+5V 5—时钟
(Plug) 插头	(Socket) 插座		

PS2 的接口如上图，除了 Pin 5 和 Pin 1 其他的引脚对解码没有什么意义。而下图是 PS2 协议的时序图。PS2 协议对数据的读取是“Clock 的下降沿”有效。PS2 时钟的频率比较慢，大约是 10Khz 左右。



第 N 位	属性
0	开始位
1~8	数据位
9	校验位
10	结束位

PS2 的一帧是 11 位。对 PS2 进行解码时，除了第 1~8 位数据位以外，其余的位都可以无视。

对编码键盘“键盘码”的简单认识

普通计算机采用的都是“编码键盘”，但是“编码键盘”的“编码方式”有分为“第一套”“第二套”和“第三套”。“第二套”的编码使用较为普遍，大致的民用键盘都是采

用“[第二套](#)”编码方式。

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1C	F0,1C	9	46	F0,46	[	54	F0,54
B	32	F0,32	'	0E	F0,0E	INSERT	E0,70	E0,F0,70
C	21	F0,21	-	4E	F0,4E	HOME	E0,6C	E0,F0,6C
D	23	F0,23	=	55	F0,55	PG UP	E0,7D	E0,F0,7D
E	24	F0,24	\	5D	F0,5D	DELETE	E0,71	E0,F0,71
F	2B	F0,2B	BKSP	66	F0,66	END	E0,69	E0,F0,69
G	34	F0,34	SPACE	29	F0,29	PG DN	E0,7A	E0,F0,7A
H	33	F0,33	TAB	0D	F0,0D	U ARROW	E0,75	E0,F0,75
I	43	F0,43	CAPS	58	F0,58	L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B	L SHFT	12	F0,12	D ARROW	E0,72	E0,F0,72
K	42	F0,42	L CTRL	14	F0,14	R ARROW	E0,74	E0,F0,74
L	4B	F0,4B	L GUI	E0,1F	E0,F0,1F	NUM	77	F0,77
M	3A	F0,3A	L ALT	11	F0,11	KP /	E0,4A	E0,F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7C	F0,7C
O	44	F0,44	R CTRL	E0,14	E0,F0,14	KP -	7B	F0,7B
P	4D	F0,4D	R GUI	E0,27	E0,F0,27	KP +	79	F0,79
Q	15	F0,15	R ALT	E0,11	E0,F0,11	KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D	APPS	E0,2F	E0,F0,2F	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	76	F0,76	KP 1	69	F0,69
U	3C	F0,3C	F1	05	F0,05	KP 2	72	F0,72
V	2A	F0,2A	F2	06	F0,06	KP 3	7A	F0,7A
W	1D	F0,1D	F3	04	F0,04	KP 4	6B	F0,6B
X	22	F0,22	F4	0C	F0,0C	KP 5	73	F0,73
Y	35	F0,35	F5	03	F0,03	KP 6	74	F0,74
Z	1A	F0,1A	F6	0B	F0,0B	KP 7	6C	F0,6C
0	45	F0,45	F7	83	F0,83	KP 8	75	F0,75
1	16	F0,16	F8	0A	F0,0A	KP 9	7D	F0,7D
2	1E	F0,1E	F9	01	F0,01	]	5B	F0,5B
3	26	F0,26	F10	09	F0,09	;	4C	F0,4C
4	25	F0,25	F11	78	F0,78	'	52	F0,52
6	36	F0,36	PRNT SCRN	E0,12, E0,7C	E0,F0, 7C,E0, F0,12	.	49	F0,49
7	3D	F0,3D	SCROLL	7E	F0,7E	/	4A	F0,4A
8	3E	F0,3E	PAUSE	E1,14,77, E1,F0,14, F0,77	-NONE-			

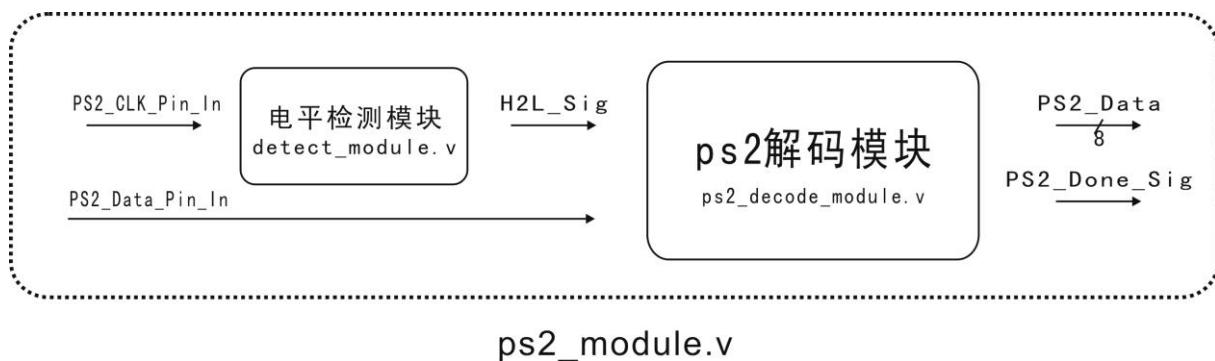
键盘的编码有“**通码**”（Make）和“**断码**”（Break）之分。看得简单一点就是，“**通码**”是某按键的“**按下事件**”，“**断码**”是某按键的“**释放事件**”。

假设，笔者按下“W”键不放，每秒大约会输出 10 个“0x1d”的“**通码**”。然后我释放“W”键，就会输出“0xF0 0x1d”的“**断码**”。编码键盘还有一个老规则，就是一次只能有一个输出而已（多个按键同时按下，只有其中一个有效）。

再假设笔者按下“W”键不放，然后我再按下“X”键不放，那么会输出“0x1d”“0x22”“0x22”……的通码。如果此时笔者放开“X”键，就会输出“0xf0 0x22”的“断码”，此时“W”键的“**通码**”已经无效。但是当笔者释放“W”键的时候，依然会输出“0xf0 0x1d”，“W”键的“**断码**”。

至于组合键“Shift + ?”，“Ctrl + ?”都是软件的工作，我们就无视吧。

实验八的目的很简单，实验主要对 PS2 进行解码。



ps2\_module.v

上图中的组合模块 ps2\_module.v 中包含了“**电平检测模块**” detect\_module.v 和 “**ps2 解码模块**” ps2\_decode\_module 。detect\_module.v 是为了检查 PS2 时钟信号的“**下降沿**”，此外也是为了使模块的建立更简单。ps2\_decode\_module.v 如字面上的意思，对每帧（**十一位一组数据**）的数据进行**解码**和**过滤**的行为，最后将一字节数据输出 PS2\_Data，然后 PS2\_Done\_Sig 产生一个高脉冲，表示完成一次性的解码操作。

detect\_module.v

```
1. module detect_module
2. (
3.     CLK, RSTn,
4.     PS2_CLK_Pin_In,
5.     H2L_Sig
6. );
7.
```

```

8.      input CLK;
9.      input RSTn;
10.     input PS2_CLK_Pin_In;
11.     output H2L_Sig;
12.
13.     /*****
14.
15.     reg H2L_F1;
16.     reg H2L_F2;
17.
18.     always @ ( posedge CLK or negedge RSTn )
19.         if( !RSTn )
20.             begin
21.                 H2L_F1 <= 1'b1;
22.                 H2L_F2 <= 1'b1;
23.             end
24.         else
25.             begin
26.                 H2L_F1 <= PS2_CLK_Pin_In;
27.                 H2L_F2 <= H2L_F1;
28.             end
29.
30.     *****/
31.
32.     assign H2L_Sig = H2L_F2 & !H2L_F1;
33.
34.     *****/
35.
36.
37. endmodule

```

嗯！这个功能模块的解释可以参考实验实验三。

### *ps2\_decode\_module*

```

1. module ps2_decode_module
2. (
3.     CLK, RSTn,
4.     H2L_Sig, PS2_Data_Pin_In,
5.     PS2_Data, PS2_Done_Sig
6. );
7.

```

## Verilog HDL 那些事儿 – 建模篇

```
8.      input CLK;
9.      input RSTn;
10.     input H2L_Sig;
11.     input PS2_Data_Pin_In;
12.     output [7:0]PS2_Data;
13.     output PS2_Done_Sig;
14.
15.     /*****/
16.
17.     reg [7:0]rData;
18.     reg [4:0]i;
19.     reg isShift;
20.     reg isDone;
21.
22.
23.     always @ ( posedge CLK or negedge RSTn )
24.         if( !RSTn )
25.             begin
26.                 rData <= 8'd0;
27.                 i <= 5'd0;
28.                 isDone <= 1'b0;
29.             end
30.         else
31.             case( i )
32.
33.                 5'd0:
34.                     if( H2L_Sig ) i <= i + 1'b1;
35.
36.                 4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8:
37.                     if( H2L_Sig ) begin i <= i + 1'b1; rData[ i-1 ] <= PS2_Data_Pin_In; end
38.
39.                 5'd9, 5'd10:
40.                     if( H2L_Sig ) i <= i + 1'b1;
41.
42.                 5'd11:
43.                     if( rData == 8'hf0 ) i <= 5'd12;
44.                     else i <= 5'd23;
45.
46.                 5'd12, 5'd13, 5'd14, 5'd15, 5'd16, 5'd17, 5'd18, 5'd19, 5'd20, 5'd21, 5'd22:
47.                     if( H2L_Sig ) i <= i + 1'b1;
48.
49.                 5'd23:
50.                     begin i <= i + 1'b1; isDone <= 1'b1; end
51.
```

```

52.           5'd24;
53.           begin i <= 5'd0; isDone <= 1'b0; end
54.
55.           endcase
56.
57.           /*****
58.
59.           assign PS2_Data = rData;
60.           assign PS2_Done_Sig = isDone;
61.
62.           *****/
63.
64. endmodule

```

ps2\_decode\_module.v 最主要的**核心**是在第 31~55 行。同样这个模块也是采用“**仿顺序操作**”的写法。步骤 i一开始会停留在 0 (33 行), 当检查到 H2L\_Sig 的变化 ... 我们知道 PS2 一帧数据的第 0 位是开始位, 所以无视。在接下来的 8 个位都是数据位, PS2 的数据位是从**最低位开始, 最高位结束**。然后对 PS2\_Data\_Pin\_In 引脚进行读值 (第 36~37 行)。

PS2 一帧数据的第 9~10 位是**校验位和结束位**, 执行无视操作 (39~40 行)。在 i 等于 11 的时候 (42 行), 我们要进行判断, 读取的“**数据是通码还是断码?**”, 如果是“**通码**” (44 行), 就进入 49 行随后产生一个高脉冲的完成信号 (49~53 行), 最后返回 33 行, 等待下一次的操作。

如果是“**断码**” (43 行), 保留“**断码的 0xf0**”, 进入 46 行随后对“**断码之后的通码**”采取无视的操作 (46~47 行), 最后产生一个高脉冲的完成信号 (49~53 行), 然后返回第 33 行。

**补充:** 断码 0xf0 也是一种键盘信息。

### *ps2\_module*

```

1. module ps2_module
2. (
3.   CLK, RSTn,
4.   PS2_CLK_Pin_In, PS2_Data_Pin_In,
5.   PS2_Data, PS2_Done_Sig
6. );
7.
8.   input CLK;
9.   input RSTn;

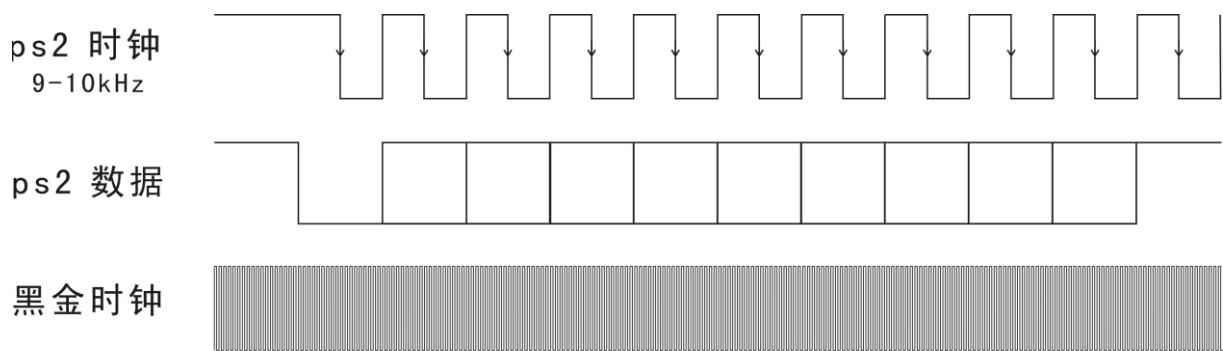
```

```
10.    input PS2_CLK_Pin_In;
11.    input PS2_Data_Pin_In;
12.    output [7:0]PS2_Data;
13.    output PS2_Done_Sig;
14.
15.    /*****/
16.
17.    wire H2L_Sig;
18.
19.    detect_module U1
20.    (
21.        .CLK( CLK ),
22.        .RSTn( RSTn ),
23.        .PS2_CLK_Pin_In( PS2_CLK_Pin_In ), // input - from top
24.        .H2L_Sig( H2L_Sig )           // output - to U2
25.    );
26.
27.    /*****
28.
29.    ps2_decode_module U2
30.    (
31.        .CLK( CLK ),
32.        .RSTn( RSTn ),
33.        .H2L_Sig( H2L_Sig ),           // input - from U1
34.        .PS2_Data_Pin_In( PS2_Data_Pin_In ), // input - from top
35.        .PS2_Data( PS2_Data ),           // output - to top
36.        .PS2_Done_Sig( PS2_Done_Sig ) // output - to top
37.    );
38.
39.    /*****
40.
41. endmodule
```

该模块是组合模块，基本上和图形一样，自己看着办吧。

实验八说明：

实验八的 PS2 解码仅对按键的“[通码](#)”和“[断码的 0xf0](#)”执行操作，相反，却对“[断码之后的通码](#)”采取无视的行为。实验八的重点主要是如何实现“[PS2 时钟采样](#)”和“[PS2 数据采样](#)”操作？



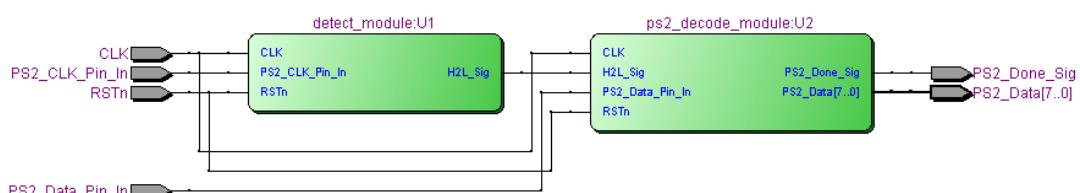
从上图中我们可以看到，黑金的时钟频率（50MHz）比起PS2的时钟频率，大约是差别5000倍。而 detect\_module.v 就是利用这5000个时钟信号去采集每个PS2时钟的下降沿（小箭头）。当 detect\_module.v 检测到ps2时钟产生下降沿，(detect\_module.v 32行)由于布尔的表达式关系，就会产生一个高脉冲经 H2L\_Sig 信号。PS2协议的一帧数据大约是11个数据位，换句话说PS2时钟接下来会产生11个下降沿，也就是说会产生11次高脉冲的H2L\_Sig。

PS2时钟在每一个上升沿的时候，PS2数据就会“设置”（移位）数据，而 ps2\_decode\_module.v 中的 31~55 行，就是对 PS2 数据的移位进行“数据采样”。假设 PS2 时钟在第一次的下降沿，自然而然 PS2 数据会“设置”第 0 位 数据（ps2\_decode\_module.v 34 行），ps2\_decode\_module.v 采取无视的操作，接下来的八位数据位才是真正需要的数据位。

有一点比较值得注意的是在 ps2\_decode\_module.v 第 42~44 行。从上面我们知道，黑金的时钟频率对 ps2 的时钟频率有 5000 倍的相差，从另一个角度去理解的话，距离每一次 PS2 时钟信号的下降沿，黑金拥有 2000 个可以执行的时钟。然而在第 42~44 行只用了一个时间去判断通码还是断码，余下还剩下 4999 个可有的空闲时钟。

也就是说，采样和被采样的频率如果相差越大，采样方可执行的空余的时间就越多，在设计上就越轻松。

完成的扩展图：

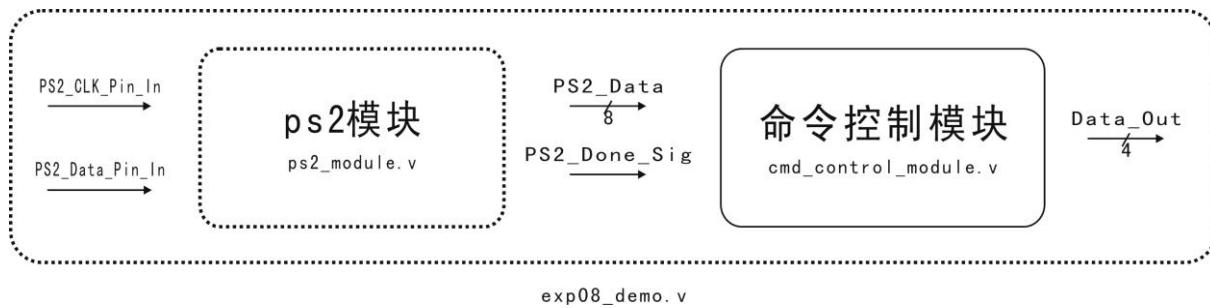


补充：笔者设计的PS2解码的特性是通码和断码的0xf0通吃，但是不吃断码之后的通码。

实验八结论：

实验八是 PS2 解码模块。然而笔者在设计上，对 ps2\_detect\_module.v 添加了 PS2\_Done\_Sig, 这个信号无疑是表示了“一次性操作”。因为编码的键盘，只要按着不放就会一直发送“通码”，如果笔者针对“通码”去设计的话，笔者永远“只能得到键盘的数据”，而得不到“完成（解码）次数”这一点。

## 实验八演示：



实验八演示中的“命令控制模块” cmd\_control\_module.v 会对“W”，“X”和“Ctrl”键作出反应。在 cmd\_control\_module.v 的输出是 4 位的信号，初始值是 4'b0001。当“W”键按下，就会产生左移的效果，“X”键则是右移。至于“Ctrl”键会产生“互换”。

*cmd\_control\_module.v*

```

1. module cmd_control_module
2. (
3.     CLK, RSTn,
4.     PS2_Done_Sig, PS2_Data,
5.     Data_Out
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input PS2_Done_Sig;
11.    input [7:0]PS2_Data;
12.    output [3:0]Data_Out;
13.

```

```

14.      *****/
15.
16.      reg [3:0]rData;
17.
18.      always @ ( posedge CLK or negedge RSTn )
19.          if( !RSTn )
20.              begin
21.                  rData <= 4'b0001;
22.              end
23.          else if( PS2_Done_Sig )
24.              case( PS2_Data )
25.
26.                  8'h1d:
27.                      rData <= { rData[2:0], rData[3] };
28.
29.                  8'h22:
30.                      rData <= { rData[0], rData[3:1] };
31.
32.                  8'h14:
33.                      rData <= { rData[0], rData[1], rData[2], rData[3] };
34.
35.              endcase
36.
37.      *****/
38.
39.      assign Data_Out = rData;
40.
41.      *****/
42.
43. endmodule

```

第 16~35 行是 cmd\_control\_module.v 的核心部分，当 PS2\_Done\_Sig 产生高脉冲（23 行），该控制模块就会对 PS2\_Data 产生反应。8'h1d 是“W”键的通码，而 8'h22 是“X”键的通码，至于 8'h14 是“Ctrl”键的通码。

“W”键，产生左移效果（26~27 行），  
 “X”键，产生右移效果（29~30 行），  
 “Ctrl”键，产生互换效果（32~33 行）。

*exp08\_demo.v*

## 1. module exp08\_demo

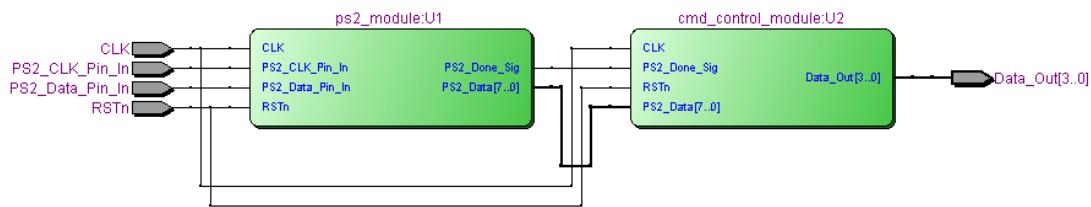
```
2.  (
3.      CLK, RSTn,
4.      PS2_CLK_Pin_In, PS2_Data_Pin_In,
5.      Data_Out
6.  );
7.
8.      input CLK;
9.      input RSTn;
10.     input PS2_CLK_Pin_In;
11.     input PS2_Data_Pin_In;
12.     output [3:0]Data_Out;
13.
14.     ****
15.
16.     wire [7:0]PS2_Data;
17.     wire PS2_Done_Sig;
18.
19. ps2_module U1
20. (
21.     .CLK( CLK ),
22.     .RSTn( RSTn ),
23.     .PS2_CLK_Pin_In( PS2_CLK_Pin_In ), // input - from top
24.     .PS2_Data_Pin_In( PS2_Data_Pin_In ), // input - from top
25.     .PS2_Data( PS2_Data ),           // output - to U2
26.     .PS2_Done_Sig( PS2_Done_Sig )    // ouput - to U2
27. );
28.
29.     ****
30.
31. cmd_control_module U2
32. (
33.     .CLK( CLK ),
34.     .RSTn( RSTn ),
35.     .PS2_Done_Sig( PS2_Done_Sig ), // input - from U1
36.     .PS2_Data( PS2_Data ),       // input - from U1
37.     .Data_Out( Data_Out )        // output - to top
38. );
39.
40.     ****
41.
42. endmodule
```

exp08\_demo.v 是 ps2\_module 和 cmd\_control\_module.v 的组合模块。ps2\_module.v 的具体介绍请参考实验七。

### 实验八演示说明:

在 cmd\_control\_module.v 中的第 23 行，PS2\_Done\_Sig 很有效被利用来控制执行的次数。而且 cmd\_control\_module.v 仅对 “W” “X” “Ctrl” 键的 “通码” 产生反应而已，不会对其他的按键的通码或者断码产生反应。换一句话说，cmd\_control\_module.v 只会在以上三个键在 “按下” 或者 “按下不放” 才会有所反应。

完成后的扩展图：



### 实验八演示结论:

笔者以前接触过很多有关 PS2 的解码实验，都非常执着于 “字符”的输出。其实无论是什么输出，只要有效的利用 “通码” 就不成问题。最重要的是如何编辑这些 “通码” 成为有效的 “命令” 或者 “触发事件” !?

还有另一点，编码键盘虽然很方便，但是编码键盘也有无法取代独立键盘的地方。在实验八演示中，在两个键同时按下的时候，仅有按下最迟的一方才享有输出的权利。相反独立按键是完全 “并行” 执行，不会出现对输出争先恐后的事件。所以在许多方面编码键盘还是不完美。

最后还有一点，就是所有拥有 PS2 接口的设备，一般上都是 “低速” 设备。这一点要好好的记住。

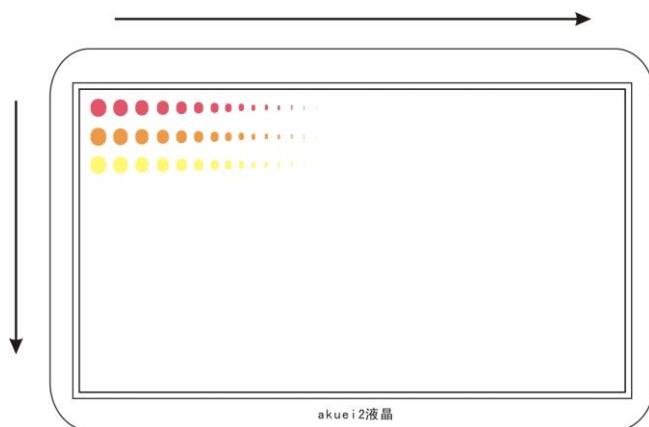
## 实验九：VGA 驱动

### 实验九之一：驱动概念

VGA 驱动实验作为 FPGA 的入门实验多少也有些挑战。VGA 是什么？估计有接触过电脑的朋友多少也是知道。粗略认识 VGA 可分为，VGA 硬件接口，和 VGA 协议。VGA 硬件接口没有什么好学的，VGA 协议才是正点。

VGA 协议主要由 5 个输入信号组成，亦是 HSYNC Signal, VSYNC Signal, RGB Signal。说简单一点，HSYNC Signal 是“[列同步信号](#)”， VSYNC Signal 是“[行同步信号](#)”，RGB Signal 是“[红色-绿色-蓝色 颜色信号](#)”。

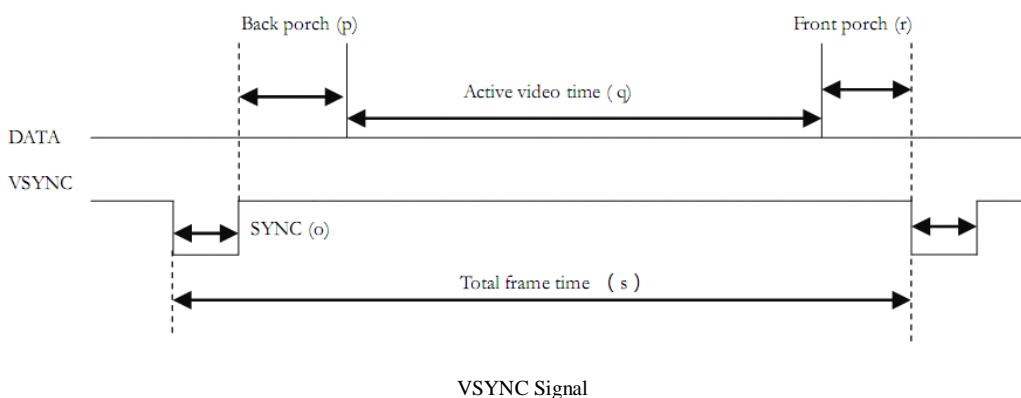
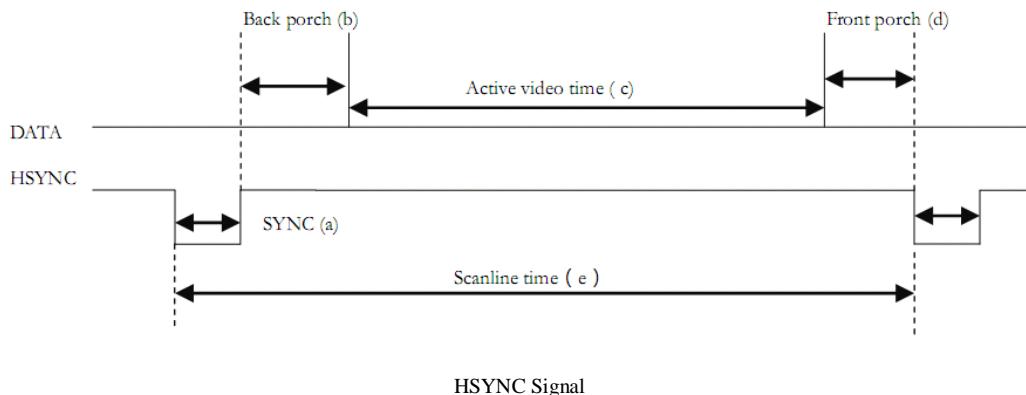
VGA 的扫描是固定的。一帧的屏幕是由“[m 行扫描](#)”和“[n 列填充](#)”组成。假设以 800 x 600 x 60Hz 为例的显示标准（800 宽 x 600 高 x 60Hz），那么宏观上它有 600 行和 800 列为一行。



扫描的次序如下：

- 扫描第 0 行 - 在第 0 行，列填充 0~799。
- 扫描第 1 行 - 在第 1 行，列填充 0 ~ 799。
- 扫描第 2 行 - 在第 2 行，列填充 0 ~ 799。
- 扫描第 m 行 - 在第 m 行，列填充 0 ~ 799。
- 扫描第 598 行 - 在第 598 行，列填充 0 ~ 799
- 直到描第 599 行- 在第 599 行，列填充 0 ~ 799

宏观上，一帧屏幕的显示是由 600 行从上至下扫描，800 列从左至右填充（这也是为什么每当电脑几乎要当机的时候，视屏显示从上之下的延迟扫描）然而微观上，一行的行扫描是由超过 800 个列填充完成。



上图是有关 HSYNC 和 VSYNC 的时序图，以 800 x 600 x 60Hz 为例，信息如下：

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

HSYNC Signal 是用来控制“列填充”，而一个 HSYNC Signal 可以分为 4 个段，也就是 a(同步段)，b(后廊段)，c(激活段)，d(前廊段)。HSYNC Signal 的 a 是拉低的 128 个列像素，b 是拉高的 88 个列像素，至于 c 是拉高的 800 个列像素，而最后的 d 是拉高的 40 个列像素。一列总共有 1056 个列像素。

VSYNC Signal 是用来控制“行扫描”。而一个 VSYNC Signal 同样可以分为 4 个段，也是 o(同步段)，p(后廊段)，q(激活段)，r(前廊段)。VSYNC Signal 的 o 是拉低的 4 个行像素，p 是拉高的 23 个行像素，至于 q 是拉高的 600 个行像素，而最后的 r 是拉高的 1 个行像素。一行总共有 628 个行像素。

**“一个行像素”是以“列像素”为单位的（以 800 x 600 x 60Hz 为例）如下所示：**

1 个行像素 = 1056 个列像素。

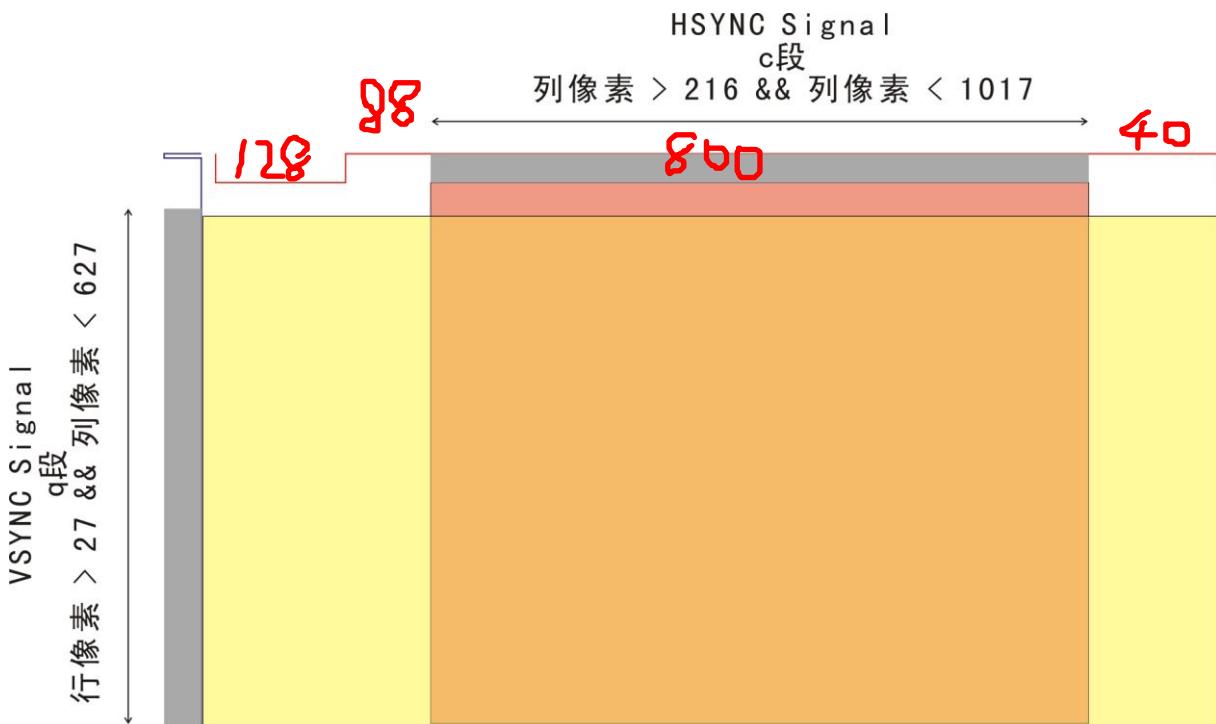
而 “一个列像素” 是以 “时间” 为单位 (以  $800 \times 600 \times 60\text{Hz}$  为例), 如下所示:

1 个列像素 =  $25 \text{ ns}$ 。  

$$1 / (1056 * 628 * 60) \\ = 25.13\text{ns}$$

1 个行像素 = 1056 个列像素 =  $1056 \times 25\text{ns} = 2.64\mu\text{s}$ 。

(以  $800 \times 600 \times 60\text{Hz}$  为例) 在上述内容读者可以发现一个事实, 要完成一行的扫描, 需要 1056 个列像素, 也就是说需要  $1056 \times 25\text{ns}$  的时间。如果要完成所有行的扫描的话, 需要  $628 \times 1056 \times 25\text{ns}$  的时间。很遗憾的是, 不是所有时间都用来显示图片, 有一部分的时间是用来同步操作。

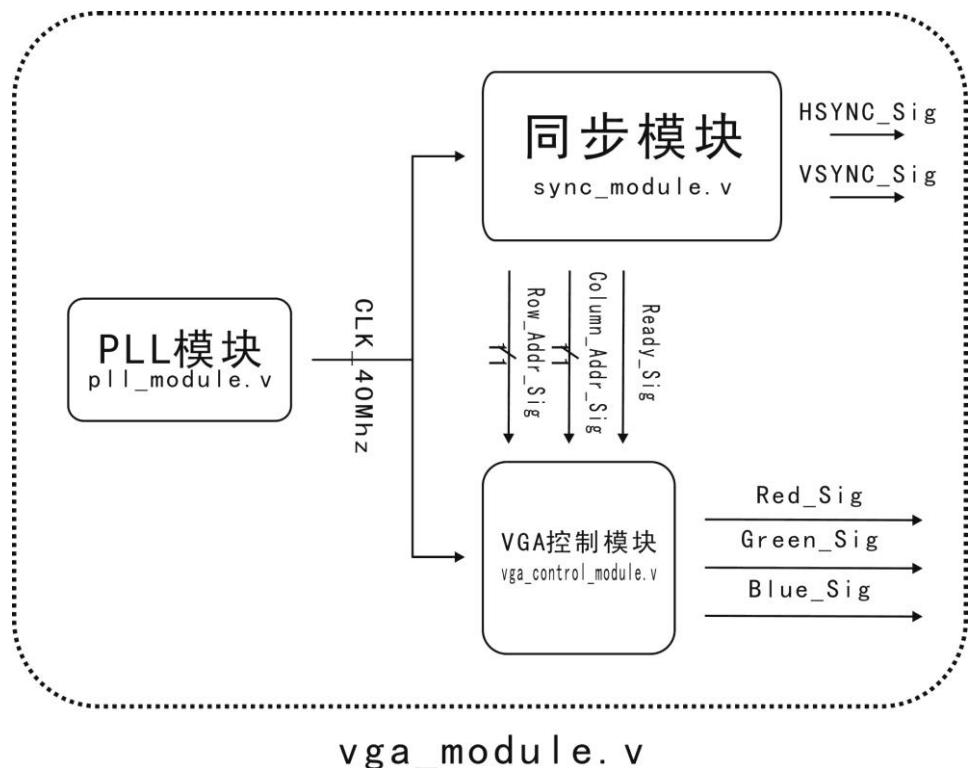


在上图表示了, HSYNC Signal 只有在的 C 段 (红色部分) 和 VSYNC Signal 的 q 段 (黄色部分) 的激活段, 数据的输入才有效。换句话说, 显示图片是发生在交叉的部分, 亦即 “有效区域”。

交叉部分的表达式可以如此描述:

$\text{列像素} > 216 \&& \text{列像素} < 1017 \&& \text{行像素} > 27 \&& \text{行像素} < 627$ 。

接下来我们以简单的实验来说明 VGA 时序规则。



`vga_module.v` 是组合模块，而且它包含了 `pll_module.v`, `sync_module.v` 和 `vga_control_module.v`。我们先说说 `pll_module.v` 的目的：

如果要驱动 VGA 为  $800 \times 600 \times 60Hz$  显示标准，我们知道这个显示标准需要的“**最小单位**”也就是“**列像素**”所占用的时间周期--**25ns**。换句话说，至少需要 40MHz 的时钟频率，而黑金搭载的时钟为 50Mhz，所以需要用到 `pll_module.v` 对黑金的源时钟进行倍频（**四倍频后五分频得到 40MHz**）。

如何实现？

$50\text{Mhz} = 20\text{ns}$  时钟周期

$40\text{Mhz} = 25\text{ns}$  时钟周期

(pll 是 FPGA 的重要硬件资源，pll 主要的功能是对源时钟编程，翻频，分频等 )

而 `sync_module.v` 是对 HSYNC Signal 和 VSYNC Signal 控制的“**功能模块**”。它扮演了“**VGA 驱动的核心**”的角色。除此之外，`sync_module.v` 还输出当前的 x 地址 (Column\_Addr\_Sig)，y 地址 (Row\_Addr\_Sig) 和有效区域信号 (Ready\_Sig)。

`vga_control_module.v` 是“**图像控制的核心**”控制模块，有关 Red, Green, Blue 信号的控制，x 地址和 y 地址的控制，图像显示控制，帧控制，完全都是在这个模块中完成操作。

说简单点，`sync_module.v` 的工作是“**显示标准控制**”，然而 `vga_control_module.v` 的工作是“**图像显示控制**”。

*sync\_module.v*

```
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 ) ) 628
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

```

由于时钟频率已经由 `pll_module.v` 进行倍频，由本来的 50Mhz 变成 40Mhz，所以时钟周期完全符合 800 x 600 x 60Hz 的显示标准。也就是说，时间周期要达到一个列像素的时间，亦即 [25ns](#)。

第 18 行定义的 `Count_H` 是对“[列像素](#)”计数的寄存器。第 20~26 行表示了“[列像素](#)”每 25ns 就会累计，直到 `Count_H` 达到 1056 值。

第 30 行定义了 `Count_V`，它是针对“[行像素](#)”计数的寄存器。而第 32~38 行描述了，“[1 个行像素等于 1056 个列像素](#)（以 800 x 600 x 60Hz 为例）”因为每 1056 个列像素，`Count_V` 就会递增（第 37 行），而第 35 行表示了 `Count_V` 的最大数，亦即 628。

在 42~51 行，表示了“[有效区域](#)”，也就是说数据输入有效的条件。`isReady` 寄存器（42 行）定义了“[有效区域](#)”的标志。

第 55 行表示了 `VSYNC Signal` 的 o 段。`VSYNC Signal` 在 o 段是保持低电平，当 o 段

之后就会拉高输出，而且 o 段保持低电平的时间是 4 个行像素。在 56 行是针对，HSYNC

Signal 的 a 段。我们知道开始的 128 个列像素都是拉低 HSYNC Signal，当 128 个列像素之后就拉高 HSYNC Signal。第 57 行是“[有效区域](#)”的输出信号。

我们都知道屏幕是以“像素”计算。为了要很好的控制图片显示在屏幕的位置，sync\_module.v 有义务输出[当前的 x 地址](#) (Column\_Addr\_Sig) 和[当前的 y 地址](#) (Row\_Addr\_Sig)。x 地址的计数是发生在 128 个 Count\_H 之后 (62 行)，然而 y 地址计数是 27 个 Count\_V 之后 (63 行) 开始计数。

Count\_H 的  $128 = \text{HSYNC Signal 的 a 段} + \text{b 段}$ 。

Count\_V 的  $27 = \text{VSYNC Signal 的 o 段} + \text{p 段}$ 。

isReady 有效区域是  $= \text{HSYNC Signal 的 C 段} * \text{VSYNC Signal 的 q 段}$ 。

Column\_Addr\_Sig  $= \text{HSYNC Signal 的 b 段之后 } 800 \text{ 个列像素}$ 。

Row\_Addr\_Sig  $= \text{VSYNC Signal 的 p 段之后的 } 600 \text{ 个行像素}$ 。

*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;
```

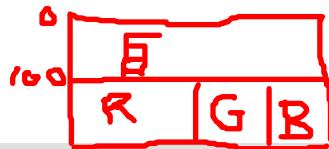
```

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

```

这个 vga\_control\_module.v 比较简单，在第 18 行定义了一个名为 isRectangle 的标志寄存器。在第 23 行，表示了在 x 地址之后，和在 y 地址 100 以内就设置 isRectangle 寄存器为逻辑 1。第 31~33 行，所有颜色信号被赋予同样的表达式，这表示了在 ~~799~~<sup>100</sup> x 100 的区域内显示白色的矩形。Ready\_Sig 有时候可以作为保险。vga\_control\_module.v 作为“图像显示控制”，在这个实验之中它显示了 高 100 x 宽 799 的矩形

vga\_module.v



```

1. module vga_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.    ****
17.
18.    wire CLK_40Mhz;
19.

```

```
20.    pll_module U1
21.    (
22.        .inclk0( CLK ),           // input - from top
23.        .c0( CLK_40Mhz )        // output - inter global
24.    );
25.
26.    /*****
27.
28.    wire [10:0]Column_Addr_Sig;
29.    wire [10:0]Row_Addr_Sig;
30.    wire Ready_Sig;
31.
32.    sync_module U2
33.    (
34.        .CLK( CLK_40Mhz ),
35.        .RSTn( RSTn ),
36.        .VSYNC_Sig( VSYNC_Sig ),      // output - to U3
37.        .HSYNC_Sig( HSYNC_Sig ),      // output - to U3
38.        .Column_Addr_Sig( Column_Addr_Sig ), // output - to U3
39.        .Row_Addr_Sig( Row_Addr_Sig ),      // output - to U3
40.        .Ready_Sig( Ready_Sig )         // output - to U3
41.    );
42.
43.    *****/
44.
45.    vga_control_module U3
46.    (
47.        .CLK( CLK_40Mhz ),
48.        .RSTn( RSTn ),
49.        .Ready_Sig( Ready_Sig ),      // input - from U2
50.        .Column_Addr_Sig( Column_Addr_Sig ), // input - from U2
51.        .Row_Addr_Sig( Row_Addr_Sig ),      // input - from U2
52.        .Red_Sig( Red_Sig ),            // output - to top
53.        .Green_Sig( Green_Sig ),        // output - to top
54.        .Blue_Sig( Blue_Sig )          // output - to top
55.    );
56.
57.    *****/
58.
59. endmodule
```

该模块是组合模块，基本上和图形没哟什么两样。

**实验九之一说明:**

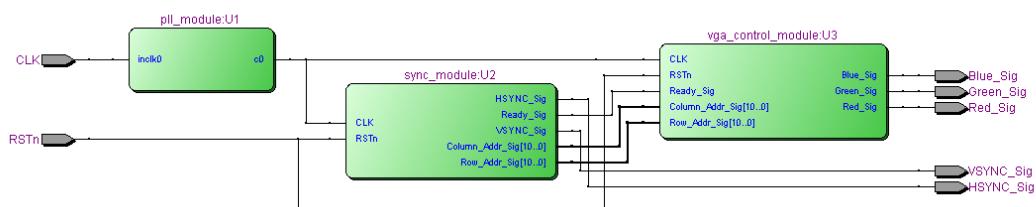
这个实验主要是驱动 VGA， 800 x 600 x 60Hz 为显示标准，同时也描述了 sync\_module.v 在 vga\_module.v 组合模块中扮演的角色。sync\_module.v 直接对 HSYNC Signal 和 VSYNC Signal 控制。同时间 sync\_module.v 也把当前的 x 地址和 y 地址告诉 vga\_control\_module.v 好让这个模块作出相关的显示控制。

除了 800 x 600 x 60Hz 的显示标准以外，还有其他的显示标准可以参考。

显示模式	时钟 ( MHz )	行时序 ( 像素数 )					帧时序 ( 行数 )				
		a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525
640x480@75	31.5	64	120	640	16	840	3	16	480	1	500
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628
800x600@75	49.5	80	160	800	16	1056	3	21	600	1	625
1024x768@60	65	136	160	1024	24	1344	6	29	768	3	806
1024x768@75	78.8	176	176	1024	16	1312	3	28	768	1	800
1280x1024@60	108.0	112	248	1280	48	1688	3	38	1024	1	1066
1280x800@60	83.46	136	200	1280	64	1680	3	24	800	1	828
1440x900@60	106.47	152	232	1440	80	1904	3	28	900	1	932

(常用时钟 Mhz 的时间周期，亦即是 1 个列像素的时间) 从上图我们可以得知 800 x 600 x 60Hz 显示标准的使用时钟是 40Mhz 。这也是实验九之一将 20Mhz 时钟翻倍至 40Mhz 的原因，因为 800 x 600 x 60Hz 显示标准的 1 个列像素需要 25ns。

完成的扩展图：

**实验九之一结论:**

实验九之一实现了简单的 VGA 驱动，同时实验也告诉我们配置 sync\_module.v 等于是配置 VGA 的显示标准。这个实验有一个重点，因为关于源时钟 50Mhz 分频后是 40Mhz，悄悄好达到 1 个列像素的时间单位 25ns。

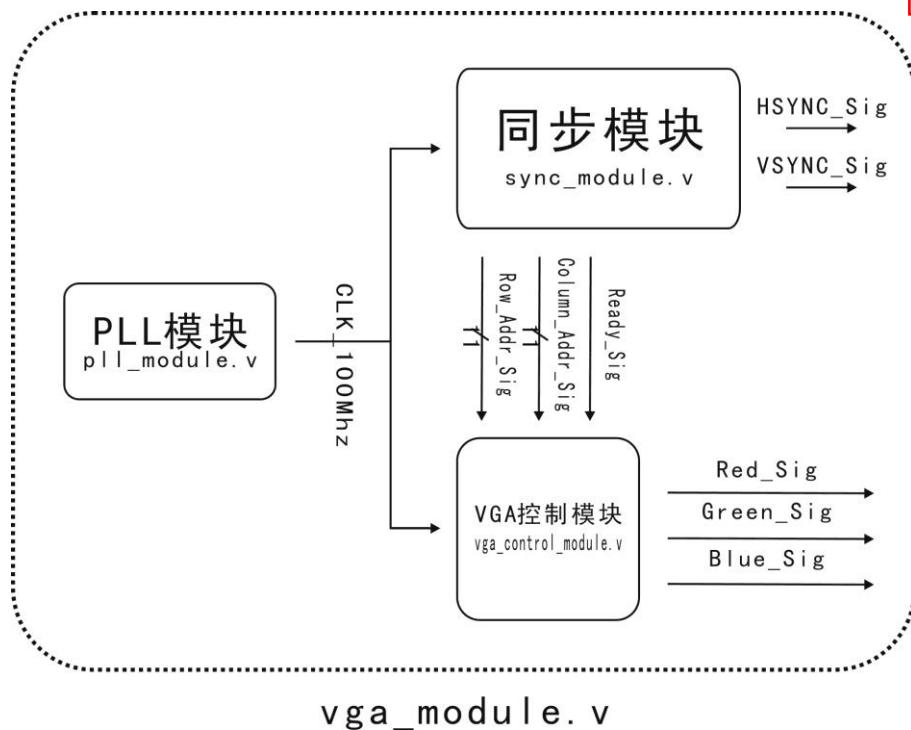
## 实验九之二：向下兼容概念

显示模式	时钟 ( MHz )	行时序 ( 像素数 )					帧时序 ( 行数 )				
		a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525

假设现在我们将源时钟 20Mhz 翻倍到 100Mhz, 640 x 480 x 60Hz 作为 VGA 的显示标准。然而 640 x 480 x 60Hz 的显示标准需要的时钟频率是 **25.175Mhz** 也就是说 1 个列像素需要的时间是 39.7 ns, 那么 100Mhz 如何向下兼容这个 640 x 480 x 60Hz 的显示标准？

时钟源20MHz  
VGA要求时钟25MHz  
那么需要先将时钟倍频到100MHz 然后每4个时钟计数为一个像素时钟(40ns)即可

$$800 * 525 * 60 \text{ Hz} = 25\text{MHz}$$



`vga_module.v` 组合模块同样是没有改变。有改变的只是 `pll_module.v` 将源时钟 20Mhz 翻倍至 100Mhz , 和 `sync_module.v` 显示要求改变致 640 x 480 x 60Hz。

其实向下兼容的概念，也就是模仿“**显示标准的主要时钟频率**”。换一句说，如果 640 x 480 x 60Hz 的主要时钟频率是 25.175Mhz 的话，亦即 1 个列像素是 39.7ns，那么可以用更高的源时钟求得 39.7ns 的定时。说得简单一点就是配置一个定时器。

假设源时钟是 100Mhz，定时器的计数 x 是：

$$\begin{aligned}
 x &= 39.7 \text{ ns} / (1 / 100\text{Mhz}) \\
 &= 3.97 \\
 &= 4
 \end{aligned}$$

*sync\_module.v*

```
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. parameter T40NS = 3'd3;
19.
20.    *****/
21.
22. reg [2:0]Count1;
23.
24. always @ ( posedge CLK or negedge RSTn )
25.     if( !RSTn )
26.         Count1 <= 3'd0;
27.     else if( Count1 == T40NS )
28.         Count1 <= 3'd0;
29.     else
30.         Count1 <= Count1 + 1'b1;
31.
32.    *****/
33.
34. reg [10:0]Count_H;
35.
36. always @ ( posedge CLK or negedge RSTn )
37.     if( !RSTn )
38.         Count_H <= 11'd0;
39.     else if( Count_H == 11'd800 )
40.         Count_H <= 11'd0;
```

```
41.         else if( Count1 == T40NS )
42.             Count_H <= Count_H + 1'b1;
43.
44.     *****/
45.
46.     reg [10:0]Count_V;
47.
48.     always @ ( posedge CLK or negedge RSTn )
49.         if( !RSTn )
50.             Count_V <= 11'd0;
51.         else if( Count_V == 11'd525 )
52.             Count_V <= 11'd0;
53.         else if( Count_H == 11'd800 )
54.             Count_V <= Count_V + 1'b1;
55.
56.     *****/
57.
58.     reg isReady;
59.
60.     always @ ( posedge CLK or negedge RSTn )
61.         if( !RSTn )
62.             isReady <= 1'b0;
63.         else if( ( Count_H >= 11'd144 && Count_H < 11'd784 ) &&
64.                 ( Count_V >= 11'd35 && Count_V < 11'd515 ) )
65.             isReady <= 1'b1;
66.         else
67.             isReady <= 1'b0;
68.
69.     *****/
70.
71.     assign VSYNC_Sig = ( Count_V <= 11'd2 ) ? 1'b0 : 1'b1;
72.     assign HSYNC_Sig = ( Count_H <= 11'd96 ) ? 1'b0 : 1'b1;
73.     assign Ready_Sig = isReady;
74.
75.
76.     *****/
77.
78.     assign Column_Addr_Sig = isReady ? Count_H - 11'd144 : 11'd0; // Count from 0
79.     assign Row_Addr_Sig = isReady ? Count_V - 11'd35 : 11'd0; // Count from 0
80.
81.     *****/
82.
83. endmodule
```

第 18 行定义了 40ns 的常量。而第 22~30 行是用于 40ns 的定时。那样一来，我们不用同样的时钟频率也能向下驱动不同的显示标准（在 41 行的定时作用）。

除此之外，640 x 480 x 60Hz 显示标准要修改的地方还不少：

最大个列像素（第 39 行），最大个行像素（第 51 行），一个行像素所占有的 n 个列像素（53 行），有效区域条件（第 63~64 行），HSYNC Signal 的 a 段拉低时间（71 行），VSYNC Signal 的 o 段拉低时间（72 行），还有 x 地址和 y 地址的输出（第 78~79 行）。

*vga\_control\_module.v*

*vga\_control\_module.v* 和实验九之一基本一样，没有任何改变，这里就不显示了。

*vga\_module.v*

这里指的是，显示“图像”-- 0 \* 100 区域显示白色，没有改变，若显示为其他图像，则也会作出相应调整

```

1. module vga_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.    ****
17.
18.    wire CLK_100Mhz;
19.
20.    pll_module U1
21.    (
22.        .inclk0( CLK ),           // input - from top
23.        .c0( CLK_100Mhz )       // output - to inter global
24.    );
25.
```

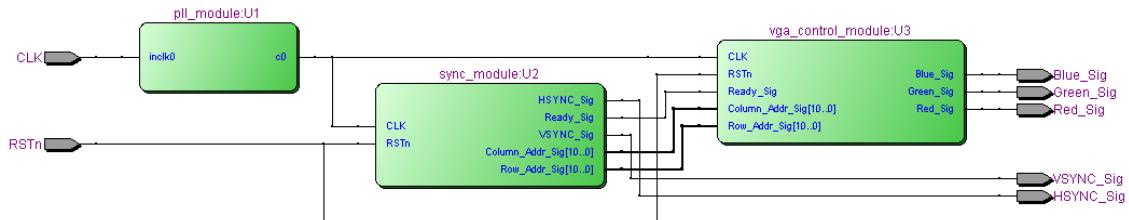
```
26.      *****/
27.
28.      wire [10:0]Column_Addr_Sig;
29.      wire [10:0]Row_Addr_Sig;
30.      wire Ready_Sig;
31.
32.      sync_module U2
33.      (
34.          .CLK( CLK_100Mhz ),
35.          .RSTn( RSTn ),
36.          .VSYNC_Sig( VSYNC_Sig ),           // output - to top
37.          .HSYNC_Sig( HSYNC_Sig ),         // output - to top
38.          .Column_Addr_Sig( Column_Addr_Sig ), // output - to U3
39.          .Row_Addr_Sig( Row_Addr_Sig ),     // output - to U3
40.          .Ready_Sig( Ready_Sig )          // output - to U3
41.      );
42.
43.      *****/
44.
45.      vga_control_module U3
46.      (
47.          .CLK( CLK_100Mhz ),
48.          .RSTn( RSTn ),
49.          .Ready_Sig( Ready_Sig ),           // input - from U2
50.          .Column_Addr_Sig( Column_Addr_Sig ), // input - from U2
51.          .Row_Addr_Sig( Row_Addr_Sig ),     // input - from U2
52.          .Red_Sig( Red_Sig ),             // output - to top
53.          .Green_Sig( Green_Sig ),          // output - to top
54.          .Blue_Sig( Blue_Sig )            // output - to top
55.      );
56.
57.      *****/
58.
59. endmodule
```

vga\_module.v 组合模块也就是修改了 CLK\_100Mhz 的命名而已（第 18, 23, 34, 47 行）。其余的和实验九之一一样。

### 实验九之二说明:

不是所有更高时钟的频率都能向下兼容，如实验九之二一样，20Mhz 的频率，为了兼容 25.175Mhz 至少需要翻频 3 倍以上。

### 完成后扩展图：



### 实验九之二结论:

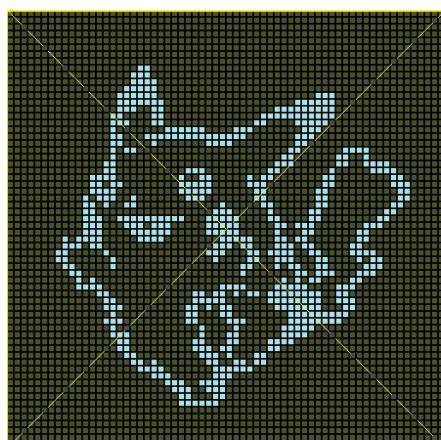
无论是实验九之一或者实验九之二，都是强调了 1 个列像素在驱动 VGA 时所扮演的角色。因为 **1 个列像素是最小的单位**。只要掌握了 1 个列像素，随之可以得到 1 个行像素，有效区域，x 地址和 y 地址，HSYNC Signal 和 VSYNC Signal 等。

## 实验九之三：点阵概念

点阵这东西，一直以来都是笔者的大爱，笔者所有 6 成的实验都是与点阵有关。笔者相信有学过 LED 点阵的同学应该对点阵有所概念。点阵嘛~就是在  $m \times n$  的面积中，实现亮（逻辑 1）和 灭（逻辑 0）效果来产生一副图像。典型的实例如：交通灯的小绿人。但是在 VGA 上实现点阵效果需要下一点功夫。因为 VGA 时序的关系，VGA 有固定的扫描次序。不同于 LED 点阵，用户可以自定义点阵的位长度，扫描方式。

开始实验之前，我们再来复习一下 VGA 扫描的次序（以  $800 \times 600 \times 60Hz$  为例）：一行的扫描是由 1056 个列像素填充，换句话说就是 1 个行像素等于 1056 个列像素。然而图片显示的“有效区域”是在 HSYNC Signal 的 c 段和 VSYNC Signal 的 q 段的交叉部分。

在点阵方面，RGB 信号全为逻辑 1 代表“亮”，相反的 RGB 信号全为逻辑 0 代表“灭”。



上面是一张  $64 \times 64$  像素的比卡丘（我最喜欢了）。点阵编码的方式是“逐行式” + “高位在前”。“逐行式”也就是逐行扫描的意思。“高位在前”的表示如下：



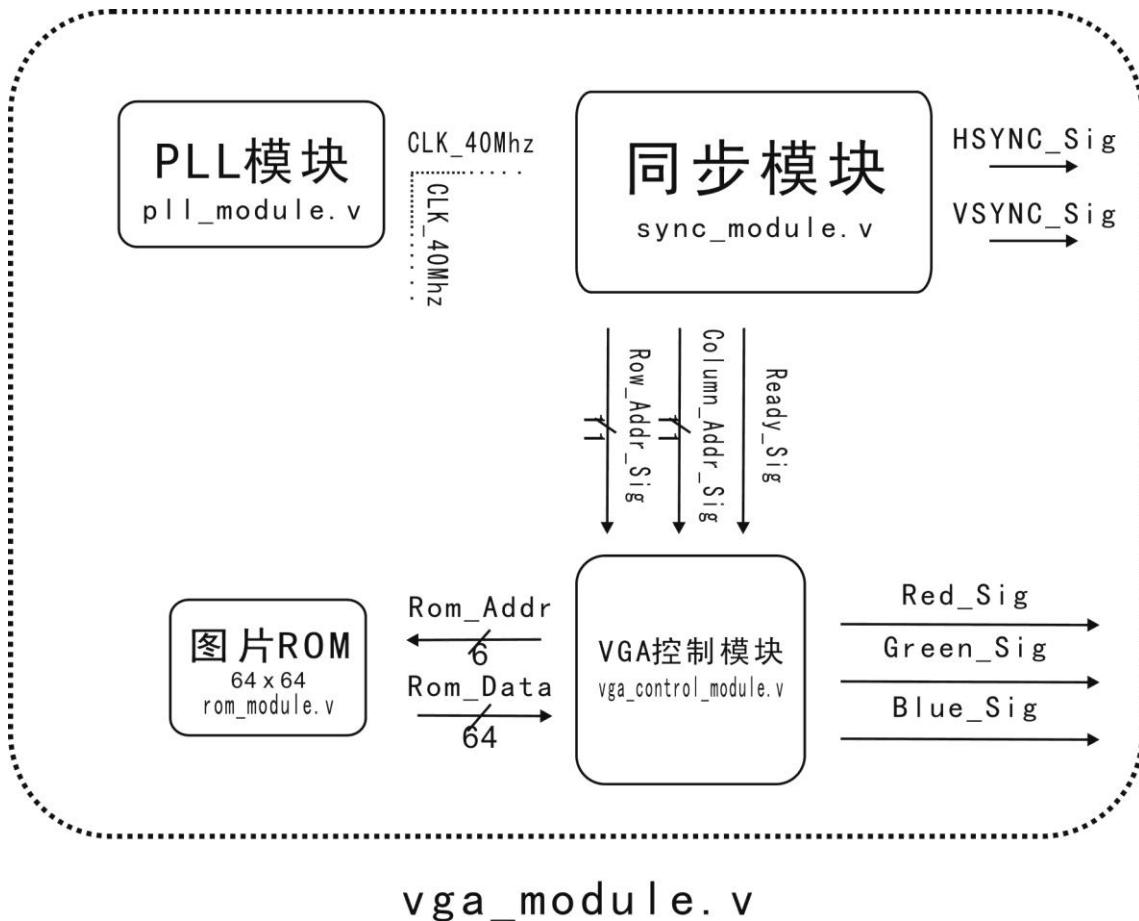
假设有一行拥有 16 个列像素，那么编码会是如此：0xf738。

至于为什么要采用“高位在前”的编码方式？原因在于“符合人类思维”和“容易理解”。

要显示一副  $64 \times 64$  像素点阵的比卡丘图像，所占用的空间是  $64 \text{ Bit (宽)} * 64 \text{ Bit (高)} * 1 \text{ Bit (颜色)}$  亦即 4096 Bit 空间容量。所以，要存放这一副图像， $64 \text{ bit} \times 64 \text{ words}$  的 rom 就行了。因为这样的配置也使得设计简单化，毕竟 1 words 就代表一行，1 Bit 就代表一列。假设笔者要读取第 32 行的第 6 列的值。我只要向 rom 发送 32 的地址，然后取出 rom 的值的第六位即可。

行	64 位点阵信息	行	64 位点阵信息
0	0000000000000000	32	001C3F07C1181800
1	0000000000000000	33	00BC1E01F1B01800
2	0000000000000000	34	00EC0003B1D83000
3	0000000000000000	35	018200071BCC3C00
4	0000000000000000	36	0181000219C61600
5	0000000000000000	37	00C1000031660E00
6	0000000000000000	38	0060000020371C00
7	0000000000000000	39	0028000000AC3000
8	0000180000000000	40	003C0001E1F0E000
9	0000380000000000	41	0006001B5BFCC000
10	00003C0000000000	42	0003001F03FCE000
11	00007C0000000000	43	0003803581FC6000
12	0000FC0000070000	44	00010031807FC000
13	0000FE00003F0000	45	00038030E07F8000
14	0000E60001FF0000	46	0001000840F80000
15	0001860007FF0000	47	0001C008C0F00000
16	0001C6000C3E0000	48	0000C018C3C00000
17	000101F8701C0000	49	0000C01D83000000
18	00018FFE0380000	50	0000600E8C000000
19	0001840780300000	51	000034066C000000
20	000160000607800	52	00001E03F0000000
21	000100000C0C800	53	0000070380000000
22	0003000001C0CF00	54	000001CF00000000
23	0003003803010780	55	000000E700000000
24	000F004C160300C0	56	0000006300000000
25	000D807C1C0300E0	57	0000007E00000000
26	000F007C0C060060	58	0000000000000000
27	001720380C060060	59	0000000000000000
28	001A6001060600C0	60	0000000000000000
29	00180003820C0180	61	0000000000000000
30	001CFFC7C30C0300	62	0000000000000000
31	001C7707C1B80E00	63	0000000000000000

上面是 64 x 64 比卡丘图像的点阵信息。



图上的 vga\_module.v 是基于实验九之一的 vga\_module.v。主要是对 vga\_control\_module 进行扩展和修改，支持从 rom\_module.v 读取数据。之外 vga\_control\_module.v 还控制图片的信息和输出。具体的操作直接看代码吧。

#### *sync\_module.v*

```

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;
```

## Verilog HDL 那些事儿 – 建模篇

```
57.     assign Ready_Sig = isReady;
58.
59.
60.     *****/
61.
62.     assign Column_Addr_Sig = isReady ? Count_H - 11'd216 : 11'd0; // Count from 0
63.     assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0; // Count from 0
64.
65.     *****/
66.
67. endmodule
```

基本上和实验九之一一模一样。显示标准是 800 x 600 x 60Hz。

*vga\_control\_module.v*

```
1. module vga_control_module
2. (
3.     CLK, RSTn,
4.     Ready_Sig, Column_Addr_Sig, Row_Addr_Sig,
5.     Rom_Data, Rom_Addr,
6.     Red_Sig, Green_Sig, Blue_Sig
7. );
8.     input CLK;
9.     input RSTn;
10.    input Ready_Sig;
11.    input [10:0]Column_Addr_Sig;
12.    input [10:0]Row_Addr_Sig;
13.
14.    input [63:0]Rom_Data;
15.    output [5:0]Rom_Addr;
16.
17.    output Red_Sig;
18.    output Green_Sig;
19.    output Blue_Sig;
20.
21.    *****/
22.
23.    reg [5:0]m;
24.
25.    always @ ( posedge CLK or negedge RSTn )
26.        if( !RSTn )
```

```

27.           m <= 6'd0;
28.           else if( Ready_Sig && Row_Addr_Sig < 64 )
29.               m <= Row_Addr_Sig[5:0];
30.
31.   ****
32.
33.   reg [5:0]n;
34.
35.   always @ ( posedge CLK or negedge RSTn )
36.       if( !RSTn )
37.           n <= 6'd0;
38.       else if( Ready_Sig && Column_Addr_Sig < 64 )
39.           n <= Column_Addr_Sig[5:0];
40.
41.   ****
42.
43.   assign Rom_Addr = m;
44.
45.   assign Red_Sig = Ready_Sig ? Rom_Data[ 6'd63 - n ] : 1'b0;
46.   assign Green_Sig = Ready_Sig ? Rom_Data[ 6'd63 - n ] : 1'b0;
47.   assign Blue_Sig = Ready_Sig ? Rom_Data[ 6'd63 - n ] : 1'b0;
48.
49.   ****
50.
51.
52. endmodule

```

在第 5 行，第 14~15 行，是针对 rom 进行修改，主要是加了 Rom\_Data 和 Rom\_Addr。第 23~29 行表示了 m 寄存器是读取 当前的 y 地址 (Row\_Addr\_Signal)，亦即当前的行。而第 33~39 行的 n 寄存器则是读取当前的 x 地址 (Column\_Addr\_Sig)，也就是当前的列。

在 43 行中，m 寄存器作为“[行寻址](#)”将值赋予 Rom\_Addr。最后 45~47 行，n 寄存器作为“[列寻址](#)”，对 rom 读取的值 Rom\_Data，从[最高位到最低位](#)判断目前的第 m 行第 n 列是“[点亮](#)”还是“[点灭](#)”。

假设一个情况，笔者要显示图片第 0 行。那么图片第 0 行的值，当然是存放在 rom 的 0 地址（详细的 rom 配置和创建过程请参考实验配置笔记）。如果笔者要在屏幕的“[第 0 y 地址](#)”和“[第 0 x 地址](#)”显示开始显示图片的话，我们可以利用“[当前 y 地址](#) (Row\_Addr\_Sig)”和“[当前 x 地址](#) (Column\_Addr\_Sig)”来告诉了我们“[当前的显示进度](#)”。

如果“[当前 y 地址](#)”是 0，直接利用“[当前 y 地址](#)”对 rom 进行“[行寻址](#)”从 rom 读取图片第 0 行的信息，然后再利“[当前 x 地址](#)”对读出图片的第 0 行信息进行“[列寻址](#)”。从高位到低位进行判断，“[该图像信息遭当前显示的 x , y 地址](#)”是否是“[点亮](#)”还是

点“[点灭](#)”。

如：图片第 0 行的信息全部都是逻辑 0，所以在“[当前 y 地址](#)” 0，和“[当前 x 地址](#)” 从 0~63 都是“[点灭](#)”的操作。上述的过程会一直重复直到 64 行的图片信息扫描完毕。

*vga\_module.v*

```
1. module vga_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.    /*****
17.
18.    wire CLK_40Mhz;
19.
20.    pll_module U1
21.    (
22.        .inclk0( CLK ),           // input - from top
23.        .c0( CLK_40Mhz )        // output - to inter global
24.    );
25.
26.    /*****
27.
28.    wire [10:0]Column_Addr_Sig;
29.    wire [10:0]Row_Addr_Sig;
30.    wire Ready_Sig;
31.
32.    sync_module U2
33.    (
34.        .CLK( CLK_40Mhz ),
```

```

35.      .RSTn( RSTn ),
36.      .VSYNC_Sig( VSYNC_Sig ),           // input - from top
37.      .HSYNC_Sig( HSYNC_Sig ),           // input - from top
38.      .Column_Addr_Sig( Column_Addr_Sig ), // output - to U4
39.      .Row_Addr_Sig( Row_Addr_Sig ),       // output - to U4
40.      .Ready_Sig( Ready_Sig )           // output - to U4
41. );
42.
43. ****
44.
45. wire [63:0]Rom_Data;
46.
47. rom_module U3
48. (
49.     .clock( CLK_40Mhz ),
50.     .address( Rom_Addr ),           // input - from U4
51.     .q( Rom_Data )               // output - to U4
52. );
53.
54. ****
55.
56. wire [5:0]Rom_Addr;
57.
58. vga_control_module U4
59. (
60.     .CLK( CLK_40Mhz ),
61.     .RSTn( RSTn ),
62.     .Ready_Sig( Ready_Sig ),           // input - from top
63.     .Column_Addr_Sig( Column_Addr_Sig ), // input - from U2
64.     .Row_Addr_Sig( Row_Addr_Sig ),       // input - from U2
65.     .Rom_Data( Rom_Data ),             // input - from U3
66.     .Rom_Addr( Rom_Addr ),           // output - to U3
67.     .Red_Sig( Red_Sig ),              // output - to top
68.     .Green_Sig( Green_Sig ),          // output - to top
69.     .Blue_Sig( Blue_Sig )            // output - to top
70. );
71.
72. ****
73.
74. endmodule

```

第 43~54 行主要是多了 rom 的实例化, 然后在第 54~72 行 vga\_control\_module.v 实例化中, 对 rom 扩展 Rom\_Data 和 Rom\_Addr 的输入输出口。

注意：第 45 行定义了位宽为 64 位的 Rom\_Data 的 wire。然而第 56 行定义了位宽为 6 的 Rom\_Addr 的 wire。这表示了 rom\_module.v 的 1 字数据是 64 位宽，Rom\_Addr 位宽为 6 表示可以支持到 64 个 words 的寻址。

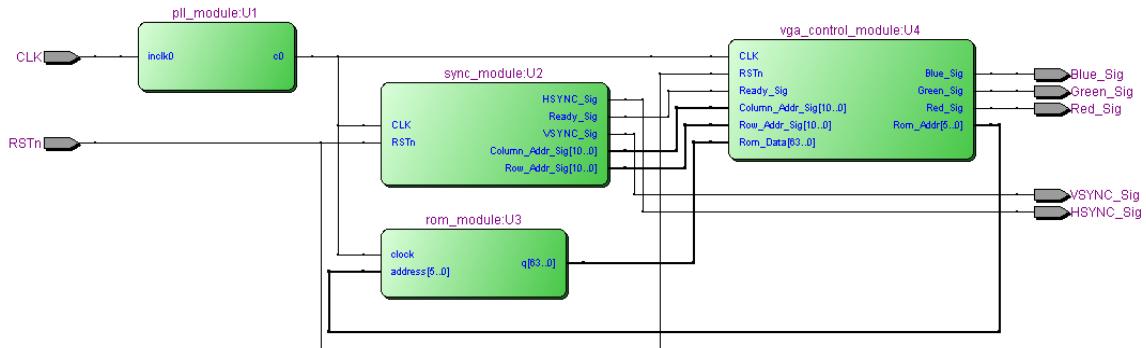
### 实验九之三说明:

实验九之三主要是对“[点阵](#)”的概念叙述一番。此外还说明了“[屏幕](#)”的x地址和y地址在图片显示上的作用。

如果要将一张图片显示在屏幕上任一个位置，需要好好利用从 sync\_module.v 输出的 Column\_Addr\_Sig 和 Row\_Addr\_Sig 信号，因为这两个信号表示了当前显示的 x 地址和 y 地址（当前显示 m 行 n 列）。

此外实验九之三还很有效的利用 Row\_Addr\_Sig 对 rom\_module.v 进行“[行寻址](#)”，然后利用 Column\_Addr\_Sig 对执行“[列寻址](#)”的操作。

完成后的扩展图：



### 实验九之三结论:

说一点题外话，在 LED 点阵中，因为 LED 点阵的扫描次序完全是用户自定义的，所以“[点阵](#)”编码的方式也是用户自定义的。不同与 LED 点阵，VGA 有一套自己的扫描次序，为了降低设计的猥琐度，点阵的编码方式还是乖乖迎合 VGA 的扫描次序。

VGA 扫描方式是逐行对列填充，亦即每一行的列填充都是从左至右。那么“高位在前”的“[点阵编码](#)”显然很符合 VGA 的“从左至右的列填充”的扫描次序。

还有一点问题点就是，关于“[用什么方式去储存图片](#)”？如果读者的图片是 16x16 像素，那么笔者很建议读者在 vga 控制模块里定义图片每一行信息的常量。如：

```
parameter Line1 = 16'hffff, Line2 = 16'h02ee .....
```

如果图片是超过这样 16 x 16 像素，那么还是建立一个 rom 来存放图片信息。不要像 C

语言那样，建立一个库文件，然后调用。Verilog HDL 语言不适合这一套。

## 实验九之四：图层概念

将一张有颜色信息的图片现在屏幕上，传统通常都会用“[颜色编码](#)”的方式。假设一个例子，某 VGA 驱动电路可以支持 8 中颜色，也就是说每一列“[点阵](#)”都使用 3Bit 来代表。颜色支持如下：

颜色	Red Signal	Green Signal	Blue Signal
黑色	0	0	0
蓝色	0	0	1
绿色	0	1	0
浅蓝色	0	1	1
红色	1	0	0
紫色	1	0	1
黄色	1	1	0
白色	1	1	1

假设我有一行 4 个列像素的信息（4 个列点阵）：



那么一行的长度需要 12 位，编码结果是：100 101 111 000。对于“[颜色编码](#)”的方式，对于硬件来说，无论是储存编码信息，还是处理编码信息，实在是有点过分。所以不怎么推荐。

在这里我们还用另一种方式可以显示颜色图片，就是“[图层概念](#)”。  
举一个例子，假设我要显示 64 x 64 只带有颜色的比卡丘（爱！爱！）：



3 色层集合



红色层



绿色层



蓝色层

基本上我会将该图片分为“基本 3 色的图层”，亦即“红色层”，“绿色层”和“蓝色层”。颜色的支持如下：

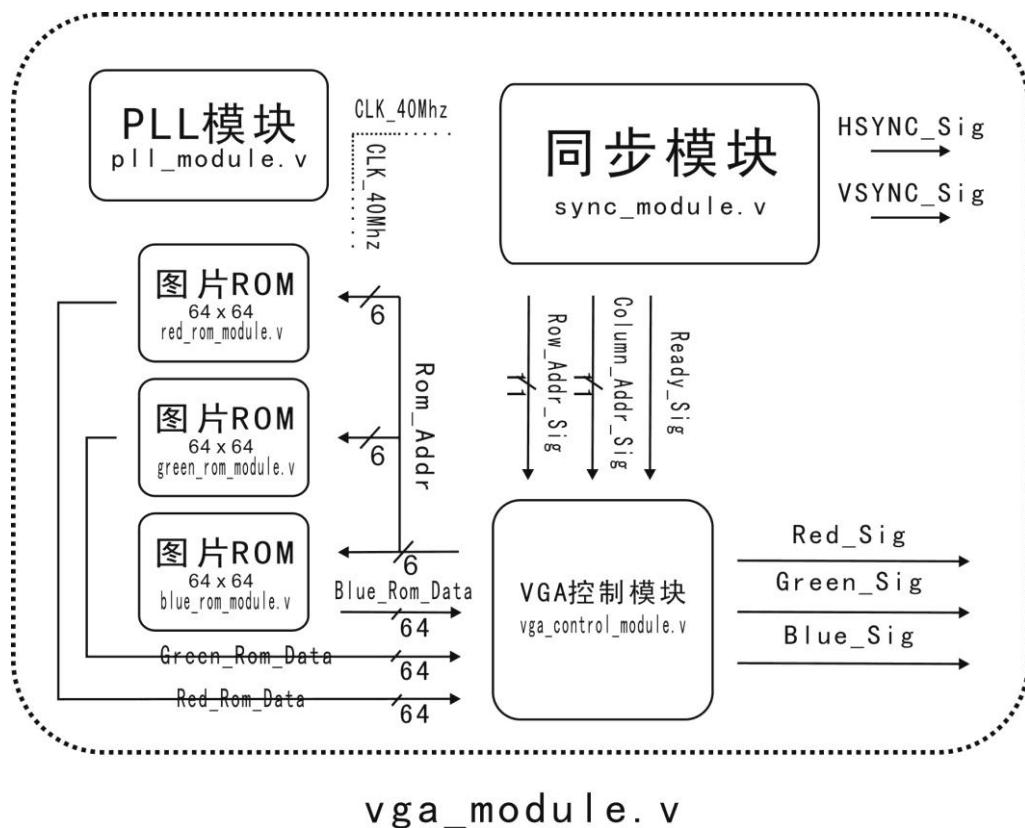
颜色	Red Signal	Green Signal	Blue Signal
红色	1	0	0
黄色	1	1	0
黑色	1	1	1
白色	0	0	0

又假设我有一行 4 个列像素的信息 (4 个列点阵):



颜色层	第四位	第三位	第二位	第一位
红色层	1	1	0	1
绿色层	0	1	0	1
蓝色层	0	0	0	1

看简单点，红色的输出，只要红色层即可。黄色的输出需要红色层和绿色层相叠加。白色也等于没有颜色，这时候不需要任何层的叠加。则黑色需要 3 个层叠加才能显示。



实验九之四主要是建立如上图的组合模块。从实验九之三的基础上，添加了 3 个 rom\_module 亦即 red\_rom\_module.v, green\_rom\_module.v 和 blue\_rom\_module.v。每

---

一个 rom\_module.v 如命名般分别储存“**红色层**”，“**绿色层**”，和“**蓝色层**”的点阵信息。最后会经过 vga\_control\_module.v 巧妙的“**叠加**”操作，然后显示一张带有颜色的图片。

*sync\_module.v*

```
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'd216 : 11'd0; // Count from 0
63.    assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0; // Count from 0
64.
65.    *****/
66.
67. endmodule

```

大致上和实验九之三一样。

*vga\_control\_module.v*

```

1. module vga_control_module
2. (
3.     CLK, RSTn,
4.     Ready_Sig, Column_Addr_Sig, Row_Addr_Sig,
5.     Red_Rom_Data, Green_Rom_Data, Blue_Rom_Data, Rom_Addr,
6.     Red_Sig, Green_Sig, Blue_Sig
7. );
8.     input CLK;
9.     input RSTn;
10.    input Ready_Sig;

```

```
11.    input [10:0]Column_Addr_Sig;
12.    input [10:0]Row_Addr_Sig;
13.
14.    input [63:0]Red_Rom_Data;
15.    input [63:0]Green_Rom_Data;
16.    input [63:0]Blue_Rom_Data;
17.    output [5:0]Rom_Addr;
18.
19.    output Red_Sig;
20.    output Green_Sig;
21.    output Blue_Sig;
22.
23.
24.    /*****
25.
26.    reg [5:0]m;
27.
28.    always @ ( posedge CLK or negedge RSTn )
29.        if( !RSTn )
30.            m <= 6'd0;
31.        else if( Ready_Sig && Row_Addr_Sig < 64 )
32.            m <= Row_Addr_Sig[5:0];
33.        else
34.            m <= 6'd0;
35.
36.    reg [5:0]n;
37.
38.    always @ ( posedge CLK or negedge RSTn )
39.        if( !RSTn )
40.            n <= 6'd0;
41.        else if( Ready_Sig && Column_Addr_Sig < 64 )
42.            n <= Column_Addr_Sig[5:0];
43.        else
44.            n <= 6'd0;
45.
46.    /*****
47.
48.    assign Rom_Addr = m;
49.
50.    assign Red_Sig = Ready_Sig ? Red_Rom_Data[ 6'd63 - n ] : 1'b0;
51.    assign Green_Sig = Ready_Sig ? Green_Rom_Data[ 6'd63 - n ] : 1'b0;
52.    assign Blue_Sig = Ready_Sig ? Blue_Rom_Data[ 6'd63 - n ] : 1'b0;
53.
54.    *****/
```

---

```

55.
56.
57. endmodule

```

第 5 行分别定义了 Red\_Rom\_Data, Green\_Rom\_Data, Blue\_Rom\_Data, 3 个 rom 的输入口, 均为 64 位宽(第 14~16 行)。第 26~46 行, 是读取当前的 m 行(y 地址-Row\_Addr\_Sig) 和当前的 n 列 (x 地址-Column\_Addr\_Sig)。

在 48 行定义了 Rom\_Addr 输出值完全是基于 m (当前的 y 地址, 当前显示的行)。然而有一点不一样的是发生在第 50~52 行, 各个 RGB Signal 都有各自的输出值 (各自都有自己 ROM 信息库)。如: Red\_Sig 它的输出完全是根据 Red\_Rom\_Data 的值, 而 Red\_Rom\_Data 的来源是 red\_rom\_module.v。

至于 n (当前的 x 地址, 当前显示的列), 是用于“**列寻址**”, 然后判断该点阵是“**点亮**”还是“**点灭**”。

*vga\_module.v*

```

1. module vga_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.    ****
17.
18.    wire CLK_40Mhz;
19.
20.    pll_module U1
21.    (
22.        .inclk0( CLK ),      // input - from top
23.        .c0( CLK_40Mhz )   // output - inter globak
24.    );

```

```
25.  
26.    ****  
27.  
28.    wire [10:0]Column_Addr_Sig;  
29.    wire [10:0]Row_Addr_Sig;  
30.    wire Ready_Sig;  
31.  
32.    sync_module U2  
(  
33.        .CLK( CLK_40Mhz ),  
34.        .RSTn( RSTn ),  
35.        .VSYNC_Sig( VSYNC_Sig ),           // input - from top  
36.        .HSYNC_Sig( HSYNC_Sig ),          // input - from top  
37.        .Column_Addr_Sig( Column_Addr_Sig ), // output - to U6  
38.        .Row_Addr_Sig( Row_Addr_Sig ),      // output - to U6  
39.        .Ready_Sig( Ready_Sig )           // output - to U6  
40.    );  
41.  
42.    ****  
43.  
44.  
45.    wire [63:0]Red_Rom_Data;  
46.  
47.    red_rom_module U3  
(  
48.        .clock( CLK_40Mhz ),  
49.        .address( Rom_Addr ),       // input - from U6  
50.        .q( Red_Rom_Data )        // output - to U6  
51.    );  
52.  
53.    ****  
54.  
55.  
56.    wire [63:0]Green_Rom_Data;  
57.  
58.    green_rom_module U4  
(  
59.        .clock( CLK_40Mhz ),  
60.        .address( Rom_Addr ),       // input - from U6  
61.        .q( Green_Rom_Data )        // output - to U6  
62.    );  
63.  
64.    ****  
65.  
66.  
67.    wire [63:0]Blue_Rom_Data;  
68.
```

```

69.     blue_rom_module U5
70.     (
71.         .clock( CLK_40Mhz ),
72.         .address( Rom_Addr ), // input - from U6
73.         .q( Blue_Rom_Data ) // output - to U6
74.     );
75.
76.     /*************************************************************************/
77.
78.     wire [5:0]Rom_Addr;
79.
80.     vga_control_module U6
81.     (
82.         .CLK( CLK_40Mhz ),
83.         .RSTn( RSTn ),
84.         .Ready_Sig( Ready_Sig ), // input - from U2
85.         .Column_Addr_Sig( Column_Addr_Sig ), // input - from U2
86.         .Row_Addr_Sig( Row_Addr_Sig ), // input - from U2
87.         .Red_Rom_Data( Red_Rom_Data ), // input - from U3
88.         .Green_Rom_Data( Green_Rom_Data ), // input - from U4
89.         .Blue_Rom_Data( Blue_Rom_Data ), // input - from U5
90.         .Rom_Addr( Rom_Addr ), // output - to U3, U4, U5
91.         .Red_Sig( Red_Sig ), // output - to top
92.         .Green_Sig( Green_Sig ), // output - to top
93.         .Blue_Sig( Blue_Sig ) // output - to top
94.     );
95.
96.     /*************************************************************************/
97.
98. endmodule

```

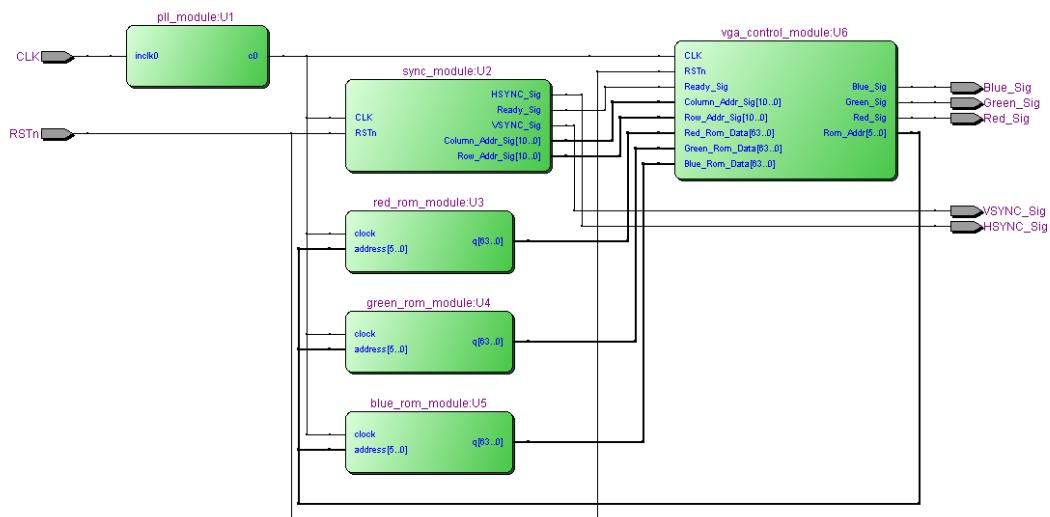
实验九之四的 vga\_module.v 比较不同。在 43~76 行中实例了 3 个 rom\_module.v 分别是 red\_rom\_module.v (47~52 行), green\_rom\_module.v (58~63 行), blue\_rom\_module.v (69~74 行)。然而各个 rom\_module.v 都有自己的输出连线 Rom\_Data (45 行, 56 行, 67 行), 这些连线的终点是 vga\_control\_module.v (87~89 行)。vga\_control\_module.v 输出的 Rom\_Addr (90 行), 3 个 rom\_module.v 同时共有 (72 行, 61 行, 50 行)。

## 实验九之四说明:

实验九之四主要是基于“**图层概念**”去实现颜色图像的显示。在试验中，创建了3个64x64的rom模块，每一个rom模块分别针对每一个层去储存信息。所谓的“**叠加操作**”也不过是微秒的使用Red\_Sig, Green\_Sig, Blue\_Sig信号而已。

说白了 vga\_control\_module.v 和在实验九之三的操作一样，同样是处理点阵信息，不过不同的是，不是处理1副图片的点阵信息，而是3副图片的点阵信息。

## 完成后的扩展图：



## 实验九之四结论:

关于“**图层**”的概念，在现实中如同不同颜色的半透明纸，叠加起来，呈现出不同颜色的效果

## 实验九之五：帧的概念

关系到“**帧**”，大家都会联想起什么？没错就是动画。但是笔者在认识众多的 VGA 实验当中，都只是单纯的注重 VGA 驱动的方法，而忽略了帧。“**帧**”对于 VGA 来说是一个重要的概念之一。屏幕显示“**逼不逼真？**”，“**有没有拖尾现象？**”，这些都关系到“**帧**”。

在一般的 VGA 显示标准中 如：800 x 600 x 60Hz 最后一字的“**60Hz**”，表示该显示标准，在 1 秒内可以显示 60 帧图像（一帧等于一副图像）。

在这里我们先玩一些计算游戏：

显示模式	时钟 ( MHz )	行时序 ( 像素数 )					帧时序 ( 行数 )				
		a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525
640x480@75	31.5	64	120	640	16	840	3	16	480	1	500
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628
800x600@75	49.5	80	160	800	16	1056	3	21	600	1	625

以 800 x 600 x 60Hz 为例。我们知道 1 个行像素等于 1056 个列像素，而一个列像素需要 25ns。换一句话说，一“**帧**”的图像 需要  $628 * 1056 * 25\text{ns}$  的时间。那么一秒内到底能显示多少“**帧**”图片？

$$\begin{aligned} \text{1 帧图像所需要的时间} &= 628 * 1056 * 25\text{ns} \\ &= 0.0165792\text{s} \end{aligned}$$

$$\begin{aligned} \text{1 秒内可以显示 n 帧图片} &= 1 / 0.0165792 \\ &= 60.3 \\ &= 60 \end{aligned}$$

再尝试！以 640 x 480 x 75Hz 为例。1 个列像素需要 31.74ns。

$$\begin{aligned} \text{1 帧图像所需要的时间} &= 500 * 840 * ( 1 / 31.5\text{MHz} ) \\ &= 0.0133333333333333 \end{aligned}$$

$$\begin{aligned} \text{1 秒内可以显示 n 帧图片} &= 1 / 0.0133333333333333 \\ &= 75 \end{aligned}$$

再再尝试！以 800 x 600 x 75Hz 为例。1 个列像素需要 20.2ns。

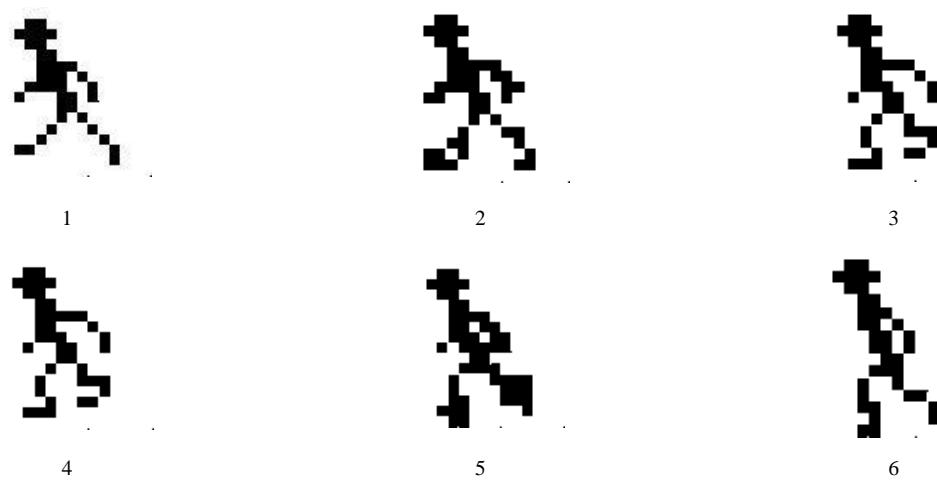
$$\begin{aligned} \text{1 帧图像所需要的时间} &= 625 * 1056 * ( 1 / 49.5\text{MHz} ) \\ &= 0.0133333333333333 \end{aligned}$$

$$\begin{aligned} \text{1 秒内可以显示 n 帧图片} &= 1 / 0.0133333333333333 \\ &= 75 \end{aligned}$$

笔者计算帧不是要研究什么“xx 标准有多少 xx 帧”，我们的目的主要是控制“一副图像可以逗留帧数”，然而实现“动画效果”。

有一点请注意，这里所说的“动画”，不是读者在电视机上看的动画，那些动画有业界的标准（好像是 1 秒 24 帧图像）。

实验九之五要显示的“动画”很简单，就是总所皆知的交通灯“小绿人”。（偷偷告诉你噢，小绿人实验在笔者学习单片机期间，研究过很长一段时间）。完整的“小绿人”太多幅图像了（18 副），这里笔者摘掉只剩前六副图像而已。



每一副小绿人的图像都是  $16 \times 16 \times 1\text{Bit}$ ，点阵信息如下：

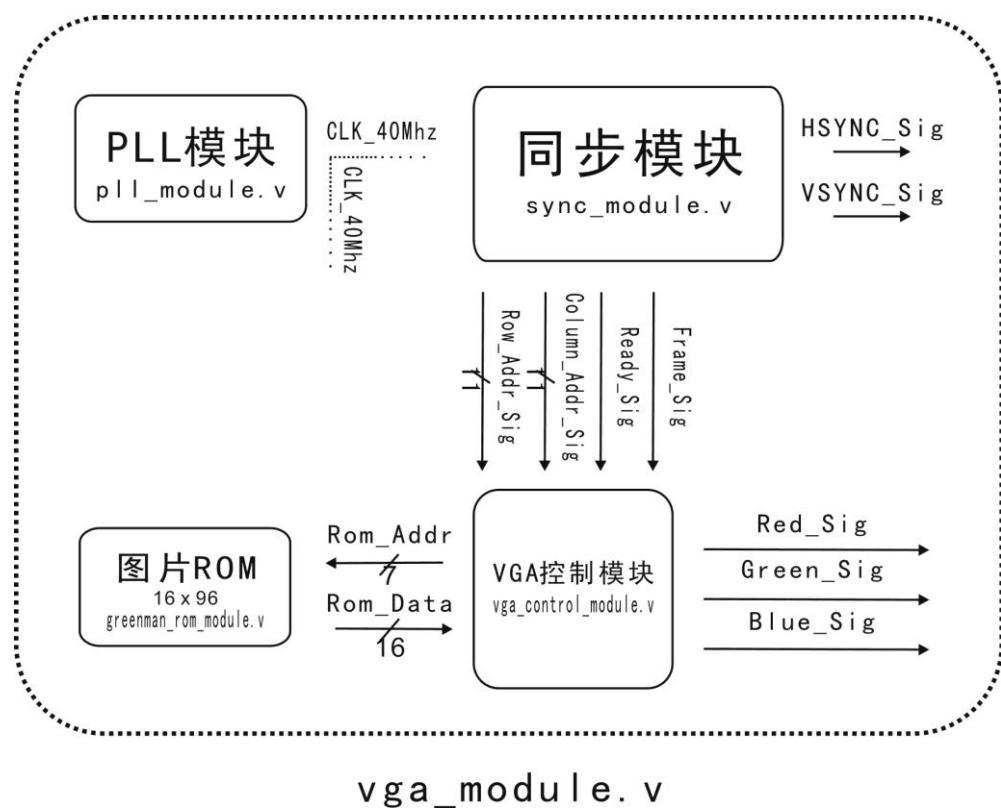
行	第一副	第二幅	第三幅	第四幅	第五副	第六幅
0	0000	0000	0000	0000	0000	0000
1	1800	1800	1800	1800	1800	1800
2	3C00	3C00	3C00	3C00	3C00	3C00
3	1800	1800	1800	1800	1800	1800
4	0C00	0C00	0C00	0C00	0C00	0C00
5	0F00	0F80	0F80	0F80	0F00	0E00
6	0E80	0EC0	0E40	0C40	0D80	0D00
7	1E40	1E60	1E20	0E20	0EC0	0E80
8	2340	3340	0300	1320	17C0	0680
9	0300	0300	0300	0300	0300	0300
10	0280	0280	0280	0280	0280	0700
11	0440	0460	0460	04E0	04F0	0900
12	0820	0C20	0410	0420	0470	08C0
13	3010	3410	0410	02C0	0670	0C20
14	0010	3830	3820	0E00	0E10	0420
15	0000	0000	0000	0000	0600	0C00

以上的点阵编码都是“高位在前” + “逐行扫描”，基本上和实验九之三一样。

现在的问题是：“如何将这些点阵信息存入 rom？”

很简单，我们只要建立  $16 \text{ bit} \times 96 \text{ words}$  的 rom 就可以了，当然这样作是有原因的。小绿人的一副图片是  $16 \times 16$  的大小，一行有 16 个列像素，而且有 16 行。也就是说，如果建立 16 Bits 的位宽，一副图片需要 16 words，而且这里有 6 副图片， $6 \times 16 = 96$  需要 96 words。

每一副图像相差有 16 个 words，而这个“相差数目”我们称为“帧偏移量”。所以呀，每当想更换一副图像，都需要加上“帧偏移量”。



实验九之五的组合模块如上。在同步模块 sync\_module.v 中多了一个输出 Frame\_Sig 信号。该信号主要是，每当完成一帧图像的显示，就产生一个高脉冲来通知 vga\_control\_module.v。vga\_control\_module.v 会依 Frame\_Sig 的高脉冲来计数“帧”。

要知道“一帧图像是否已经显示完？”，在 sync\_module.v 之中就要好好的利用“行像素”计数器 Count\_V。以  $800 \times 600 \times 60\text{hz}$  为例，当行计数器 Count\_V 计数到 628 的时候，亦即一帧图像已经显示完毕。

*sync\_module.v*

```
1. module sync_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig, Ready_Sig, Frame_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 Frame_Sig;
14.    output [10:0]Column_Addr_Sig;
15.    output [10:0]Row_Addr_Sig;
16.
17.    /*****
18.
19.    reg [10:0]Count_H;
20.
21.    always @ ( posedge CLK or negedge RSTn )
22.        if( !RSTn )
23.            Count_H <= 11'd0;
24.        else if( Count_H == 11'd1056 )
25.            Count_H <= 11'd0;
26.        else
27.            Count_H <= Count_H + 1'b1;
28.
29.    *****/
30.
31.    reg [10:0]Count_V;
32.
33.    always @ ( posedge CLK or negedge RSTn )
34.        if( !RSTn )
35.            Count_V <= 11'd0;
36.        else if( Count_V == 11'd628 )
37.            Count_V <= 11'd0;
38.        else if( Count_H == 11'd1056 )
39.            Count_V <= Count_V + 1'b1;
40.
```

```

41.      *****/
42.
43.      reg isReady;
44.
45.      always @ ( posedge CLK or negedge RSTn )
46.          if( !RSTn )
47.              isReady <= 1'b0;
48.          else if( ( Count_H >= 11'd216 && Count_H < 11'd1017 ) &&
49.                  ( Count_V >= 11'd27 && Count_V < 11'd627 ) )
50.              isReady <= 1'b1;
51.          else
52.              isReady <= 1'b0;
53.
54.      *****/
55.
56.      assign VSYNC_Sig = ( Count_V <= 11'd4 ) ? 1'b0 : 1'b1;
57.      assign HSYNC_Sig = ( Count_H <= 11'd128 ) ? 1'b0 : 1'b1;
58.      assign Ready_Sig = isReady;
59.      assign Frame_Sig = ( Count_V == 11'd628 ) ? 1'b1 : 1'b0;
60.
61.      *****/
62.
63.      assign Column_Addr_Sig = isReady ? Count_H - 11'd216 : 11'd0; // Count from 0
64.      assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0; // Count from 0
65.
66.      *****/
67.
68. endmodule

```

第 4, 13 行定义了 Frame\_Sig , 然而 Frame\_Sig 会产生高脉冲当 1 帧图片显示结束 (59 行)。我们知道一个事实, 以 800 x 600 x 60Hz, 当行像素计数器 Count\_V 计数到 628, 的时候表示, 最后一行已经扫描完毕。

*vga\_control\_module.v*

```

1. module vga_control_module
2. (
3.     CLK, RSTn,
4.     Ready_Sig, Frame_Sig, Column_Addr_Sig, Row_Addr_Sig,
5.     Rom_Data, Rom_Addr,
6.     Red_Sig, Green_Sig, Blue_Sig
7. );
8.     input CLK;

```

```
9.      input RSTn;
10.     input Ready_Sig;
11.     input Frame_Sig;
12.     input [10:0]Column_Addr_Sig;
13.     input [10:0]Row_Addr_Sig;
14.
15.     input [15:0]Rom_Data;
16.     output [6:0]Rom_Addr;
17.
18.     output Red_Sig;
19.     output Green_Sig;
20.     output Blue_Sig;
21.
22.     /*****
23.
24.     reg [4:0]m;
25.
26.     always @ ( posedge CLK or negedge RSTn )
27.         if( !RSTn )
28.             m <= 5'd0;
29.         else if( Ready_Sig && Row_Addr_Sig < 16 )
30.             m = Row_Addr_Sig[4:0] ;
31.         else
32.             m = 5'd0;
33.
34.     reg [4:0]n;
35.
36.     always @ ( posedge CLK or negedge RSTn )
37.         if( !RSTn )
38.             n <= 5'd0;
39.         else if( Ready_Sig && Column_Addr_Sig > 2 && Column_Addr_Sig < 19 )
40.             n = Column_Addr_Sig[4:0] - 5'd3;
41.         else
42.             n = 5'd0;
43.
44.     /*****
45.
46.     parameter FRAME = 10'd60;
47.
48.     /*****
49.
50.     reg [9:0]Count_Frame;
51.
52.     always @ ( posedge CLK or negedge RSTn )
```

```
53.         if( !RSTn )
54.             Count_Frame <= 10'd0;
55.         else if( Count_Frame == FRAME )
56.             Count_Frame <= 10'd0;
57.         else if( Frame_Sig )
58.             Count_Frame <= Count_Frame + 1'b1;
59.
60.     *****/
61.
62.     reg [6:0]rAddr;
63.     reg [3:0]i;
64.
65.     always @ ( posedge CLK or negedge RSTn )
66.         if( !RSTn )
67.             begin
68.                 rAddr <= 7'd0;
69.                 i <= 4'd0;
70.             end
71.         else
72.             case ( i )
73.
74.                 4'd0 :
75.                     if( Count_Frame == FRAME ) i <= 4'd1;
76.                     else rAddr <= 7'd0;
77.
78.                 4'd1 :
79.                     if( Count_Frame == FRAME ) i <= 4'd2;
80.                     else rAddr <= 7'd16;
81.
82.                 4'd2 :
83.                     if( Count_Frame == FRAME ) i <= 4'd3;
84.                     else rAddr <= 7'd32;
85.
86.                 4'd3 :
87.                     if( Count_Frame == FRAME ) i <= 4'd4;
88.                     else rAddr <= 7'd48;
89.
90.                 4'd4 :
91.                     if( Count_Frame == FRAME ) i <= 4'd5;
92.                     else rAddr <= 7'd48;
93.
94.                 4'd5 :
95.                     if( Count_Frame == FRAME ) i <= 4'd6;
96.                     else rAddr <= 7'd64;
```

```
97.
98.          4'd6 :
99.          if( Count_Frame == FRAME ) i <= 4'd0;
100.         else rAddr <= 7'd80;
101.
102.         endcase
103.
104.     /*****
105.
106.     assign Rom_Addr = rAddr + m;
107.
108.     assign Red_Sig = Ready_Sig ? Rom_Data[ 5'd15 - n ] : 1'b0;
109.     assign Green_Sig = Ready_Sig ? Rom_Data[ 5'd15 - n ] : 1'b0;
110.     assign Blue_Sig = Ready_Sig ? Rom_Data[ 5'd15 - n ] : 1'b0;
111.
112.    *****/
113.
114.
115.endmodule
```

第 4, 11 行定义了 Frame\_Sig 的入口。此外，为了能更好的显示“[动画效果](#)”，图像显，y 地址从 0 (29 行)，x 地址从 2 (第 39 行) 开始。在 46 行，定义了 FRAME 的常量。然而在 50~58 行，是 FRAME 的计数器。每当 Frame\_Sig 接收到一个高脉冲 (57 行)，该计数器就递增，直到达到与常量 FRAME 相同的值 (55 行) 就会赋值为 0。

第 60~104 行是“[切换图像](#)”的操作，每当一副图像显示 10 帧 (根据 FRAME 常量)，就会更换下一副图像。而更换图像的操作就是为 rAddr 添加“[帧偏移量](#)” 16。

根据 green\_rom\_module.v 的配置，我们知道：

第一幅图像的起始地址是 00，而行寻址是 00~15。  
第二幅图像的起始地址是 16，而行寻址是 16~31。  
第三幅图像的起始地址是 32，而行寻址是 32~47。  
第四幅图像的起始地址是 48，而行寻址是 48~63。  
第五幅图像的起始地址是 64，而行寻址是 64~79。  
第六幅图像的起始地址是 80，而行寻址是 80~95。

在第 106 行，Rom\_Addr 输出的值是 当前“[第 n 图像起始地址](#)” + “[m 行寻址](#)”。

假设一个情况：在模块开始运作的时候，Addr 的值是 0 也就是说第一幅图片被显示。第一幅图片会以点阵的方式显示。当第一章图片重复显示 10 次的时候 (10 帧)，Addr 的值会赋予 16，也就是第二副图像被显示，而且同样以点阵的方式显示。直到显示 10 帧以后，下一张图像会被选中。以上的顺序步骤会一直执行，直到显示完第 6 副图像，然后 Addr 会再一次赋予 0 值。也就是说，“[动画](#)”会从新开始显示，“[动画](#)”会重复到永远 ...

vga\_module.v

```
1. module vga_module
2. (
3.     CLK, RSTn,
4.     VSYNC_Sig, HSYNC_Sig,
5.     Red_Sig, Green_Sig, Blue_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    output VSYNC_Sig;
11.    output HSYNC_Sig;
12.    output Red_Sig;
13.    output Green_Sig;
14.    output Blue_Sig;
15.
16.    /*************************************************************************/
17.
18.    wire CLK_40Mhz;
19.
20.    pll_module U1
21.    (
22.        .inclk0( CLK ),      // input - from top
23.        .c0( CLK_40Mhz ) // output - to inter global
24.    );
25.
26.    /*************************************************************************/
27.
28.    wire Frame_Sig;
29.    wire [10:0]Column_Addr_Sig;
30.    wire [10:0]Row_Addr_Sig;
31.    wire Ready_Sig;
32.
33.    sync_module U2
34.    (
35.        .CLK( CLK_40Mhz ),
36.        .RSTn( RSTn ),
37.        .VSYNC_Sig( VSYNC_Sig ),    // input - from top
38.        .HSYNC_Sig( HSYNC_Sig ),    // input - from top
```

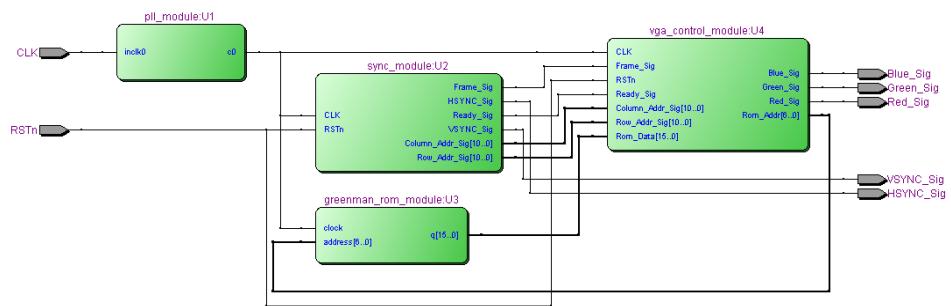
```
39.      .Frame_Sig( Frame_Sig ),           // output - to U4
40.      .Column_Addr_Sig( Column_Addr_Sig ), // output - to U4
41.      .Row_Addr_Sig( Row_Addr_Sig ),       // output - to U4
42.      .Ready_Sig( Ready_Sig )             // output - to U4
43. );
44.
45. ****
46.
47. wire [15:0]Rom_Data;
48.
49. greenman_rom_module U3
50. (
51.     .clock( CLK_40Mhz ),
52.     .address( Rom_Addr ),    // input - from U4
53.     .q( Rom_Data )         // output - to U4
54. );
55.
56. ****
57.
58. wire [6:0]Rom_Addr;
59.
60. vga_control_module U4
61. (
62.     .CLK( CLK_40Mhz ),
63.     .RSTn( RSTn ),
64.     .Ready_Sig( Ready_Sig ),           // input - from U2
65.     .Frame_Sig( Frame_Sig ),         // input - from U2
66.     .Column_Addr_Sig( Column_Addr_Sig ), // input - from U2
67.     .Row_Addr_Sig( Row_Addr_Sig ),       // input - from U2
68.     .Rom_Data( Rom_Data ),           // input - from U3
69.     .Rom_Addr( Rom_Addr ),           // output - to U3
70.     .Red_Sig( Red_Sig ),             // output - to top
71.     .Green_Sig( Green_Sig ),          // output - to top
72.     .Blue_Sig( Blue_Sig )            // output - to top
73. );
74.
75. ****
76.
77. endmodule
```

上面的源码就是 vga\_module.v 这个组合模块。其实也没有什么特别的，只要好好的浏览“[图形](#)”，既然而然很容易明白。

### 实验九之五说明:

实验九之五描述了“帧”对“动画效果”的影响。如果以 800 x 600 x 60Hz 现实标准为例，我们可以知道一帧图像所需要的时间是 0.0167s，在试验中一副图像重复了 60 帧，也就是说一副图像逗留的时间是 1s 左右。结果有 6 副图像，“一次动画”的时间需要 6s。

### 完成后扩展图：



### 实验九之五结论:

笔者也不知道要说什么了，当读者掌握 5 个实验的概念后，已经证明读者对 VGA 的驱动已经掌握在手了。

## 实验九说明：

VGA 的屏幕看简单点即是一副大点阵屏幕，但是这个点阵屏幕可以支持 RGB 彩色。但是为了很好与 VGA 接口沟通，我们必须了解 “[VGA 的时序](#)”。当中 HSYNC Signal 等于 行控制信号，VSYNC Signal 等于 列控制信号。然和 RGB Signal 是 颜色控制信号。

实验九的五个实验都是描述 VGA 的基本概念和知识。其中 “[功能模块](#)” sync\_module.v 控制着 “[显示标准](#)”，“[控制模块](#)” vga\_control\_module.v 是执行着 “[图像控制](#)” 的操作，“[功能模块](#)” rom\_module.v，是用来储存图像信息。

## 实验九结论：

实验九的设计完全基于 “[低级建模](#)” 的准则。尤其在实验九之四中，“[低级建模](#)” 完全把该实验发挥到很高的级别。除此之外，实验九之三和五，也表示了 “[低级建模](#)” 在 “[代码整洁度](#)” 和 “[理解上](#)” 都有很好的效果。

还有一点就是 “[控制模块](#)” 的作用。在 5 个实验当中，唯有控制模块的修改最大，如：“[点阵操作](#)” 和 “[动画效果](#)” 等，都是控制模块在操作。

典型的 HDL 教科书中，才不会要读者了解 “[模块的性质](#)”。没有性质的模块，常常会使得初学者在设计上和理解容易陷入 “[混乱](#)”。反之，如果 “[模块含有性质](#)” 的话，在设计和理解方面，思路会而外的清晰。

在以前还没有琢磨出这样的建模技巧之前，笔者真的无法完成上述的例程，那怕是简单的设计，即使笔者很侥幸的写出一两样东西，估计 “[结果](#)” 也不会好到那里去。

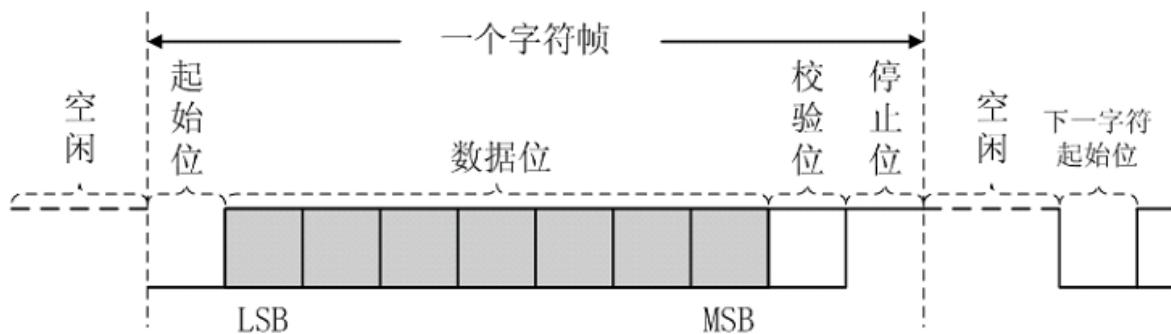
至于现在，笔者尝试使用 “[低级建模](#)” 去完成程式设计，感觉都是得心应手，而且设计越发的清晰。虽然 “[低级建模](#)” 的建模量确实很多，但是可以把它看成是一种修行练功。

笔者也不再说什么了，再说下去笔者也会认为自己是老王卖瓜了。更多有趣，更多好玩的事情，还是需要读者自己在设计的过程中去体会吧。

### 3.4 实验十：串口模块

单片机？串口？这些已经是众所周知的组合了吧。但是有一点，读者是否深入了解过串口传输上的小细节呢？我们先抛开硬件不谈（基本上没有什么好谈），在传统的（单片机）串口实验，对串口的调用常常都是对寄存器进行配置和查询。

实际上的串口，在传输的期间到底发生了什么事情，这些东西在传统的(单片机)串口实验上是不晓得的？反之使用 Verilog HDL 对串口建模，读者会从底层窥探到它。



上图串口传输的时序图。串口传输数据都是一帧数据 11 位。

位	位作用
0	起始位
1~7	数据位
9	校验位
10	停止位

在串口的总线上“高电平”是默认的状态，当一帧数据的开始传输必须先拉低电平，这就是第 0 位的作用。第 0 位过后就是 8 个数据位，这八个数据位才是一帧数据中最有意义的东西。最后的两位是校验位和停止位，作用如同命名般一样，基本上是没有重要意义。

串口传输还有另一个重要参数就是“波特率”。很多朋友都误解“波特率”是串口传输的传输速度，这样的理解在宏观上是无误。但是在微观上“波特率”就是串口传输中“一个位的周期”，换句话说亦是“一个位所逗留的时间”。

常用的波特率有 9600 bps 和 115200 bps ( bit per second )。“9600 bps” 表示每秒可以传输 9600 位。但是经过公式计算“一个位的周期”就会暴露出来。

$$\begin{aligned} \text{一个位的周期} &= 1 / \text{bps} \\ &= 1 / 9600 \end{aligned}$$

$$= 0.00010416666666667$$

从上述的公式, 我们明白一个事实 9600 bps ,一位数据占用 0.0001041666666667s 时间。如果是一帧 11 位的数据, 就需要

$$0.0001041666666667 \times 11 = 0.0011458333333334$$

那么一秒钟内可以传输

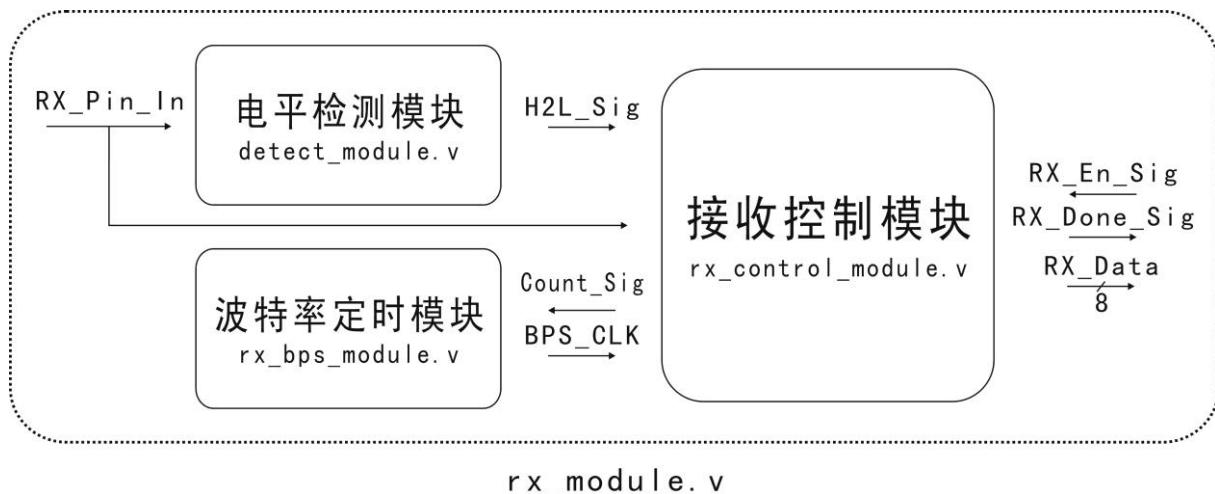
$$1 / 0.0011458333333334 = 872.727272727268$$

872.7272727268 个帧数据。

当然这只是在数字上计算出来而已, 但是实际上还有许多看不见的延迟因数。

## 实验十之一：串口接收模块

关系到串口接收, 基本上就是 “[采样](#)” 的操作。当你明白 “bps” 的意义后, 对于理解串口的接收, 非常的简单。



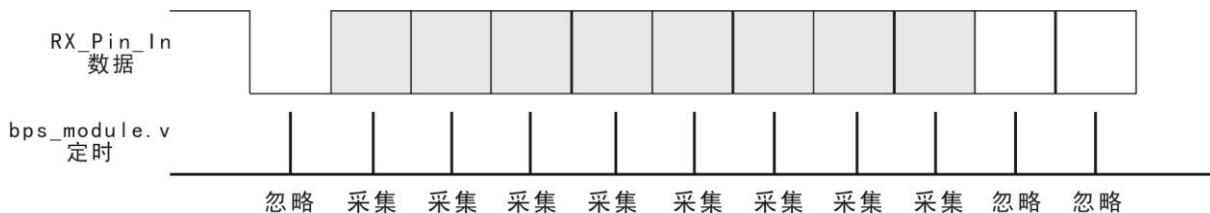
实验十之一是要建立如上图的 rx\_module.v 。Rx\_module.v 是一个组合模块, 主要是包含 detect\_module.v , bps\_module.v 和 rx\_control\_module.v , 3 个功能模块。

detect\_module.v 的输入是连结物理引脚 rx, 它主要检测一帧数据的第 0 位, 也就是起始位, 然后产生一个高脉冲经 H2L\_Sig 给 rx\_control\_module.v , 以表示一帧数据接收工作已经开始。

rx\_bps\_module.v 是产生波特率定时的功能模块。换一句话说, 它是配置波特率的模块。当 rx\_control\_module.v 拉高 Count\_Sig, bps\_module.v 经 BPS\_CLK 对 rx\_control\_module.v 产生定时。

`rx_control_module.v` 是核心控制模块。针对串口的配置主要是 1 帧 11 位的数据，重视八位数据位，无视起始位，校验位和结束位。当 `RX_En_Sig` 拉高，这个模块就开始工作，它将采集来自 `RX_Pin_In` 的数据，当完成一帧数据接收的时候，就会产生一个高脉冲给 `RX_Done_Sig`。

说道“采集”，它到底是如何实现采集的呢？



首先，读者猜猜看，在什么时候数据是最稳定？如上图所示，数据采集都是在“**每位数据的中间**”进行着。在上图中 `RX_Pin_In` 输入一帧数据，当 `detect_module.v` 检测到低电平（起始位），`rx_control_module.v` 和 `rx_bps_module.v` 就产生定时（与 `RX_Pin_In` 的波特率是一致）。然而 `rx_bps_module.v` 产生的定时是在每个位时间的中间。

在第 0 位数据，采取忽略的态度，然后接下来的 8 位数据位都被采集，最后校验位和停止位，却是采取了忽略的操作。有一点你必须好好注意，串口传输数据“**从最低位开始，到最高位结束**”。

为了有效控制 `rx_module.v` 所以 `RX_En_Sig` 和 `RX_Done_Sig` 都是被必须的。  
(`RX_En_Sig` 和 `RX_Done_Sig` 类似仿顺序操作的控制信号- 第四章)

#### *detect\_module.v*

```

1. module detect_module
2. (
3.   CLK, RSTn,
4.   RX_Pin_In,
5.   H2L_Sig
6. );
7.   input CLK;
8.   input RSTn;
9.   input RX_Pin_In;
10.  output H2L_Sig;
11.
12.  *****/
13.
14.  reg H2L_F1;
```

```
15.     reg H2L_F2;
16.
17.     always @ ( posedge CLK or negedge RSTn )
18.         if( !RSTn )
19.             begin
20.                 H2L_F1 <= 1'b1;
21.                 H2L_F2 <= 1'b1;
22.             end
23.         else
24.             begin
25.                 H2L_F1 <= RX_Pin_In;
26.                 H2L_F2 <= H2L_F1;
27.             end
28.
29.     /*************************************************************************/
30.
31.     assign H2L_Sig = H2L_F2 & !H2L_F1;
32.
33.     /*************************************************************************/
34.
35. endmodule
```

嗯！detect\_module.v 这个功能模块，读者也非常眼熟了吧，而该功能模块是为了检查电平由高变低。当检测到电平又高变低，在第 31 行就会输出高脉冲。

*rx\_bps\_module.v*

```
1. module rx_bps_module
2. (
3.     CLK, RSTn,
4.     Count_Sig,
5.     BPS_CLK
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input Count_Sig;
11.    output BPS_CLK;
12.
13.    /*************************************************************************/
14.
15.    reg [12:0]Count_BPS;
```

```

16.
17.     always @ ( posedge CLK or negedge RSTn )
18.         if( !RSTn )
19.             Count_BPS <= 13'd0;
20.         else if( Count_BPS == 12'd5207 )
21.             Count_BPS <= 13'd0;
22.         else if( Count_Sig )
23.             Count_BPS <= Count_BPS + 1'b1;
24.         else
25.             Count_BPS <= 13'd0;
26.
27.     /*****
28.
29.     assign BPS_CLK = ( Count_BPS == 13'd2604 ) ? 1'b1 : 1'b0;
30.
31.     *****/
32.
33. endmodule

```

9600 bps 传输速度使一位数据的周期是  $0.0001041666666667\text{s}$ 。以 50Mhz 时钟频率要得到上述的定时需要：

$$N = 0.0001041666666667 / (1 / 50\text{Mhz}) = 5208$$

如果从零开始算起  $5208 - 1$  亦即 5207 个计数（20 行）。然而，采集数据要求“在周期的中间”，那么结果是  $5208 / 2$ ，结果等于 1041（29 行）。基本上 rx\_bps\_module.v 只有在 Count\_Sig 拉高的时候（22 行），模块才会开始计数。

#### *rx\_control\_module.v*

```

1. module rx_control_module
2. (
3.     CLK, RSTn,
4.     H2L_Sig, RX_Pin_In, BPS_CLK, RX_En_Sig,
5.     Count_Sig, RX_Data, RX_Done_Sig
6.
7. );
8.
9.     input CLK;
10.    input RSTn;
11.
12.    input H2L_Sig;

```

## Verilog HDL 那些事儿 – 建模篇

```
13.      input RX_En_Sig;
14.      input RX_Pin_In;
15.      input BPS_CLK;
16.
17.      output Count_Sig;
18.      output [7:0]RX_Data;
19.      output RX_Done_Sig;
20.
21.
22.      /*************************************************************************/
23.
24.      reg [3:0]i;
25.      reg [7:0]rData;
26.      reg isCount;
27.      reg isDone;
28.
29.      always @ ( posedge CLK or negedge RSTn )
30.          if( !RSTn )
31.              begin
32.                  i <= 4'd0;
33.                  rData <= 8'd0;
34.                  isCount <= 1'b0;
35.                  isDone <= 1'b0;
36.              end
37.          else if( RX_En_Sig )
38.              case ( i )
39.
40.                  4'd0 :
41.                      if( H2L_Sig ) begin i <= i + 1'b1; isCount <= 1'b1; end
42.
43.                  4'd1 :
44.                      if( BPS_CLK ) begin i <= i + 1'b1; end
45.
46.                  4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8, 4'd9 :
47.                      if( BPS_CLK ) begin i <= i + 1'b1; rData[ i - 2 ] <= RX_Pin_In; end
48.
49.                  4'd10 :
50.                      if( BPS_CLK ) begin i <= i + 1'b1; end
51.
52.                  4'd11 :
53.                      if( BPS_CLK ) begin i <= i + 1'b1; end
54.
55.                  4'd12 :
56.                      begin i <= i + 1'b1; isDone <= 1'b1; isCount <= 1'b0; end
```

```

57.
58.          4'd13 :
59.          begin i <= 1'b0; isDone <= 1'b0; end
60.
61.          endcase
62.
63.      /*****
64.
65.      assign Count_Sig = isCount;
66.      assign RX_Data = rData;
67.      assign RX_Done_Sig = isDone;
68.
69.
70.      *****/
71.
72. endmodule

```

在 37~61 行, 是 rx\_control\_module.v 的核心控制功能。在 37 行, 表示了如果 RX\_En\_Sig 如果没有拉高, 这个模块是不会工作。(在 26 行定义了 isCount 标志寄存器, 为了使能 rx\_bps\_module.v 输出采集定时信号) 当 rx\_control\_module.v 模块被使能, 该模块就会处于就绪状态, 一旦 detect\_module.v 检查到由高变低的电平变化(41 行), 会使步骤 i 进入第 0 位采集, 然而 isCount 标志寄存器同时也会被设置为逻辑 1, rx\_bps\_module.v 便会开始产生波特率的定时。

我们知道 rx\_bps\_module.v 产生的定时是在“[每位数据的中间](#)”。在 43~44 行, 第一次的定时采集时第 0 位数据(起始位), 保持忽略态度。在 46~47 行, 定时采集的是八位数据位, 每一位数据位会依低位到最高位储存入 rData 寄存器。

49~53 行, 是最后两位的定时采集(校验位, 停留位), 同时采取忽略的态度。当进入 55~59 行, 这表示一帧数据的采集工作已经结束。最后会产生一个完成信号的高脉冲, 同时间 isCount 会被设置为逻辑 0, 亦即停止 rx\_bps\_module.v 的操作。

至于 65~67 行, isCount 标志寄存器是 Count\_Sig 输出的驱动器, rData 寄存器是 RX\_Data 输出的驱动器, 最后 isDone 标志寄存器是 RX\_Done\_Sig 输出的驱动寄存器。

### *rx\_module.v*

```

1. module rx_module
2. (
3.     CLK, RSTn,
4.     RX_Pin_In, RX_En_Sig,

```

```
5.      RX_Done_Sig, RX_Data
6.  );
7.
8.      input CLK;
9.      input RSTn;
10.
11.     input RX_Pin_In;
12.     input RX_En_Sig;
13.
14.     output [7:0]RX_Data;
15.     output RX_Done_Sig;
16.
17.
18. /*****
19.
20. wire H2L_Sig;
21.
22. detect_module U1
23. (
24.     .CLK( CLK ),
25.     .RSTn( RSTn ),
26.     .RX_Pin_In( RX_Pin_In ),      // input - from top
27.     .H2L_Sig( H2L_Sig )          // output - to U3
28. );
29.
30. *****/
31.
32. wire BPS_CLK;
33.
34. rx_bps_module U2
35. (
36.     .CLK( CLK ),
37.     .RSTn( RSTn ),
38.     .Count_Sig( Count_Sig ),    // input - from U3
39.     .BPS_CLK( BPS_CLK )        // output - to U3
40. );
41.
42. *****/
43.
44. wire Count_Sig;
45.
46. rx_control_module U3
47. (
48.     .CLK( CLK ),
```

```

49.      .RSTn( RSTn ),
50.
51.      .H2L_Sig( H2L_Sig ),           // input - from U1
52.      .RX_En_Sig( RX_En_Sig ),     // input - from top
53.      .RX_Pin_In( RX_Pin_In ),    // input - from top
54.      .BPS_CLK( BPS_CLK ),        // input - from U2
55.
56.      .Count_Sig( Count_Sig ),     // output - to U2
57.      .RX_Data( RX_Data ),        // output - to top
58.      .RX_Done_Sig( RX_Done_Sig ) // output - to top
59.
60. );
61.
62. ****
63.
64.
65. endmodule

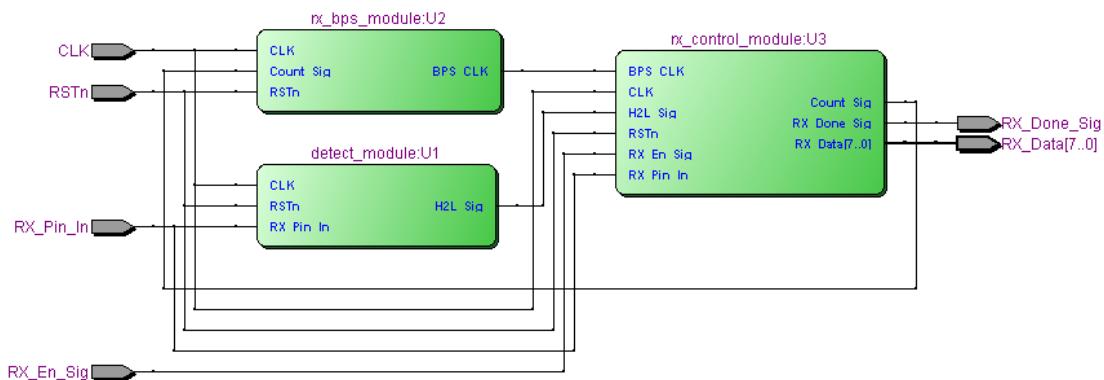
```

#### 实验十之一说明:

当 rx\_module.v 组合模块组织一切以后只保留的输入输出有：输入信号 RX\_Pin\_In 和 RX\_En\_Sig；输出信号 RX\_Data 和 RX\_Done\_Sig。在试验中，rx\_bps\_module.v 扮演着“**配置波特率**”，然而 rx\_control\_module.v 的工作则对“**数据作出过滤**”。

在典型的应用上“1位起始位，8位数据位，1位校验位，和1位停止位”已经成为主流。当然波特率的选择是 9600 bps 还是 115200 bps 这视需要而定。

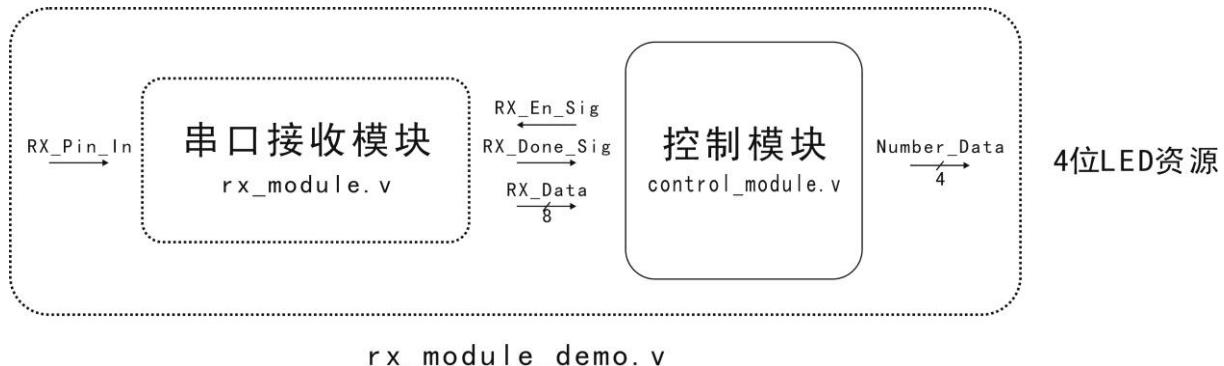
#### 完成后的扩展图：



实验十之一结论：

“定时采集”的部分，比较难搞明白，但是深思熟虑过后，读者会发现这样的理解是很  
有逻辑。

## 实验十之一演示：



这一章我们要建立如上图 rx\_module\_demo.v 组合模块，它里边包含了一个控制模块对 rx\_module.v 的调用。一开始 control\_module.v 会拉高 RX\_En\_Sig 使能 rx\_module.v。当有一阵数据经 RX\_Pin\_In 传入，rx\_module.v 就会接收，然后过滤后的数据输出致 RX\_Data，然后再产生一个高脉冲给 RX\_Done\_Sig。当 control\_module.v 接收到 RX\_Done\_Sig 的高脉冲，拉低 RX\_En\_Sig，并且将 RX\_Data 的“前四位”输出致 4 位 LED 资源。然后过程再一次重复不断。

*control\_module.v*

```

1. module control_module
2. (
3.     CLK, RSTn,
4.     RX_Done_Sig,
5.     RX_Data,
6.     RX_En_Sig,
7.     Number_Data
8. );
9.
10.    input CLK;
11.    input RSTn;
12.    input RX_Done_Sig;
13.    input [7:0]RX_Data;
14.    output RX_En_Sig;
15.    output [7:0]Number_Data;
16.
17.    /*****
18.
19.    reg [7:0]rData;

```

```
20.    reg isEn;
21.
22.    always @ ( posedge CLK or negedge RSTn )
23.        if( !RSTn )
24.            rData <= 8'd0;
25.        else if( RX_Done_Sig )
26.            begin rData <= RX_Data; isEn <= 1'b0; end
27.        else isEn <= 1'b1;
28.
29.    /*****
30.
31.    assign Number_Data = rData;
32.    assign RX_En_Sig = isEn;
33.
34.    *****/
35.
36. endmodule
```

在 22~27 行就是这个控制模块的核心功能了。一开始的时候（27 行）就将 isEn 设置为逻辑 1，这个标志寄存器在 32 行驱动着 RX\_En\_Sig。换句话说，此时的 rx\_module.v 已经进入就绪状态，然后 control\_module.v 等待着 RX\_Done\_Sig 反馈（25 行）。

一旦一帧数据接收完毕，RX\_Done\_Sig 就会产生高脉冲，然后 rData 就会寄存 RX\_Data 的值。同时间 isEn 被设置为逻辑 0（26 行）。在下一个来临，control\_module.v 会再一次设置 isEn 为逻辑 1，已做好接收下一组数据的准备。

### *rx\_module\_demo.v*

```
1. module rx_module_demo
2. (
3.     CLK, RSTn,
4.     RX_Pin_In,
5.     Number_Data
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input RX_Pin_In;
11.    output [3:0]Number_Data;
12.
13.    *****/
14.
```

```
15.     wire RX_Done_Sig;
16.     wire [7:0]RX_Data;
17.
18.     rx_module U1
19.     (
20.         .CLK( CLK ),
21.         .RSTn( RSTn ),
22.         .RX_Pin_In( RX_Pin_In ),           // input - from top
23.         .RX_En_Sig( RX_En_Sig ),          // input - from U2
24.         .RX_Done_Sig( RX_Done_Sig ),      // output - to U2
25.         .RX_Data( RX_Data )             // output - to U2
26.     );
27.
28.     /*****
29.
30.     wire RX_En_Sig;
31.     wire [7:0]Output_Data;
32.
33.     control_module U2
34.     (
35.         .CLK( CLK ),
36.         .RSTn( RSTn ),
37.         .RX_Done_Sig( RX_Done_Sig ),    // input - from U1
38.         .RX_Data( RX_Data ),           // input - from U1
39.         .RX_En_Sig( RX_En_Sig ),       // output - to U1
40.         .Number_Data( Output_Data )   // output - to top
41.     );
42.
43.     /*****
44.
45.     assign Number_Data = Output_Data[3:0];
46.
47. endmodule
```

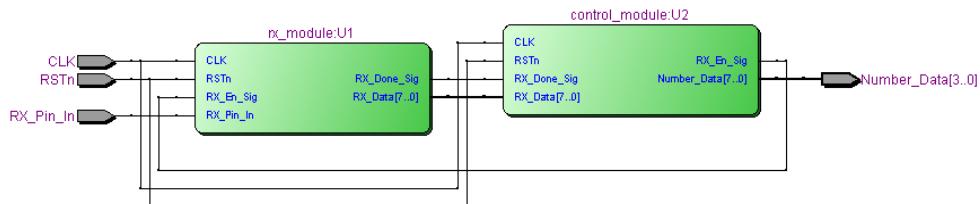
这是实验十之一演示的组合模块，基本上没有什么特别的。连线关系也和“[图形](#)”一样。在 45 行，对 Number\_Data 的输出驱动，取致 Output\_Data 前四位。

## 实验十之一演示说明:



这个演示主要是在“串口调试助手”以“十六进制”发送数据。如果发送 0F，那么四位 LED 就全部都会亮起来；如果输出 0A，那么只有第四位和第二位 LED 亮了起来。

## 完成后扩展图：

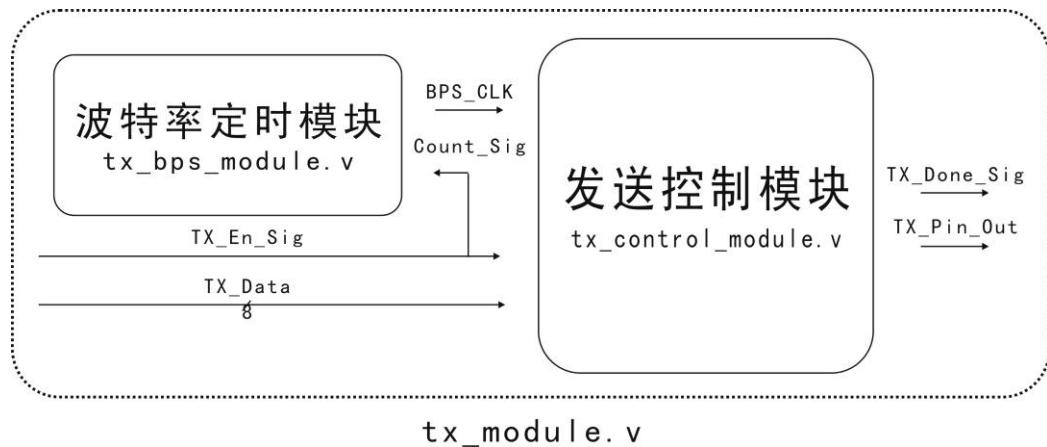


## 实验十之一演示结论:

这个演示，不过是演示 rx\_module.v 是如何调用而已。

## 实验十之二：串口发送模块

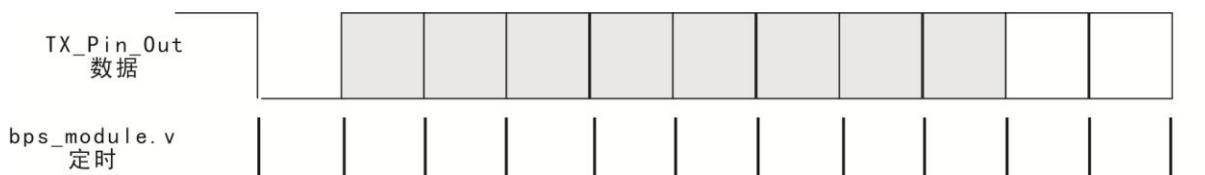
经实验十之一之后，读者基本上是不是已经掌握“波特率”了？在这一章实验“波特率”依然扮演着很重要的角色。



上图是 tx\_module.v 组合模块。与 rx\_module.v 比较 tx\_module.v 少了 detect\_module.v，因为串口发送一帧数据的时候用不着检测什么。

tx\_bps\_module.v 是作为“定时”的功能。当 TX\_En\_Sig 拉低的时候，它是处于睡眠的状态。一旦 TX\_En\_Sig 拉高电平，那么 tx\_bps\_module.v 就开始计数。然后定时产生一个高脉冲经 BPS\_CLK 给 tx\_control\_module.v。

tx\_control\_module.v 控制模块是最为中心的一部分，当 TX\_En\_Sig 拉高电平，同时间 tx\_bps\_module.v 也会开始计数。tx\_control\_module.v 将 TX\_Data 的值，按 tx\_bps\_module.v 产生的定时，有节奏的往 TX\_Pin\_Out 发送。当一帧数据发送完毕后，就反馈一个 TX\_Done\_Sig 的高脉冲。



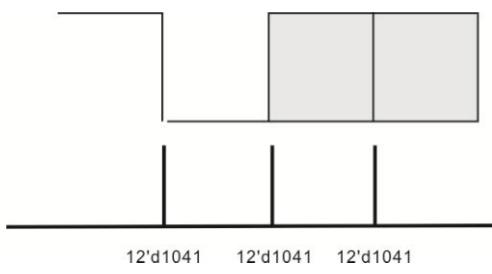
tx\_module.v 和 rx\_module.v 比较不同的是：rx\_module.v 是“定时采集”，然而 tx\_module.v 是“定时发送”。假设我配置的波特率为 9600 bps，那么每隔 0.00010416666666667 tx\_bps\_module.v 这个模块就会产生一个高脉冲给 tx\_control\_module.v，该控制模块会按这个节拍将数据一位一位的数据发送出去。

一帧数据有 11 位，那么 tx\_bps\_module.v 需要产生 12 次定时。

*tx\_bps\_module.v*

```
1. module tx_bps_module
2. (
3.     CLK, RSTn,
4.     Count_Sig,
5.     BPS_CLK
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input Count_Sig;
11.    output BPS_CLK;
12.
13.    ****
14.
15.    reg [12:0]Count_BPS;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            Count_BPS <= 13'd0;
20.        else if( Count_BPS == 13'd5207 )
21.            Count_BPS <= 13'd0;
22.        else if( Count_Sig )
23.            Count_BPS <= Count_BPS + 1'b1;
24.        else
25.            Count_BPS <= 13'd0;
26.
27.    ****
28.
29.    assign BPS_CLK = ( Count_BPS == 13'd2604 ) ? 1'b1 : 1'b0;
30.
31.    ****
32.
33. endmodule
```

*tx\_bps\_module.v* 配置为 9600 bps 波特率（20 行），内容基本上和 *rx\_bps\_module.v* 大同小异。不同的是，它现在的工作是“**定时发送**”。读者可能会对 29 行的代码产生疑问“怎么定时发送是发生在 Count\_BPS 计数的一半？”



左图是产生 3 个“**定时发送**”。每个“**定时发送**”是在计数 12'd2604 发生。读者尝试数数看，两个“**定时发送**”之间到底相差了多少个计数？没错，是 12'd5208 个计数。这下明白怎么一回事了吧！上一个定时的产生与下一个定时产生的之间才是重点，也就是说“**一位数据的周期**”定义在两个定时的之间。

### *tx\_control\_module.v*

```

1. module tx_control_module
2. (
3.     CLK, RSTn,
4.     TX_En_Sig, TX_Data, BPS_CLK,
5.     TX_Done_Sig, TX_Pin_Out
6.
7. );
8.
9.     input CLK;
10.    input RSTn;
11.
12.    input TX_En_Sig;
13.    input [7:0]TX_Data;
14.    input BPS_CLK;
15.
16.    output TX_Done_Sig;
17.    output TX_Pin_Out;
18.
19.    /*****
20.
21.    reg [3:0]i;
22.    reg rTX;
23.    reg isDone;
24.
25.    always @ ( posedge CLK or negedge RSTn )
26.        if( !RSTn )
27.            begin
28.                i <= 4'd0;
29.                rTX <= 1'b1;

```

```
30.           isDone     <= 1'b0;
31.       end
32.   else if( TX_En_Sig )
33.       case ( i )
34.
35.           4'd0 :
36.               if( BPS_CLK ) begin i <= i + 1'b1; rTX <= 1'b0; end
37.
38.           4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8 :
39.               if( BPS_CLK ) begin i <= i + 1'b1; rTX <= TX_Data[ i - 1 ]; end
40.
41.           4'd9 :
42.               if( BPS_CLK ) begin i <= i + 1'b1; rTX <= 1'b1; end
43.
44.           4'd10 :
45.               if( BPS_CLK ) begin i <= i + 1'b1; rTX <= 1'b1; end
46.
47.           4'd11 :
48.               if( BPS_CLK ) begin i <= i + 1'b1; isDone <= 1'b1; end
49.
50.           4'd12 :
51.               begin i <= 1'b0; isDone <= 1'b0; end
52.
53.       endcase
54.
55.   /*************************************************************************/
56.
57.   assign TX_Pin_Out = rTX;
58.   assign TX_Done_Sig = isDone;
59.
60.   /*************************************************************************/
61.
62. endmodule
```

比起 rx\_control\_module.v , tx\_control\_module.v 相对比较简单。当 TX\_En\_Sig 被拉高（32 行）该控制模块就会开始工作了（同时间 tx\_bps\_module.v 也会开始计数）。每当 tx\_bps\_module.v 产生一个定时的高脉冲致 BPS\_CLK, tx\_control\_module.v 都会发送一位数据。

第 0 位数据是起始位，所以 rTX 寄存器被赋值与逻辑 0（36 行）。接下来的八位都是数据位，tx\_control\_module.v 按 TX\_Data 的值从最低位到最高位，将数据赋值给 rTX 寄存器（39 行）。最后两位数据则是校验位和停止位，如果没有什么特别需求，就随便填上逻辑 1 就行了（41~45 行）。最后产生一个 TX\_Done\_Sig 的高脉冲（47~51 行）。

在 58 行 TX\_Done\_Sig 的输出是被 isDone 这个标志寄存器驱动着，然而 TX\_Pin\_Out 的输出是由 rTX 这个寄存器驱动。

*tx\_module.v*

```
1. module tx_module
2. (
3.     CLK, RSTn,
4.     TX_Data, TX_En_Sig,
5.     TX_Done_Sig, TX_Pin_Out
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input [7:0]TX_Data;
11.    input TX_En_Sig;
12.    output TX_Done_Sig;
13.    output TX_Pin_Out;
14.
15.
16. /*****
17.
18. wire BPS_CLK;
19.
20. tx_bps_module U1
21. (
22.     .CLK( CLK ),
23.     .RSTn( RSTn ),
24.     .Count_Sig( TX_En_Sig ),      // input - from U2
25.     .BPS_CLK( BPS_CLK )        // output - to U2
26. );
27.
28. *****/
29.
30. tx_control_module U2
31. (
32.     .CLK( CLK ),
33.     .RSTn( RSTn ),
34.     .TX_En_Sig( TX_En_Sig ),      // input - from top
35.     .TX_Data( TX_Data ),        // input - from top
36.     .BPS_CLK( BPS_CLK ),        // input - from U2
37.     .TX_Done_Sig( TX_Done_Sig ), // output - to top
38.     .TX_Pin_Out( TX_Pin_Out )   // output - to top
```

```
39.      );
40.
41.      *****/
42.
43. endmodule
```

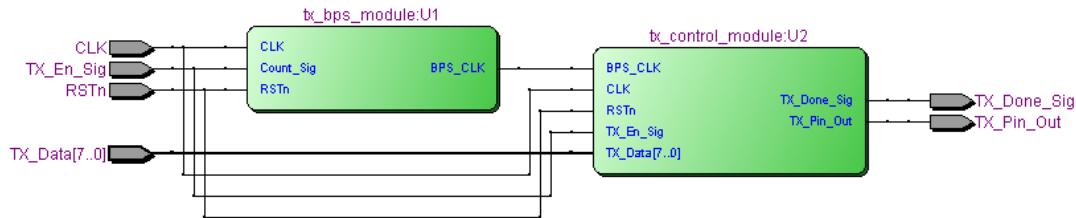
在组合模块 tx\_module.v 中的连线，基本上和“图形”是一模一样。但是有一点比较特别的是在 24 行，Count\_Sig 输入（驱动）是由 TX\_En\_Sig 直接连线。

实验十之二说明：

发送模块 tx\_module.v 基本上比 rx\_module.v 简单来得多，只要好好的掌握“**定时采集**”和“**定时发送**”的区别就可以驾驭它。

tx\_bps\_module.v 依然是扮演着配置“**波特率**”的角色。tx\_control\_module.v 的功能是对一字节数据，包装后以串口一帧的格式发送出去。至于要不要加入“**校验位**”，取“**多少数据位**”，或者取“**多少停留位**”，视实验要求而定，没有强迫性的。典型的应用下基本上有 1 个起始位，8 个数据位，1 个校验位和 1 个停止位。如果没有什么特别的要求，校验位可以随便填。

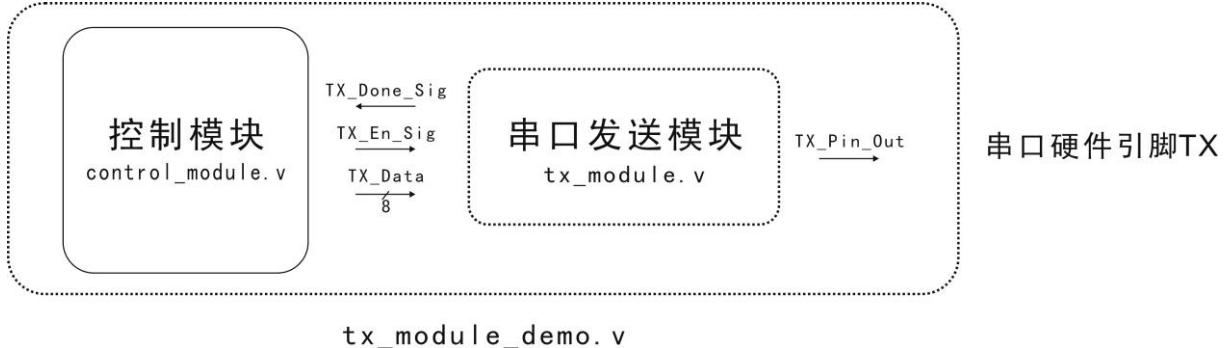
完成后的扩展图：



实验十之二结论：

tx\_module.v 为了更好的调用它，在建模的时候建立 TX\_En\_Sig 和 TX\_Done\_Sig 控制信号（TX\_En\_Sig 和 TX\_Done\_Sig 信号类似于仿顺序操作的 Start\_Sig 和 Done\_Sig 信号-第四章）对往后的调用非常有用。

## 实验十之二演示：



上图是实验十之二演示的组合模块。在 tx\_module\_demo.v 中的 control\_module.v 主要是每秒往 tx\_module.v 发送 0x31 的数据。

control\_module.v 每秒会往 TX\_Data 发送数据 0x31，然后拉高 TX\_Done\_Sig 使 tx\_module.v 开始工作。当 tx\_module.v 发往一帧数据以后，就会对 TX\_Done\_Sig 产生一个高脉冲，以示发送完毕。当 control\_module.v 接收到 tx\_module.v 反馈的 TX\_Done\_Sig，它就会拉低 tx\_module.v（上述的动作基本上在 1 秒内完成）。然后等待下一秒的到来，再重复一样的动作 .....

*control\_module.v*

```

1. module control_module
2. (
3.     CLK, RSTn,
4.     TX_Done_Sig,
5.     TX_En_Sig, TX_Data
6.
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input TX_Done_Sig;
12.    output TX_En_Sig;
13.    output [7:0]TX_Data;
14.
15.    ****
16.

```

```
17. parameter T1S = 26'd49_999_999;
18.
19. *****/
20.
21. reg [25:0]Count_Sec;
22.
23. always @ ( posedge CLK or negedge RSTn )
24.   if( !RSTn )
25.     Count_Sec <= 26'd0;
26.   else if( Count_Sec == T1S )
27.     Count_Sec <= 26'd0;
28.   else
29.     Count_Sec <= Count_Sec + 1'b1;
30.
31. *****/
32.
33. reg isEn;
34. reg [7:0]rData;
35.
36. always @ ( posedge CLK or negedge RSTn )
37.   if( !RSTn )
38.     begin
39.       isEn <= 1'b0;
40.       rData <= 8'h31;
41.     end
42.   else if( TX_Done_Sig )
43.     begin
44.       rData <= 8'h31;
45.       isEn <= 1'b0;
46.     end
47.   else if( Count_Sec == T1S )
48.     isEn <= 1'b1;
49.
50. *****/
51.
52. assign TX_Data = rData;
53. assign TX_En_Sig = isEn;
54.
55. *****/
56.
57. endmodule
```

第 17 行是定义 1 秒的常量，在 23~29 是 1 秒的定时器。control\_module.v 主要是每秒发送 0x31 的数据，换句话说就是每秒设置一次 isEn 标志寄存器。当 isEn 被设置后，

tx\_module.v 就会开始工作，发完一帧数据位 TX\_Done\_Sig 会产生高脉冲。这使得（42行），control\_module.v 会重新赋值 rData 寄存器 然后复位 isEn 标志寄存器。直到下一秒的到来，isEn 标志寄存器会再一次被设置。

*tx\_module\_demo.v*

```
1. module tx_module_demo
2. (
3.     CLK, RSTn,
4.     TX_Pin_Out
5. );
6.
7.     input CLK;
8.     input RSTn;
9.     output TX_Pin_Out;
10.
11.    /*****
12.
13.    wire [7:0]TX_Data;
14.    wire TX_En_Sig;
15.
16.    control_module U1
17.    (
18.        .CLK( CLK ),
19.        .RSTn( RSTn ),
20.        .TX_Done_Sig( TX_Done_Sig ), // input - from U2
21.        .TX_En_Sig( TX_En_Sig ), // output - to U2
22.        .TX_Data( TX_Data ) // output - to U2
23.    );
24.
25.    /*****
26.
27.    wire TX_Done_Sig;
28.
29.    tx_module U2
30.    (
31.        .CLK( CLK ),
32.        .RSTn( RSTn ),
33.        .TX_Data( TX_Data ), // input - from U1
34.        .TX_En_Sig( TX_En_Sig ), // input - from U1
35.        .TX_Done_Sig( TX_Done_Sig ), // output - to U1
36.        .TX_Pin_Out( TX_Pin_Out ) // output - to top
```

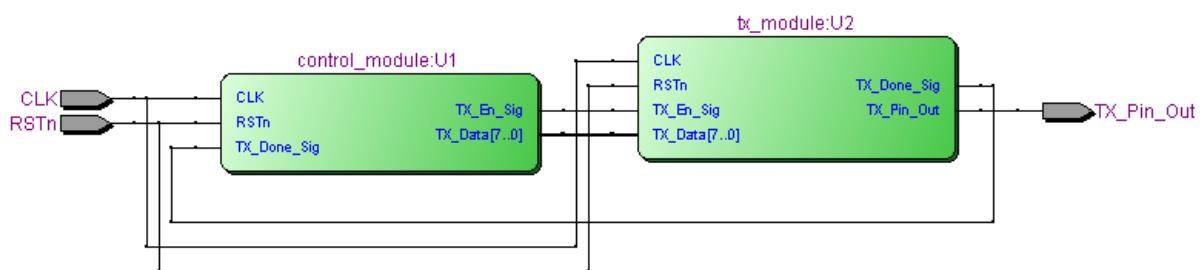
```
37.      );
38.
39.      *****/
40.
41.
42. endmodule
```

该组合模块比较简单，没有什么特别的。

### 实验十之二演示说明:

在实验十之二演示中，“串口调试助手”会一直显示 1... 如果以“十六进制”显示的话，结果会是 "31"。每一次发送的时间间隔是 1 秒。

完成的扩展图：



### 实验十之二演示结论:

这个演示是演示该 **tx\_module.v** 如何调用。

## 实验十说明：

在前面笔者也说过了，串口是全双工执行的。在建模的时候，理应考虑“**发送**”和“**接收**”是分开建立，亦即“**发送模块**”和“**接收模块**”各一个，互不干涉。

无论是“**发送模块**”还是“**接收模块**”，xx\_bps\_module.v 执行着波特率的定时（定时采集与定时发送）。然而 xx\_control\_module.v 执行一帧数据的发送或接收的处理。

此外为了要很好的调用该模块，我们应该好好考虑控制信号 xx\_En\_Sig 和 xx\_Done\_Sig 的实现。还记的传统的（单片机）串口实验吗？我们都是使用“**中断**”或者“**查询寄存器**”的方式控制串口的工作状态。反之在实验十中，我们使用 xx\_En\_Sig 和 xx\_Done\_Sig 来控制发送模块和接收模块（串口）。

## 实验十结论：

在实验十当中笔者将一个串口，分为两个模块，亦即“**发送模块**”和“**接收模块**”。然而发送模块|接收模块是由特定的功能模块和控制模块组织起来。

事实山，实验十已经涉及许多有关“**仿顺序操作**”的基本知识了（以往的实验同样也涉及一点）。“**仿顺序操作**”的“**模块构造**”会有标志性的 Start\_Sig 和 Done\_Sig，而且仿顺序操作的写法都会伴随着步骤 i 出现。

## 总结：

当读者看到这里，就表示读者已经掌握“[低级建模](#)”的基本技巧了。其中的重点是“[图形](#)”，“[低级建模的准则](#)”和“[模块的性质](#)”在建模上的作用。这些东西笔者在第二章，详细的描述过，因为“[它们](#)”是“[低级建模](#)”最重要的支柱。

第三章的重点是为了强化读者在第二章所学到的东西。在这一章中的所有试验，我们可以看到“控制模块”，“功能模块”，“组合模块”之间合作，将一个完整的模块有结构的表达出来。不仅于此，活用这三个模块可以使得 Verilog HDL 语言的建模更简单，更多可能性。

在这里我们稍微深入理解低级建模的准则“[一个模块一个功能](#)”。如实验七，数码管的取位模块和加码模块都是个别在一个模块中，其中笔者用图形将它们的形状（性质）表达出来，然后用连线来表示它们之间的关系。

假设，这两个模块混在一起的话，我们会遇见如下的问题：

第一，绘制图形不但变得麻烦起来，而且图形的表达能力也会降下。  
第二，在组合模块上，连线会变得更复杂。

问题当然不只这一些，假设我们把问题带到实验九-VGA 驱动，那么笔者问读者“你有没有信心将所有试验都实现？”。低级建模的准则不是一般的重要而是非常重要。为了后期的建模，这一准则一定要好好遵守。

# 第四章：低级建模 - 仿顺序操作

## 4.1 基本思路

C 语言的编程在理解上我们可以把它看成“[顺序操作](#)”。因为它就如同吃饭一样，有步骤的，张口，将饭入口，咬碎，吞下。Verilog HDL 语言在本质上它不同与 C 语言，要实现“[顺序操作](#)”，实际上是不可能的，但是我们可以模仿“[顺序操作](#)”的操作模式，亦即“[仿顺序操作](#)”。

要实现“[仿顺序操作](#)”如果没有任何建模技巧支持的话，结果是很糟糕的。幸运的是，在第二章~第三章读者们已经有“低级建模”是作为后台和准备，要实现“[仿顺序操作](#)”一点也不难。在这里我们以 SOS 信号实验 来认识“[仿顺序操作](#)”的基本思路。

如果以 C 语言来实现 SOS 信号实验，思路（写法）会是如下：

```
//建立 S 函数
void S_Function( void ) { ... }

//建立 O 函数
void O_Function( void ) { ... }

//建立 SOS 函数
void SOS_Function( void )
{
    S_Function();
    O_Function();
    S_Function();
}

//调用 SOS 函数
int main()
{
    SOS_Function();
    return 0;
}
```

先建立最基本的两个函数 S\_Function() 和 O\_Function，然后再建立一个函数 SOS\_Funcntion() 有次序的调用两个基本函数。最后在主函数中直接调用 SOS\_Function()。在肉眼看不见的 main 函数中，C 语言已经在底层而且很隐藏的执行“函数调用”和“函数返回”等指令来实现上述的功能。但是 Verilog HDL 语言没有这个便利，那么要 Verilog HDL 语言要如何模仿“[顺序操作](#)”该如何呢？

## 实验十一：SOS 信号之三

首先我们先要建立功能拟似 S 函数 和 O 函数的功能模块。

*s\_module.v*



```
1. module s_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Done_Sig,
6.     Pin_Out
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input Start_Sig;
12.    output Done_Sig;
13.    output Pin_Out;
14.
15.    /*************************************************************************/
16.
17.    parameter T1MS = 16'd49_999;
18.
19.    /*************************************************************************/
20.
21.    reg [15:0]Count1;
22.
23.    always @ ( posedge CLK or negedge RSTn )
24.        if( !RSTn )
25.            Count1 <= 16'd0;
26.        else if( Count1 == T1MS )
27.            Count1 <= 16'd0;
28.        else if( isCount )
```

```
29.          Count1 <= Count1 + 1'b1;
30.      else if( !isCount )
31.          Count1 <= 16'd0;
32.
33.      /*************************************************************************/
34.
35.      reg [9:0]Count_MS;
36.
37.      always @ ( posedge CLK or negedge RSTn )
38.          if( !RSTn )
39.              Count_MS <= 10'd0;
40.          else if( Count_MS == rTimes )
41.              Count_MS <= 10'd0;
42.          else if( Count1 == T1MS )
43.              Count_MS <= Count_MS + 1'b1;
44.
45.      /*************************************************************************/
46.
47.      reg [3:0]i;
48.      reg rPin_Out;
49.      reg [9:0]rTimes;
50.      reg isCount;
51.      reg isDone;
52.
53.      always @ ( posedge CLK or negedge RSTn )
54.          if( !RSTn )
55.              begin
56.                  i <= 4'd0;
57.                  rPin_Out <= 1'b0;
58.                  rTimes <= 10'd1000;
59.                  isCount <= 1'b0;
60.                  isDone <= 1'b0;
61.              end
62.          else if( Start_Sig )
63.              case( i )
64.
65.                  4'd0, 4'd2, 4'd4:
66.                      if( Count_MS == rTimes ) begin rPin_Out <= 1'b0; isCount <= 1'b0; i <= i + 1'b1; end
67.                      else begin isCount <= 1'b1; rPin_Out <= 1'b1; rTimes <= 10'd100; end
68.
69.                  4'd1, 4'd3, 4'd5:
70.                      if( Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
71.                      else begin isCount <= 1'b1; rTimes <= 10'd50; end
72.
```

```
73.          4'd6:  
74.          begin isDone <= 1'b1; i <= 4'd7; end  
75.  
76.          4'd7:  
77.          begin isDone <= 1'b0; i <= 4'd0; end  
78.  
79.          endcase  
80.  
81.          /**************************************************************************/  
82.  
83.          assign Done_Sig = isDone;  
84.          assign Pin_Out = !rPin_Out;  
85.  
86.          /**************************************************************************/  
87.  
88. endmodule
```

第 15~45 行是 1ms 的计数器。第 17 行定义了 1ms 的常量，21~31 行是 1ms 的定时器，然而 isCount 标志寄存器使能着该定时器 (28|30 行)。35~43 行是计数器，计数的结果由 rTimes 寄存器控制 (40 行)。

47~51 行是各个寄存器的声明。i 寄存器控制着执行步骤(47 行)，rPin\_Out 寄存器用于驱动 Pin\_Out 输出 (48 行)，然而 isDone 是用于驱动 Done\_Sig 的输出 (51 行)。

(注意 rTimes 的复位值是 0 以外，这一点请留意)第 62 行到 79 行是 s\_module.v 的核心部分，第 62 行是“**使能**”的功能，当 Start\_Sig 拉高以后 s\_module.v 才能工作。65~69 行是“延时的写法”，i 等于 0, 2, 4 时就设置 rPin\_Out 寄存器为逻辑 1，而延迟的时间是 100ms (67 行)。相反的，当 i 等于 1, 3, 5 的时候就设置 rPin\_Out 寄存器为逻辑 0，延迟的时间是 50ms (71 行)。

上述写法的思路如下：

当 i 等于 0 时，由于 if 条件未成立 (66 行)，那么 isCount 就是设置为 1，同时 rTimes 也会被赋予 100。当 isCount 设置为 1 时，定时器就会开始计数并且产生定时 (28 行)，计数器 (35~43 行) 每 1ms 的间隔都会计数一次 (42 行)，直到计数与 rTimes 的值一样 (40 行) 就会被清零。在同一个瞬间，if 条件成立，并且将 isCount 设置为逻辑 0，同时间将 rPin\_Out 寄存器拉低，最后 i 寄存器递增来表示下一个步骤 (66 行)。

最后在 73~77 行产生一个高脉冲的 Done\_Sig。

*o\_module.v*



```
1. module o_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Done_Sig,
6.     Pin_Out
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input Start_Sig;
12.    output Done_Sig;
13.    output Pin_Out;
14.
15.   ****
16.
17. parameter T1MS = 16'd49_999;
18.
19.   ****
20.
21. reg [15:0]Count1;
22.
23. always @ ( posedge CLK or negedge RSTn )
24.     if( !RSTn )
25.         Count1 <= 16'd0;
26.     else if( Count1 == T1MS )
27.         Count1 <= 16'd0;
28.     else if( isCount )
29.         Count1 <= Count1 + 1'b1;
30.     else if( !isCount )
31.         Count1 <= 16'd0;
32.
```

## Verilog HDL 那些事儿 – 建模篇

```
33.      *****/
34.
35.      reg [9:0]Count_MS;
36.
37.      always @ ( posedge CLK or negedge RSTn )
38.          if( !RSTn )
39.              Count_MS <= 10'd0;
40.          else if( Count_MS == rTimes )
41.              Count_MS <= 10'd0;
42.          else if( Count1 == T1MS )
43.              Count_MS <= Count_MS + 1'b1;
44.
45.      *****/
46.
47.      reg [3:0]i;
48.      reg rPin_Out;
49.      reg [9:0]rTimes;
50.      reg isCount;
51.      reg isDone;
52.
53.      always @ ( posedge CLK or negedge RSTn )
54.          if( !RSTn )
55.              begin
56.                  i <= 4'd0;
57.                  rPin_Out <= 1'b0;
58.                  rTimes <= 10'd1000;
59.                  isCount <= 1'b0;
60.                  isDone <= 1'b0;
61.              end
62.          else if( Start_Sig )
63.              case( i )
64.
65.                  4'd0, 4'd2, 4'd4:
66.                      if( Count_MS == rTimes ) begin rPin_Out <= 1'b0; isCount <= 1'b0; i <= i + 1'b1; end
67.                      else begin isCount <= 1'b1; rPin_Out <= 1'b1; rTimes <= 10'd400; end
68.
69.                  4'd1, 4'd3, 4'd5:
70.                      if( Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
71.                      else begin isCount <= 1'b1; rTimes <= 10'd50; end
72.
73.                  4'd6:
74.                      begin isDone <= 1'b1; i <= 4'd7; end
75.
76.                  4'd7:
```

```

77.           begin isDone <= 1'b0; i <= 4'd0; end
78.
79.       endcase
80.
81.   /*************************************************************************/
82.
83.   assign Done_Sig = isDone;
84.   assign Pin_Out = !rPin_Out;
85.
86.   /*************************************************************************/
87.
88. endmodule

```

基本上和 s\_module.v 大同小异，不同的只是延迟的时间（rTimes 赋值比较多）比较长（67 行）。因为我们知道 o 的摩斯密码比 s 的摩斯密码的时间长。

#### sos\_control\_module



接下来我们要建立一个控制 s\_module.v 和 o\_module.v 执行次序的控制模块 sos\_control\_module.v。

```

1. module sos_control_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     S_Done_Sig, O_Done_Sig,
6.     S_Start_Sig, O_Start_Sig,
7.     Done_Sig
8. );
9.
10.    input CLK;
11.    input RSTn;
12.    input Start_Sig;
13.    input S_Done_Sig, O_Done_Sig;
14.    output S_Start_Sig, O_Start_Sig;

```

```
15.     output Done_Sig;
16.
17.     /*****/
18.
19.     reg [3:0]i;
20.     reg isO;
21.     reg isS;
22.     reg isDone;
23.
24.     always @ ( posedge CLK or negedge RSTn )
25.         if( !RSTn )
26.             begin
27.                 i <= 4'd0;
28.                 isO <= 1'b0;
29.                 isS <= 1'b0;
30.                 isDone <= 1'b0;
31.             end
32.         else if( Start_Sig )
33.             case( i )
34.
35.                 4'd0:
36.                     if( S_Done_Sig ) begin isS <= 1'b0; i <= i + 1'b1; end
37.                     else isS <= 1'b1;
38.
39.                 4'd1:
40.                     if( O_Done_Sig ) begin isO <= 1'b0; i <= i + 1'b1; end
41.                     else isO <= 1'b1;
42.
43.                 4'd2:
44.                     if( S_Done_Sig ) begin isS <= 1'b0; i <= i + 1'b1; end
45.                     else isS <= 1'b1;
46.
47.                 4'd3:
48.                     begin isDone <= 1'b1; i <= 4'd4; end
49.
50.                 4'd4:
51.                     begin isDone <= 1'b0; i <= 4'd0; end
52.
53.             endcase
54.
55.     /*****
56.
57.     assign S_Start_Sig = isS;
58.     assign O_Start_Sig = isO;
```

```

59. assign Done_Sig = isDone;
60.
61. ****
62.
63. endmodule

```

在 20~21 行定义了 isS 和 isO 的标志寄存器。 isS 用于驱动 S\_Start\_Sig (57 行), isO 用于驱动 O\_Start\_Sig (58 行)。

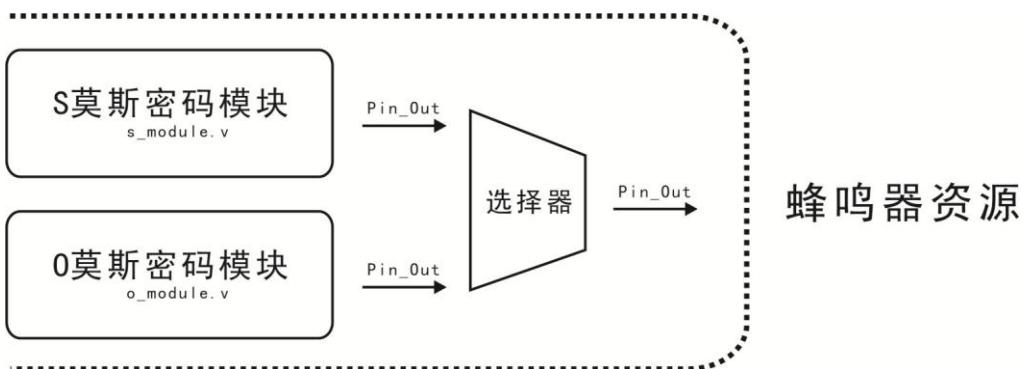
在 32~53 行是核心部分。32 行是“使能”，只要 Start\_Sig 不拉高，该模块就不会工作。我们知道“SOS 的产生次序”是先产生 S，产生 O，然后再产生 S，然而 35~45 行的“仿顺序操作”便是如此执行。

假设一个情况：

当 i 等于 0 时，由于 if 条件不满足 (36 行)，然后 isS 寄存器会被设置为 1 (37 行)。然后 sos\_module.v 就会等待 S\_Done\_Sig 的高脉冲反馈。当 S\_Done\_Sig 反馈高脉冲，也就是说 s\_module.v 已经完成操作，这也表示 if 条件满足了 (36 行)。然后 isS 标志寄存器会被拉低，i 寄存器会递增以表示下一步操作。

上述的情况会一直执行，直到 42~43 行。最后 Done\_Sig 会产生一个高脉冲。

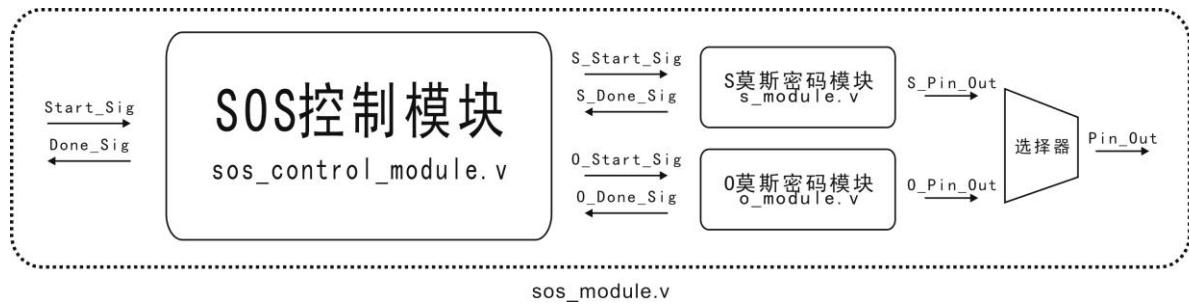
两仪性和多义性的问题：



在组合 s\_module.v, o\_module.v 和 sos\_control\_module.v 之前，存在着两仪性或者多义性的问题。从图上可以知道 s\_module.v 和 o\_module.v 的 Pin\_Out 输出是共享着蜂鸣器资源，但是有一个事实我们必须了解，就是“一个时期仅有一个输出”这个道理。为了有效协调 Pin\_Out 的输出，就添加“选择器”来控制输出。

(有关选择器的详细内容，可以参考笔者之前写过的笔记 - Verilog HDL 建模技巧 - 低级建模 · 仿顺序操作 (思路篇))

sos\_module.v



上图是组合模块 `sos_module.v`。对模块的调用有 `Start_Sig` 和 `Done_Sig` 信号，然而，在输出的一端，添加了“**选择器**”来协调输出。

```
1. module sos_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Done_Sig,
6.     Pin_Out
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input Start_Sig;
12.    output Done_Sig;
13.    output Pin_Out;
14.
15.    ****
16.
17.    wire S_Done_Sig;
18.    wire S_Pin_Out;
19.
20.    s_module U1
21.    (
22.        .CLK( CLK ),
23.        .RSTn( RSTn ),
```

```
24.      .Start_Sig( S_Start_Sig ),      // input - from U3
25.      .Done_Sig( S_Done_Sig ),       // output - to U3
26.      .Pin_Out( S_Pin_Out )        // output - to selector
27.
28. );
29.
30. *****/
31.
32. wire O_Done_Sig;
33. wire O_Pin_Out;
34.
35. o_module U2
36. (
37.     .CLK( CLK ),
38.     .RSTn( RSTn ),
39.     .Start_Sig( O_Start_Sig ),      // input - from U3
40.     .Done_Sig( O_Done_Sig ),       // output - to U3
41.     .Pin_Out( O_Pin_Out )        // output - to selector
42. );
43.
44. *****/
45.
46. wire S_Start_Sig;
47. wire O_Start_Sig;
48.
49. sos_control_module U3
50. (
51.     .CLK( CLK ),
52.     .RSTn( RSTn ),
53.     .Start_Sig( Start_Sig ),      // input - from top
54.     .S_Done_Sig( S_Done_Sig ),    // input - from U1
55.     .O_Done_Sig( O_Done_Sig ),    // input - from U2
56.     .S_Start_Sig( S_Start_Sig ),  // output - to U1
57.     .O_Start_Sig( O_Start_Sig ),  // output - to U2
58.     .Done_Sig( Done_Sig )        // output - to top
59. );
60.
61. *****/
62.
63. //selector
64.
65. reg Pin_Out;
66.
67. always @ ( * )
```

```
68.      if( S_Start_Sig ) Pin_Out = S_Pin_Out;      // select from U1
69.      else if( O_Start_Sig ) Pin_Out = O_Pin_Out; // select from U2
70.      else Pin_Out = 1'bx;
71.
72.      ****
73.
74. endmodule
```

第 15~72 行基本上和“图形”一样，实例化了 s\_module.v，o\_module.v 和 sos\_control\_module.v 然后按照“图形”连线。

在 62~72 行是“选择器”Pin\_Out 寄存器是用来驱动 Pin\_Out 输出（65 行）。在 67 行表示“任何时候都会变化”。然而 S\_Start\_Sig 和 O\_Start\_Sig 信号，作为选择器的辨别信号。当 S\_Start\_Sig 拉高时，是 S\_Pin\_Out 驱动着 Pin\_Out 输出，当 O\_Start\_Sig 拉高的时候，是 O\_Pin\_Out 驱动着 Pin\_Out 输出。

但是有一点必须注意，就是 第 70 行，每一个选择器都需要一个“默认状态”，如果没有添加这一行，会出现很多编译警告。

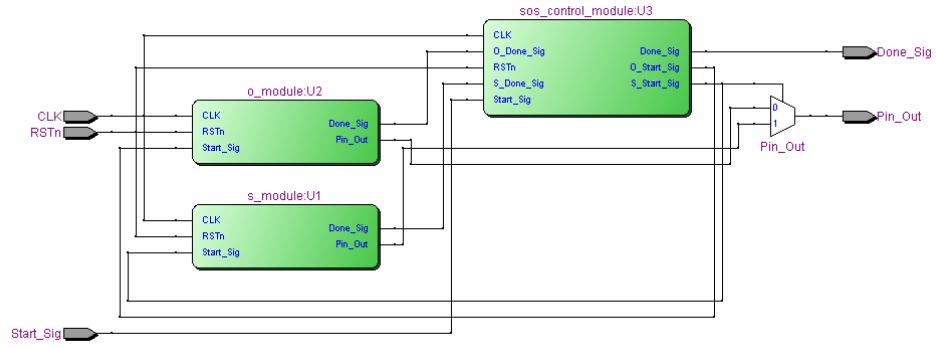
实验十一说明：

实验十一是以个简单的“仿顺序操作”的实例。上述的建模完全是模仿“顺序操作” - c 语言，去实现 SOS 功能。

“仿顺序操作”的建模都有标志性的“模块构造”，就是所有模块都拥有“Start\_Sig”和“Done\_Sig”。“Start\_Sig”如同 C 语言中的调用指令，“Done\_Sig”如同 C 语言的返回指令。这两个信号的存在就是为了控制模块的调用。

最后一点就是“选择器”，选择器在组合模块里面的存在真的很需要，因为“仿顺序操作”的模块，往往对“输出资源”存在“两仪性”或者“多义性”的问题。然而“选择器”还充分的使用“xx\_Start\_Sig”信号来达到“协调控制”。所以说“组合模块”，不仅是“组合”的工作而已，有时候要处理“组合”以外的工作。

完整的扩展图：



### 实验十一结论：

“仿顺序操作”的模块基本上用不着理解 Start\_Sig 和 Done\_Sig 之间发生了什么事都可以调用。

如果读者要求更详细的信息，笔者很建议读者温习“Verilog HDL 建模技巧 - 低级建模 · 仿顺序操作（思路篇）”这一本笔记。这一本笔记是笔者当时研究“仿顺序操作”的时候所完成的一本笔记，故记录比较详细。

“仿顺序操作”的建模，可能会使初次触碰的读者比较不习惯。笔者希望读者可以耐性去思考和理解。实际上“仿顺序操作”的建模很简单，因为“低级建模”已经充分的简化建模的难度。如果读者在第二章和第三章的基础实验掌握好的话，估计是没有什么困难而言。

### 实验十一演示：

这个演示就对实验十一完成的 sos\_module.v 调用吧。

*sos\_module\_demo.v*

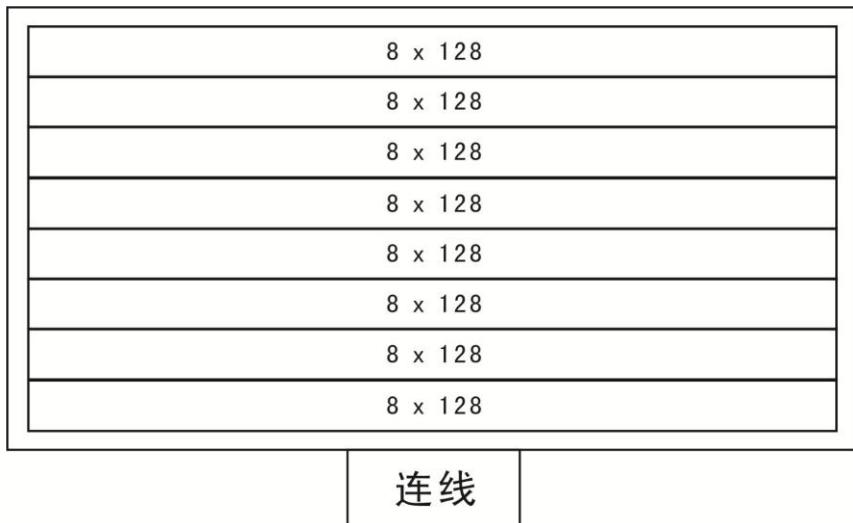
```
1. module sos_module_demo
2. (
3.     CLK, RSTn,
4.     Pin_Out
5. );
6.
7.     input CLK;
8.     input RSTn;
9.     output Pin_Out;
10.
11.    /*****
12.
13.    reg isStart;
14.
15.    always @ ( posedge CLK or negedge RSTn )
16.        if( !RSTn )
17.            isStart <= 1'b1;
18.        else if( Done_Sig )
19.            isStart <= 1'b0;
20.
21.    *****/
22.
23.    wire Done_Sig;
24.
25.    sos_module U1
26.    (
27.        .CLK( CLK ),
28.        .RSTn( RSTn ),
29.        .Start_Sig( isStart ),
30.        .Done_Sig( Done_Sig ),
31.        .Pin_Out( Pin_Out )
32.    );
33.
34.    *****/
35. endmodule
```

在 13 行定义了 `isStart` 标志寄存器，是用于是使能 `sos_module.v` (29 行)。一开始的时候 `isStart` 被复位为 1，然后 `sos_module.v` 就被使能并且开始工作 (17 行)。直到 `sos_module.v` 完成操作，就会经 `Done_Sig` 返回一个高脉冲。当检测到该脉冲 (18 行)，`isStart` 就会被设置为逻辑 0，最后 `sos_module.v` 就会保持沉默。换一句说就是，一开始 `sos_module.v` 会操作一次，然后操作完毕后就停止操作了。

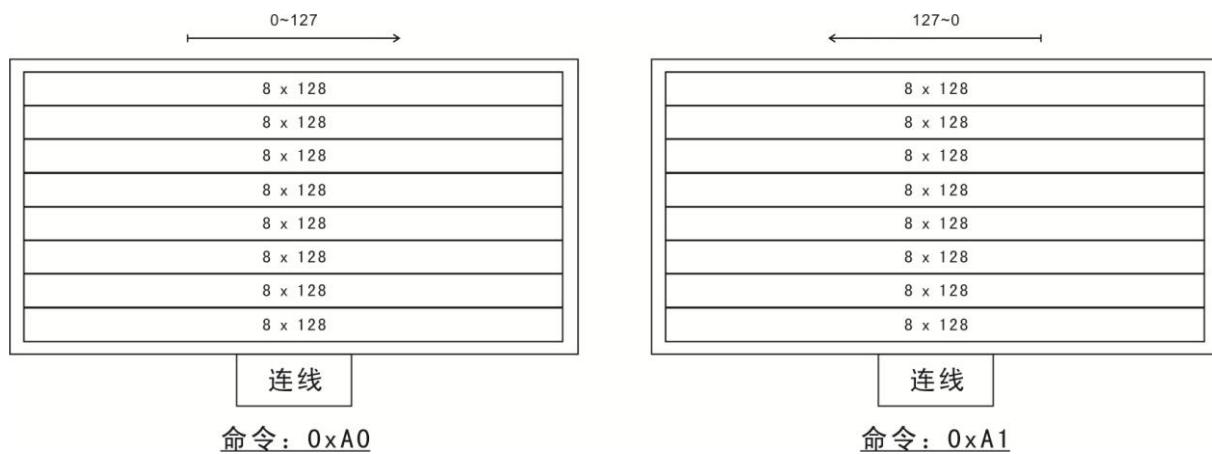
## 4.2 实验十二：12864（ST7565P）液晶驱动

### 显示概念

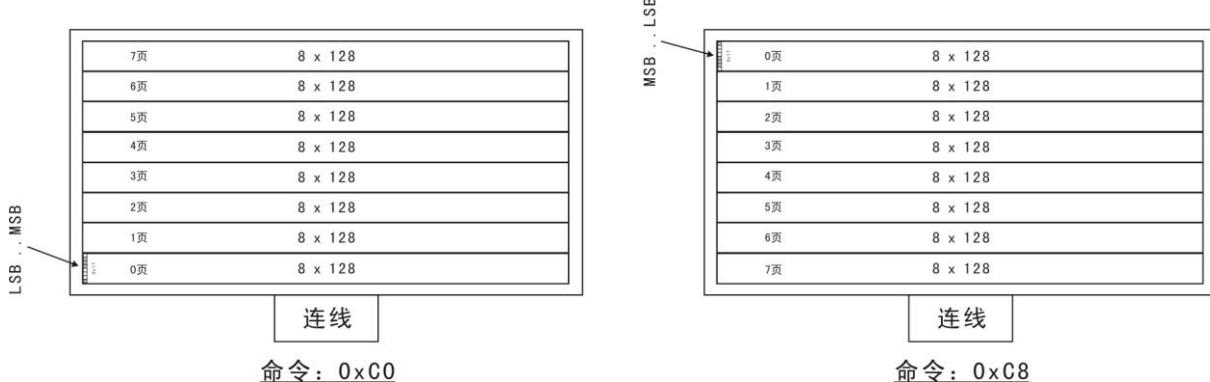
含有 ST7565P 芯片的液晶是没有字库支持的。但是，我们要的只是液晶绘图功能而已。液晶的“[显示](#)”，液晶的“[扫描次序](#)”全部都与 CGRAM 分配有很大的关系。我们先了解“[扫描次序](#)”吧。



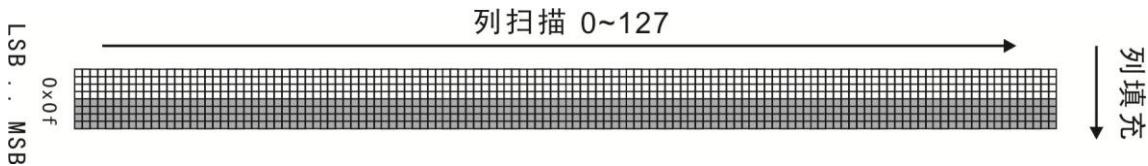
宏观上一副液晶是“[64 高 x 128 宽](#)”，它是由芯片 ST7565P 分成“[8 个 8 高 x 128 宽的页](#)”。至于液晶的“[扫描次序](#)”就与 4 个命令有关系。



上图表示了，当命令为 0xA0 列填充是“[自左向右](#)”，如果命令是 0xA1 列填充是“[自右向左](#)”。总归，这两个命令控制了“[列填充次序](#)”



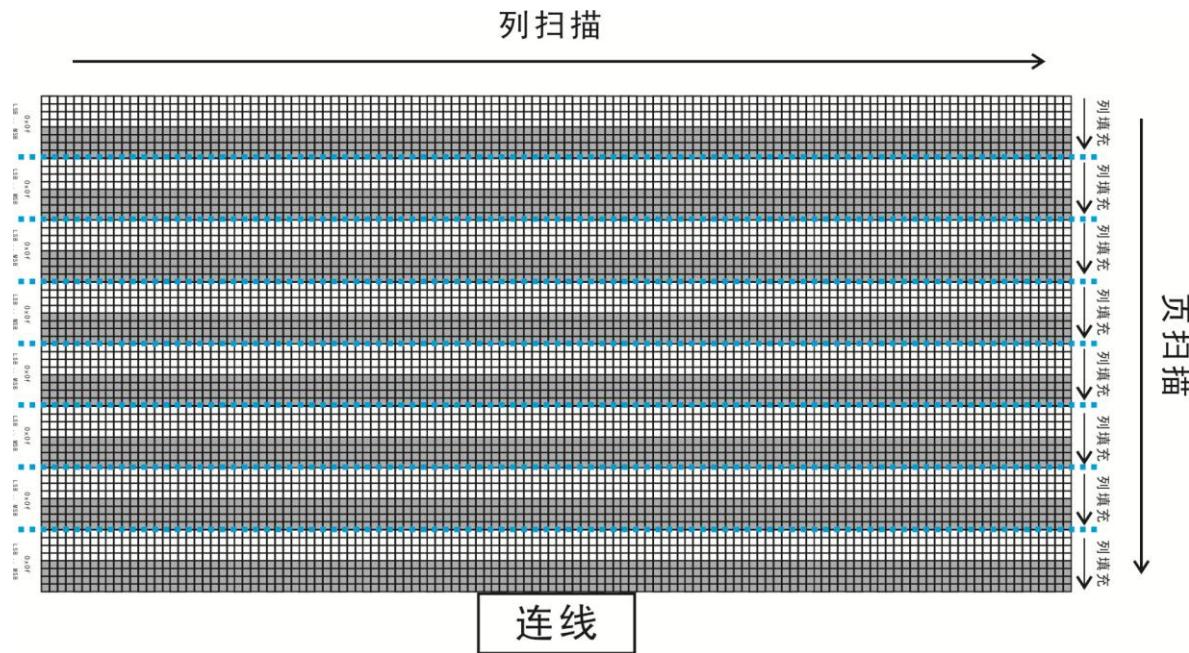
除了控制列填充的命令以外，当然还有控制“**页扫描次序**”的命令（每一页的高度是一个字节）。如上图，命令 0xC0 控制页扫描是“**从下至上**”，然而命令 0xc8 控制页扫描“**由上至下**”。无论页扫描的次序是“**从上至下**”还是“**从下至上**”，然而每一页的列填充，都是“**低位开始高位结束**”



关于列填充的问题。我们知道每“**一页**”都是由“**一个字节高 x 128 位宽**”组成。换句话说“**一页**”都是由“**一个字节数据，列填充 128 次**”。如上图中所示。假设“**页扫描次序**”是由上至下，填充的值是 0x0f，那么经过 128 次的“**列填充**”以后，一页的扫描结果会是如上图所示。

关于 ST7565P 芯片，命令，和液晶扫描它们之间的关系而已，我们简单来总结一下：

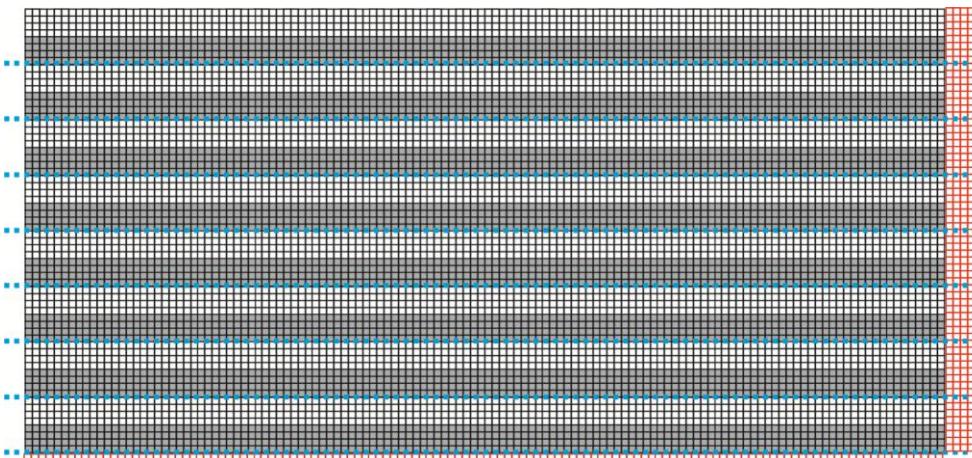
- (一) CGRAM 分布是由 8 页组成。
- (二) 每一页是由一个字节填充 128 次组成。
- (三) 列填充次序与命令 0xA0 与 0xA1 有关。
- (四) 页扫描次序与命令 0xC0 与 0xC8 有关。
- (五) 列填充字节的高位低位关系与页扫描次序（命令）有关。
- (六) 一页的高度是 8 位（一字节）。



上图所示是“[页扫描](#)”由上至下，“[列填充](#)”由左至右，列填充值是 [0x0f](#)。在 CGRAM 分布方面。CGRAM 可以说是由  $8 \text{ bits} \times 1024 \text{ words}$ ，如果以“[页](#)”去分配，也就是说  $8 \text{ page} \times 8 \text{ bits} \times 128 \text{ words}$ ，那么“[页](#)”和“[页](#)”之间的偏移量就是 128。这一点要好好的记住。

那么关于“[列地址](#)”和“[页地址](#)”又是如何呢？

事实上 CGRAM 的建立不可能是  $8 \text{ page} \times 8 \text{ bits} \times 128 \text{ words}$  那么完美的，必定有而外的列和页是不在显示的范围内，亦即第 8 页和第 128~131 列（如果页和列从 0 开始计算）。虽然说完成一次列填充，列地址会自动递增，然而 ST7565P 对于列地址的控制显得很笨蛋。



假设一开始我们设置“[页地址 0 和列地址 0 作为起始地址](#)”，当列填充到 127（如果从 0 开始计算），列地址会自动递增至 128，这显然不是显示范围了（红色部分）。所以呀，每一次完成 128 次的列填充，就要“[重新设置列起始地址和下一个页地址](#)”。

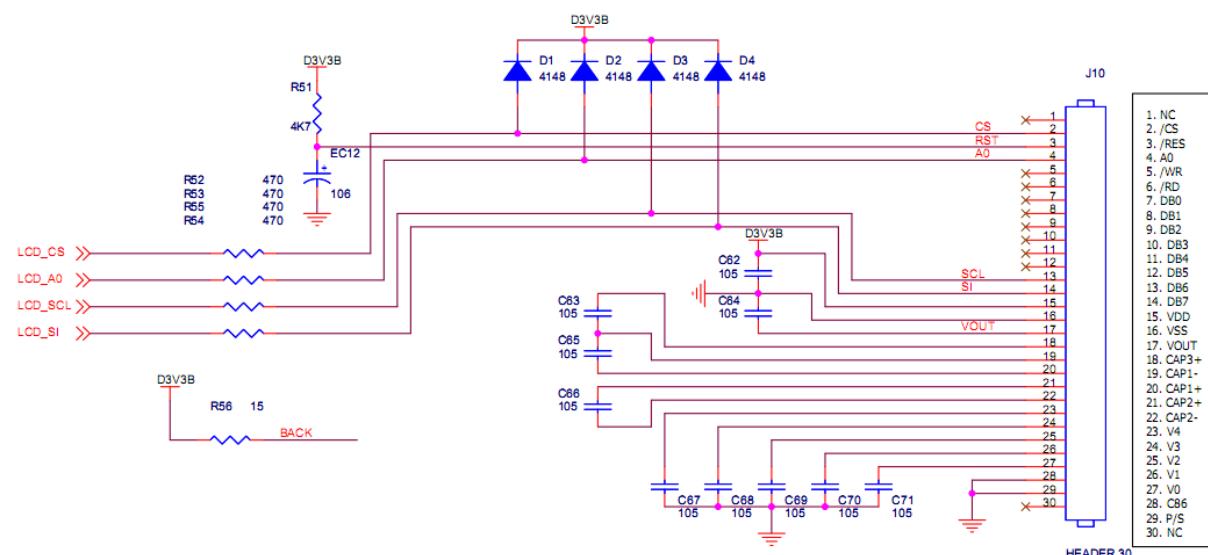
关于设置页地址的命令很简单，就是 `0xb?`。其中“`?`”就是页地址的设置处。假设输入 `0xb0`，也就是页地址 0。

关于设置列地址的命令是 `0x1?` 和 `0x0?`。命令 `0x1?` 的“`?`”是列地址的“**高四位**”，而，`0x0?` 的“`?`”是列地址的“**低四位**”。假设输入 `0x10, 0x00`，也就是说列地址是 `8'b 0000_0000`，亦即 0。

假设我要设置页地址 1（`0000 0001`），和列地址 65（`0100 0001`）。那么我需要输入：

```
0xb1;  
0x14;  
0x01;
```

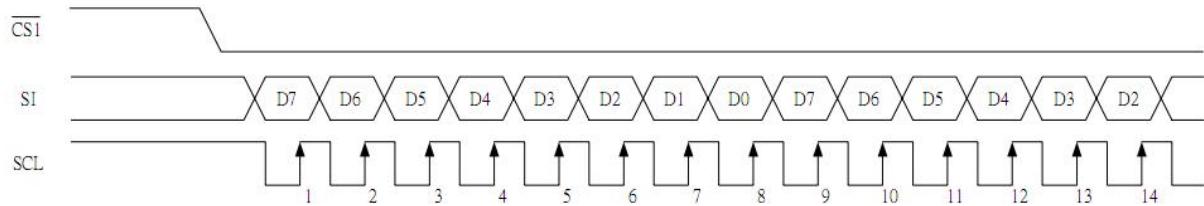
通过几页的内容，笔者只是要读者明白 ST7565P 芯片驱动液晶的显示规则和一些基本命令而已，真正的好戏儿在后头。



上图是在黑金开发板上的 12864 液晶原理图。对于串行输入模式的液晶来说，重要的引脚有 P/S, CS, A0, DB6 (SCL) 和 DB7 (SDI) 而已。ST7565P 芯片可以支持 3 种传输模式，当然最简单的传输模式还是 SPI 模式，然而控制“**传输模式的引脚**”就是 P/S。当 P/S 被拉低时就是表示“**串行传输模式**”。

CS 是使能信号（低电平有效）。A0 是命令或者数据决定信号（0 = 命令，1 = 数据）。SCL 是串行时钟信号，SI 是串行输入信号。至于其他的引脚属性自己去查相关的数据手册吧，这里只说重要的引脚而已。

在 SPI 传输中有分为主机和从机之分。主机的定义，有 CS 使能权，产生串行时钟。反之从机的定义是 CS 被使能，接收串行时钟。故黑金开发板上的 12864 液晶资源是从机，然而 FPGA 是主机。



在这里我们要干的事情只有一件而已，就是主机（FPGA）向从机（液晶资源）写数据。上图是向 ST7565P 芯片串行写入数据的时序图。SI 端，SCL 端和 CS 端都是由主机输出，从机输入（液晶读入数据）。从图中我们可以明白，从机读取（锁存）数据都是 CS 被拉低，并且发生在 SCL 信号的上升沿。

在顺序操作上（以 C 语言为例），ST7565P 芯片液晶的简易驱动的概念如下：

```
// 建立最基本的传输函数
SPI_Send{ unsigned char Data } {}

//建立传输数据函数
Send_Data( unsigned char Data)
{
    A0 = 1; SPI_Send( Data );
    .....
}

//建立传输命令函数
Send_Command( unsigned char Data )
{
    A0 = 0; SPI_Send( Data );
    .....
}

//建立初始化函数
Initial_Function()
{
    //液晶显示初始化配置
    Send_Command( 0xaf ); //液晶使能
    Send_Command( 0x40 ); //开始显示
    Send_Command( 0xa6 ); //此命令表达 1 = 点亮, 0 = 点灭

    //扫描次序配置
    Send_Command( 0xa0 ); //列扫描向左至右
    Send_Command( 0xc8 ); //行扫描从上至下
}
```

```

//内部电源配置
Send_Command( 0xa4 );
Send_Command( 0xa2 );
Send_Command( 0x2f );
Send_Command( 0x24 );
Send_Command( 0x81 ); //背光 LED 配置命令
Send_Command( 0x24 ); //背光 LED 配置值
}

//绘图函数
Draw_Fucntion()
{
    for( int page = 0; page < 8; page++ )
    {
        Send_Command( 0xb0 | page ); //设置页地址
        Send_Command( 0x10 );      //设置列地址 “高四位” - 0000
        Send_Command( 0x00 );      //设置列地址 “第四位” - 0000

        for( int x = 0 ; x < 128; x ++ ) Send_Data( *pic++ );
    }
}

//主函数
int main( void )
{
    Initial_Function();
    Draw_Function();
    whiel(1); //停止
}

```

在顺序操作中，我们会先建立最基本的 SPI\_Send() 函数，然后基于 SPI\_Send() 函数又建立 Send\_Data() 和 Send\_Command() 等函数。接下来，会基于 Send\_Command() 函数建立 Initial\_Function() 函数，和基于 Send\_Data() 函数建立 Draw\_Fucntion() 函数。最后在主函数中调用 Initial\_Function() 和 Draw\_Function() 函数。

在 4-1 章笔者说过了，顺序操作如同吃饭那样，有“[步骤的概念](#)”，然而顺序操作的语言很多都是高级语言，一些调用的指令都是被隐性处理，如函数的调用指令和返回指令等。在上述的内容中，高级函数无视了许多隐性指令，只是简单的多次嵌入低层函数，就能形成 Initial\_Function() 和 Draw\_Function() 等高级函数。然而 Verilog HDL 语言只能用建模和控制信号（Start\_Sig 和 Done\_Sig）模仿这些而已 .....

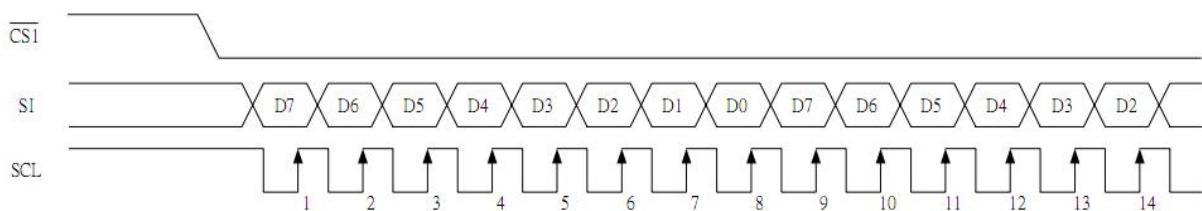
### SPI 发送模块

在这里，我们要在主机上建立，一个向从机写入数据的 SPI 发送模块，首先我们先从从 C 语言上了解几个主机在 SPI 写操作上容易被疏忽的小细节：

我们知道 SPI 设备在传输都有一个规则，SCL 时钟信号在“[上升沿](#)”的时候是“[锁存数据](#)”，SCL 时钟信号在“[下降沿](#)”是“[设置数据](#)”。在这里我们 SPI 主机（FPGA），写操作要干的工作就是在“[拉高 SCL 时钟信号之前](#)”设置数据（移位数据），设置数据之后，再拉高时钟信号。但是我们常常会忽略了一些具体的细节。

<pre>SPI_Send( unsigned char Data ) {     CS = 0; SCL = 0;      for( int i = 0; i &lt; 8; i++ )     {         if( Data &amp; 0x80 ) SI = 1;         else SI = 0;         Data &lt;&lt;= 1;         SCL = 0;         SCL = 1;     } }</pre>	<pre>SPI_Send( unsigned char Data ) {     CS = 0; SCL = 1;      for( int i = 0; i &lt; 8; i++ )     {         SCL = 0;         if( Data &amp; (7-i) ) SI = 1;         else SI = 0;          SCL = 1;     } }</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

上面有两个主机的 SPI\_Send 函数（写函数），左边的写法是最常用，但是也是最容易忽略小细节。相比右边的写法比较谨慎，在最低的程度上符合一写小细节。



再来，我们继续引用 ST7565P 芯片的写入时序图，分析并且区分上边的两个写法。关于 SCL 信号在空闲的时候总是处于高电平。当主机开始向从机写入数据，主机会先拉低 CS 信号，再拉低 SCL 信号，然后“[设置](#)”数据，亦即主机（FPGA）更新 SI 的数据（主机数据移位操作），最后再拉高 SCL 信号。同一时间，从机会因为 SCL 的上升沿变化，从机（液晶资源）“[锁存](#)”（从机读取数据操作）SI 上的数据。

很明显左边的写法没有符合这些细节，然而右边的写法却符合这些细节。无论是左边的

写法还是右边的写法，都忽略了一个致命的细节，两种写法都无法确定 SPI 时钟信号的时钟频率。当然我们可以基于上述的写法产生更笨拙的写法，如下：

```
SPI_Send( unsigned char Data )
{
    CS = 0; SCL = 1;

    for( int i = 0; i < 8; i++ )
    {
        SCL = 0; Delay_US(10); //添加延迟函数

        if( Data & (7-i) ) SI = 1;
        else SI = 0;

        SCL = 1; Delay_US(10); //添加延迟函数
    }
}
```

哦！这样的写法只会浪费单片机宝贵的处理资源 ... 除非这个单片机有置入实时操作系统，否则那样的活儿将会是非常的糟糕。

在这里我们明白到，如果用顺序操作语言驱动 SPI 设备的写入，在步骤上要尽量迎合时序图。此外，我们还明白到，顺序操作语言为了确定 SPI 的时钟信号的频率（时钟周期）之际，必须添加延迟函数，但是延迟函数会大量的减低单片机的使用率（浪费处理器的处理资源）。

接下来，看笔者如何使用 Verilog HDL 语言来模仿 SPI\_Send() 函数，而且不失一些 SPI 写入操作上的小小细节，并且可以确定 SPI 时钟信号的周期。



上图所示是要建立的功能模块 spi\_write\_module.v，亦即是主机的 SPI 发送模块。为了最大发挥 Verilog HDL 语言特性，SPI\_Data 和 SPI\_Out 的位配置如下：

SPI_Data			SPI_Out			
[9]	[8]	[7 .. 0]	[3]	[2]	[1]	[0]
CS	A0	Data	CS	A0	SCL	SI

*spi\_write\_module.v*

```
1. module spi_write_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     SPI_Data,
6.     Done_Sig,
7.     SPI_Out
8. );
9.
10.    input CLK;
11.    input RSTn;
12.    input Start_Sig;
13.    input [9:0]SPI_Data;
14.    output Done_Sig;
15.    output [3:0]SPI_Out; // [3]CS [2]A0 [1]CLK [0]DO
16.
17.    ****
18.
19. parameter T0P5US = 5'd24;//0.5us
20.
21. ****
22.
23. reg [4:0]Count1;
24.
25. always @ ( posedge CLK or negedge RSTn )
26.     if( !RSTn )
27.         Count1 <=5'd0;
28.     else if( Count1 == T0P5US )
29.         Count1 <= 5'd0;
30.     else if( Start_Sig )
31.         Count1 <= Count1 + 1'b1;
32.     else
33.         Count1 <= 5'd0;
34.
35. ****
36.
37. reg [4:0]i;
38. reg rCLK;
39. reg rDO;
40. reg isDone;
```

```

41.
42.      always @ ( posedge CLK or negedge RSTn )
43.          if( !RSTn )
44.              begin
45.                  i <= 5'd0;
46.                  rCLK <= 1'b1;
47.                  rDO <= 1'b0;
48.                  isDone <= 1'b0;
49.              end
50.          else if( Start_Sig )
51.              case( i )
52.
53.                  5'd0, 5'd2, 5'd4, 5'd6, 5'd8, 5'd10, 5'd12, 5'd14:
54.                      if( Count1 == T0P5US ) begin rCLK <= 1'b0; rDO <= SPI_Data[ 7 - ( i >> 1 ) ]; i <= i + 1'b1; end
55.
56.                  5'd1, 5'd3, 5'd5, 5'd7, 5'd9, 5'd11, 5'd13, 5'd15 :
57.                      if( Count1 == T0P5US ) begin rCLK <= 1'b1; i <= i + 1'b1; end
58.
59.                  5'd16:
60.                      begin isDone <= 1'b1; i <= i + 1'b1; end
61.
62.                  5'd17:
63.                      begin isDone <= 1'b0; i <= 5'd0; end
64.
65.              endcase
66.
67.          /*****
68.
69.          assign Done_Sig = isDone;
70.          assign SPI_Out = { SPI_Data[9], SPI_Data[8], rCLK, rDO };
71.
72.          *****/
73.
74.      endmodule

```

SCL 的时钟频率定义为 1Mhz , 也就是说周期时间是 1us , 半周期就是 0.5us 。如果以 50Mhz 来定时, 那么计数的结果是 25。在 19 行定义了 0.5us 的常量, 第 23~33 行是 0.5us 的定时器。但是比较不同的是, 这个定时器平时不工作, 当 Start\_Sig 拉高的时候才开始计数 (第 30 行)。

第 37~65 行是 spi\_write\_module.v 的核心功能。i 寄存器表示操作步骤, rCLK 寄存器表示 SCL 然而 rDO 寄存器表示 SI 。如同前面所述那样, SCL 时钟信号, 处于空闲状态时是出于高电平, 所以 rCLK 复位值是逻辑 1 (46 行)。

在这里稍微提醒一下：

SPI\_Data：第 9 位表示 CS，第 8 位表示 A0, 第 7..0 位表示一字节数据。

SPI\_Out：第 3 位表示 CS，第 2 位表示 A0，第 1 位表示 SCL，第 0 位表示 SI。

当 Start\_Sig 拉高的同时，定时器开始计数（30 行），该模块也开始执行（50 行）。

当第一个 0.5us 定时产生的时候（54 行），也就是第一个时钟的前半周期，亦即下降沿，rCLK 设置为逻辑 0。根据 SPI 传输的规则，下降沿的时候主机设置数据，rDO 赋予 SPI\_Data 信号的第 7 位（SPI 传输是从最高位开始，最低位结束），最后 i 递增以示下一个步骤。

当 i 等于 1 的时候并且定时产生（56 行），这表示第一个时钟的后半周期，亦即上升沿，rCLK 设置为逻辑 1。在 SPI 传输的规则中上升沿的时候，从机锁存数据。然后 i 递增以示下一个步骤。

上述的步骤会一直重复到第八次，直到一字节的数据发送完毕。最后会产生一个完成信号（59~63 行）。

这里有一个表达式需要说明一下：

$i >> 1$ ：表示  $i / 2$ 。右移操作也代表了 “ $i / j^2$ ” ( $j$  是右移次数)。

假设  $8 >> 2$ ，亦即  $8 / 2^2$  等于 2。  $8 >> 3$ ，亦即  $8 / 2^3$  等于 1。

最后还有一个重点就是 SPI\_Out 的驱动（70 行）。在上面我已经重复过 SPI\_Out 是占 4 位的输出。而且每一个位都有意义。

SPI\_Out 第 3 位：表示了 CS，所以直接由 SPI\_Data 的第 9 位驱动。

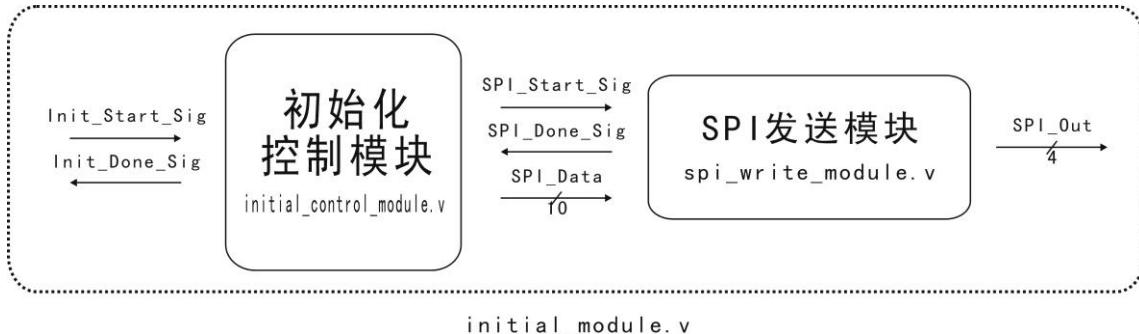
SPI\_Out 第 2 位：表示了 A0，同样也是直接由 SPI\_Data 的第 8 位驱动。

SPI\_Out 第 1 位：表示了 SCL，以寄存器 rCLK 来驱动。

SPI\_Out 第 0 位：表示了 SI，以寄存器 rDO 来驱动。

这样的目的是简化连线的复杂度。我们知道 Verilog HDL 语言的位操作是很强大。懂得善用，会对建模提到很大的帮助。

## 初始化模块



乍看 initial\_module.v 既包含了 initial\_control\_module.v 和 spi\_write\_module.v。initial\_module.v 是用来模仿 Initial\_Function()。spi\_write\_module.v 前面已经说过了，至于 initial\_control\_module.v 我们知道我们需要一个控制模块来执行初始化的步骤，而这个控制模块就是这个初衷。

*initial\_control\_module.v*

```

1. module initial_control_module
2. (
3.   CLK, RSTn,
4.   Start_Sig,
5.   SPI_Done_Sig,
6.   SPI_Start_Sig,
7.   SPI_Data,
8.   Done_Sig
9. );
10.
11.   input CLK;
12.   input RSTn;
13.   input Start_Sig;
14.   input SPI_Done_Sig;
15.   output SPI_Start_Sig;
16.   output [9:0]SPI_Data;
17.   output Done_Sig;
18.
19.   *****/
20.
21.   reg [3:0]i;
22.   reg [9:0]rData;
```

```
23.    reg isSPI_Start;
24.    reg isDone;
25.
26.    always @ ( posedge CLK or negedge RSTn )
27.        if( !RSTn )
28.            begin
29.                i <= 4'd0;
30.                rData <= { 2'b11, 8'h2f };
31.                isSPI_Start <= 1'b0;
32.                isDone <= 1'b0;
33.            end
34.
35.        else if( Start_Sig )
36.            case( i )
37.
38.                4'd0:
39.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
40.                    else begin rData <= { 2'b00, 8'haf }; isSPI_Start <= 1'b1; end
41.
42.                4'd1:
43.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
44.                    else begin rData <= { 2'b00, 8'h40 }; isSPI_Start <= 1'b1; end
45.
46.                4'd2:
47.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
48.                    else begin rData <= { 2'b00, 8'ha6 }; isSPI_Start <= 1'b1; end
49.
50.                4'd3:
51.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
52.                    else begin rData <= { 2'b00, 8'ha0 }; isSPI_Start <= 1'b1; end
53.
54.                4'd4:
55.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
56.                    else begin rData <= { 2'b00, 8'hc8 }; isSPI_Start <= 1'b1; end
57.
58.                4'd5:
59.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
60.                    else begin rData <= { 2'b00, 8'ha4 }; isSPI_Start <= 1'b1; end
61.
62.                4'd6:
63.                    if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
64.                    else begin rData <= { 2'b00, 8'ha2 }; isSPI_Start <= 1'b1; end
65.
66.                4'd7:
```

```

67.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
68.         else begin rData <= { 2'b00, 8'h2f }; isSPI_Start <= 1'b1; end
69.
70.         4'd8:
71.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
72.         else begin rData <= { 2'b00, 8'h24 }; isSPI_Start <= 1'b1; end
73.
74.         4'd9:
75.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
76.         else begin rData <= { 2'b00, 8'h81 }; isSPI_Start <= 1'b1; end
77.
78.         4'd10:
79.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
80.         else begin rData <= { 2'b00, 8'h24 }; isSPI_Start <= 1'b1; end
81.
82.         4'd11:
83.         begin rData <= { 2'b11, 8'h2f }; isDone <= 1'b1; i <= i + 1'b1; end
84.
85.         4'd12:
86.         begin isDone <= 1'b0; i <= 4'd0; end
87.
88.     endcase
89.
90. ****
91.
92.     assign Done_Sig = isDone;
93.     assign SPI_Start_Sig = isSPI_Start;
94.     assign SPI_Data = rData;
95.
96. ****
97.
98. endmodule

```

第 11~17 行定义了输出和输入口相关的信息，具体和图形一样。在 22 行定义了 rData 寄存器，它是用来驱动 SPI\_Data (94 行)。第 23 行定义了 isSPI\_Start 标志寄存器，如命名般一样，是用来驱动 SPI\_Start\_Sig，换句话就是 SPI 发送模块的使能信号。

第 26~88 是该模块的核心部分。当上一层将 Start\_Sig 拉高的时候（注意：initial\_control\_module.v 的 Start\_Sig 外部连线是 Initial\_Start\_Sig），该模块就开始工作（35 行）。（全核心功能都是使用“仿顺序操作”的写法）

前三个命令是液晶的“[显示配置命令](#)”（38~48 行），然而我们知道要对液晶写数据的时候，CS 和 A0 都必须拉低，由于 SPI\_Data 位分配的关系。rData 寄存器第 9 .. 8 位都是赋予 2'b00。

假设 i 等于 0。那么会发送第一个命令，亦即 0xaf（39 行）一开始由于条件 if 没有达到，(40 行) rData 会被赋予 2'b00, 8'haf，并且 isSPI\_Start 会设置为逻辑 1，这时候 SPI 发送模块就会开始工作。直到 SPI 发送模块发送一字节数据，并且反馈一个完成信号的高脉冲 (SPI\_Done\_Sig)，if 条件就会成立 (39 行)，然后 isSPI\_Start 就会被设置为逻辑 0，然后 i 递增以示下一步骤。

类似上面的操作会一直重复，直到完成发送 3 个“显示配置命令”，2 个“扫描次序配置命令”，和 6 个“内部电源配置命令”(38~80 行)。最后该模块会反馈一个完成信号给上一层模块 (82~86 行)，并且 (83 行) 复位 rData 寄存器（前两位必须设置为逻辑 1，而后八位可以是任意值）。

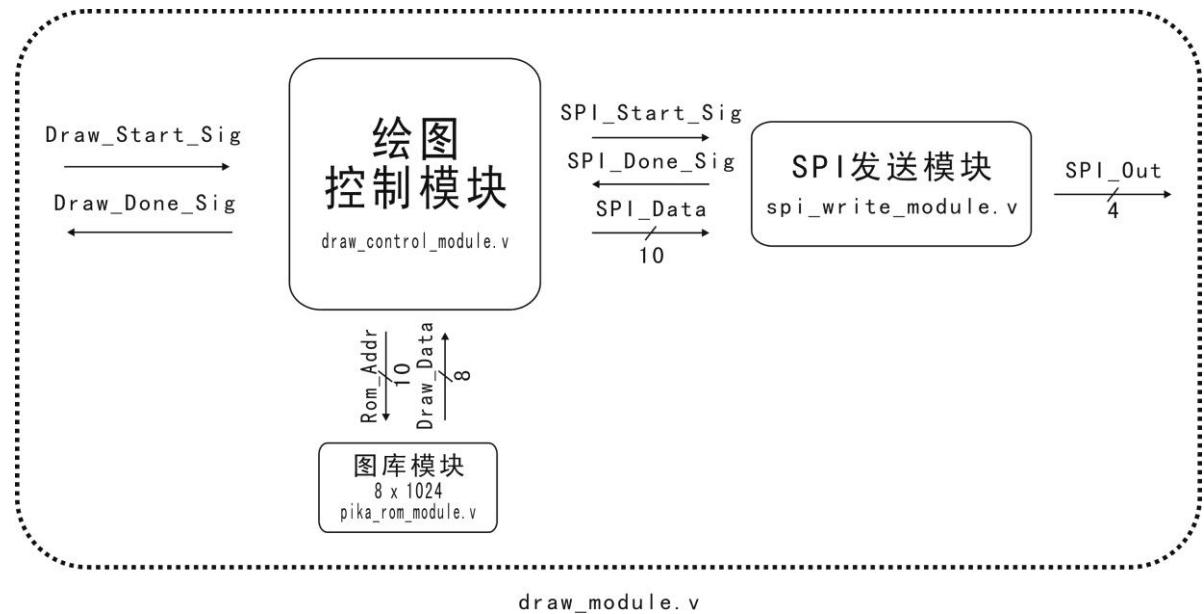
*initial\_module.v*

```
1. module initial_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Done_Sig,
6.     SPI_Out
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input Start_Sig;
12.    output Done_Sig;
13.    output [3:0]SPI_Out; // [3]CS [2]A0 [1]CLK [0]DO
14.
15.    ****
16.
17.    wire SPI_Start_Sig;
18.    wire [9:0]SPI_Data;
19.
20.    initial_control_module U1
21.    (
22.        .CLK( CLK ),
23.        .RSTn( RSTn ),
24.        .Start_Sig( Start_Sig ),           // input - from top
25.        .SPI_Done_Sig( SPI_Done_Sig ),   // input - from U2
26.        .SPI_Start_Sig( SPI_Start_Sig ), // output - to U2
27.        .SPI_Data( SPI_Data ),          // output - to U2
```

```
28.      .Done_Sig( Done_Sig )          // output - to top
29. );
30.
31.  *****/
32.
33. wire SPI_Done_Sig;
34.
35. spi_write_module U2
36. (
37.     .CLK( CLK ),
38.     .RSTn( RSTn ),
39.     .Start_Sig( SPI_Start_Sig ),    // input - from U1
40.     .SPI_Data( SPI_Data ),        // input - from U1
41.     .Done_Sig( SPI_Done_Sig ),    // output - to U1
42.     .SPI_Out( SPI_Out )         // output - to top
43. );
44.
45. *****/
46.
47. endmodule
```

initial\_module.v 是 initial\_control\_module.v 和 spi\_write\_module 的组合模块。连线关系基本上和“[图形一样](#)”。

## 绘图模块



draw\_module.v 是一个组合模块，它包含 spi\_write\_module.v , draw\_control\_module.v 和 pika\_rom\_module.v。pika\_rom\_module.v 是一个 8 bits x 1024 words 的 rom。

draw\_control\_module.v 控制模块主要是控制绘图的所有操作步骤，然而 pika\_rom\_module.v 包含了所需要的图像信息。该控制模块对 spi\_write\_module.v 的链接也和 initial\_control\_module.v 一样。

### draw\_control\_module.v

```
1. module draw_control_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     Draw_Data,
6.     SPI_Done_Sig,
7.     SPI_Start_Sig,
8.     SPI_Data,
9.     Rom_Addr,
10.    Done_Sig
11. );
12.
13.    input CLK, RSTn;
```

```

14.      input Start_Sig;
15.      input [7:0]Draw_Data;
16.      input SPI_Done_Sig;
17.      output SPI_Start_Sig;
18.      output [9:0]SPI_Data;
19.      output [9:0]Rom_Addr;
20.      output Done_Sig;
21.
22.      *****/
23.
24.      reg [5:0]i;
25.      reg [7:0]x;
26.      reg [3:0]y;
27.      reg [9:0]rData;
28.      reg isSPI_Start;
29.      reg isDone;
30.
31.      always @ ( posedge CLK or negedge RSTn )
32.          if( !RSTn )
33.              begin
34.                  i <= 6'd0;
35.                  x <= 8'd0;
36.                  y <= 4'd0;
37.                  rData <= { 2'b11, 8'h00 };
38.                  isSPI_Start <= 1'b0;
39.                  isDone <= 1'b0;
40.              end
41.          else if( Start_Sig )
42.              case( i )
43.
44.                  6'd0, 6'd4, 6'd8, 6'd12, 6'd16, 6'd20, 6'd24, 6'd28:
45.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
46.                      else begin rData <= { 2'b00, 4'hb, y }; isSPI_Start <= 1'b1; end
47.
48.                  6'd1, 6'd5, 6'd9, 6'd13, 6'd17, 6'd21, 6'd25, 6'd29:
49.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
50.                      else begin rData <= { 2'b00, 4'h1, 4'h0 }; isSPI_Start <= 1'b1; end
51.
52.                  6'd2, 6'd6, 6'd10, 6'd14, 6'd18, 6'd22, 6'd26, 6'd30:
53.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
54.                      else begin rData <= { 2'b00, 4'h0, 4'h0 }; isSPI_Start <= 1'b1; end
55.
56.                  6'd3, 6'd7, 6'd11, 6'd15, 6'd19, 6'd23, 6'd27, 6'd31:
57.                      if( x == 8'd128 ) begin y <= y + 1'b1; x <= 8'd0; i <= i + 1'b1; end

```

## Verilog HDL 那些事儿 – 建模篇

```
58.          else if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; x <= x + 1'b1; end
59.          else begin rData <= { 2'b01, Draw_Data }; isSPI_Start <= 1'b1; end
60.
61.          6'd32:
62.          begin rData <= { 2'b11, 8'd0 }; y <= 4'd0; isDone <= 1'b1; i <= i + 1'b1; end
63.
64.          6'd33:
65.          begin isDone <= 1'b0; i <= 6'd0; end
66.
67.      endcase
68.
69.  *****/
70.
71. assign SPI_Start_Sig = isSPI_Start;
72. assign Rom_Addr = x + (y << 7);
73. assign SPI_Data = rData;
74. assign Done_Sig = isDone;
75.
76. *****/
77.
78. endmodule
```

第 13~20 行的定义基本上都和“[图形](#)”一样，除了 Start\_Sig 和 Done\_Sig 比较特别，它们的外部连线是 Draw\_Start\_Sig 和 Draw\_Done\_Sig。

第 31~67 行是该模块的核心部分，但是别被它吓到了，它不过是充气而已。在这里我们简单复习一下在“[顺序操作](#)”中的 Draw\_Function() 的操作次序。

```
Draw_Fucntion()
{
    for( int page = 0; page < 8; page++)
    {
        Send_Command( 0xb0|page); //页地址配置
        Send_Command( 0x10 );    //列地址高四位配置
        Send_Command( 0x00 );    //列地址第四位配置

        for( int x = 0; x < 128; x++ ) Send_Data( *p++ ); //发送 128 次列填充
    }
}
```

上述的 Draw\_Function() 函数将“绘图操作”的次序表达得一了百了，而且该函数中最大作用就是 for 循环，很可惜 Verilog HDL 语言是不推荐使用 for 循环。(不要问笔者为什么，很多的参考书上都是这样写的，如果用笔者的话说，for 循环不适合 Verilog HDL 语言的风格)。

在 Draw\_Function() 函数之中，第一个 for 循环控制 page，亦即页。并且在每一个页的开始都重新配置页地址和列地址。至于第二个 for 循环是用于控制 128 次的列填充。

那么 Verilog HDL 语言该如何呢？

在 24~29 行中生命了相关寄存器，i 控制执行步骤，x 控制列扫描地址（列填充次数），y 控制页扫描次数，rData 是用来驱动 SPI\_Out (73 行)，而 isSPI\_Start 是用来驱动 SPI\_Start\_Sig。当 Start\_Sig 被拉高的时候 (41 行)，该控制模块就开始工作。

我们先假设一个情况：

当 i 等于 0 的时候，由于 if 条件不成立 (45 行)。经“顺序操作”关系，必须先设置页地址，rData 被赋予 2'b00 (CS=0, A0 = 0, 亦即发送命令) 和 y 寄存器的值，Y 寄存器复位值是 8'd0。然后 isSPI\_Start 寄存器被设置为逻辑 1 (46 行)。此时 SPI 发送模块开始工作。

当 SPI 发送完一字节的数据，就会反馈一个高脉冲至完成信号 SPI\_Done\_Sig。此时 if 条件就会成立(45 行)，isSPI\_Start 寄存器被设置为 0，然后 i 递增以示下一步骤。

当页地址设置完毕后，接下来的操作就要设置列地址。48~50 行是设置列地址的高四位，52~54 行是设置列地址的低四位。具体操作和设置页地址一样。不一样的是，每一次设置“新一页”，列地址都必须复原为 00。

当页地址和列地址设置 okay 后，接下来就是 128 次的列填充操作了。x 寄存是用来控制“列填充次数”，同期间也充当“列扫描地址（列寻址）”。一开始的时候，由于 if 条件和 else if 条件同样无法达成 (57~58 行)，rData 会被赋予 2'b01 (CS = 0, A0 = 1, 亦即数据) 和八位数据，然而八位数据 Draw\_Data 的取值是来至 pika\_rom\_module.v 的地址 0 的值。同期间 isSPI\_Start 被设置为逻辑 1，亦即 SPI 发送模块被使能。

**在这里我们先暂停一下！**

为了迎合 CGRAM 的分配，pika\_rom\_module.v 同样也是采用一样的分配方式。pika\_rom\_module.v 是 8 bits x 1024 words 的分配。如果要迎合 CGRAM 的分配方式 pika\_rom\_module.v 可以这样定义 8 pages x 8 bits x 128 words，这也就说每一页之间的页偏移量是 128 个 words。在 72 行是 Rom\_Addr 的输出，至于为什么驱动的表达式是  $x + (y \ll 7)$  呢？“ $y \ll 7$ ”等价于“ $y * 128$ ”，我们知道，y 寄存器代表了液晶设置的当前页，x 寄存器代表了液晶当前的列填充次数。

为了从 pika\_rom\_module.v 中读取到正确的 Draw\_Data 值，“ $x + (y \ll 7)$ ”表达式也成为了 Rom 地址的转换表达式。

笔者假设一个情况：

当 y 寄存器等于 0 值时，亦即液晶已经就绪列填充第 0 页。而第 0 页的列填充值是在 pika\_rom\_module.v 的 0~127 中。假设模块开始填充第 0 列，经过表达式转换后：  
 $(0 + (0 << 7)) = 0$ ; 也就是说第 0 页第 0 列的填充值在 pika\_rom\_module.v 的地址 0。

再假设一个情况：

当 y 寄存器的值等于 3，亦即液晶正在填充第 3 页，然而第 3 页的列填充值是在 pika\_rom\_module.v 的 384~491 中。假设列填充在 63 开始，经过表达式转换后：  
 $(63 + (3 << 7)) = 447$ ; 也就是说第 3 页第 63 列的填充值在 pika\_rom\_module.v 的地址 447。

**好了继续上面的话题。**

直到 SPI 发送模块完成一字节数据的发送后，就会反馈一个高脉冲的完成信号。此时（58 行）else if 条件成立，isSPI\_Start 被设置为逻辑 0，x 寄存器递增，以示下一个列填充。这个时候，会再一次进入 else（59 行），isSPI\_Start 被设置为逻辑 1，rData 被赋值为 2'b01 和 Draw\_Data，由于 72 行表达式的关系，这时候列填充的值是来至 pika\_rom\_module.v 地址 1 的值。

上述的内容会一直重复到 x 递增至 128 次，直到 if 条件成立（57 行）y 寄存器递增，以示下一个页地址，然后 x 寄存器被赋值为 0，最后 i 递增以示下一个步骤。

“[设置页地址，设置初始列地址，128 次的列填充](#)”这样的过程会一直重复，直到完成液晶全部 8 个页的所有列填充，draw\_control\_module.v 就会产生一个完成信号（61~65 行），同期间也会复位 rData 和 y 寄存器的值（62 行）。

就这样一副 12864 液晶扫描完毕。

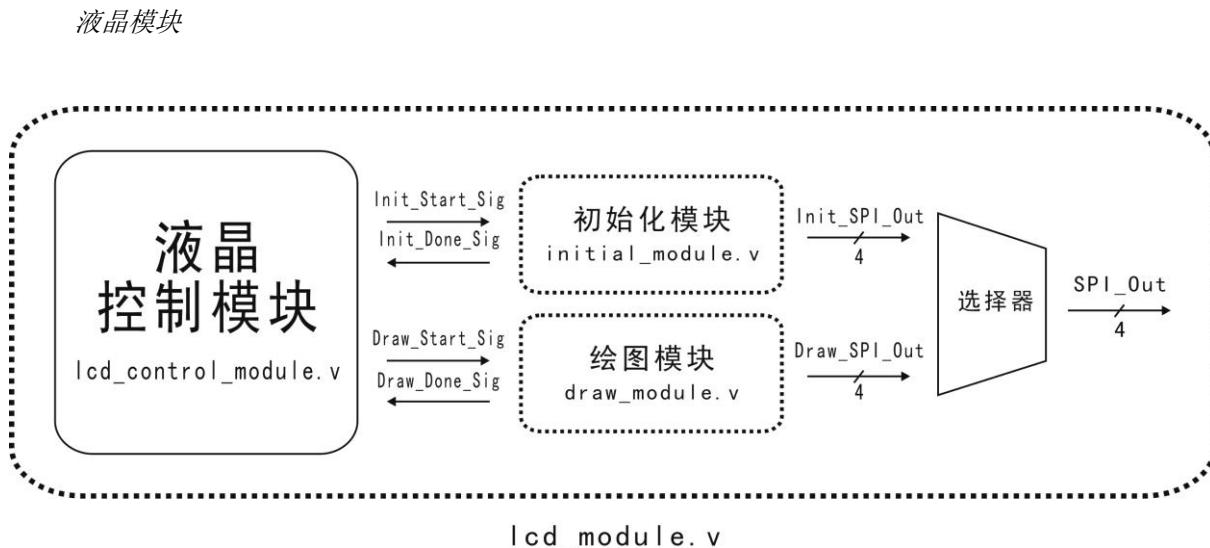
#### *draw\_module.v*

```
1. module draw_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     SPI_Out,
6.     Done_Sig
7. );
8.
9.     input CLK;
10.    input RSTn;
11.    input Start_Sig;
```

```
12.      output [3:0]SPI_Out;
13.      output Done_Sig;
14.
15.      *****/
16.
17.      wire SPI_Start_Sig;
18.      wire [9:0]SPI_Data;
19.      wire [9:0]Rom_Addr;
20.
21.      draw_control_module U1
22.      (
23.          .CLK( CLK ),
24.          .RSTn( RSTn ),
25.          .Start_Sig( Start_Sig ),           // input - from top
26.          .Draw_Data( Draw_Data ),         // input - from U2
27.          .SPI_Done_Sig( SPI_Done_Sig ),   // input - from U3
28.          .SPI_Start_Sig( SPI_Start_Sig ), // output - to U3
29.          .SPI_Data( SPI_Data ),          // output - to U3
30.          .Rom_Addr( Rom_Addr ),          // output - to U2
31.          .Done_Sig( Done_Sig )           // output - to top
32.      );
33.
34.      *****/
35.
36.      wire [7:0]Draw_Data;
37.
38.      pika_rom_module U2
39.      (
40.          .clock( CLK ),
41.          .address( Rom_Addr ),        // input - from U1
42.          .q( Draw_Data )             // output - to U1
43.      );
44.
45.      *****/
46.
47.      wire SPI_Done_Sig;
48.
49.      spi_write_module U3
50.      (
51.          .CLK( CLK ),
52.          .RSTn( RSTn ),
53.          .Start_Sig( SPI_Start_Sig ), // input - from U1
54.          .SPI_Data( SPI_Data ),     // input - from U1
55.          .Done_Sig( SPI_Done_Sig ), // output - to U1
```

```
56.         .SPI_Out( SPI_Out )          // output - to tp[  
57.     );  
58.  
59.     /*****  
60.  
61. endmodule
```

Draw\_module.v 是一个组合模块，具体上和“图形”一样。



上图中的 lcd\_module.v 是已经完成的组合模块。Lcd\_control\_module.v 顾名思义就是控制液晶操作的控制模块。我们来看一下该模块与“顺序操作”的关系。

```
main()  
{  
    Initial_Function();  
    Draw_Function();  
  
    while(1);  
}
```

在主函数中，先调用 Initial\_Function()，然后再调用 Draw\_Function()。然而液晶控制模块的操作却是如此，先使能 initial\_module.v 然后再使能 draw\_module.v，最后保持沉默。

无论是 initial\_module.v 或者 draw\_module.v 它们各自都拥有 spi\_write\_module.v 。为了协调它们共享输出资源，选择器是必须的（至于选择器与 4-1 章的一样）。

整个组合模块比较简单，只要照“顺序操作”的思路，就能理解它们。

*lcd\_control\_module.v*

```
1. module lcd_control_module
2. (
3.     CLK, RSTn,
4.     Init_Done_Sig, Draw_Done_Sig,
5.     Init_Start_Sig, Draw_Start_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input Init_Done_Sig, Draw_Done_Sig;
11.    output Init_Start_Sig, Draw_Start_Sig;
12.
13.    /***** */
14.
15.    reg [3:0]i;
16.    reg isInit;
17.    reg isDraw;
18.
19.    always @ ( posedge CLK or negedge RSTn )
20.        if( !RSTn )
21.            begin
22.                i <= 4'd0;
23.                isInit <= 1'b0;
24.                isDraw <= 1'b0;
25.            end
26.        else
27.            case( i )
28.
29.                4'd0:
30.                    if( Init_Done_Sig ) begin isInit <= 1'b0; i <= i + 1'b1; end
31.                    else isInit <= 1'b1;
32.
33.                4'd1:
34.                    if( Draw_Done_Sig ) begin isDraw <= 1'b0; i <= i + 1'b1; end
35.                    else isDraw <= 1'b1;
36.
37.                4'd2:
38.                    i <= 4'd2;
39.
```

```
40.         endcase
41.
42.     /*****
43.
44.     assign Init_Start_Sig = isInit;
45.     assign Draw_Start_Sig = isDraw;
46.
47.     *****/
48.
49. endmodule
```

在 16~17 行定义了使能初始化模块和绘图模块的标志寄存器 isInit 和 isDraw。在 27~40 行是该控制模块的核心部分。一开始先使能初始化模块 (29~31 行)，当初始化完成后 (30 行)，便使能绘图模块 (33~35 行)。绘图完成后 (34 行)，便停止 (37~38 行)。

*lcd\_module.v*

```
1. module lcd_module
2. (
3.     CLK, RSTn,
4.     SPI_Out
5. );
6.
7.     input CLK;
8.     input RSTn;
9.     output [3:0]SPI_Out;
10.
11.    *****/
12.
13.    wire Init_Start_Sig;
14.    wire Draw_Start_Sig;
15.
16.    lcd_control_module U1
17.    (
18.        .CLK( CLK ),
19.        .RSTn( RSTn ),
20.        .Init_Done_Sig( Init_Done_Sig ),      // input - from U2
21.        .Draw_Done_Sig( Draw_Done_Sig ),    // input - from U3
22.        .Init_Start_Sig( Init_Start_Sig ),    // output - to U2
23.        .Draw_Start_Sig( Draw_Start_Sig )     // output - to U3
24.    );
25.
```

```

26. *****/
27.
28. wire Init_Done_Sig;
29. wire [3:0]Init_SPI_Out;
30.
31. initial_module U2
32. (
33.     .CLK( CLK ),
34.     .RSTn( RSTn ),
35.     .Start_Sig( Init_Start_Sig ), // input - from U1
36.     .Done_Sig( Init_Done_Sig ), // output - to U1
37.     .SPI_Out( Init_SPI_Out ) // output - to selector
38. );
39.
40. *****/
41.
42. wire Draw_Done_Sig;
43. wire [3:0]Draw_SPI_Out;
44.
45. draw_module U3
46. (
47.     .CLK( CLK ),
48.     .RSTn( RSTn ),
49.     .Start_Sig( Draw_Start_Sig ), // input - from U1
50.     .Done_Sig( Draw_Done_Sig ), // output - to U1
51.     .SPI_Out( Draw_SPI_Out ) // output - to selector
52. );
53.
54. *****/
55.
56. reg [3:0]SPI_Out;
57.
58. always @ ( * )
59.     if( Init_Start_Sig ) SPI_Out = Init_SPI_Out; // drive by U2
60.     else if( Draw_Start_Sig ) SPI_Out = Draw_SPI_Out; // drive by U3
61.     else SPI_Out <= 3'bx;
62.
63. *****/
64.
65. endmodule

```

lcd\_module.v 组合模块基本上和“图形”一样，自己看着办吧。

实验十二说明:

这个实验是模仿“顺序操作”来驱动液晶资源。Initial\_module.v 组合模块，包含了 initial\_control\_module.v 和 spi\_write\_module.v；initial\_control\_module.v 控制初始化的次序，spi\_write\_module 将数据以 SPI 模式输出。

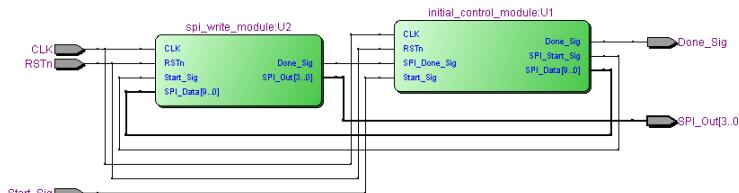
draw\_module.v 组合模块包含了 draw\_control\_module.v，pika\_rom\_module.v 和 spi\_write\_module.v。Rom 模块储存了 64 x 128 的图片信息，控制模块控制了显示图片的步骤，最后 SPI 发送模块将数据以 SPI 模式输出。

lcd\_module.v 同时拥有 initial\_module.v 和 draw\_module.v 这两个组合模块，然后利用自身的 lcd\_control\_module.v 去控制“先使能 initial\_module.v 后使能 draw\_module.v”。

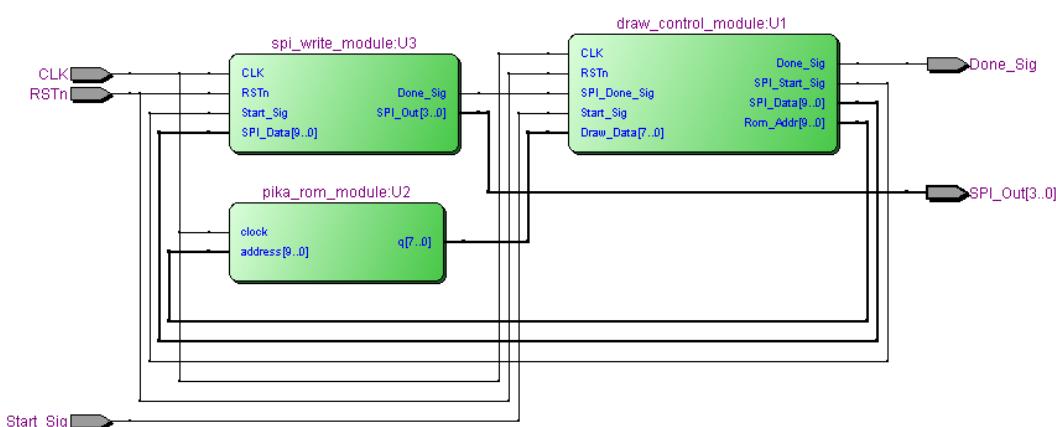
宏观上，lcd\_module.v 都是由一层一层不同功能的组合模块组合而成，实际上它却是按照“顺序操作”的执行步骤建模而成。

“仿顺序操作”始终有一个避免不了的问题，那就是“两仪性或者多义性”，initial\_module.v 或者 draw\_module.v 它们自身都拥有 spi\_write\_module.v，两个模块不可能同时拥有一样的输出资源，这时候就需要选择器的帮助。

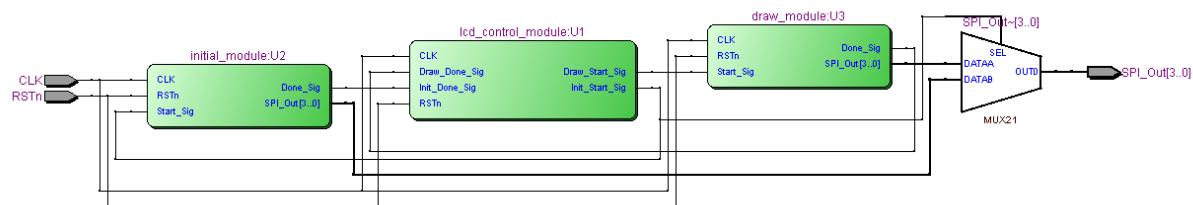
完成后的扩展图：



initial\_module.v



draw\_module.v



lcd\_module.v

### 实验十二结论：

在宏观上“**仿顺序操作**”是利用建模模仿“**顺序操作**”。在微观上“**仿顺序操作**”是利用Verilog HDL语言本身的特性去模仿“**顺序操作**”。在上述的内容中，“**顺序操作**”语言无法顾及spi传输的小细节，但是Verilog HDL语言却顾及到这些小细节。

最后我们来讨论一个问题：

网络上初学者常常误会“Verilog HDL语言是硬件描述语言的关系，所以代码是不可以精简...”Verilog HDL语言的代码量是多是少不是本质的问题，反之如何维护代码的结构和风格才是本质的问题。如果自己写的代码精简但是不在乎代码的结构和风格，结果这东西估计只有自己受用而已，因为只有自己看得懂，别人却看不懂！（这种感受，估计很多新手都尝受过...）

## 4.3 命令式的仿顺序操作

什么是命令式的仿顺序操作！？在明白这东西之前，我们先看一个简单的例子：  
假设笔者要建立 可以产生 SSS, SOS, OSO, 000 这四种模块。如果模仿 C 语言，函数的写法会是如下：

```
//基础函数
S_Function(){...}
O_Function(){...}

//基于基础函数创建的函数
SSS_Function()
{
    S_Function(); S_Function(); S_Function();
}

SOS_Fucntion()
{
    S_Function(); O_Function(); S_Function();
}

OSO_Fucntion()
{
    O_Function(); S_Function(); O_Function();
}

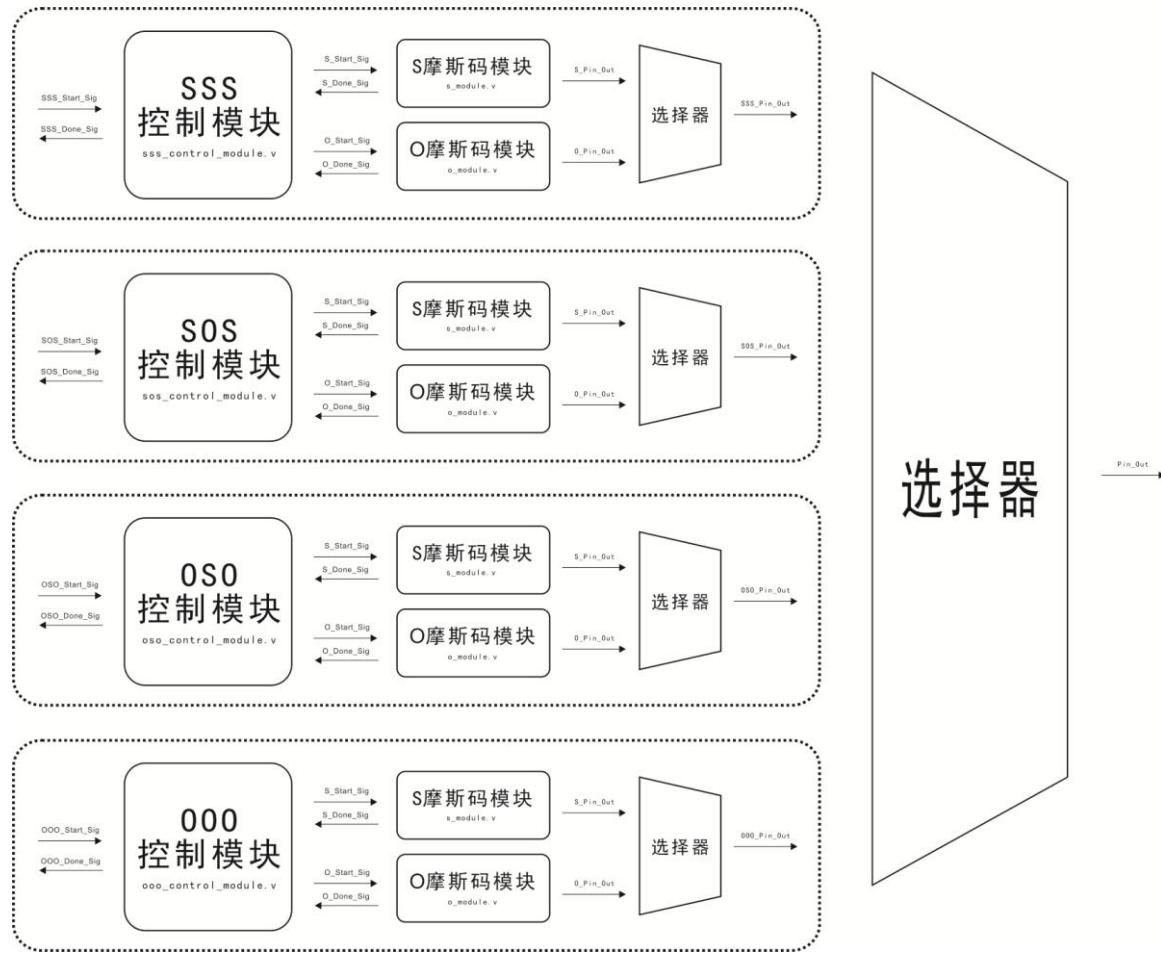
OOO_Fucntion()
{
    O_Function(); O_Function(); O_Function();
}
```

我们会很自然的，以 S\_Function() 和 O\_Fucntion() 为基础，再建立出四个新的 SOS\_Function(), SSS\_Function(), OSO\_Function(), 和 OOO\_Fucntion()。在顺序语言上（如 C 语言），这样的方法当然，没有问题，但是在 Verilog HDL 语言上呢？

笔者说过“**仿顺序操作**”归根究底不是单纯模仿顺序操作而已，而是利用 Verilog HDL 语言本身的特质，去模仿顺序操作。4.1 章~4.2 章的实验，所使用的方法最大限度只是可以支持“少数函数”的“仿顺序操作”而已。

我们来看看，如果以 4.1 章~4.2 章为基础，来实现上述内容的话，会是什么样一个结果。

---



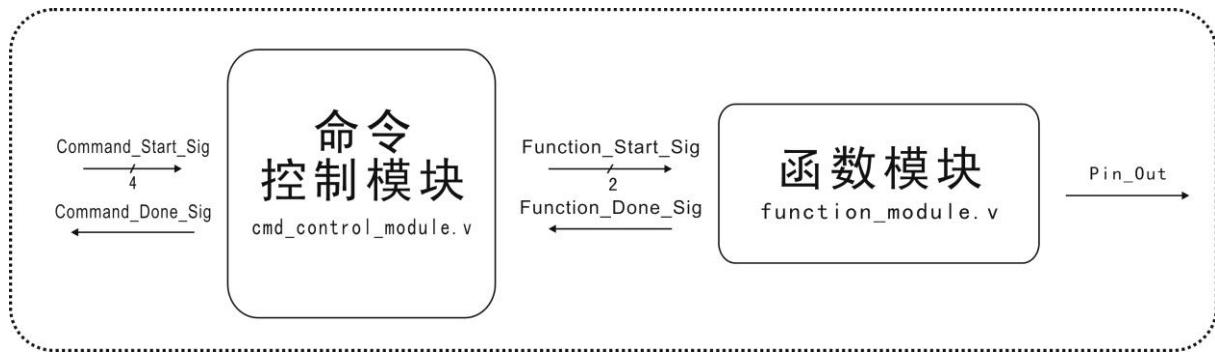
上图够可怕吧！上图包含了 SSS 组合模块，SOS 组合模块，OSO 组合模块和 OOO 模块。然而每一个组合模块都包含各自的“**控制模块**”，“**S 摆斯码模块**”，“**O 摆斯码模块**”和“**选择器**”。最后，每一个组合模块的输出，还需要一个输出选择器来协调操作。

我们从另一个方面来分析它的缺点：

- (一) 模块的重复，资源的消耗。
- (二) 建模量多，连线设计繁多。
- (三) 模块调用的难度。

如果读者有笔者这样的耐性，这样的劳动当然是没有问题啦。但是，实际上笔者也觉得这样的建模方法非常“猥琐”，而且模块的连线也很困难。所以我们需要另一种“仿顺序操作”建模的方法，毕竟 4-1 章 和 4-2 章的方法只适合小型的“仿顺序操作”而已。**这个方法既是“命令式的仿顺序操作”。**

何谓“命令式的仿顺序操作？”我们来看看下面的一张图，就可以知道个大概：



如上图！结果我们可以把上述的建模精简到这样的程度。在这里笔者把最基本的产生 S 摩斯密码和 O 摩斯码的功能集合在函数模块中。然后模仿高级函数如 SSS ... SOS 等，则可以利用命令控制模块，根据 SSS ... SOS 的执行步骤去控制（驱动 Function\_Start\_Sig）函数模块。最后如果要实现 SSS ... SOS 等功能，只要根据命令控制模块的命令（Command\_Start\_Sig），去调它即可。

（嗯 ... 还是直接看代码比较直接。）

*function\_module.v*

```
module function_module
(
    .....
    Function_Start_Sig,
    Function_Done_Sig,
    Pin_Out
);
    .....
    input [1:0]Function_Start_Sig;
    input Function_Done_Sig;
    output Pin_Out;

    /***** //定时器和延时器 *****/
    //定时器和延时器
    .....
    /***** //定时器和延时器 *****/

```

```

reg [3:0]i;
reg rPin_Out;
reg isDone;
......

always @ ( posedge or CLK or negedge RSTn )
if( !RSTn )
begin
    i <= 4'd0;
    rPin_Out <= 1'b0;
    isDone <= 1'b0;
    .....
end
====> else if( Function_Start_Sig[1] )
case( i )

```

// S 摩斯码产生

.....

```

4'd 9:
begin isDone <= 1'b1; i <= i + 1'b1; end

```

```

4'd10:
begin isDone <= 1'b0; i <= 4'd0;

```

endcase

```

====> else if( Function_Start_Sig[0] )
case( i )

```

// 0 摩斯码产生

.....

```

4'd 9:
begin isDone <= 1'b1; i <= i + 1'b1; end

```

```

4'd10:
begin isDone <= 1'b0; i <= 4'd0;

```

endcase

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

```
assign Function_Done_Sig = isDone;
assign Pin_Out = rPin_Out;

/***********************/

endmodule
```

function\_module.v 关键的部分是 Function\_Start\_Sig 的位宽和 else if 部分。  
Function\_Start\_Sig 的每一位“位宽”都代表不同的“Start\_Sig”。

Function_Start_Sig[ 1..0 ]	
位命令	功能
10	S 莫斯码产生
01	O 摩斯码产生

然而“Done\_Sig”和以往一样，没有任何变化。假设我要产生 S 模式码，那么我只要往 Function\_Start\_Sig 输入 2'b10 即可。

*cmd\_control\_module.v*

```
module cmd_control_module
(
    .....
    Command_Start_Sig,
    Command_Done_Sig,
    Function_Start_Sig,
    Function_Done_Sig
);

    input [3:0]Command_Start_Sig;
    output Command_Done_Sig;
    output [1:0]Function_Start_Sig;
    input Function_Done_Sig;

    /*****



    reg [3:0]i;
    reg [1:0]isStart;
    reg isDone;

    always @ ( posedge CLK or negedge RSTn )
```

```

if( !RSTn )
begin
    i <= 4'd0;
    isStart <= 2'b00;
    isDone <= 1'b0;
end
==> else if( Start_Sig[3] ) // 产生 SSS
case( i )

    4'd0, 4'd1, 4'd2 :
if( Fucntion_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b10;

    4'd3:
begin isDone <= 1'b1; i <= i + 1'b1; end

    4'd4:
begin isDone <= 1'b0; i <= i + 1'b1; end

endcase
==> else if( Start_Sig[2] ) // 产生 SOS
case( i )

    4'd0, 4'd2 :
if( Fucntion_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b10;

    4'd1 :
if( Fucntion_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b01;

    4'd3:
begin isDone <= 1'b1; i <= i + 1'b1; end

    4'd4:
begin isDone <= 1'b0; i <= i + 1'b1; end

endcase
==> else if( Start_Sig[1] ) // 产生 OSO
case( i )

    4'd0, 4'd2 :
if( Fucntion_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b01;

```

```

4'd1 :
if( Function_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b10;

4'd3:
begin isDone <= 1'b1; i <= i + 1'b1; end

4'd4:
begin isDone <= 1'b0; i <= i + 1'b1; end

endcase
==> else if( Start_Sig[1] ) // 产生 OOO
    case( i )

4'd0, 4'd1, 4'd2 :
if( Function_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
else isStart <= 2'b01;

4'd3:
begin isDone <= 1'b1; i <= i + 1'b1; end

4'd4:
begin isDone <= 1'b0; i <= i + 1'b1; end

endcase

/***********************/

assign Function_Start_Sig = isStart;
assign Command_Done_Sig = isDone;

/***********************/

endmodule

```

和 function\_module.v 一样 command\_control\_module 关键的部分是 Command\_Start\_Sig 的“位宽”和 else if 部分，位分配如下：

Command_Start_Sig[ 3..0 ]	
位命令	功能
1000	产生 SSS
0100	产生 SOS

0010	产生 OSO
0001	产生 OOO

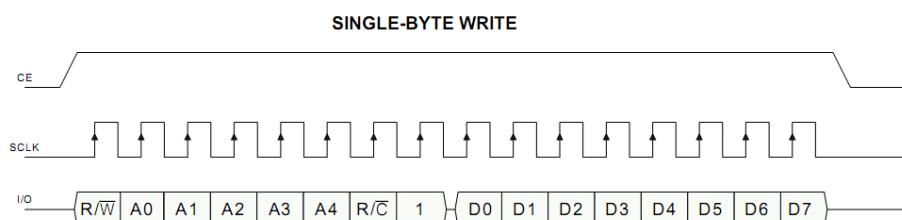
假设笔者输入 Command\_Start\_Sig 是 0100,

- 一、对 function\_module.v 输入 2'b10，产生 S 摩斯码，返回 Function\_Done\_Sig。
- 二、对 function\_module.v 输入 2'b01，产生 O 摩斯码，返回 Function\_Done\_Sig。
- 三、对 function\_module.v 输入 2'b10，产生 S 摩斯码，返回 Function\_Done\_Sig。
- 四、返回 Command\_Done\_Sig。

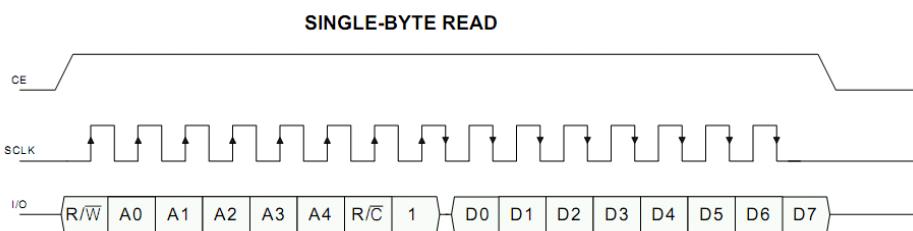
“命令式仿顺序操作”的基本思路就那么简单，接下来我们以一个实验来说明。

### 实验十三：DS1302 实时时钟驱动

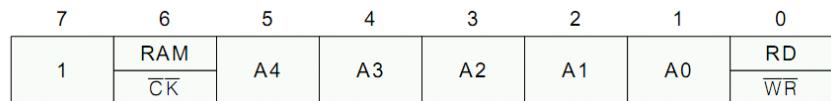
虽说 DS1302 相关的资料都是在网络满天飞，在这里笔者还是介绍一些重点。实时时钟芯片，大家应该明白是什么吧，就是一种控制时钟的芯片。一旦初始化后，它就会随着现实的时钟一直计数。要明白 DS1302 芯片最主要的关键，就是“[传输时序](#)”和“[芯片本身的寄存器分配](#)”。



上图是 DS1302 芯片写操作的时序图。第一个字节是“[访问寄存器的地址](#)”，第二个字节是“[写数据](#)”。在写操作的时候，都是“[上升沿有效](#)”，然而还有一个条件，就是 CE (/RST) 信号必须拉高。（数据都是从 LSB 开始发送，亦即是最低位开始至最高位结束）



上图是 DS1302 芯片读操作的时序图。基本上和写操作的时序图大同小异，区别的地方就是在第二个字节时“[读数据](#)”的动作。第二字节读数据开始时，SCLK 信号都是“[下降沿有效](#)”。嗯，别忘了 CE (/RST) 信号同样是必须拉高。（第一节数据是从 LSB 开始输出，第二节数据是从 LSB 开始读入）



无论是读操作还是写操作，在时序图中，第一个字节都是“访问寄存器的地址”，然而这一字节数据有自己的格式。

BIT 7 固定。

BIT 6 表示是访问寄存器本身，还是访问 RAM 空间。

BIT 5..1 表示是寄存器|RAM 空间的地址。

BIT 0 表示是访问寄存器本身是写操作，还是读操作。

REGISTER ADDRESS	REGISTER DEFINITION
<b>A. CLOCK</b>	
SEC	00-59 CH 10 SEC SEC
MIN	00-59 0 10 MIN MIN
HR	01-12 12/ 00-23 24 0 10 A/P HR HR
DATE	01-28/29 01-30 0 0 10 DATE DATE 01-31
MONTH	01-12 0 0 0 10 M MONTH
DAY	01-07 0 0 0 0 0 DAY
YEAR	00-99 10 YEAR YEAR
CONTROL	WP 0 0 0 0 0 0 0
TRICKLE CHARGER	TCS TCS TCS TCS DS DS RS RS
CLOCK BURST	1 0 1 1 1 1 1 RD W

上图是寄存器地址的全家福。笔者顺便强调一下，Verilog HDL 语言有的是很强的位操作，“访问寄存器的地址”可以这样表示：

```
{ 2'b10 , 5'd Addr, 1'b RD/W }
```

(这样就可以提高一些解读性)我们知道 BIT 7 是固定的位, 然而 BIT 6 表示“[访问 RAM 空间还是访问寄存器](#)”。在寄存器地址的全家福中, BIT 6 都是清一色的为“[逻辑 0](#)”。

假设要写秒寄存器, 那么笔者可以这样输入:

```
{ 2'b10, 5'd0, 1'b0 }
```

再假设我要读秒寄存器, 那么笔者可以这样输入:

```
{ 2'b10, 5'd0, 1'b1 }
```

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	RANGE
CH		10 Seconds		Seconds				00-59
		10 Minutes		Minutes				00-59
12/24	0	10 AM/PM	Hour	Hour				1-12/0-23
0	0	10 Date		Date				1-31
0	0	0	10 Month	Month				1-12
0	0	0	0	0	Day			1-7
10 Year				Year				00-99
WP	0	0	0	0	0	0	0	—
TCS	TCS	TCS	TCS	DS	DS	RS	RS	—

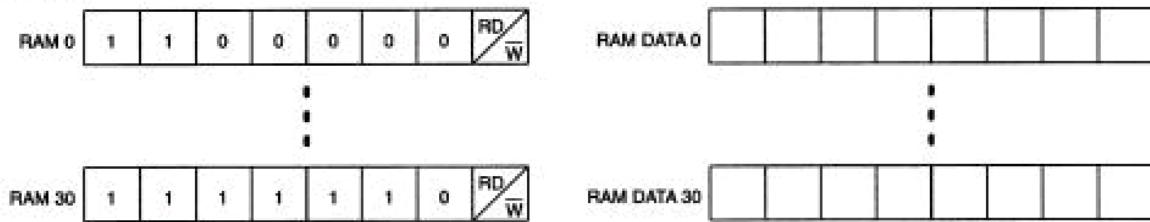
上图表达了每一个寄存器的字节配置。秒寄存器(第一个), 前高四位(BIT7 除外), 表示“[秒的十位](#)”, 低四位表示“[秒的个位](#)”。其他的寄存器的字节配置也是如此。但是有3个寄存器比较特别, 那就是“[秒寄存器](#)”, “[时寄存器](#)”, “[控制寄存器](#)”(最后两个)。

[秒寄存器的最高位\(BIT7\)](#), 如果写入“[逻辑 0](#)” DS1302 芯片就开始启动, 反之就关闭。

时寄存器的最高位(BIT7), 表示了“[逻辑 1 是 12 小时进制](#)”, “[逻辑 0 是 24 小时进制](#)”, 笔者认为, 还是 24 小时进制比较方便工作。

控制寄存器的最高位(BIT7), 如果写入“[逻辑 0](#)”表示关闭写保护, 写入“[逻辑 1](#)”表示打开写保护。所以呀, 每当要变更寄存器的内容之前, 就要关闭写保护。

## B. RAM



上图是 RAM 的全家福。RAM 的空间有  $2^5 - 2 = 0 \sim 30$ ，亦即 31 words x 8 bits 的空间。由于是访问 RAM，所以“访问寄存器的地址”的 BIT6 必须是逻辑 1。RAM 地址的范围如下：

```
{ 2'b11, 5'd0, 1'b RD/W } ~ { 2'b11, 5'd30, 1'b RD/W }
```

=====

接下来是驱动 DS1302 的简单操作概念：

如果笔者要关闭写保护，那么笔者需要的操作如下：

```
先写第一节地址 - { 2'b10, 5'd 7, 1'b0 }
后写第二节数据 - { 8'h00 }
```

如果笔者要在时寄存器写入 10 十进制( 如果以 24 小时进制 )，那么笔者需要如下的操作：

```
先写第一节地址 - { 2'b10, 5'd2, 1'b0 }
后写第二节数据 - { 4'h1, 4'h0 }
```

如果笔者要在分寄存器写入 20 十进制，那么笔者需要的操作如下：

```
先写第一节地址 - { 2'b10, 5'd1, 1'b0 }
后写第二节数据 - { 4'h2, 4'h0 }
```

如果笔者要在秒寄存器写入 33 十进制，那么笔者需要的操作如下：

```
先写第一节地址 - { 2'b10, 5'd0, 1'b0 }
后写第二节数据 - { 4'h3, 4'h3 }
```

( 在这里我们知道，秒寄存器的最高位，控制着 DS1302 芯片的启动和关闭，所以秒寄存器的配置都是留在最后才操作 )

如果我要在地址 20, RAM 空间写入 0xff 的数据，那么笔者需要如下的操作：

先写第一节地址 - { 2'b11, 5'd 20, 1'b0 }

后写第二节数据 - { 8'hff }

如果笔者要在时寄存器读出的“**十位和个位**”(如果以 24 小时进制), 那么笔者需要如下的操作:

先写第一节地址 - { 2'b10, 5'd2, 1'b1 }

后读第二节数据 - { 4'h 读出时十位, 4'h 读出时个位 }

如果笔者要在秒寄存器读出“**十位和个位**”, 那么笔者需要的操作如下:

先写第一节地址 - { 2'b10, 5'd1, 1'b1 }

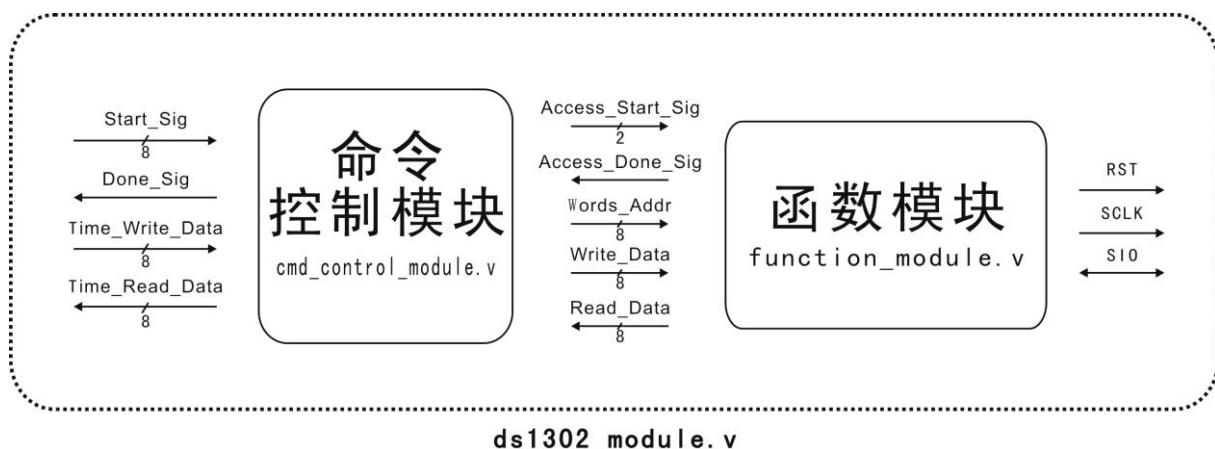
后读第二节数据 - { 4'h 读出秒十位, 4'h 读出秒个位 }

如果笔者要在地址 20, RAM 空间读出数据, 那么笔者需要如下的操作:

先写第一节地址 - { 2'b11, 5'd 20, 1'b1 }

后读第二节数据 - { 8'h 读出数据 }

*ds1302\_module.v*



上图是我们要建立的组合模块 *ds1302\_module.v*。首先我们把焦点放在 *function\_module.v*。我们知道如果以“**命令式仿顺序操作**”, 函数模块, 必须包含“**两个最基本的函数**”, 亦即“**写字节函数**”和“**读字节函数**”。

在上图我们可见 函数模块 的开始信号 *Access\_Start\_Sig* 的位宽有两位, 它们分别是:

Access_Start_Sig [ 1..0 ]	
位命令	功能

10	写字节操作
01	读字节操作

此外，还有由上层模块输入的 Words\_Addr 和 Write\_Data，亦即“[写一字节操作](#)”中所要求的“[第一字节](#)”和“[第二字节](#)”数据。Read\_Data 和 Access\_Done\_Sig 分别是返回的“读出数据”和“完成信号”。

具体的操作，我们还是直接看代码吧！

*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.
14.     input CLK;
15.     input RSTn;
16.     input [1:0]Start_Sig;
17.     input [7:0]Words_Addr;
18.     input [7:0]Write_Data;
19.     output [7:0]Read_Data;
20.     output Done_Sig;
21.     output RST;
22.     output SCLK;
23.     inout SIO;
24.
25.     ****
26.
27. parameter T0P5US = 5'd24;//0.5us,50M*0.5us-1=24
28.
29.     ****
30.
31. reg [4:0]Count1;
```

```
32.  
33.    always @ ( posedge CLK or negedge RSTn )  
34.        if( !RSTn )  
35.            Count1 <= 5'd0;  
36.        else if( Count1 == T0P5US )  
37.            Count1 <= 5'd0;  
38.        else if( Start_Sig[0] == 1'b1 || Start_Sig[1] == 1'b1 )  
39.            Count1 <= Count1 + 1'b1;  
40.        else  
41.            Count1 <= 5'd0;  
42.  
43.    /*****  
44.  
45.    reg [5:0]i;  
46.    reg [7:0]rData;  
47.    reg rSCLK;  
48.    reg rRST;  
49.    reg rSIO;  
50.    reg isOut;  
51.    reg isDone;  
52.  
53.    always @ ( posedge CLK or negedge RSTn )  
54.        if( !RSTn )  
55.            begin  
56.                i <= 6'd0;  
57.                rData <= 8'd0;  
58.                rSCLK <= 1'b0;  
59.                rRST <= 1'b0;  
60.                rSIO <= 1'b0;  
61.                isOut <= 1'b0;  
62.                isDone <= 1'b0;  
63.            end  
64.        else if( Start_Sig[1] )  
65.            case( i )  
66.  
67.                0 :  
68.                    begin rSCLK <= 1'b0; rData <= Words_Addr; rRST <= 1'b1;isOut <= 1'b1; i <= i + 1'b1; end  
69.  
70.                1, 3, 5, 7, 9, 11, 13, 15 :  
71.                    if( Count1 == T0P5US ) i <= i + 1'b1;  
72.                    else begin rSIO <= rData[ (i >> 1) ]; rSCLK <= 1'b0; end  
73.  
74.                2, 4, 6, 8, 10, 12, 14, 16 :  
75.                    if( Count1 == T0P5US ) i <= i + 1'b1;
```

## Verilog HDL 那些事儿 – 建模篇

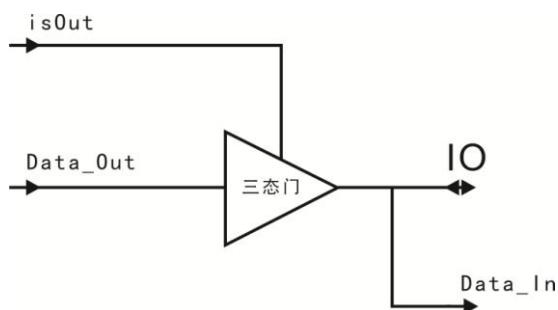
```
76.           else begin rSCLK <= 1'b1; end
77.
78.           17 :
79.           begin rData <= Write_Data; i <= i + 1'b1; end
80.
81.           18, 20, 22, 24, 26, 28, 30, 32 :
82.           if( Count1 == T0P5US ) i <= i + 1'b1;
83.           else begin rSIO <= rData[ (i >> 1) - 9 ]; rSCLK <= 1'b0; end
84.
85.           19, 21, 23, 25, 27, 29, 31, 33 :
86.           if( Count1 == T0P5US ) i <= i + 1'b1;
87.           else begin rSCLK <= 1'b1; end
88.
89.           34 :
90.           begin rRST <= 1'b0; i <= i + 1'b1; end
91.
92.           35 :
93.           begin isDone <= 1'b1; i <= i + 1'b1; end
94.
95.           36 :
96.           begin isDone <= 1'b0; i <= 6'd0; end
97.
98.       endcase
99.   else if( Start_Sig[0] )
100.       case( i )
101.
102.           0 :
103.           begin rSCLK <= 1'b0; rData <= Words_Addr; rRST <= 1'b1; isOut <= 1'b1; i <= i + 1'b1; end
104.
105.           1, 3, 5, 7, 9, 11, 13, 15 :
106.           if( Count1 == T0P5US ) i <= i + 1'b1;
107.           else begin rSIO <= rData[ (i >> 1) ]; rSCLK <= 1'b0; end
108.
109.           2, 4, 6, 8, 10, 12, 14, 16 :
110.           if( Count1 == T0P5US ) i <= i + 1'b1;
111.           else begin rSCLK <= 1'b1; end
112.
113.           17 :
114.           begin isOut <= 1'b0; i <= i + 1'b1; end
115.
116.           18, 20, 22, 24, 26, 28, 30, 32 :
117.           if( Count1 == T0P5US ) i <= i + 1'b1;
118.           else begin rSCLK <= 1'b1; end
119.
```

```

120.           19, 21, 23, 25, 27, 29, 31, 33 :
121.           if( Count1 == T0P5US ) begin i <= i + 1'b1; end
122.           else begin rSCLK <= 1'b0; rData[ (i >> 1) - 9 ] <= SIO; end
123.           34 :
124.           begin rRST <= 1'b0; isOut <= 1'b1; i <= i + 1'b1; end
125.
126.
127.           35 :
128.           begin isDone <= 1'b1; i <= i + 1'b1; end
129.
130.           36 :
131.           begin isDone <= 1'b0; i <= 6'd0; end
132.
133.       endcase
134.
135.   *****/
136.
137.   assign Read_Data = rData;
138.   assign Done_Sig = isDone;
139.
140.   assign RST = rRST;
141.   assign SCLK = rSCLK;
142.   assign SIO = isOut ? rSIO : 1'bz;
143.
144.   *****/
145.
146. endmodule

```

第 1~23 行表示了该模块输入输出口，注意 SIO 是 IO 口（11 行）。说道 IO，



上图是一个 IO 的硬件设计。如果要使 IO 输出，这时候 `isOut` 必须拉高，同时间 `Data_Out` 的数据就会输出。如果要使 IO 为输入，这时候需要拉低 `isOut`，然而三态门会输出高阻态将“**输出**”载止，从 IO 口输入的数据就会经向 `Data_In`。如果使用 Verilog HDL 语言来表示：

```
assign IO = isOut ? Data_Out : 1'bz;
assign Data_In = IO;
```

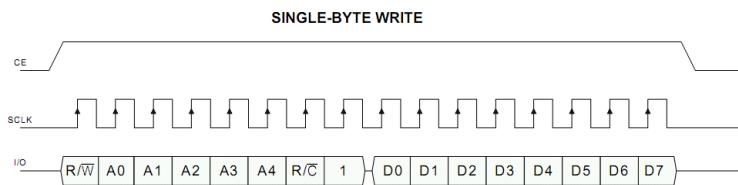
在 142 行，定义了 SDA 这个 IO 口，是由 isOut 这个寄存器控制着“**输入输出**”。当 isOut 为逻辑 1 时，该 IO 口是输出状态，反之是输入状态。然而 IO 的读取调用，可以直接在操作中读取（122 行）。当然我们也可以这样：

```
wire SDA_In;
assign SDA_In = SIO;
assign SIO = isOut ? rSIO : 1'bz;
```

然后在读取调用的地方，可以这样写，结果也是一样。

```
rData[ ( i >> 1 ) - 9 ] = SDA_In;
```

在 45~51 行定义了相关的寄存器，i 是指示着执行步骤，rData 用来暂存数据，rSCLK 用来驱动 SCLK，rRST 用来驱动 RST，rSIO 用来驱动 SIO 的输出，isOut 用来控制 IO 口的方向，最后的 isDone 是完成标志，亦即用来反馈完成信息。

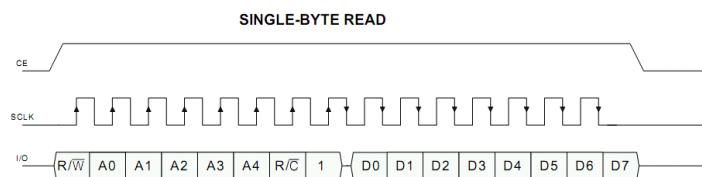


在 65~98 行是 Start\_Sig 为 2'b10 的时候，亦即是“**写字节操作**”。这段内容之中，代码完全是按照时序图执行。

在步骤 0 的时候，对 rData，rSCLK，rRST（CE），isOut 等寄存器进行初始化，**这一点很重要**（68 行）。

然后在步骤 1~16 之中，将“**第一个字节数据**”，亦即“**访问寄存器地址字节**”发送出去。传输规则和 SPI 有点相似，都是**时间下降沿设置数据，时间上升沿锁存数据**。（70~76 行）此外在步骤 17 将 rData 设置为“**第二个字节数据**”。（79 行）

然后重复如同步骤 1~16 那样（81~87 行），将“**第二个字节数据**”发送出去。在步骤 34，对 rRST（CE）拉低，以示“**写字节操作**”已经结束。最后在步骤 35~36 反馈完成信号。



在 99~133 行也是 Start\_Sig 为 2'b01 的时候，亦即是“**读字节操作**”。在读操作中，第一字节和第二字节数据显然，对时间沿的敏感不同。

在步骤 0 的时候，对 rData, rSCLK, rRST (CE), isOut 等寄存器进行初始化，**这一点很重要** (103 行)。

然后在步骤 1~16 之中，将“**第一个字节数据**”，亦即“**访问寄存器地址字节**”发送出去。这时的数据锁存发生在时间的上升沿。(105~111 行) 在步骤 17 对 IO 口的方向，改变为输入，亦即将 isOut 设置为逻辑 0。(114 行)

在步骤 18~33 之间是“**读取一个字节数据的操作**”，该动作时时间的下降沿，对 SIO 信后读取数据 (116~122 行)。在这里笔者再强调一下，DS1302 芯片，数据的传输都是从 LSB 开始到 MSB 结束。

最后在步骤 35 对 rRST 的拉低，以示“**读字节数据**”操作已经结束。然后恢复 IO 口为输出，亦即拉高 isOut 寄存器 (125 行)，然后产生一个完成信号 (127~131 行)。

在 122 行，从 DS1302 芯片读取的数据会暂存在 rData 这个寄存器，然后该寄存器会驱动 Read\_Data 这个信号线 (137 行)。

#### *cmd\_control\_module.v*

接下来我们要探讨的就是 cmd\_control\_module.v。从“**图形**”看来，cmd\_control\_module.v 是 function\_module.v 的调用模块。从顺序操作上看来 cmd\_control\_module.v 的功能如下（以下只是伪代码，希望读者不要太认真，为了给读者一个感知的认识）：

```
Function_Module( Command, Addr, Data )
{
    case( Command )
    {
        2'b10 : Write_Function( Addr, Data );           // Write operation
        2'b11 : Read_Fucntion( Addr ) { return Data; } // Read operation
    }
}

CMD_Control_Module( Command )
{
    case( Command )
    {
        8'b10000000 : Function_module( 2'b10, {2'b10, 5'd0, 1'b0}, 8'h00 ); // Unprotect
        8'b01000000 : Function_module( 2'b10, {2'b10, 5'd2, 1'b0}, 8'h00 ); // Write hour
        8'b00100000 : Function_module( 2'b10, {2'b10, 5'd1, 1'b0}, 8'h00 ); // Write minit
    }
}
```

## Verilog HDL 那些事儿 – 建模篇

```
8'b00010000 : Function_module( 2'b10, {2'b10,5'd0,1'b0}, 8'h00 ); // Write second  
8'b00001000 : Function_module( 2'b10, {2'b10,5'd0,1'b0}, 8'h80 ); // Protect  
8'b00000100 : Function_module( 2'b01, {2'b10,5'd2,1'b1} ); // Read hour  
8'b00000010 : Function_module( 2'b01, {2'b10,5'd1,1'b1} ); // Read minit  
8'b00000001 : Function_module( 2'b01, {2'b10,5'd0,1'b1} ); // Read second  
}  
}
```

从上面的伪代码看来 CMD\_Control\_Module 反应出 cmd\_control\_module 是利用 Start\_Sig 的 8 位位宽来定义 8 中不同的操作。而且在这 8 个不同的操作之中，都对 function\_module.v 都有不同的操作。

(位宽对命令分配如下)

Start_Sig[ 7..0 ]	
位命令	功能
1000_0000	关闭写保护
0100_0000	变更时寄存器
0010_0000	变更分寄存器
0001_0000	变更秒寄存器
0000_1000	开启写保护
0000_0100	读取时寄存器
0000_0010	读取分寄存器
0000_0001	读取秒寄存器

```
1. module cmd_control_module  
2. (  
3.     CLK, RSTn,  
4.  
5.     Start_Sig,  
6.     Done_Sig,  
7.  
8.     Time_Write_Data,  
9.     Time_Read_Data,  
10.  
11.    Access_Done_Sig,  
12.    Access_Start_Sig,  
13.  
14.    Read_Data,  
15.    Words_Addr,  
16.    Write_Data  
17.  
18. );
```

```
19.
20.     input CLK;
21.     input RSTn;
22.
23.     input [7:0]Start_Sig;
24.     output Done_Sig;
25.
26.     input [7:0]Time_Write_Data;
27.     output [7:0]Time_Read_Data;
28.
29.     input Access_Done_Sig;
30.     output [1:0]Access_Start_Sig;
31.
32.     input [7:0]Read_Data;
33.     output [7:0]Words_Addr;
34.     output [7:0]Write_Data;
35.
36.     *****/
37.
38.     reg [7:0]rAddr;
39.     reg [7:0]rData;
40.
41.     always @ ( posedge CLK or negedge RSTn )
42.         if( !RSTn )
43.             begin
44.                 rAddr <= 8'd0;
45.                 rData <= 8'd0;
46.             end
47.         else
48.             case( Start_Sig[7:0] )
49.
50.                 8'b1000_0000 : // Write unprotect
51.                 begin rAddr <= { 2'b10, 5'd7, 1'b0 }; rData <= 8'h00; end
52.
53.                 8'b0100_0000 : // Write hour
54.                 begin rAddr <= { 2'b10, 5'd2, 1'b0 }; rData <= Time_Write_Data; end
55.
56.                 8'b0010_0000 : // Write minit
57.                 begin rAddr <= { 2'b10, 5'd1, 1'b0 }; rData <= Time_Write_Data; end
58.
59.                 8'b0001_0000 : // Write second
60.                 begin rAddr <= { 2'b10, 5'd0, 1'b0 }; rData <= Time_Write_Data; end
61.
62.                 8'b0000_1000 : // Write protect
```

## Verilog HDL 那些事儿 – 建模篇

```
63.          begin rAddr <= { 2'b10, 5'd7, 1'b0 }; rData <= 8'b1000_0000; end
64.
65.          8'b0000_0100 : // Read hour
66.          begin rAddr <= { 2'b10, 5'd2, 1'b1 };end
67.
68.          8'b0000_0010 : // Read minit
69.          begin rAddr <= { 2'b10, 5'd1, 1'b1 };end
70.
71.          8'b0000_0001 : // Read second
72.          begin rAddr <= { 2'b10, 5'd0, 1'b1 };end
73.
74.      endcase
75.
76.  *****/
77.
78. reg [1:0]i;
79. reg [7:0]rRead;
80. reg [1:0]isStart;
81. reg isDone;
82.
83. always @ ( posedge CLK or negedge RSTn )
84.     if( !RSTn )
85.         begin
86.             i <= 2'd0;
87.             rRead <= 8'd0;
88.             isStart <= 2'b00;
89.             isDone <= 1'b0;
90.         end
91.     else if( Start_Sig[7:3] ) // Write action
92.         case( i )
93.
94.             0 :
95.                 if( Access_Done_Sig ) begin isStart <= 2'b00; i <= i + 1'b1; end
96.                 else begin isStart <= 2'b10; end
97.
98.             1 :
99.                 begin isDone <= 1'b1; i <= i + 1'b1; end
100.
101.            2 :
102.                 begin isDone <= 1'b0; i <= 2'd0; end
103.
104.         endcase
105.     else if( Start_Sig[2:0] ) // Read action
106.         case( i )
```

```

107.
108.          0 :
109.          if( Access_Done_Sig )begin rRead <= Read_Data; iStart <= 2'b00; i <= i + 1'b1;end
110.          else begin iStart <= 2'b01; end
111.
112.          1 :
113.          begin iDone <= 1'b1; i <= i + 1'b1; end
114.
115.          2 :
116.          begin iDone <= 1'b0; i <= 2'd0; end
117.
118.      endcase
119.
120.  *****/
121.
122. assign Done_Sig = iDone;
123.
124. assign Time_Read_Data = rRead;
125.
126. assign Access_Start_Sig = iStart;
127.
128. assign Words_Addr = rAddr;
129. assign Write_Data = rData;
130.
131. *****/
132.
133. endmodule

```

第 3~34 行的接口定义和“[图形](#)”是一致的。在 38~39 行定义了针对 Words\_Addr 和 Write\_Data 的 ( rAddr 和 rData ) 暂存寄存器。换句话说，rAddr 寄存器是用来驱动 Words\_Addr 信号，rData 寄存器是用来驱动 Write\_Data 信号。

我们知道在 8 位 Start\_Sig 的位宽之中，Start\_Sig[7..3] 是写操作，反之 Start\_Sig[2..0] 是读操作。在 DS1302 芯片的时序中“[写操作](#)”的第一个字节需要“[访问寄存器的地址](#)”，第二个字节是“[写数据](#)”。

然而在 48~74 行针对这一内容，执行 rAddr 和 rData 寄存器值的赋值。如在 8'b1000\_0000 的时候，是“[关闭写保护](#)”的操作，换句话说就是要往“[控制寄存器](#)”写入“[数据 8'h00](#)”。故对 rAddr 和 rData 赋值 {2'b10, 5'd7, 1'b0}，8'h00。再举一个例子，当 Start\_Sig 等价于 8'b0100\_0000 的时候，即表示对“[时寄存器](#)”，“[写入数据](#)”。这时候对 rAddr 赋予早已经预定好的值，亦即 { 2'b10, 5'd2, 1'b0 }。然而不同的是，rData 被赋予的值，是从上层发来的 Time\_Write\_Data。

至于 Start\_Sig[2..0] 是表示“[读操作](#)”，在 DS1302 芯片的时序中，读操作只需要写入“[第](#)

“一字节数据”，第二字节数据是从 DS1302 读来的。举个例子，如当 Start\_Sig 等价于 8'b0000\_0001 是表示从“[秒寄存器读出数据](#)”，所以关于这个操作 rAddr 被赋予 { 2'b10, 5'd0, 1'b1 }。

在 76~120 行，是该模块的具体操作。i 寄存器表示执行步骤，rRead 寄存器是读出数据的暂存寄存器，isStart 寄存器是用于驱动 Access\_Start\_Sig，亦即是对 function\_module.v 控制的寄存器（78~80 行）。

在这里笔者再重申一下 Start\_Sig[7..3] 是“写操作”，Start\_Sig[2..0] 是“读操作”。

假设一个情况，当 Start\_Sig 等价于 8'b1000\_0000 的时候，我们知道这是“[关闭写保护](#)”的操作。在同一时间 rAddr 和 rData 都会被赋值（51 行）。然后 91 行的 if 条件就会成立。那么一次性的写操作就会发生（94~102 行），当一次性的写操作完成后，它会反馈完成信号。

我们再假设一个情况，当 Start\_Sig 等价于 8'b0000\_0001 的时候，这表示“[从秒寄存器读取数据](#)”。在同一个瞬间 rAddr 会被赋予相关的值（72 行），然后在 105 行 if 条件就会成立，在 108~116 行就会完成一次的“[读字节数据](#)”的操作。

当完成一次性的“[读字节数据](#)”，读取到的数据就会被暂存在 rRead 寄存器（109 行），最后反馈一个完成信号。以示上一层模块“[一次性的读数据操作](#)”已经完成。嗯！终于完成对 function\_module.v 和 cmd\_control\_module.v 的解释了。最后的工作就是把它们组合成为 ds1302\_module.v。

#### ds1302\_module.v

```
1. module ds1302_module
2. (
3.     CLK, RSTn,
4.
5.     Start_Sig,
6.     Done_Sig,
7.
8.     Time_Write_Data,
9.     Time_Read_Data,
10.
11.    RST,
12.    SCLK,
13.    SIO
14.
15. );
16.
```

```
17.    input CLK;
18.    input RSTn;
19.
20.    input [7:0]Start_Sig;
21.    output Done_Sig;
22.
23.    input [7:0]Time_Write_Data;
24.    output [7:0]Time_Read_Data;
25.
26.    output RST;
27.    output SCLK;
28.    inout SIO;
29.
30.   *****/
31.
32.   wire [7:0]Words_Addr;
33.   wire [7:0]Write_Data;
34.   wire [1:0]Access_Start_Sig;
35.
36.   cmd_control_module U1
37. (
38.     .CLK( CLK ),
39.     .RSTn( RSTn ),
40.     .Start_Sig( Start_Sig ),           // input - from top
41.     .Done_Sig( Done_Sig ),           // output - to top
42.     .Time_Write_Data( Time_Write_Data ), // input - from top
43.     .Time_Read_Data( Time_Read_Data ), // output - to top
44.     .Access_Done_Sig( Access_Done_Sig ), // input - from U2
45.     .Access_Start_Sig( Access_Start_Sig ), // output - to U2
46.     .Read_Data( Read_Data ),           // input - from U2
47.     .Words_Addr( Words_Addr ),           // output - to U2
48.     .Write_Data( Write_Data )           // output - to U2
49. );
50.
51.   *****/
52.
53.   wire [7:0]Read_Data;
54.   wire Access_Done_Sig;
55.
56.   function_module U2
57. (
58.     .CLK( CLK ),
59.     .RSTn( RSTn ),
60.     .Start_Sig( Access_Start_Sig ), // input - from U1
```

```

61.      .Words_Addr( Words_Addr ),           // input - from U1
62.      .Write_Data( Write_Data ),           // input - from U1
63.      .Read_Data( Read_Data ),           // output - to U1
64.      .Done_Sig( Access_Done_Sig ),       // output - to U1
65.      .RST( RST ),                      // output - to top
66.      .SCLK( SCLK ),                    // output - to top
67.      .SIO( SIO )                      // output - to top
68. );
69.
70. ****
71.
72. endmodule

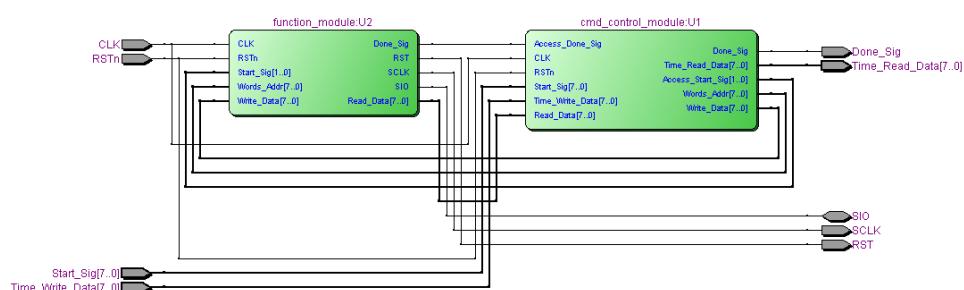
```

组合模块 ds1302\_module.v 基本和“图形”没有什么大不同，自己看着办吧。

### 实验十三说明：

在实验十三中 function\_module.v 包含了底层的基本操作，如果从顺序操作的角度看来 function\_module.v 包含了底层函数。然而 cmd\_control\_module.v 不是以“**函数的形式**”，基于底层函数去创建更高层的函数，相反的 cmd\_control\_module.v 是以“**位命令的形式**”去“**模仿更高层函数的执行步骤和操作**”。

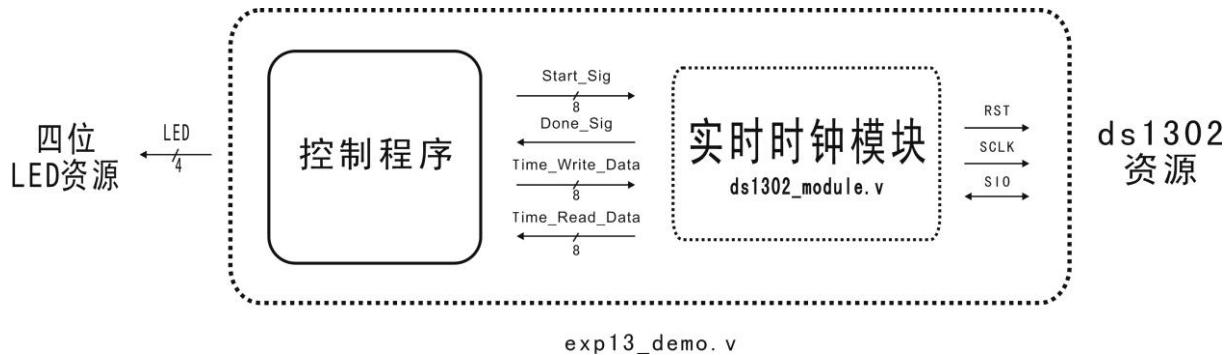
### 完成后的扩展图：



### 实验十三结论：

这个实验和以往的实验很不同，实验十三重点不是在于 DS1302 芯片的驱动，主要是使用 Verilog HDL 语言本身的位操作来强化“Start\_Sig”并且配合“**命令式仿顺序操作**”的设计。笔者为了使笔记精简，所以仅为 cmd\_control\_module.v 配置 8 个命令而已，实际上可以更多。这要读者自己看着办。

## 实验十三演示：



这个演示主要是演示对 `ds1302_module.v` 的调用。控制模块具体操作如下

- 一、关闭写保护，亦即发送命令 `8'b1000_0000`;
- 二、变更时寄存器，亦即发送命令 `8'b0100_0000`;
- 三、变更分寄存器，亦即发送命令 `8'b0010_0000`;
- 四、变更秒寄存器，亦即发送命令 `8'b0001_0000`;
- 五、最后永远读取秒寄存器的值，亦即发送命令 `8'b0000_0001`。然后将秒个位往四位 LED 资源发送。

`exp13_demo.v`

```

1. module exp13_demo
2. (
3.     CLK, RSTn,
4.     RST,
5.     SCLK,
6.     SIO,
7.     LED
8. );
9.
10.    input CLK;
11.    input RSTn;
12.    output RST;
13.    output SCLK;
14.    inout SIO;
15.    output [3:0]LED;
16.

```

## Verilog HDL 那些事儿 – 建模篇

```
17.      *****/
18.
19.      reg [3:0]i;
20.      reg [7:0]isStart;
21.      reg [7:0]rData;
22.      reg [3:0]rLED;
23.
24.      always @ ( posedge CLK or negedge RSTn )
25.          if( !RSTn )
26.              begin
27.                  i <= 4'd0;
28.                  isStart <= 8'd0;
29.                  rData <= 8'd0;
30.                  rLED <= 4'd0;
31.              end
32.          else
33.              case( i )
34.
35.                  0:
36.                      if( Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
37.                      else begin isStart <= 8'b1000_0000; rData <= 8'h00; end
38.
39.                  1:
40.                      if( Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
41.                      else begin isStart <= 8'b0100_0000; rData <= { 4'd1, 4'd2 }; end
42.
43.                  2:
44.                      if( Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
45.                      else begin isStart <= 8'b0010_0000; rData <= { 4'd2, 4'd2 }; end
46.
47.                  3:
48.                      if( Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
49.                      else begin isStart <= 8'b0001_0000; rData <= { 4'd2, 4'd2 }; end
50.
51.                  4:
52.                      if( Done_Sig ) begin rLED <= Time_Read_Data[3:0]; isStart <= 8'd0; i <= 4'd4; end
53.                      else begin isStart <= 8'b0000_0001; end
54.
55.              endcase
56.
57.      *****/
58.
59.      wire Done_Sig;
60.      wire [7:0]Time_Read_Data;
```

```
61.  
62.      ds1302_module U1  
63.      (  
64.          .CLK( CLK ),  
65.          .RSTn( RSTn ),  
66.          .Start_Sig( isStart ),  
67.          .Done_Sig( Done_Sig ),  
68.          .Time_Write_Data( rData ),  
69.          .Time_Read_Data( Time_Read_Data ),  
70.          .RST( RST ),  
71.          .SCLK( SCLK ),  
72.          .SIO( SIO )  
73.      );  
74.  
75.      /**************************************************************************/  
76.  
77.      assign LED = rLED;  
78.  
79.      /**************************************************************************/  
80.  
81.  
82. endmodule
```

在步骤 0 (35~37 行), 该模块向 ds1302\_module.v 发送关闭写保护的命令, 已经 8'b1000\_000。然后步骤 1~3 分别对, 时寄存器, 分寄存器和秒寄存器写入数据, 分别是写入 12 时, 22 分, 22 秒 (39~49 行)。在步骤 4, 会一直从秒寄存器读取, 实时时钟的秒值, 换句话说该模块会一直对 ds1302\_module.v 发送 8'b0000\_0001 的命令。每当完成“一次读字节操作”, 就会对 rLED 赋予“秒个位”的值。最后由 rLED 寄存器驱动 LED 信号。

## 总结：

当读者把笔记看到这里，笔者只能说恭喜了。基本上读者对“**低级建模**”的掌握更上一步了。经第二章到第四章，读者已经逐步掌握了“**低级建模**”的知识。笔记第二章要求掌握“**低级建模**”的基本概念；笔记第三章要求掌握“**低级建模**”的建模基础。然而笔记的第四章要求掌握“**仿顺序操作**”这一概念。无论是笔记的那一章，都是“**低级建模**”重要的一部分，谁也不可缺少谁。

读者可能在笔记的第四章中隐约了解到为什么“**仿顺序操作**”是 Verilog HDL 语言不可或缺的一部分。之所以“**仿顺序操作**”是 Verilog HDL 语言不可缺少的一部分，因为 Verilog HDL 是一个拥有并行性质的语言，多少对于的“**顺序操作**”都会有点力不从心。此外，很多“**控制**”或者“**驱动**”的工作，都和顺序操作有关，仿顺序操作的存在就是为了补足这一点。

笔者再强调一下“**仿顺序操作**”在宏观上是模仿“**顺序操作**”，微观上是使用 Verilog HDL 语言本身的性质去模仿“**顺序操作**”。目前在读者眼中看见的是“**仿顺序操作**”的假像，该假象会使读者把“**仿顺序操作**”看成“把大象放进步骤有几个步骤 … ”的概念。实际上，“**仿顺序操作**”的概念是“**时间点**”。

目前的读者可能会不理解笔者的这一番话真正的意义，读者不理解不用劲，读者只要懂得使用“**仿顺序操作**”就可以了。往后，当读者继续深入了解 Verilog HDL 语言，读者就会明白这一番话的真正意义。

到目前为止，这些实验都是准备功夫而已“**低级建模的精彩是满足当前的建模，为后期的建模带来准备。**”换句话说“**仿顺序操作**”不过是为后期做好准备的一部分而已。至于什么是“**后期的建模**”。下一章笔记，读者就会知晓了。

## 第五章：低级建模-封装（接口建模）

这里我们先讨论这样一个话题：

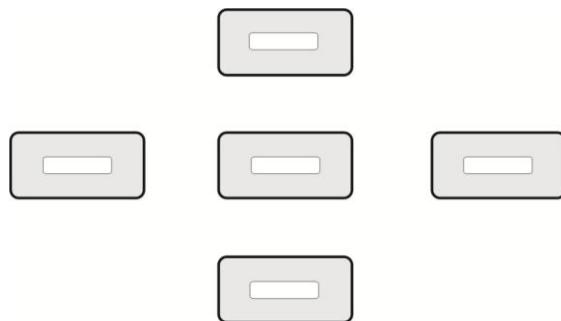
一层高楼，我们必须从地基开始建起。第一楼的“**地基**”必须稳固，而且结构必须良好。不然的话，第二楼的建筑工作就有困难。

低级建模就是这样一回事，早期的建模，我们只是针对每某个硬件资源，建立一个“**基础**”而已。这个“**基础**”虽然可以调用，但是却不是真正的完成品。如果“**基础**”要成为完成品，那么该“**基础**”就要执行所谓的“**封装**”。

“**封装**”的定义可以是很多，笔者暂时先把它定义为“某个基础的最后建模工作”。

### 5.1 实验十四 - 独立按键封装

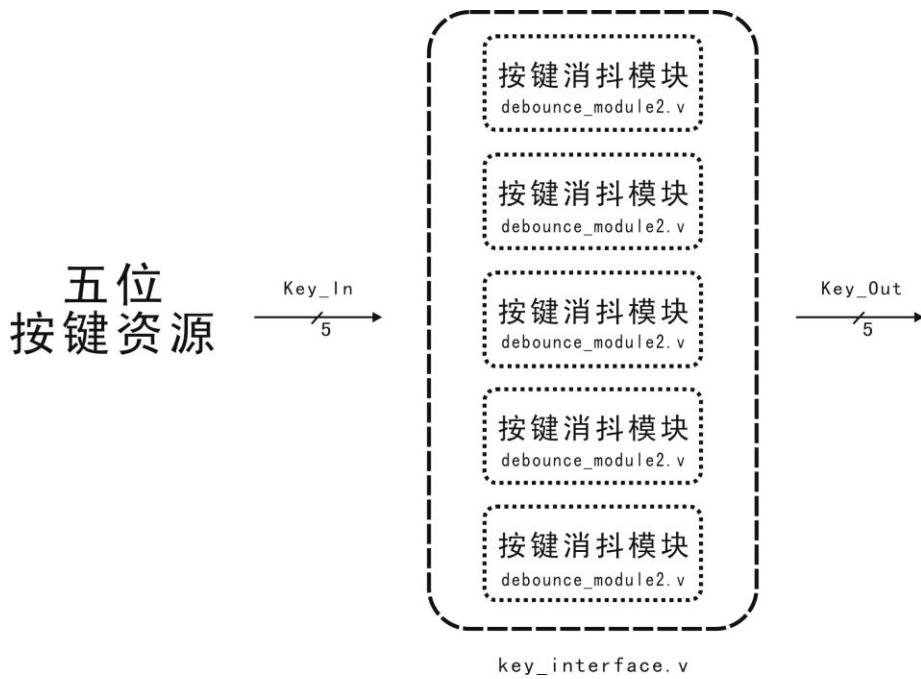
请回忆一下，在实验三和实验四是我们不是建立了“**按键消抖**”的功能模块了吗？估计读者们在早期可能会误会“**实验这样简单就完了？**”。实验三和实验四就宛如“**按键的地基**”而已，因为“**消抖功能**”对所有按键来说是必须拥有的功能。但是在实验三和实验四却还没有针对黑金开发板上的“**按键资源**”，执行所谓“**竣工**”意义上的建模。



**在针对某一个硬件资源的封装之前，不同的硬件资源都有不同的考虑。**如黑金开发板上的 5 个独立按键，如果笔者要为它们封装的话，我们必须考虑什么？

- (一) 按键的功能 - 按键按下消抖，按键按下产生高脉冲，按键释放消抖。
- (二) 按键的数目 - 5 个按键资源。

然而实验四的 debounce\_module2.v 符合如上的功能，那么我们只要基于该模块，执行 5 次的实例化，然后再组合就会完成“**针对黑金开发板的独立按键**”的封装工作。



上图是基于实验四 debounce\_module2.v 经过 5 次实例化后，再以 key\_interface.v 组合模块执行封装而成的“[按键接口](#)”。

*key\_interface.v*

```
1. module key_interface
2. (
3.     input CLK,
4.     input RSTn,
5.     input [4:0]Key_In,
6.     output [4:0]Key_Out
7. );
8.
9. // [4]Up [3]Down [2]Left [1]Right [0]Middle
10.
11. /***** *****/
12.
13. debounce_module2 U1_Up // Debounded key up
14. (
15.     .CLK( CLK ),
16.     .RSTn( RSTn ),
17.     .Pin_In( Key_In[4] ),
18.     .Pin_Out( Key_Out[4] )
```

```
19. );
20. *****/
21. debounce_module2 U2_Down // Debounded key down
22. (
23.     .CLK( CLK ),
24.     .RSTn( RSTn ),
25.     .Pin_In( Key_In[3] ),
26.     .Pin_Out( Key_Out[3] )
27. );
28. *****/
29. debounce_module2 U3_Left // Debounded key left
30. (
31.     .CLK( CLK ),
32.     .RSTn( RSTn ),
33.     .Pin_In( Key_In[2] ),
34.     .Pin_Out( Key_Out[2] )
35. );
36. *****/
37. debounce_module2 U4_Right // Debounded key right
38. (
39.     .CLK( CLK ),
40.     .RSTn( RSTn ),
41.     .Pin_In( Key_In[1] ),
42.     .Pin_Out( Key_Out[1] )
43. );
44. *****/
45. debounce_module2 U5_Middle // Debounded key middle
46. (
47.     .CLK( CLK ),
48.     .RSTn( RSTn ),
49.     .Pin_In( Key_In[0] ),
50.     .Pin_Out( Key_Out[0] )
51. );
52. *****/
53. 
```

---

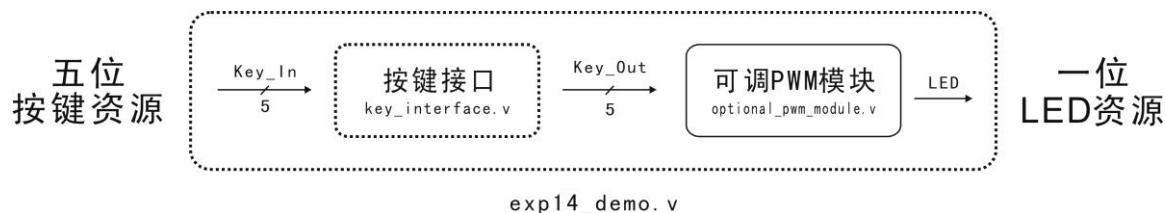
63. endmodule

上面代码就是利用实验四的 debounce\_module2.v 多次实例化的结果。

Key\_In[4..0] 和 Key\_Out[4..0] 位分配的如下。

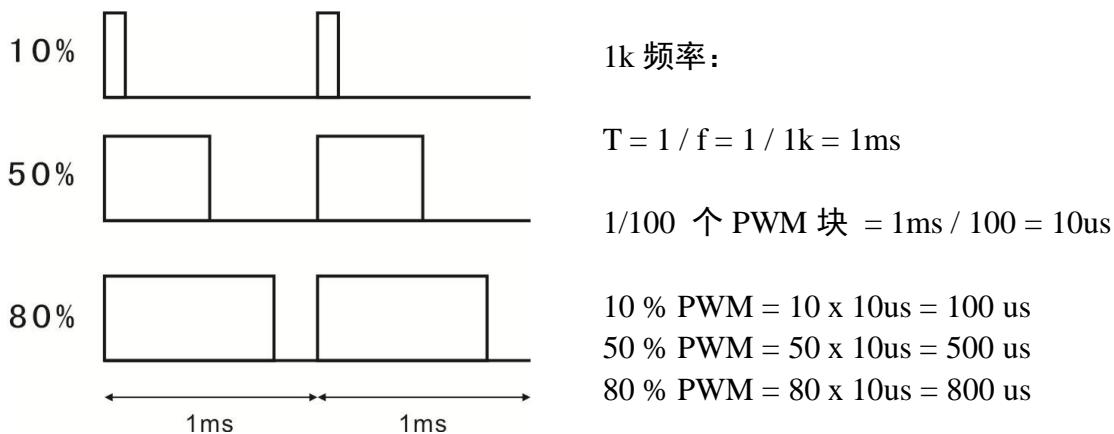
按键资源	Key_In[4..0]	Key_Out[4..0]
↑上	Key_In[4]	Key_Out[4]
↓下	Key_In[3]	Key_Out[3]
←左	Key_In[2]	Key_Out[2]
→右	Key_In[1]	Key_Out[1]
· 中	Key_In[0]	Key_Out[0]

## 实验十四演示：



实验十四演示，主要是利用 5 位按键分别去调制“可调 PWM 模块”。我们知道按键接口的作用，每当按下某一个按钮，某一个输出就是输出一个高脉冲。在这个演示中，最陌生应该是 optional\_pwm\_module.v 这个模块吧。

“可调 PWM 模块”笔者在初期的笔记打算加入的，但是考虑到初期的问题，才延后到这里。PWM 实验对于 Verilog HDL 语言来说是一个经典的实验。那么，什么是 PWM？我们来简单认识一下：



PWM 信号在宏观上是拥有不同占空比比率的信号周期。从上图我们可以看到，假设笔者要求 1k 的频率方波信号，那么一个周期就是 1ms。再假设笔者求得占空比的比率是

可以从 1~100% 之间调节，那么笔者必须将一个周期的时间再分为 100 份，亦即 10us 一个 PWM 块。

上图中指示了 3 个不同占空比的 PWM 信号：

10% PWM 信号，高电平保持时间 100us，低电平保持时间 900us。

50% PWM 信号，高电平保持时间 500us，低电平保持时间 500us。

80% PWM 信号，高电平保持时间 800us，低电平保持时间 200us。

PWM 信号在电流（电压）的调节上是非常方便的。一些简单的计算可以是如下：

假设某输出口的驱动能力是 10mA，当仅有 10% PWM 的情况下，驱动能力降落仅剩原有的 10% 能力，亦即 1mA。

在实验十四的演示中，利用 optional\_pwm\_module.v 控制 LED 发光亮度。然而 optional\_pwm\_module.v 的 1k 输出频率，PWM 块分为 256 份 (0~255)，而不是典型的 100 份 PWM 块。

$$T = 1 / f = 1 / 1\text{kHz} = 1\text{ms}$$

$$256 \text{ 份 PWM 块} = 1\text{ms} / 256 = 3.9\mu\text{s}$$

如果以 50Mhz 的频率产生 3.9us 的定时：

$$N = (3.9 \times 10^{-6}) / (1 / 50 \times 10^6) \\ = 195$$

=====

我们知道 key\_interface.v 的 Key\_In[4..0] 和 Key\_Out[4..0] 位分配是如下：

[4]Up(Button) [3]Down(Button) [2]Left(Button) [1]Right(Button) [0]Middle(Button)

我们以 key\_interface.v 的位分配，针对 optional\_pwm\_module.v 的 PWM 调节如下：

位分配	按键映射	optional_pwm_module.v 的调节功能
[4]	Up(Button)	PWM 块 +10
[3]	Down(Button)	PWM 块 -10
[2]	Left(Button)	PWM 块 -1
[1]	Right(Button)	PWM 块 +1
[0]	Middle(Button)	PWM 块 1/2

具体的内容，我们直接看源码。

*optional\_pwm\_module.v*

```
1. module optional_pwm_module
2. (
3.     input CLK,
4.     input RSTn,
5.     input [4:0]Option_Key,
6.     output LED
7. );
8.
9.     ****
10.
11.    parameter SEGMENT = 8'd195;// 3.9us
12.
13.    ****
14.
15.    reg [7:0]C1;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            C1 <= 8'd0;
20.        else if( C1 == SEGMENT )
21.            C1 <= 8'd0;
22.        else
23.            C1 <= C1 + 1'b1;
24.
25.    ****
26.
27.    reg [7:0]System_Seg;
28.
29.    always @ ( posedge CLK or negedge RSTn )
30.        if( !RSTn )
31.            System_Seg <= 8'd0;
32.        else if( System_Seg == 8'd255 )
33.            System_Seg <= 8'd0;
34.        else if( C1 == SEGMENT )
35.            System_Seg <= System_Seg + 1'b1;
36.
37.    ****
38.
39.    reg [7:0]Option_Seg;
40.
41.    always @ ( posedge CLK or negedge RSTn )
```

```
42.         if( !RSTn )
43.
44.             Option_Seg <= 8'd0;
45.
46.         else if( Option_Key[4] ) // Key up = Segment + 10
47.
48.             if( Option_Seg < 8'd245) Option_Seg <= Option_Seg + 8'd10;
49.             else Option_Seg <= 8'd255;
50.
51.         else if( Option_Key[3] ) // key down = Segment - 10
52.
53.             if( Option_Seg > 8'd10) Option_Seg <= Option_Seg - 8'd10;
54.             else Option_Seg <= 8'd0;
55.
56.         else if( Option_Key[2] ) // key left = Segment + 1
57.
58.             if( Option_Seg < 8'd255) Option_Seg <= Option_Seg + 8'd1;
59.             else Option_Seg <= 8'd255;
60.
61.         else if( Option_Key[1] ) // key down = Segment - 1
62.
63.             if( Option_Seg > 8'd0) Option_Seg <= Option_Seg - 8'd1;
64.             else Option_Seg <= 8'd0;
65.
66.         else if( Option_Key[0] ) // key middle = Segment = half
67.
68.             Option_Seg <= 8'd127;
69.
70.     *****/
71.
72.     assign LED = ( System_Seg < Option_Seg ) ? 1'b1 : 1'b0;
73.
74.     *****/
75.
76. endmodule
```

在源码第 11 行，定义了以 50Mhz 频率定时 3.9us 的常量。15~23 行是 3.9us 的定时器。第 27~35 行是 256 块 PWM 的计数器。37~70 行是该模块的核心功能：

当 Option\_Key（亦即 key\_interface.v Key\_Out 的连线口）其中某一位产生高脉冲的时候，都会有各种的功,。37~70 行简略下会如下表：

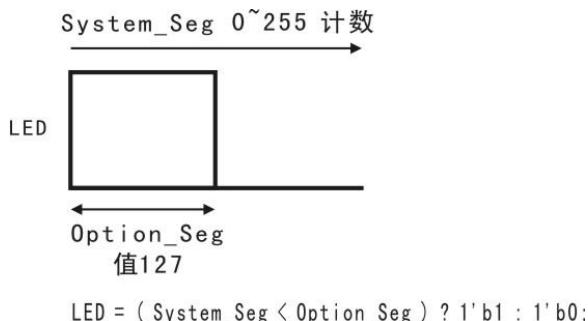
key_interface.v Key_Out[4..0]	optional_pwm_module.v Option_Key[4..0]	按键映射	功能
[4]	[4]	Up	PWM 块 +10
[3]	[3]	Down	PWM 块 -10
[2]	[2]	Left	PWM 块 -1
[1]	[1]	Right	PWM 块 +1
[0]	[0]	Middle	PWM 块 1/2

寄存器 Option\_Seg 表示了可调节的 PWM 块。假设 Option\_Seg 的值调节至 127，已经是 127 / 255 也就是 50% PWM。

然而寄存器 System\_Seg 是自动递增的 PWM 块，每 3.9us 定时 System\_Seg 就会自动递增 (34~35 行)。当 System\_Seg 从 0 值递增至 255 值，这就表示  $3.9\text{us} \times 256 = 1\text{ms}$ ，亦即一个周期已经计数完毕。然后又从下一个周期开始计数。(32~33 行)

假设笔者要产生 50 % PWM 的信号，必然 Option\_Seg 的值必须调制为 127。然后再经 72 行表达式的关系，就会产生 50% 占空比的 PWM 信号。PWM 产生过程如下：

当 System\_Seg 计数器从 0 ~ 127 递增的时候，由于 72 行的表达式 ( $\text{System\_Seg} < \text{Option\_Seg}$ ) 的关系，在 System\_Seg 计数器的值在 0~127 之间，LED 的输出都是高电平。当 System\_Seg 计数器的值递增超过 127 之后，亦即 128~255 之间，LED 的输出都是低电平。



左图是 72 行表达式的关系图。Option\_Seg 可以看成是一个比对常量，当 System\_Seg 小于 Option\_Seg 常量值 LED 的输出是高电平。当 System\_Seg 的值超过 Option\_Seg 的常量值，LED 的输出就是低电平。

在这里我们稍微来讨论 46~68 行的功能。

当 Option\_Key[4] 产生一个高脉冲，如果 Option\_Seg 的值小于 245，那么 Option\_Seg 的值就增加 10 。反之如果 Option\_Seg 的值大于 245，Option\_Seg 就直接赋予 255。(49~49 行)

当 Option\_Key[3] 产生一个高脉冲，如果 Option\_Seg 的值大于 9，那么 Option\_Seg 的值就减少 10 。反之如果 Option\_Seg 的值小于 9，Option\_Seg 就直接赋予 0。(51~54 行)

当 Option\_Key[2] 产生一个高脉冲, 如果 Option\_Seg 的值小于 255, 那么 Option\_Seg 的值就增加 1。反之如果 Option\_Seg 的值大于等于 255, Option\_Seg 就直接赋予 255。(56~59 行)

当 Option\_Key[1] 产生一个高脉冲, 如果 Option\_Seg 的值大于 0, 那么 Option\_Seg 的值就减少 1。反之如果 Option\_Seg 的值小于等于 0, Option\_Seg 就直接赋予 0。(61~64 行)

当 Option\_Key[0] 产生一个高脉冲, Option\_Seg 就直接赋予 127。(66~68 行)

*exp14\_demo.v*

```
1. module key_interface_demo
2. (
3.     input CLK,
4.     input RSTn,
5.     input [4:0]Key_In,
6.     output LED
7. );
8.
9. ****
10.
11. wire [4:0]Key_Out;
12.
13. key_interface U1
14. (
15.     .CLK( CLK ),
16.     .RSTn( RSTn ),
17.     .Key_In( Key_In ),      // input - from top
18.     .Key_Out( Key_Out )    // output - to U2
19. );
20.
21. ****
22.
23. optional_pwm_module U2
24. (
25.     .CLK( CLK ),
26.     .RSTn( RSTn ),
27.     .Option_Key( Key_Out ), // input - from U1
28.     .LED( LED )           // output - to top
29. );
30.
```

```

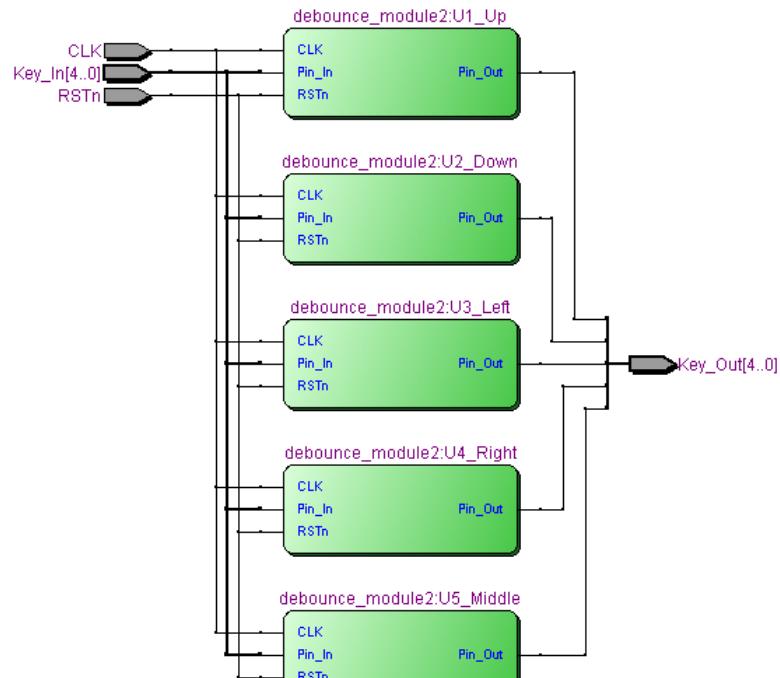
31.   *****/
32.
33. endmodule

```

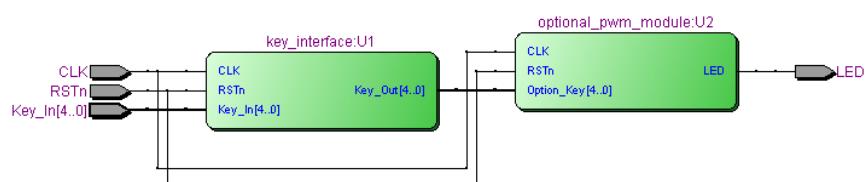
#### 实验十四演示说明:

这个演示主要是演示 key\_interface.v 如何调用而已。然而 key\_interface.v 的重点就是 Key\_Out[4..0] 每个位的为分配。

#### 完成扩展图:



key\_interface.v



key\_interface\_demo.v

### 实验十四演示结论：

这个实验比较简单，主要是利用实验四的 debounce\_module2.v 然后针对 5 位的按键资源执行实例化成为 key\_interface.v，然后在 key\_interface\_demo.v 中调用。这章的实验，对于封装来讲，算是最简单的一种吧。前面笔者已经说过了，不同的封装都有不同的考虑。

## 5.2 实验十五：数码管封装

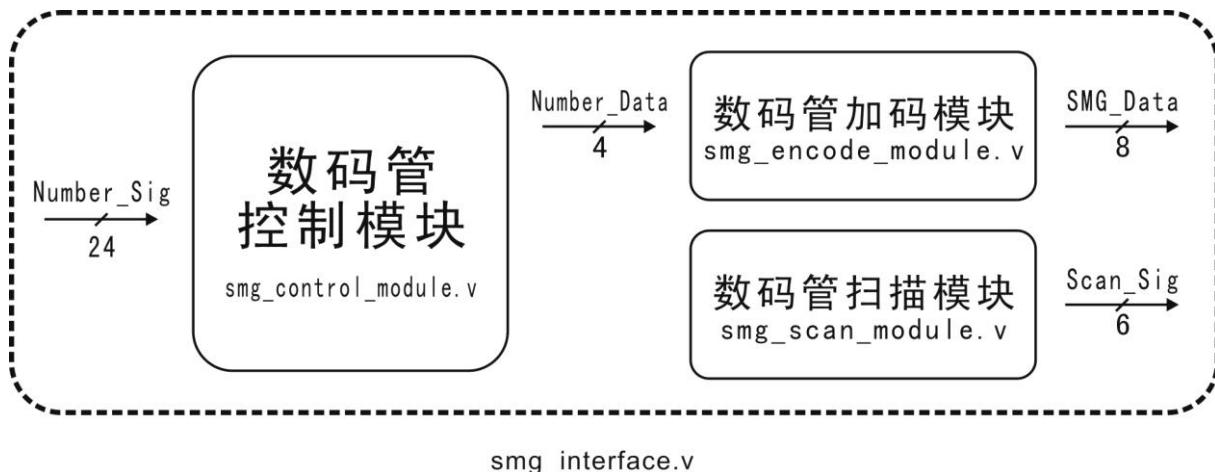
数码管的驱动实验在实验七中不是干过了吗？但是驱动和封装是不同一回事（5.1 章说过了）。在实验七，笔者针对“十进制”“两位数码管”，去完成驱动的设计。在过程中我们没有对黑金开发板上数码管资源考虑过如何封装。

在这一章的实验，我们要充分的使用 Verilog HDL 语言强大的“位操作”来简化数码管的封装工作。

数码管位数	6 位
每一位停留时间	1 ms
一次性扫描时间	6 ms
扫描频率	166.67 Hz

（具体的内容请回顾 3.1 章）

上表是封装数码管所要求的属性。下面是封装的“图形”。



上图的组合模块-数码管接口 `smg_interface.v` 中，输入信号 `Number_Sig` 占了 24 位宽，然而 `Number_Sig` 的位分配如下表：

Number_Sig[23:0]	位代表（从左边数起）
Number_Sig[23:20]	第一位 数字   数码管
Number_Sig[19:16]	第二位 数字   数码管
Number_Sig[15:12]	第三位 数字   数码管
Number_Sig[11:8]	第四位 数字   数码管
Number_Sig[7:4]	第五位 数字   数码管
Number_Sig[3:0]	第六位 数字   数码管

为什么每一位数字|数码管，都用 4 位位宽来代表呢？其实这个 idea 是来至 ds1302 芯片。我们知道每数码管可以支持显示 0~F，正是因为如此，如果我们用每 4 位位宽来代表某一个数码管显示的信号，那么可以避免“使用除法或者求余运算符执行十进制的取位操作”。一来方便设计，而来减少资源。

举个例子：24'h123456，亦即 0001 0010 0011 0100 0101 0110。

smg\_encode\_module.v 在这里的功能就是就是将数字 0~F 加码为数码管码。然而，比较特别的是，smg\_control\_module.v 和 smg\_scan\_module.v 有并行操作的性质。smg\_interface.v 的大致操作如下：

假设我往 Number\_Sig 输入 24'h123456

在 T1，smg\_control\_module.v 会将 Number\_Sig[23:20] 送往至 smg\_encode\_module.v 加码并且送往数码管。在同一时间 smg\_scan\_module.v 会扫描第一位数码管（使能）。

在 T2，smg\_control\_module.v 会将 Number\_Sig[19:16] 送往至 smg\_encode\_module.v 加码并且送往数码管，在同一时间 smg\_scan\_module.v 会扫描第二位数码管（使能）。

在 T3，smg\_control\_module.v 会将 Number\_Sig[15:12] 送往至 smg\_encode\_module.v 加码并且送往数码管，在同一时间 smg\_scan\_module.v 会扫描第三位数码管（使能）。

在 T4，smg\_control\_module.v 会将 Number\_Sig[11:8] 送往至 smg\_encode\_module.v 加码并且送往数码管，在同一时间 smg\_scan\_module.v 会扫描第四位数码管（使能）。

在 T5，smg\_control\_module.v 会将 Number\_Sig[7:4] 送往至 smg\_encode\_module.v 加码并且送往数码管，在同一时间 smg\_scan\_module.v 会扫描第五位数码管（使能）。

在 T6，smg\_control\_module.v 会将 Number\_Sig[3:0] 送往至 smg\_encode\_module.v 加码并且送往数码管，在同一时间 smg\_scan\_module.v 会扫描第六位数码管（使能）。

在 T1 的时候第一位数码管会显示 1。在 T2 的时候第二位数码管会显示 2，其他的依此类推。最后在 T6 的时候，第六位数码管会显示 6。

就这样一次性的扫描（六位数码管全扫描）就完成。啊，别忘了！每位数码管扫描停留的时间（使能的时间）大约是 1ms。所以一次性扫描所需要的时间大约是 6ms，亦即在每一秒内，一组 6 位的数码管会扫描 166 次左右。

smg\_interface.v 具体的操作还是直接看源码吧。

*smg\_control\_module.v*

```
1. module smg_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.     input [23:0]Number_Sig,
6.     output [3:0]Number_Data
7. );
8.
9.     /*************************************************************************/
10.
11.    parameter T1MS = 16'd49999;
12.
13.    /*************************************************************************/
14.
15.    reg [15:0]C1;
16.
17.    always @ ( posedge CLK or negedge RSTn )
18.        if( !RSTn )
19.            C1 <= 16'd0;
20.        else if( C1 == T1MS )
21.            C1 <= 16'd0;
22.        else
23.            C1 <= C1 + 1'b1;
24.
25.    /*************************************************************************/
26.
27.    reg [3:0]i;
28.    reg [3:0]rNumber;
29.
30.    always @ ( posedge CLK or negedge RSTn )
31.        if( !RSTn )
32.            begin
33.                i <= 4'd0;
34.                rNumber <= 4'd0;
35.            end
36.        else
37.            case( i )
38.
39.                0:
40.                    if( C1 == T1MS ) i <= i + 1'b1;
```

```
41.           else rNumber <= Number_Sig[23:20];
42.
43.           1:
44.             if( C1 == T1MS ) i <= i + 1'b1;
45.             else rNumber <= Number_Sig[19:16];
46.
47.           2:
48.             if( C1 == T1MS ) i <= i + 1'b1;
49.             else rNumber <= Number_Sig[15:12];
50.
51.           3:
52.             if( C1 == T1MS ) i <= i + 1'b1;
53.             else rNumber <= Number_Sig[11:8];
54.
55.           4:
56.             if( C1 == T1MS ) i <= i + 1'b1;
57.             else rNumber <= Number_Sig[7:4];
58.
59.           5:
60.             if( C1 == T1MS ) i <= 4'd0;
61.             else rNumber <= Number_Sig[3:0];
62.
63.       endcase
64.
65.   *****/
66.
67.   assign Number_Data = rNumber;
68.
69.   *****/
70.
71. endmodule
```

在 11 行是 1ms 的常量，第 15~23 行则是 1ms 的定时器。第 27~63 行是该控制模块的核心部分。rNumber 是每一位数字的暂存器（28 行）用来驱动 Number\_Data（67 行）。（39~61 行）每隔 1ms 该控制模块就会将不同位的数字往 Number\_Data 输出。

*smg\_encode\_module.v*

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

```
5.      input [3:0]Number_Data,
6.      output [7:0]SMG_Data
7.  );
8.
9.  /*************************************************************************/
10.
11. parameter _0 = 8'b1100_0000, _1 = 8'b1111_1001, _2 = 8'b1010_0100,
12.           _3 = 8'b1011_0000, _4 = 8'b1001_1001, _5 = 8'b1001_0010,
13.           _6 = 8'b1000_0010, _7 = 8'b1111_1000, _8 = 8'b1000_0000,
14.           _9 = 8'b1001_0000;
15.
16. /*************************************************************************/
17.
18. reg [7:0]rSMG;
19.
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
40.
41. /*************************************************************************/
42.
43. assign SMG_Data = rSMG;
44.
45. /*************************************************************************/
46.
47. endmodule
```

smg\_encode\_module.v 和实验七相比更简化了许多。

*smg\_scan\_module.v*

```
1. module smg_scan_module
2. (
3.     input CLK,
4.     input RSTn,
5.     output [5:0]Scan_Sig
6. );
7.
8.     /*****
9.
10.    parameter T1MS = 16'd49999;
11.
12.    *****/
13.
14.    reg [15:0]C1;
15.
16.    always @ ( posedge CLK or negedge RSTn )
17.        if( !RSTn )
18.            C1 <= 16'd0;
19.        else if( C1 == T1MS )
20.            C1 <= 16'd0;
21.        else
22.            C1 <= C1 + 1'b1;
23.
24.    *****/
25.
26.    reg [3:0]i;
27.    reg [5:0]rScan;
28.
29.    always @ ( posedge CLK or negedge RSTn )
30.        if( !RSTn )
31.            begin
32.                i <= 4'd0;
33.                rScan <= 6'b100_000;
34.            end
35.        else
36.            case( i )
37.
38.                0:
```

```
39.          if( C1 == T1MS ) i <= i + 1'b1;
40.          else rScan <= 6'b011_111;
41.
42.          1:
43.          if( C1 == T1MS ) i <= i + 1'b1;
44.          else rScan <= 6'b101_111;
45.
46.          2:
47.          if( C1 == T1MS ) i <= i + 1'b1;
48.          else rScan <= 6'b110_111;
49.
50.          3:
51.          if( C1 == T1MS ) i <= i + 1'b1;
52.          else rScan <= 6'b111_011;
53.
54.          4:
55.          if( C1 == T1MS ) i <= i + 1'b1;
56.          else rScan <= 6'b111_101;
57.
58.          5:
59.          if( C1 == T1MS ) i <= 4'd0;
60.          else rScan <= 6'b111_110;
61.
62.
63.      endcase
64.
65.  *****/
66.
67. assign Scan_Sig = rScan;
68.
69.  *****/
70.
71.
72. endmodule
```

同样和实验七相比，数码管扫描模块没有了“[行扫描](#)”的概念，而是简化至仅有“[列扫描](#)”。第 10 行是 1ms 的常量声明，在 14~22 行是 1ms 的定时器。该模块和 smg\_control\_module.v 一样，都是每隔 1ms 都有一个动作。smg\_scan\_module.v 每隔 1ms 就会使能不同的数码管（38~60 行）。然而数码管实际的扫描顺序是自左向右。在位操作的角度上，逻辑 0 从最高位到最低位交替移位。

*smg\_interface.v*

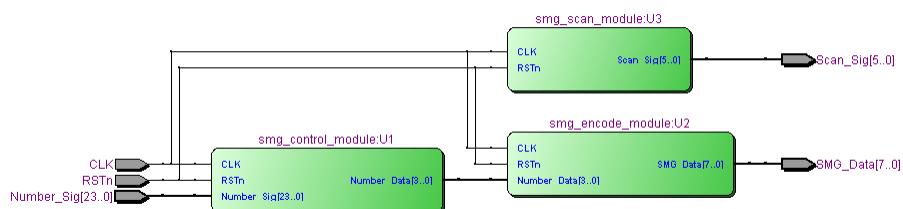
```
1. module smg_interface
2. (
3.     input CLK,
4.     input RSTn,
5.     input [23:0]Number_Sig,
6.     output [7:0]SMG_Data,
7.     output [5:0]Scan_Sig
8. );
9.
10.    /*************************************************************************/
11.
12.    wire [3:0]Number_Data;
13.
14.    smg_control_module U1
15.    (
16.        .CLK( CLK ),
17.        .RSTn( RSTn ),
18.        .Number_Sig( Number_Sig ),      // input - from top
19.        .Number_Data( Number_Data )   // output - to U2
20.    );
21.
22.    /*************************************************************************/
23.
24.    smg_encode_module U2
25.    (
26.        .CLK( CLK ),
27.        .RSTn( RSTn ),
28.        .Number_Data( Number_Data ),  // input - from U2
29.        .SMG_Data( SMG_Data )       // output - to top
30.    );
31.
32.    /*************************************************************************/
33.
34.    smg_scan_module U3
35.    (
36.        .CLK( CLK ),
37.        .RSTn( RSTn ),
38.        .Scan_Sig( Scan_Sig ) // output - to top
39.    );
40.
41.    /*************************************************************************/
```

42.  
43.  
44.  
45. endmodule

### 实验十五说明:

这个实验也没有什么特别的，最重要还是得搞懂 Number\_Sig 的位分配。

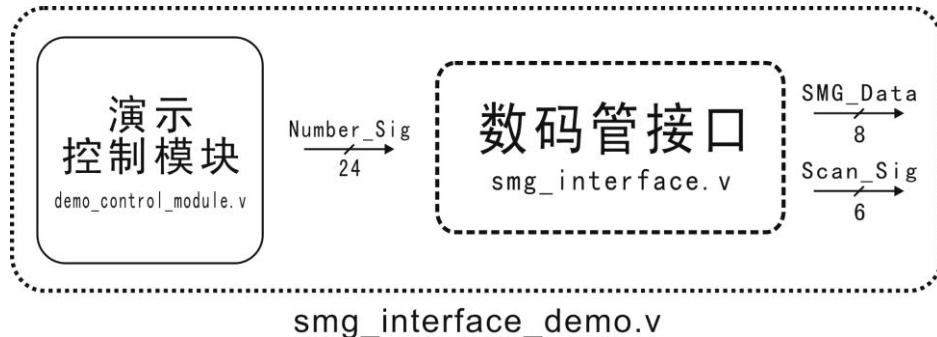
### 完成扩展图:



### 实验十五结论:

实验十五和实验十四不同的是，实验十四的 `key_interface.v` 我们要考虑如何调用它的输出，然而实验十五的 `smg_interface.v` 我们则要考虑输入调用它的输入。

## 实验十五演示：



上图是实验十五演示要实现的“[图形](#)”。演示中会建立一个名为 `demo_control_module.v` 输出 `24'h000000 ~ 24'h999999` 用来驱动 `smg_interface.v` 的输入。具体的内容还是直接看代码：

*demo\_control\_module.v*

```
1. module demo_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.     output [23:0]Number_Sig
6. );
7.
8. //*****
9.
10. parameter T100MS = 23'd4_999_999;
11.
12. //*****
13.
14. reg [22:0]C1;
15.
16. always @ ( posedge CLK or negedge RSTn )
17.     if( !RSTn )
18.         C1 <= 23'd0;
19.     else if( C1 == T100MS )
20.         C1 <= 23'd0;
21.     else
```

```

22.          C1 <= C1 + 1'b1;
23.
24.      /*************************************************************************/
25.
26.      reg [3:0]i;
27.      reg [23:0]rNum;
28.      reg [23:0]rNumber;
29.
30.      always @ ( posedge CLK or negedge RSTn )
31.          if( !RSTn )
32.              begin
33.                  i <= 4'd0;
34.                  rNum <= 24'd0;
35.                  rNumber <= 24'd0;
36.              end
37.          else
38.              case( i )
39.
40.                  0:
41.                      if( C1 == T100MS ) begin rNum[3:0] <= rNum[3:0] + 1'b1;i <= i + 1'b1; end
42.
43.                  1:
44.                      if( rNum[3:0] > 4'd9 ) begin rNum[7:4] <= rNum[7:4] + 1'b1; rNum[3:0] <= 4'd0; i <= i + 1'b1; end
45.                      else i <= i + 1'b1;
46.
47.                  2:
48.                      if( rNum[7:4] > 4'd9 )begin rNum[11:8] <= rNum[11:8] + 1'b1; rNum[7:4] <= 4'd0; i <= i + 1'b1; end
49.                      else i <= i + 1'b1;
50.
51.                  3:
52.                      if( rNum[11:8] > 4'd9 ) begin rNum[15:12] <= rNum[15:12] + 1'b1; rNum[11:8] <= 4'd0;i <= i + 1'b1; end
53.                      else i <= i + 1'b1;
54.
55.                  4:
56.                      if( rNum[15:12] > 4'd9 ) begin rNum[19:16] <= rNum[19:16] + 1'b1; rNum[15:12] <= 4'd0; i <= i + 1'b1; end
57.                      else i <= i + 1'b1;
58.
59.                  5:
60.                      if( rNum[15:12]>4'd9 ) begin rNum[19:16] <= rNum[19:16] + 1'b1; rNum[15:12] <= 4'd0; i <= i + 1'b1;end
61.                      else i <= i + 1'b1;
62.
63.                  6:
64.                      if( rNum[19:16] > 4'd9 ) begin rNum[23:20] <= rNum[23:20] + 1'b1; rNum[19:16] <= 4'd0; end
65.                      else i <= i + 1'b1;

```

```
66.  
67.      7:  
68.          if( rNum[23:20] > 4'd9 ) begin rNum <= 24'd0; i <= i + 1'b1; end  
69.          else i <= i + 1'b1;  
70.  
71.      8:  
72.          begin rNumber <= rNum; i <= 4'd0; end  
73.  
74.      endcase  
75.  
76.      /*****  
77.  
78.      assign Number_Sig = rNumber;  
79.  
80.      *****/  
81.  
82.  endmodule
```

第 8~24 行之间，包含了 100ms 定时的常量（10 行）和 100ms 的定时器（14~22 行）。在 26~74 就是该模块的核心部分。寄存器 rNum 操作空间（27 行），然而 rNumber 是用于驱动 Number\_Sig（78 行）。每隔 100ms 的定时都会是 rNum 递增（41 行），43~69 行之间就会执行“4 位宽”数字之间的“进位操作”。

我们假设一个情况，当 rNum 的值是 24'h000009，然后在下一个 100ms 的定时钟，rNum 的值就会 +1 操作。在 44 行，if 条件就会成立，rNum[3:0] 就会被赋值为零，然后 rNum[7:4] 就会执行 +1 操作，rNum 的值成为 24'h000010。接下来的几个步骤也会执行类似的操作。

在 67~69 行表示了当 rNum 的值超过 24'h999999 的时候，就会恢复为 24'h000000。

在这里我们有一个问题？为什么不直接使用 rNum 驱动 Number\_Sig 而是选择使用 rNumber 寄存器来驱动 Number\_Sig。如果我们把 rNum 当着 Number\_Sig 的驱动对象，在 i 步骤 1~7 之间，由于“进位操作”的关系，会使得 Number\_Sig 的输出产生许多毛刺，因此才使用 rNumber 驱动 Number\_Sig。当 rNum 完成“进位操作”以后，再赋值与 rNumber，由 rNumber 驱动 Number\_Sig（72 行）。

*smg\_interface\_demo.v*

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

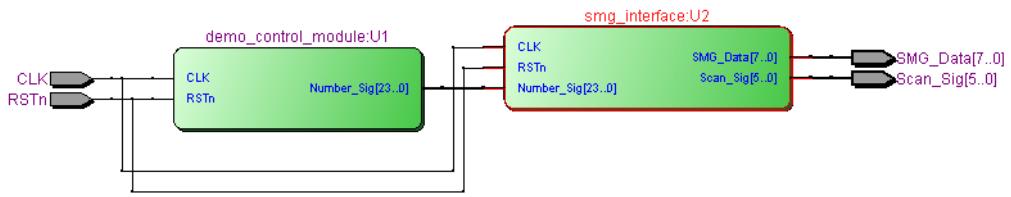
```
5.      output [7:0]SMG_Data,
6.      output [5:0]Scan_Sig
7.  );
8.
9.  *****/
10.
11.     wire [23:0]Number_Sig;
12.
13.     demo_control_module U1
14.     (
15.         .CLK( CLK ),
16.         .RSTn( RSTn ),
17.         .Number_Sig( Number_Sig ) // output - to U2
18.     );
19.
20. *****/
21.
22.     smg_interface U2
23.     (
24.         .CLK( CLK ),
25.         .RSTn( RSTn ),
26.         .Number_Sig( Number_Sig ), // input - from U1
27.         .SMG_Data( SMG_Data ), // output - to top
28.         .Scan_Sig( Scan_Sig ) // output - to top
29.     );
30.
31. *****/
32.
33. endmodule
```

上面是组合模块 smg\_interface \_demo.v 的内容，基本上和“[图形](#)”是一样的。自己看着办吧。

#### 实验十五演示说明：

这个演示说明了如何调用 smg\_interface.v 的输入，而且还说明了，善用位操作会简化设计和建模。

完成的扩展图：



实验十五演示结论：

数码管接口的调用演示。

### 5.3 实验十六：蜂鸣器封装

当读者看到这章，不要笑出来，笔者连蜂鸣器也不放过。在前面（5.1 和 5.2）的试验中，无论是独立键盘，还是数码管，它们要封装，它们都有自己的考虑，那么蜂鸣器考虑什么？这家伙那么单调，只要拉低电平，这家伙就会被驱动了。既然蜂鸣器那么单调，我们就使它不单调，我们可以在蜂鸣器的封装中，为它加入产生 S 摩斯码和 O 模式码的“功能”。

我们利用 4.3 章的“命令式仿顺序操作”的方法来建立它的功能模块。（请稍微复习一下）



等等！这就不是一个模块吗！？这样也称得上封装？

在这里我们需要重新为“**接口**”加入一个“**新的定义**”。在 5.1 和 5.2 章的试验中，我们定义了“**接口**”是“**最后的工程**”。不错“**封装**”却是针对某个资源的最后建模工程，但是读者有没有发现在 5.1 和 5.2 章的实验还包含着一个信息？那就是封装过后的接口模块“**都有独立性**”这一个事实。

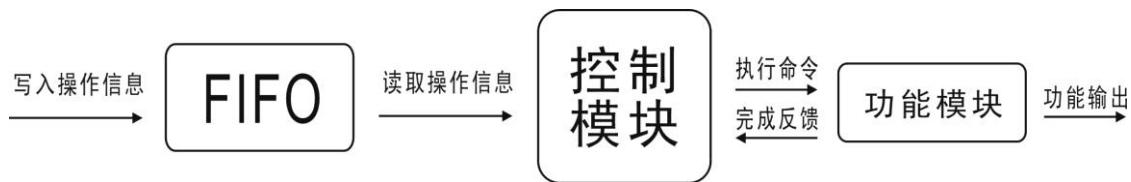
如果从另“**管理学**”的角度去理解“**接口的独立性**”，读者可以把它看成“**部门**”。在现实中我们知道“**部门和部门之间**”都是独立的，而且每一个部门都是有自己的内部操作。那么，如果要把这个观点放入“**蜂鸣器的封装**”里边，又应该如何实现呢？



在这里我们就需要用到“**FIFO**”。顾名思义 FIFO 就是“**先入先读**”的意思。但是在宏观来看 FIFO 是双向口的 RAM，FIFO 可分为两方，“**左方是写**”和“**右方是读**”。然后经过一些内部加工，读和写可以在同时发生。

FIFO 被需要是有目的的。我们重新复习一下仿顺序操作的概念。当某一个下层模块被使能时（Start\_Sig 等于 1），上一层模块就必须等待下层模块直到完成工作，才能执行下一个操作。为了避免上述的内容，我们必须借用 FIFO 的力量。我们可以尝试这样想“**如果我把操作信息全部缓冲到 FIFO 里的话 ...**”，那么上一层的模块可以将全部操作信息缓冲到 FIFO 里面，就可以摆脱“**反馈信息**”的“**束缚**”。这话怎么讲呢？

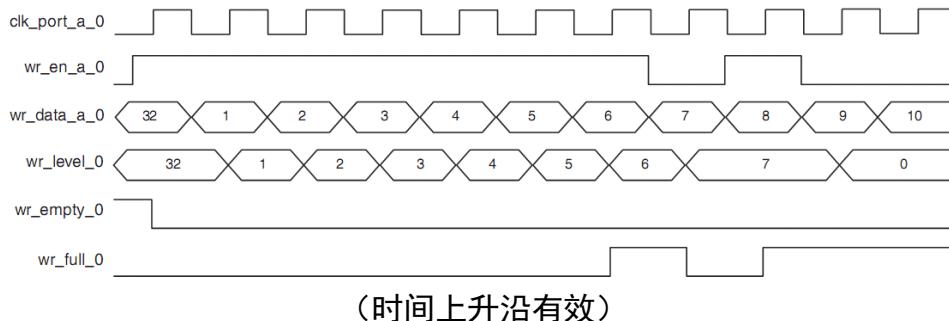
每当下一层模块完成内部的工作以后，可以直接从 FIFO 里面读取操作信息，用不着缠着上一层模块，等待反馈给予反应然后下达新的操作指令。



如果以“图形”来表示，那么结果会是如图上。上图大致的操作如下：

- (一) 首先上一层模块可以往 FIFO 里边写入大量的操作信息。然后上一层模块可以执行其他的操作，而不必在乎“反馈信息”。
- (二) 当 FIFO 里面存在信息，控制模块会从 FIFO 里面读取信息。信息被过滤以后，转换为执行命令，故启动功能模块。
- (三) 功能模很快便开始工作，直到工作结束，并且反馈完成信息给控制模块。
- (四) 当控制模块接收到从功能模块反馈回来的完成信息，会再度从 FIFO 读取信息，重复上述一样的动作，直到 FIFO 里面的信息全部读取完毕。

估计笔者们都对 FIFO 不怎么熟悉吧（笔者也是），那么我们就稍微的来理解一下 FIFO 的时序和操作（下面的内容是以接近实际 FIFO 芯片作为例子）：

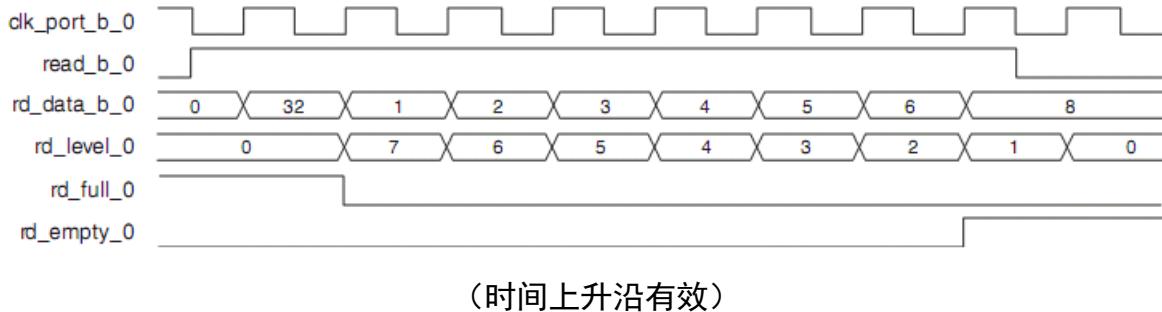


上图是 8 个深度的 FIFO。在 FIFO 初始化的时候 `wr_en_a_0` 拉高，然而 `Wr_data_a_0` 和 `wr_level_0` 都呈现“32”的信息。但是在第二个时钟的时候，信息 1 就被存入深度 1。第三个时钟信息 2 存入深度 2。直到第七个时钟，当写入信息 6 到深度 6 的时候，FIFO 的状态反应出“忙”，故此拉高 `wr_full_0` 信号，在同一个时间 `we_en_a_0` 被拉低。在第九个时钟 `wr_en_a_0` 拉高，信息 8 被写入深度 7。由于 FIFO 出现饱和状态，所以拉高 `wr_full_0` 信号。

在上述的内容中，我们发现在初始化的时候 FIFO 的深度 0 是用来预备初始化信息。然而在第九个时钟的时候，基本上 FIFO 已经饱和了。因此 FIFO 没有理由再往里面写入信息了，所以拉高 `wr_en_a_0` 也没有任何意义。

// wr\_en 写使能, wr\_data 写数据, wr\_full 写饱和信号。

下图是 FIFO 读数据的时序图:



在初始化的时候 rd\_data\_b\_0 呈现初始化信息“32”。在第二个时间中，从 FIFO 的深度 7 读出信息 1。在第三个时间中，从深度 6 读出信息 2。直到第八个时间从深度 1 读出信息 8 以后，FIFO 反应出“FIFO 没有信息了”，所以就拉高 rd\_empty\_0 信号。

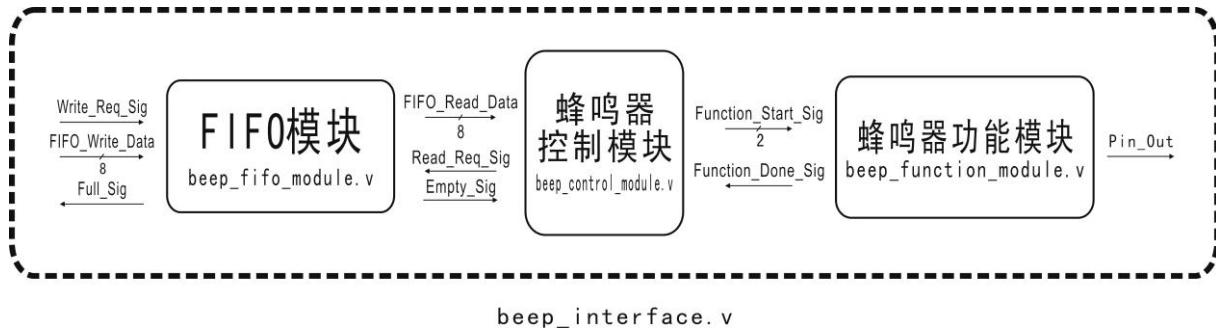
// rd\_en 读使能, rd\_data 读数据, rd\_empty 读空信号。

上述 FIFO 读写内容的大致“显示效果”如下:

T1	32							
T2	1	32						
T3	2	1	32					
T4	3	2	1	32				
T5	4	3	2	1	32			
T6	5	4	3	2	1	32		
T7	6	5	4	3	2	1	32	
T8	8	6	5	4	3	2	1	32

T1		8	6	5	4	3	2	1	32
T2			8	6	5	4	3	2	1
T3				8	6	5	4	3	2
T4					8	6	5	4	3
T5						8	6	5	4
T6							8	6	5
T7								8	6
T8									8

在时序图上出现的信息 32 是不会在 Quartus II 生成的 IP 出现的，这一点请放心。  
笔者再强调一下，上面的内容是以接近实际 FIFO 芯片的时序作为蓝图而已。Quartus II 生成的 FIFO 还是自身建立的 FIFO，只要空间为满就拉高 Full\_Sig 反之只要空间为空就拉高 Empty\_Sig，不会出现什么初始值 32 之类的东西（复位的关系），这一点请记住。



上图 `beep_interface.v` 组合模块中：

- (一) FIFO 拥有 16 个深度。
- (二) 蜂鸣器控制模块的主要功能是从 FIFO 读入一个信息经 FIFO\_Read\_Data, 蜂鸣器控制模块, 根据从 FIFO 读取的信息来使能 “**蜂鸣器功能模块**”;

在这一章的实验中 “**蜂鸣器功能模块**” 只包含两个功能：就是产生 S 摩斯码和 O 摩斯码。所以 `Command_Sig` 或者 `Function_Start_Sig` 都是 2 个位宽。

<code>Function_Start_Sig = 2'b10</code>	产生 S 摩斯码
<code>Function_Start_Sig = 2'b01</code>	产生 O 模式吗

整个 `beep_interface.v` 说穿了就是 “**如何调用 FIFO**” 而已， FIFO 的信号有：

- (一) `Write_Req_Sig` 等价于 `write_en`；
- (二) `Read_Req_Sig` 等价于 `read_en`；
- (三) `Full_Sig` 等价于 `write_full`；
- (四) `Empty_Sig` 等价于 `read_empty`；
- (五) `FIFO_Write_Data` 等价于 `write_data`；
- (六) `FIFO_Read_Data` 等价于 `read_data`；

那么这个 `beep_interface.v` 具体是如何运作还是直接看代码好。

#### *beep\_function\_module.v*

```

1. module beep_function_module
2. (
3.     input CLK,
4.     input RSTn,
5.     input [1:0]Start_Sig,
```

```
6.      output Done_Sig,
7.      output Pin_Out
8.  );
9.
10.     /*************************************************************************/
11.
12.     parameter T1MS = 16'd49_999;
13.
14.     /*************************************************************************/
15.
16.     reg [15:0]Count1;
17.
18.     always @ ( posedge CLK or negedge RSTn )
19.         if( !RSTn )
20.             Count1 <= 16'd0;
21.         else if( Count1 == T1MS )
22.             Count1 <= 16'd0;
23.         else if( isCount )
24.             Count1 <= Count1 + 1'b1;
25.         else if( !isCount )
26.             Count1 <= 16'd0;
27.
28.     /*************************************************************************/
29.
30.     reg [9:0]Count_MS;
31.
32.     always @ ( posedge CLK or negedge RSTn )
33.         if( !RSTn )
34.             Count_MS <= 10'd0;
35.         else if( Count_MS == rTimes )
36.             Count_MS <= 10'd0;
37.         else if( Count1 == T1MS )
38.             Count_MS <= Count_MS + 1'b1;
39.
40.     /*************************************************************************/
41.
42.     reg [3:0]i;
43.     reg rPin_Out;
44.     reg [9:0]rTimes;
45.     reg isCount;
46.     reg isDone;
47.
48.     always @ ( posedge CLK or negedge RSTn )
49.         if( !RSTn )
```

## Verilog HDL 那些事儿 – 建模篇

```
50.          begin
51.              i <= 4'd0;
52.              rPin_Out <= 1'b0;
53.              rTimes <= 10'd1000; // 注意，应该初始一个“大”的值
54.              isCount <= 1'b0;
55.              isDone <= 1'b0;
56.          end
57.      else if( Start_Sig[1] ) // S
58.          case( i )
59.
60.              4'd0, 4'd2, 4'd4:
61.                  if( Count_MS == rTimes ) begin rPin_Out <= 1'b0; isCount <= 1'b0; i <= i + 1'b1; end
62.                  else begin isCount <= 1'b1; rPin_Out <= 1'b1; rTimes <= 10'd100; end
63.
64.              4'd1, 4'd3, 4'd5:
65.                  if( Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
66.                  else begin isCount <= 1'b1; rTimes <= 10'd50; end
67.
68.              4'd6:
69.                  begin isDone <= 1'b1; i <= 4'd7; end
70.
71.              4'd7:
72.                  begin isDone <= 1'b0; i <= 4'd0; end
73.
74.          endcase
75.      else if( Start_Sig[0] ) // O
76.          case( i )
77.
78.              4'd0, 4'd2, 4'd4:
79.                  if( Count_MS == rTimes ) begin rPin_Out <= 1'b0; isCount <= 1'b0; i <= i + 1'b1; end
80.                  else begin isCount <= 1'b1; rPin_Out <= 1'b1; rTimes <= 10'd400; end
81.
82.              4'd1, 4'd3, 4'd5:
83.                  if( Count_MS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
84.                  else begin isCount <= 1'b1; rTimes <= 10'd50; end
85.
86.              4'd6:
87.                  begin isDone <= 1'b1; i <= 4'd7; end
88.
89.              4'd7:
90.                  begin isDone <= 1'b0; i <= 4'd0; end
91.
92.          endcase
93.
```

```

94.      *****/
95.
96.      assign Done_Sig = isDone;
97.      assign Pin_Out = !rPin_Out;
98.
99.      *****/
100.
101. endmodule

```

第 5 行中的[1:0]Start\_Sig 表示了该功能模块包含了两个功能。10~40 行是 1ms 定时器到 1 秒计数器。57~74 行是 S 摩斯码的产生，反之 75~92 行是 O 摩斯码的产生。在 97 行，由于实际的蜂鸣器的控制被挂着 PNP 三极管（低电平有效），rPin\_Out 的输出必须取反。

*beep\_control\_module.v*

```

1.  module beep_control_module
2.  (
3.      input CLK,
4.      input RSTn,
5.
6.      input [7:0]FIFO_Read_Data,
7.
8.      input Empty_Sig,
9.      output Read_Req_Sig,
10.
11.     input Function_Done_Sig,
12.     output [1:0]Function_Start_Sig
13. );
14.
15. *****/
16.
17. reg [3:0]i;
18. reg [1:0]rCmd;
19. reg isRead;
20. reg [1:0]isStart; // [1]S [0]O
21.
22. always @ ( posedge CLK or negedge RSTn )
23.     if( !RSTn )
24.         begin
25.             i <= 4'd0;
26.             rCmd <= 2'b00;
27.             isRead <= 1'b0;

```

## Verilog HDL 那些事儿 – 建模篇

```
28.           isStart <= 2'b0;
29.       end
30.   else
31.       case( i )
32.
33.           0:
34.               if( !Empty_Sig ) i <= i + 1'b1;
35.
36.           1:
37.               begin isRead <= 1'b1; i <= i + 1'b1; end
38.
39.           2:
40.               begin isRead <= 1'b0; i <= i + 1'b1; end
41.
42.           3:
43.               begin
44.
45.                   if( FIFO_Read_Data == 8'h1B ) rCmd <= 2'b10;      //s
46.                   else if( FIFO_Read_Data == 8'h44 ) rCmd <= 2'b01; // o
47.                   else rCmd <= 2'b00;
48.
49.                   i <= i + 1'b1;
50.               end
51.
52.           4:
53.               if( rCmd == 2'b00 ) i <= 4'd0;
54.               else i <= i + 1'b1;
55.
56.           5:
57.               if( Function_Done_Sig ) begin rCmd <= 2'b00; isStart <= 2'b00; i <= 4'd0; end
58.               else isStart <= rCmd;
59.
60.
61.       endcase
62.
63.   /*************************************************************************/
64.
65.   assign Read_Req_Sig = isRead;
66.   assign Function_Start_Sig = isStart;
67.
68.   /*************************************************************************/
69.
70.
71. endmodule
```

31~61 行是该控制模块的核心部分。在 33 行，步骤 0 先判断 FIFO 是否为空，如果 FIFO 不是为空 Empty\_Sig 信号就被拉低。如果 FIFO 不为空 if 条件成立，i 递增以示下一步骤。

36~40 行是读 FIFO 的操作，在这里笔者先简单的介绍一下。在前面的 FIFO 时序图中，当 FIFO 为读状态，如果 Read\_Req\_Sig ( read\_en ) 不拉高，就无法把数据读出来。FIFO 读数据是根据每一个时钟的上升沿，如果 Read\_Req\_Sig ( read\_en ) 拉高，那么在这一个上升沿中，数据就会被读出来。换句话说，我们可以利用 Read\_Req\_Sig 来充当读取数据的“锁匙”。

```
1: begin isRead <= 1'b1; i <= i + 1'b1; end
2: begin isRead <= 1'b0; i <= i + 1'b1; end
```

如上的步骤 1 至 2 表示从 FIFO 读取“一个深度的数据”。( isRead 驱动着 Read\_Req\_Sig - 65 行。) 在 36~40 行正是如上的操作。在 37 行将 isRead 拉高，从 FIFO 读取数据，然后在 40 行将 isRead 拉低，那么一个简单的 FIFO 读操作就完成了。

在步骤 3 (42~50 行) SOS 命令 的加码操作。如果从 FIFO 读出的信息是 8'h1B 那么它会被加码为 2'b10，亦即执行“S 摩斯码产生”的命令 (45 行)。反之从 FIFO 读出的信息是 8'h44 那么它会被加码为 2'b01，亦即执行“0 摆斯码产生”的命令(46 行)。其余的信息都加码为 2'b00 - 没有操作 (47 行)。最后 i 递增以示下一个步骤 (49 行)。

步骤 4 (52-54 行)先判断 rCmd 的值是否为 8'h00，如果 rCmd 的值为 8'h00，表示无效命令，然后步骤 i 清零，返回步骤 0 (53 行)。如果 rCmd 的值不是 8'h00 的话，i 递增以示下一步骤 (54 行)。

在步骤 5，56~58 行是使能 beep\_function\_module.v 的工作。isStart 是作为 Function\_Start\_Sig 的驱动 (58 行)，isStart 赋予 rCmd 的值，。接下来的工作就和“[仿顺序操作](#)”一样，直到下一层模块反馈完成信号，isStart 和 rCmd 会被清零 (57 行)。i 也会清零然后返回步骤 0。

*beep\_interface.v*

```
1. module beep_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_Req_Sig,
7.     input [7:0]FIFO_Write_Data,
8.     output Full_Sig,
9.
```

```
10.      output Pin_Out
11. );
12.
13.      ****
14.
15.      wire [7:0]FIFO_Read_Data;
16.      wire Empty_Sig;
17.
18.      beep_fifo_module U1
19. (
20.          .clock( CLK ),
21.          .wrreq( Write_Req_Sig ),    // input - from top
22.          .data( FIFO_Write_Data ), // input - from top
23.          .full( Full_Sig ),       // output - to top
24.          .rdreq( Read_Req_Sig ),   // input - from U3
25.          .q( FIFO_Read_Data ),    // output - to U2 U3
26.          .empty( Empty_Sig )      // output - to U3
27. );
28.
29.      ****
30.
31.      wire Read_Req_Sig;
32.      wire [1:0]Function_Start_Sig;
33.
34.      beep_control_module U3
35. (
36.          .CLK( CLK ),
37.          .RSTn( RSTn ),
38.          .FIFO_Read_Data( FIFO_Read_Data ),    // input - to U2 From U1
39.          .Empty_Sig( Empty_Sig ),               // input - from U1
40.          .Read_Req_Sig( Read_Req_Sig ),        // output - to U1
41.          .Function_Done_Sig( Function_Done_Sig ), // input - from U4
42.          .Function_Start_Sig( Function_Start_Sig ) // output - to U4
43. );
44.
45.      ****
46.
47.      wire Function_Done_Sig;
48.
49.      beep_function_module U4
50. (
51.          .CLK( CLK ),
52.          .RSTn( RSTn ),
53.          .Start_Sig( Function_Start_Sig ),     // input - from U3
```

```

54.      .Done_Sig( Function_Done_Sig ),      // output- to U3
55.      .Pin_Out( Pin_Out )                  // output - to top
56. );
57.
58. ****
59.
60. endmodule

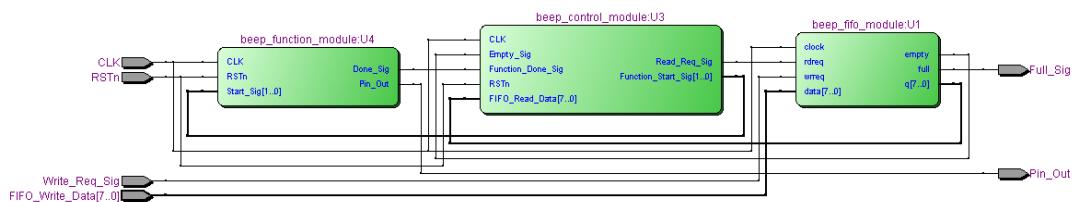
```

beep\_interface.v 组合模块的结果和“图形”一样。自己看着办吧。

实验十六说明:

实验十六的重点，就是“[如何对 FIFO 读取](#)”而已。其余的部分都是以往实验的复习。

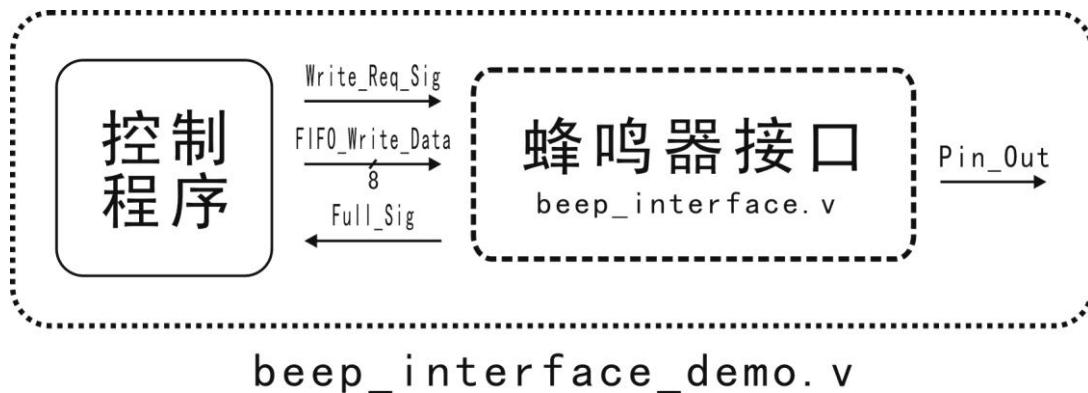
完成扩展图:



实验十六结论:

在某种程度来说，实验十六的蜂鸣器接口，标准操作信息是“[键盘通码](#)”。

## 实验十六演示：



在上图的组合模块中，控制程序对蜂鸣器接口写入“8'h1B, 8'h44, 8'h1B”这三个信息，然后就停止操作。接着，蜂鸣器接口却会发出“**SOS 信号**”实际上对蜂鸣器接口的调用就是“[如何对 FIFO 写入信息](#)”这一回事。

`beep_interface_demo.v`

```

1. module beep_interface_demo
2. (
3.     input CLK,
4.     input RSTn,
5.     output Pin_Out
6. );
7.
8.     ****
9.
10.    reg [3:0]i;
11.    reg isWrite;
12.    reg [7:0]rData;
13.
14.    always @ ( posedge CLK or negedge RSTn )
15.        if( !RSTn )
16.            begin
17.                i <= 4'd0;
18.                isWrite <= 1'b0;
19.                rData <= 8'd0;
20.            end
21.        else
22.            case( i )

```

```
23.  
24.      ****  
25.  
26.      /*  
27.  
28.      0 :  
29.      if( !Full_Sig ) begin isWrite <= 1'b1; rData <= 8'h1B; i <= i + 1'b1; end  
30.  
31.      1:  
32.      if( !Full_Sig ) begin rData <= 8'h44; i <= i + 1'b1; end  
33.  
34.      2:  
35.      if( !Full_Sig ) begin rData <= 8'h1B; i <= i + 1'b1; end  
36.  
37.      3:  
38.      begin isWrite <= 1'b0; i <= 4'd3; end  
39.      */  
40.  
41.      ****  
42.  
43.      0:  
44.      if( !Full_Sig ) begin isWrite <= 1'b1; rData <= 8'h1B; i <= i + 1'b1; end  
45.  
46.      1:  
47.      begin isWrite <= 1'b0; i <= i + 1'b1; end  
48.  
49.      2:  
50.      if( !Full_Sig ) begin isWrite <= 1'b1; rData <= 8'h44; i <= i + 1'b1; end  
51.  
52.      3:  
53.      begin isWrite <= 1'b0; i <= i + 1'b1; end  
54.  
55.      4:  
56.      if( !Full_Sig ) begin isWrite <= 1'b1; rData <= 8'h1B; i <= i + 1'b1; end  
57.  
58.      5:  
59.      begin isWrite <= 1'b0; i <= 4'd5; end  
60.  
61.      ****  
62.  
63.      endcase  
64.  
65.      ****  
66.
```

```
67.     wire Full_Sig;
68.
69.     beep_interface U1
70.     (
71.         .CLK( CLK ),
72.         .RSTn( RSTn ),
73.         .Write_Req_Sig( isWrite ),
74.         .FIFO_Write_Data( rData ),
75.         .Full_Sig( Full_Sig ),
76.         .Pin_Out( Pin_Out )
77.     );
78.
79.     /*************************************************************************/
80.
81. endmodule
```

这份源码有分为两种对 FIFO 写入的办法。

### 办法一：

在拉高 isWrite 的三个时间内，同时写入 3 份信息。然后再追加写入一份 8'h00 以使得达到强制挤出的效果。写完后拉低 isWrite，以示一次性的写入操作已经结束 (26~39 行)。

### 办法二：

办法比较傻瓜，但是解读性却很好。把 isWrite 当着写入数据的钥匙。在拉高 isWrite 的同时写入数据，然后在下一个步骤拉低 isWrite 以示一次性的写数据操作已经完成。(43~59 行)。

如果给笔者选择，笔者会选择办法二。原因是办法二比较 “[和蔼可亲](#)”。此外还有一个重点，就是 Full\_Sig 这个信号是反馈出 beep\_interface.v 的 FIFO 空间状态。如果 Full\_Sig 被拉高，亦即 beep\_interface.v 中的 FIFO 全部 16 个深度 空间已经被写满了。所以每一次要向 beep\_interface.v 写入信息的时候，必须判断 Full\_Sig 是否是被拉高的状态 (44, 50, 56 行)。

### 实验十六演示说明：

在这个演示中，我们向 FIFO 写入 3 个信息，亦即 SOS 的通码 (S - 8'h1B 和 O - 8'h44)。当下载程序在黑金开发板后就会听到 SOS 的信息。

实验十六演示结论：

在这一章中我们明白到，“**接口**”的定义除了“**最后的工程**”以外还有“**独立**”这一个定义。实际上 FIFO 是用来缓冲两个时间域不同的访问，在某种程度上“**缓冲数据**”的作用有如“仓库”。所以我们可以利用这个“**仓库**”作为“**接口的输入**”，以致“**接口**”有独立的性质。

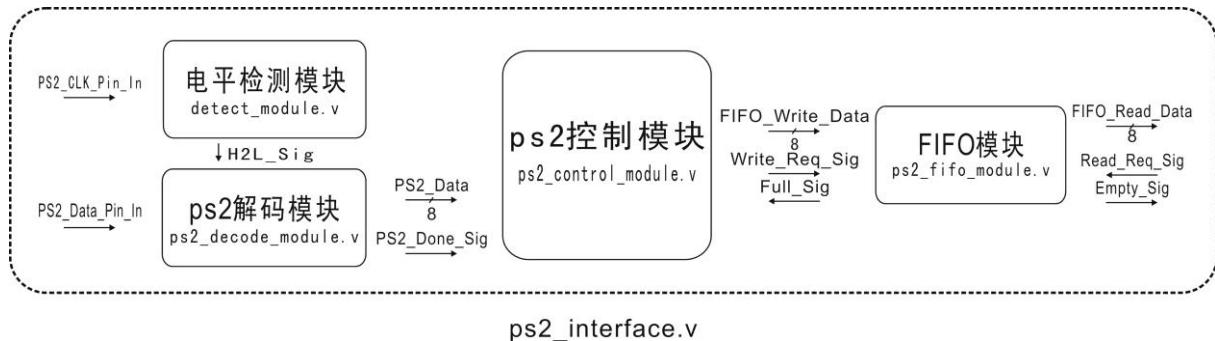
所以呀，当上一层模块调用某个“**接口**”时，只要把“**信息**”写入“**仓库**”即可，上一层模块用不着与该接口“**互动**”而被“**束缚着**”。该接口如果发现“**仓库**”有信息，就处理信息，如果“**仓库**”里没有信息就作罢。

## 5.4 实验十七：PS2 封装

有关 PS2 驱动（解码）的实验我们已经在实验八完成了，这一章我们要将 PS2 封装。在这里笔者稍微重复一下“[封装（接口）的定义](#)”：

- (一) 最后的工程。
- (二) 使模块独立。

在 5.3 章中，我们对蜂鸣器的封装中调用了 FIFO 作为信息输入的缓冲，而使得蜂鸣器接口独立于上一层模块。相反的这一章，我们要在 PS2 封装中，引入 FIFO 作为信息输出的缓冲。



上图是 `ps2_interface.v` - PS2 接口，PS2 控制模块的左方是实验八“[PS2 解码](#)”中，出现的电平检测模块和 PS2 解码模块。反之，PS2 控制模块的右方是 FIFO 模块。左方是对“[PS2 的一帧数据](#)”解码，右方是对“[解码过后的数据](#)”进行缓冲。然而，PS2 控制模块在中间协调。(FIFO 的深度是 16)

`detect_module.v`

```

1. module detect_module
2. (
3.     CLK, RSTn,
4.     PS2_CLK_Pin_In,
5.     H2L_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input PS2_CLK_Pin_In;

```

```

11.    output H2L_Sig;
12.
13.    ****
14.
15.    reg H2L_F1;
16.    reg H2L_F2;
17.
18.    always @ ( posedge CLK or negedge RSTn )
19.        if( !RSTn )
20.            begin
21.                H2L_F1 <= 1'b1;
22.                H2L_F2 <= 1'b1;
23.            end
24.        else
25.            begin
26.                H2L_F1 <= PS2_CLK_Pin_In;
27.                H2L_F2 <= H2L_F1;
28.            end
29.
30.    ****
31.
32.    assign H2L_Sig = H2L_F2 & !H2L_F1;
33.
34.    ****
35.
36.
37. endmodule

```

*ps2\_decode\_module.v*

```

1. module ps2_decode_module
2. (
3.     CLK, RSTn,
4.     H2L_Sig, PS2_Data_Pin_In,
5.     PS2_Data, PS2_Done_Sig
6. );
7.
8.     input CLK;
9.     input RSTn;
10.    input H2L_Sig;
11.    input PS2_Data_Pin_In;
12.    output [7:0]PS2_Data;

```

## Verilog HDL 那些事儿 – 建模篇

```
13.      output PS2_Done_Sig;
14.
15.      ****
16.
17.      reg [7:0]rData;
18.      reg [4:0]i;
19.      reg isShift;
20.      reg isDone;
21.
22.
23.      always @ ( posedge CLK or negedge RSTn )
24.          if( !RSTn )
25.              begin
26.                  rData <= 8'd0;
27.                  i <= 5'd0;
28.                  isDone <= 1'b0;
29.              end
30.          else
31.              case( i )
32.
33.                  5'd0:
34.                      if( H2L_Sig ) i <= i + 1'b1;
35.
36.                  4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8:
37.                      if( H2L_Sig ) begin i <= i + 1'b1; rData[ i-1 ] <= PS2_Data_Pin_In; end
38.
39.                  5'd9, 5'd10:
40.                      if( H2L_Sig ) i <= i + 1'b1;
41.
42.                  5'd11:
43.                      if( rData == 8'hf0 ) i <= 5'd12;
44.                      else i <= 5'd23;
45.
46.                  5'd12, 5'd13, 5'd14, 5'd15, 5'd16, 5'd17, 5'd18, 5'd19, 5'd20, 5'd21, 5'd22:
47.                      if( H2L_Sig ) i <= i + 1'b1;
48.
49.                  5'd23:
50.                      begin i <= i + 1'b1; isDone <= 1'b1; end
51.
52.                  5'd24:
53.                      begin i <= 5'd0; isDone <= 1'b0; end
54.
55.              endcase
56.
```

```
57.      ****
58.
59.      assign PS2_Data = rData;
60.      assign PS2_Done_Sig = isDone;
61.
62.      ****
63.
64. endmodule
```

*ps2\_control\_module.v*

```
1. module ps2_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.     input PS2_Done_Sig,
6.     input [7:0]PS2_Data,
7.
8.     input Full_Sig,
9.     output Write_Req_Sig,
10.    output [7:0]FIFO_Write_Data
11. );
12.
13.
14. ****
15.
16. reg [3:0]i;
17. reg isReq;
18.
19. always @ ( posedge CLK or negedge RSTn )
20.     if( !RSTn )
21.         begin
22.             i <= 4'd0;
23.             isReq <= 1'b0;
24.         end
25.     else
26.         case( i )
27.
28.             0:
29.                 if( PS2_Done_Sig && !Full_Sig ) i <= i + 1'b1;
30.
31.             1:
```

```
32.           begin isReq <= 1'b1; i <= i + 1'b1; end
33.
34.           2:
35.           begin isReq <= 1'b0; i <= 4'd0; end
36.
37.       endcase
38.
39.   /*************************************************************************/
40.
41.   assign FIFO_Write_Data = PS2_Data;
42.   assign Write_Req_Sig = isReq;
43.
44.   /*************************************************************************/
45.
46. endmodule
```

ps2\_control\_module.v 这个控制模块比较简单，核心部分在 26~37 行。在 29 行，如果 PS2\_Done\_Sig 信号产生高脉冲而且 Full\_Sig 拉低（FIFO 存在空闲空间），i 递增以示下一个步骤。

在这里提醒一下，PS2\_Data 输入是直接驱动 FIFO\_Write\_Data 输出（41 行）。

在 32 行 isReq 标志寄存器拉高（isReq 该寄存器是用于驱动 Write\_Req\_Sig，亦即 FIFO 写请求信号-42 行），i 递增以示下一个步骤。在 35 行 isReq 拉低，以示一次性的 FIFO 的写操作已经结束。i 赋值为 0 以示重新等待下一次的写操作。

注意：PS2\_Data 输入信号，直接驱动 FIFO\_Write\_Data（41 行）。

*ps2\_interface.v*

```
1. module ps2_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input PS2_CLK_Pin_In,
7.     input PS2_Data_Pin_In,
8.
9.     input Read_Req_Sig,
10.    output Empty_Sig,
11.    output [7:0]FIFO_Read_Data
12. );
```

```
13.
14.    *****/
15.
16.    wire H2L_Sig;
17.
18.    detect_module U1
19.    (
20.        .CLK( CLK ),
21.        .RSTn( RSTn ),
22.        .PS2_CLK_Pin_In( PS2_CLK_Pin_In ), // input - from top
23.        .H2L_Sig( H2L_Sig )           // output - to U2
24.    );
25.
26.    *****/
27.
28.    wire [7:0]PS2_Data;
29.    wire PS2_Done_Sig;
30.
31.    ps2_decode_module U2
32.    (
33.        .CLK( CLK ),
34.        .RSTn( RSTn ),
35.        .H2L_Sig( H2L_Sig ),           // input - from U1
36.        .PS2_Data_Pin_In( PS2_Data_Pin_In ), // Input - from top
37.        .PS2_Data( PS2_Data ),          // output - to U3
38.        .PS2_Done_Sig( PS2_Done_Sig ) // output - to U3
39.    );
40.
41.    *****/
42.
43.    wire Write_Req_Sig;
44.    wire [7:0]FIFO_Write_Data;
45.
46.    ps2_control_module U3
47.    (
48.        .CLK( CLK ),
49.        .RSTn( RSTn ),
50.        .PS2_Done_Sig( PS2_Done_Sig ), // input - from U2
51.        .PS2_Data( PS2_Data ),          // input - from U2
52.        .Full_Sig( Full_Sig ),          // input - to U4
53.        .Write_Req_Sig( Write_Req_Sig ), // output - to U4
54.        .FIFO_Write_Data( FIFO_Write_Data ) // output - to U4
55.    );
56.
```

```

57.      *****/
58.
59.      wire Full_Sig;
60.
61.      ps2_fifo_module U4
62.      (
63.          .clock( CLK ),
64.          .data( FIFO_Write_Data ),      // input - from U3
65.          .wrreq( Write_Req_Sig ),      // input - from U3
66.          .full( Full_Sig ),          // output - to U3
67.          .rdreq( Read_Req_Sig ),      // input - from top
68.          .q( FIFO_Read_Data ),       // output - to top
69.          .empty( Empty_Sig )         // output - to top
70.      );
71.
72.      *****/
73.
74.
75. endmodule

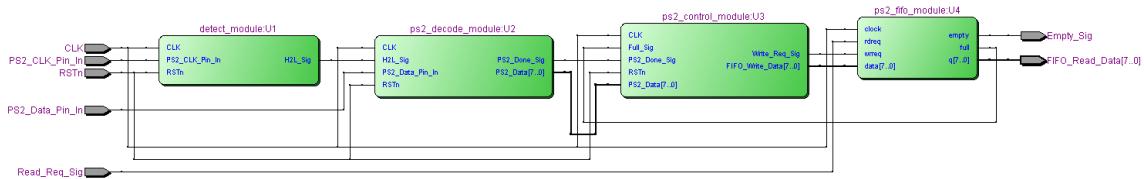
```

ps2\_interface.v 这个组合模块，基本上和“[图形](#)”一样。

### 实验十七说明：

这个实验比较简单，主要是基于实验八再加入 ps 控制模块和 FIFO 模块，然后封装而成。封装过后的 PS2 接口，我们只要考虑如何将“[从 PS2 接口的 FIFO 读取数据](#)”而已。

完成后的扩展图：



### 实验十七结论：

实验十七和实验十六比较，一个是将输出数据缓冲至 FIFO，另外一个是将输入数据缓冲至 FIFO。

## 实验十七演示：



在这个演示中，主要是演示如何调用 ps2\_interface.v。在 ps2\_interface\_demo.v 中，除了实例化 ps2\_interface.v 以外，还添写了对 ps2\_interface.v 调用的控制程序。最后该控制程序将读取到的数据的“**前四位**”输出至 LED 资源。

### *ps2\_interface\_demo.v*

```

1. module ps2_interface_demo
2. (
3.     input CLK,
4.     input RSTn,
5.     input PS2_CLK_Pin_In,
6.     input PS2_Data_Pin_In,
7.     output [3:0]LED
8. );
9.
10. ****
11.
12. reg [1:0]i;
13. reg [3:0]rLED;
14. reg isRead;
15.
16. always @ ( posedge CLK or negedge RSTn )
17.     if( !RSTn )
18.         begin
19.             i <= 2'd0;
20.             rLED <= 4'd0;
21.             isRead <= 1'b0;
22.         end
23.     else
24.         case( i )
25.

```

## Verilog HDL 那些事儿 – 建模篇

```
26.          0:  
27.              if( !Empty_Sig ) i <= i + 1'b1;  
28.  
29.          1:  
30.              begin isRead <= 1'b1; i <= i + 1'b1; end  
31.  
32.          2:  
33.              begin isRead <= 1'b0; i <= i + 1'b1; end  
34.  
35.          3:  
36.              if( FIFO_Read_Data != 8'hf0 ) begin rLED <= FIFO_Read_Data[3:0]; i <= 2'd0; end  
37.              else i <= 2'd0;  
38.  
39.      endcase  
40.  
41.  /**************************************************************************/  
42.  
43.  wire Empty_Sig;  
44.  wire [7:0]FIFO_Read_Data;  
45.  
46.  ps2_interface U1  
47.  (  
48.      .CLK( CLK ),  
49.      .RSTn( RSTn ),  
50.      .PS2_CLK_Pin_In( PS2_CLK_Pin_In ),  
51.      .PS2_Data_Pin_In( PS2_Data_Pin_In ),  
52.      .Read_Req_Sig( isRead ),  
53.      .Empty_Sig( Empty_Sig ),  
54.      .FIFO_Read_Data( FIFO_Read_Data )  
55.  );  
56.  
57.  /**************************************************************************/  
58.  
59.  assign LED = rLED;  
60.  
61.  /**************************************************************************/  
62.  
63. endmodule
```

在 27 行中，先判断 FIFO 是否为空，如果 FIFO 不为空的话就拉高 isRead。步骤 1~2（29~33 行）是从 FIFO 出一个数据至 FIFO\_Read\_Data。在步骤 3（35~37 行），在 36 行的 if 条件是用来判断从 FIFO 读取的数据是不是断码（8'hF0）？如果不是，将通码的低四位寄存与 rLED，i 清零然后返回步骤 0。否则 i 清零然后返回步骤 0。

实验十七演示说明:

PS2 接口的调用。

实验十七演示结论:

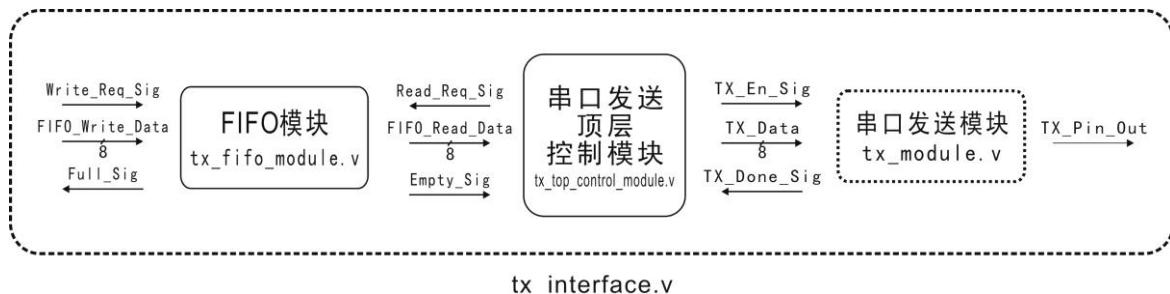
该演示比较简单，没有什么好补充的，但是从演示结果我们可以知道一个事实。每当我们按下 A 按键然后又释放 A 按键的时候，LED 资源会显示 A 按键的通码，而不是断码 8'hf0（别忘了 ps2\_decode\_module.v 是断码的 0xf0 和通码都吃，但是不吃断码之后的通码）。但是在 36 行，if 条件把断码的 0xf0 过滤了。

## 5.5 实验十八：串口发送|接收 封装

在 5.3 章的蜂鸣器封装实验中，介绍了 FIFO 在封装中用于缓冲输入信息，从而使得该接口可以独立于上一层模块。相反的，在 5.4 章的 PS2 封装试验中，FIFO 用于输出信息的缓冲。

本章实验基本上和上两章实验很类似，FIFO 都是用于输入缓冲和输出缓冲，从而使得接口独立于上一层模块。在这里笔者再重复一下：当我们在练习单片机的时候，常常会误会，串口发送和串口接收都是基于在一起，实际上它们是各自独立的。所以串口发送和串口接收必须拥有各自的接口。

**串口发送接口：**



tx\_interface.v 组合模块中的“**串口发送模块**”是在以前的实验中完成的。此外在该组合模块中，最左方的 FIFO 模块（16 个深度）是用于缓冲输入信息。然而中间的控制模块，是用于“协调控制” FIFO 模块和串口发送模块(哎~名字很囧)。

“**图形**”已经将 tx\_interface.v 表达的非常直接了，具体的操作我们还是直接看源码：

*tx\_top\_control\_module.v*

```

1. module tx_top_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Empty_Sig,
7.     input [7:0]FIFO_Read_Data,
8.     output Read_Req_Sig,
9.

```

```
10.     input TX_Done_Sig,
11.     output [7:0]TX_Data,
12.     output TX_En_Sig
13. );
14.
15. ****
16.
17. reg [1:0]i;
18. reg isRead;
19. reg isTX;
20.
21. always @ ( posedge CLK or negedge RSTn )
22.     if( !RSTn )
23.         begin
24.             i <= 2'd0;
25.             isRead <= 1'b0;
26.             isTX <= 1'b0;
27.         end
28.     else
29.         case( i )
30.
31.         0:
32.             if( !Empty_Sig ) i <= i + 1'b1;
33.
34.         1:
35.             begin isRead <= 1'b1; i <= i + 1'b1; end
36.
37.         2:
38.             begin isRead <= 1'b0; i <= i + 1'b1; end
39.
40.         3:
41.             if( TX_Done_Sig ) begin isTX <= 1'b0; i <= 2'd0; end
42.             else isTX <= 1'b1;
43.
44.
45.         endcase
46.
47. ****
48.
49. assign Read_Req_Sig = isRead;
50. assign TX_En_Sig = isTX;
51. assign TX_Data = FIFO_Read_Data;
52.
53. ****
```

```
54.  
55.  
56. endmodule
```

第 21~45 行是该控制模块的核心部分。当 i 步骤等于 0 的时候（31 行），如果 FIFO 模块不为空的话，i 就递增以示下一个步骤。

在 34~38 行，isRead 在步骤 1 被拉高，然后在步骤 2 又被拉低，该意思是从 FIFO 读取一个深度的数据。有一点，必须注意的是，在 11 行定义的 TX\_Data 输出是直接由 FIFO\_Read\_Data（7 行）驱动（51 行）。

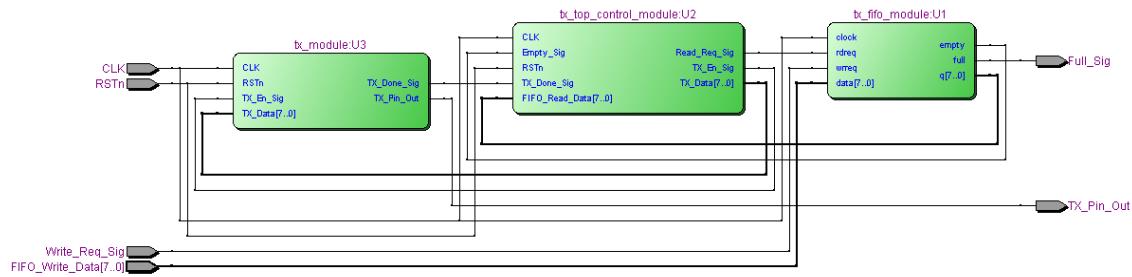
在 40 行，步骤 3 将 isTX 标志寄存器拉高，以示使能（启动）串口发送模块（42 行）。当串口发送模块完成一次性的发送操作后，会反馈一个完成信号，然后 if 条件（41 行）会成立，isTX 被拉低，和 i 被赋予 0，以示重复另一次发送操作。

*tx\_interface.v*

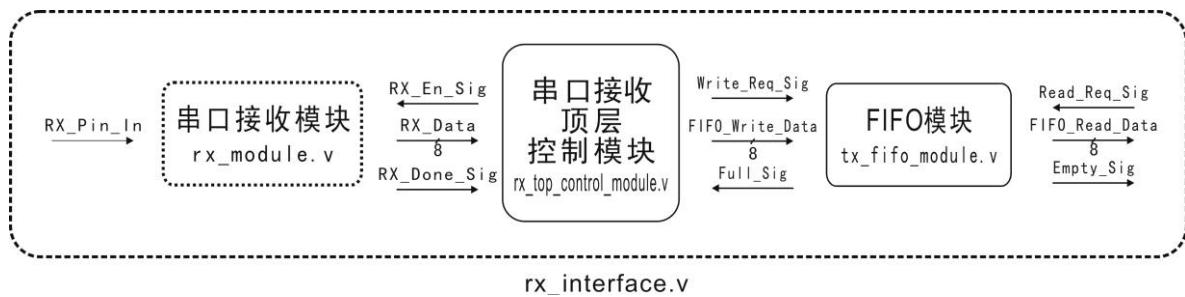
```
1. module tx_interface  
2. (  
3.     input CLK,  
4.     input RSTn,  
5.  
6.     input Write_Req_Sig,  
7.     input [7:0]FIFO_Write_Data,  
8.     output Full_Sig,  
9.  
10.    output TX_Pin_Out  
11. );  
12.  
13.    /**************************************************************************/  
14.  
15.    wire [7:0]FIFO_Read_Data;  
16.    wire Empty_Sig;  
17.  
18.    tx_fifo_module U1  
19.    (  
20.        .clock( CLK ),  
21.        .wrreq( Write_Req_Sig ), // input - from top  
22.        .data( FIFO_Write_Data ), // input - from top  
23.        .full( Full_Sig ), // output - to top  
24.        .rdreq( Read_Req_Sig ), // input - from U2  
25.        .q( FIFO_Read_Data ), // output - to U2  
26.        .empty( Empty_Sig ) // output - to U2
```

```
27. );
28.
29. *****/
30.
31. wire Read_Req_Sig;
32. wire [7:0]TX_Data;
33. wire TX_En_Sig;
34.
35. tx_top_control_module U2
36. (
37.     .CLK( CLK ),
38.     .RSTn( RSTn ),
39.     .Empty_Sig( Empty_Sig ),           // input - from U1
40.     .FIFO_Read_Data( FIFO_Read_Data ), // input - from U1
41.     .Read_Req_Sig( Read_Req_Sig ),     // output - to U1
42.     .TX_Done_Sig( TX_Done_Sig ),      // input - from U3
43.     .TX_Data( TX_Data ),             // output - to U3
44.     .TX_En_Sig( TX_En_Sig )          // output - to U3
45. );
46.
47. *****/
48.
49. wire TX_Done_Sig;
50.
51. tx_module U3
52. (
53.     .CLK( CLK ),
54.     .RSTn( RSTn ),
55.     .TX_Data( TX_Data ),           // input - from U2
56.     .TX_En_Sig( TX_En_Sig ),       // input - from U2
57.     .TX_Done_Sig( TX_Done_Sig ),   // output - to U2
58.     .TX_Pin_Out( TX_Pin_Out )     // output - to top
59. );
60.
61. *****/
62.
63.
64. endmodule
```

完成后的扩展图：



## 串口接收接口：



上图是 rx\_interface.v 的组合模块。串口接收模块是基于实验十建模而成的。 rx\_interface.v 的功能大致上如下：

该控制模块一开始就使能串口接收模块，当串口接收模块完成一次性的读取操作以后，就会反馈数据 RX\_Data 和完成信号 RX\_Done\_Sig。当串口接收顶层控制模块（名字依然很囧）接收到串口接收模块反馈的完成信号，就会不使能串口接收模块。然后该控制模块就会将 经 RX\_Data 反馈回来的数据缓冲致 FIFO 模块。

*rx\_top\_control\_module.v*

```

1. module rx_top_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input RX_Done_Sig,
7.     input [7:0]RX_Data,
8.     output RX_En_Sig,
9.
10.    input Full_Sig,
11.    output Write_Req_Sig,
```

```
12.     output [7:0]FIFO_Write_Data
13. );
14.
15. ****
16.
17.     reg [1:0]i;
18.     reg isWrite;
19.     reg isRX;
20.
21.     always @ ( posedge CLK or negedge RSTn )
22.         if( !RSTn )
23.             begin
24.                 i <= 2'd0;
25.                 isWrite <= 1'b0;
26.                 isRX <= 1'b0;
27.             end
28.         else
29.             case( i )
30.
31.                 0:
32.                     if( RX_Done_Sig ) begin isRX <= 1'b0; i <= i + 1'b1; end
33.                     else isRX <= 1'b1;
34.
35.                 1:
36.                     if( !Full_Sig ) i <= i + 1'b1;
37.
38.                 2:
39.                     begin isWrite <= 1'b1; i <= i + 1'b1; end
40.
41.                 3:
42.                     begin isWrite <= 1'b0; i <= 2'd0; end
43.
44.             endcase
45.
46. ****
47.
48.     assign RX_En_Sig = isRX;
49.     assign Write_Req_Sig = isWrite;
50.     assign FIFO_Write_Data = RX_Data;
51.
52. ****
53.
54. endmodule
```

在 21~44 行是该控制模块的核心功能。在步骤 0 (31) 行, isRX 被拉高, 此时串口接收模块就被使能 (32 行)。当串口接收模块完成一次性的读取操作, 串口接收模块就会反馈一个完成信号。然而在 32 行 if 条件就会成立, isRX 被拉低, 然后 i 递增以示下一个步骤。

在 36 行步骤 1, 如果 FIFO 不为满状态 (Full\_Sig 拉低), i 递增以示下一个步骤。

在 38~42 行, 在步骤 2, isWrite 被拉高, 然后在步骤 3, isWrite 被拉低。在这里需要注意一点, FIFO\_Write\_Data 信号直接由 RX\_Data 驱动。当串口接收模块反馈完成信号以后, RX\_Data 已经将数据就绪好在 FIFO\_Write\_Data 信号上。一旦 isWrite 被拉高又被拉低, 已经就绪的 RX\_Data 数据会被写入 FIFO。

当完成一次性的数据读取操作, i 会被赋予 0, 以示重新执行以上的所有步骤。

*rx\_interface.v*

```
1. module rx_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input RX_Pin_In,
7.
8.     input Read_Req_Sig,
9.     output [7:0]FIFO_Read_Data,
10.    output Empty_Sig
11. );
12.
13. ****
14.
15. wire [7:0]RX_Data;
16. wire RX_Done_Sig;
17.
18. rx_module U1
19. (
20.     .CLK( CLK ),
21.     .RSTn( RSTn ),
22.     .RX_Pin_In( RX_Pin_In ),      // input - from top
23.     .RX_En_Sig( RX_En_Sig ),    // input - from U2
24.     .RX_Data( RX_Data ),        // output - to U2
25.     .RX_Done_Sig( RX_Done_Sig ) // output - to U2
26. );
```

```

27.
28. *****/
29.
30. wire RX_En_Sig;
31. wire Write_Req_Sig;
32. wire [7:0]FIFO_Write_Data;
33.
34. rx_top_control_module U2
35. (
36.     .CLK( CLK ),
37.     .RSTn( RSTn ),
38.     .RX_Done_Sig( RX_Done_Sig ),      // input - from U1
39.     .RX_Data( RX_Data ),            // input - from U1
40.     .RX_En_Sig( RX_En_Sig ),        // output - to U1
41.     .Full_Sig( Full_Sig ),         // input - from U3
42.     .Write_Req_Sig( Write_Req_Sig ), // output - to U3
43.     .FIFO_Write_Data( FIFO_Write_Data ) // output - to U3
44. );
45.
46. *****/
47.
48. wire Full_Sig;
49.
50. rx_fifo_module U3
51. (
52.     .clock( CLK ),
53.     .wrreq( Write_Req_Sig ),      // input - from U2
54.     .data( FIFO_Write_Data ),    // input - from U2
55.     .full( Full_Sig ),          // output - to U2
56.     .rdreq( Read_Req_Sig ),     // input - from top
57.     .q( FIFO_Read_Data ),       // output - to top
58.     .empty( Empty_Sig )         // output - to top
59. );
60.
61. *****/
62.
63. endmodule

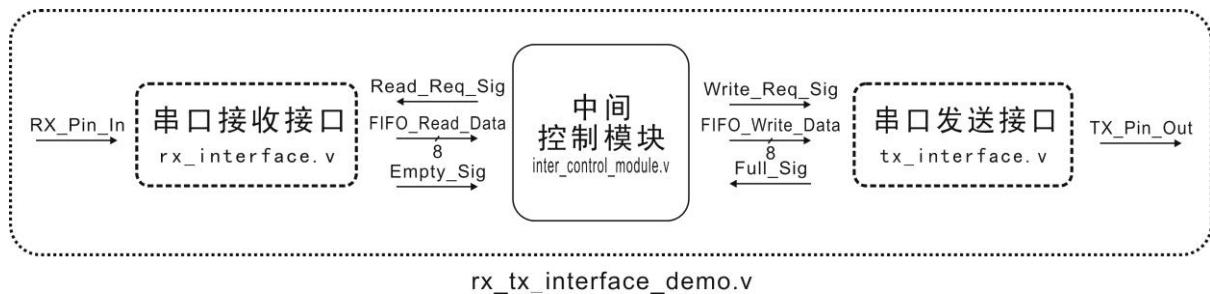
```

该组合模块和“图形”几乎是一样，哎笔者也懒得说什么了...

完成的扩展图：



## 实验十八演示：



在这个演示中主要是演示如何调用串口接收接口和串口发送接口，事实上这个演示时没有实际的意义。当串口接收接口完成数据读取后，就会将数据缓冲在自己的 FIFO 里。然后中间控制模块就会判断该 FIFO 的状态，再从 FIFO 中读取数据，然后将数据写入串口发送接口的 FIFO 里。

中间控制模块作用如同中介，将数据从一方移去另一方，完全没有介入串口接收|发送接口的操作之中。从另一个角度去看，在 `rx_tx_interface_demo.v` 中的接口（串口接收接口和串口发送接口）和中间模块控制模块，完全是独立的。

*inter\_control\_module.v*

```

1. module inter_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Empty_Sig,
7.     input [7:0]FIFO_Read_Data,
8.     output Read_Req_Sig,
9.
10.    input Full_Sig,
11.    output [7:0]FIFO_Write_Data,
12.    output Write_Req_Sig
13. );
14.
15. ****
16.
17. reg [2:0]i;
18. reg isRead;

```

```
19.    reg isWrite;
20.
21.    always @ ( posedge CLK or negedge RSTn )
22.        if( !RSTn )
23.            begin
24.                i <= 3'd0;
25.                isRead <= 1'b0;
26.                isWrite <= 1'b0;
27.            end
28.        else
29.            case( i )
30.
31.                0:
32.                    if( !Empty_Sig ) i <= i + 1'b1;
33.
34.                1:
35.                    begin isRead <= 1'b1; i <= i + 1'b1; end
36.
37.                2:
38.                    begin isRead <= 1'b0; i <= i + 1'b1; end
39.
40.                3:
41.                    if( !Full_Sig ) i <= i + 1'b1;
42.
43.                4:
44.                    begin isWrite <= 1'b1; i <= i + 1'b1; end
45.
46.                5:
47.                    begin isWrite <= 1'b0; i <= 3'd0; end
48.
49.
50.            endcase
51.
52.    /*************************************************************************/
53.
54.    assign Read_Req_Sig = isRead;
55.    assign Write_Req_Sig = isWrite;
56.    assign FIFO_Write_Data = FIFO_Read_Data;
57.
58.    /*************************************************************************/
59.
60. endmodule
```

在 29~50 行是中间控制模块的主要功能。在步骤 0，先判断 rx\_interface.v 的 FIFO 是否为空，如果不为空 if 条件就成立，i 就递增以示下一个步骤（32 行）。

在 34~38 行，表示从 rx\_interface.v 读取一个深度的数据。

当完成从 rx\_interface.v 的 FIFO 读取一个深度的数据以后，数据在 FIFO\_Read\_Data 信号上已经就绪。在步骤 3，先判断 tx\_interface.v 的 FIFO 是否为满状态？如果 tx\_interface.v 的 FIFO 不为满状态，i 会递增以示下一个步骤。

在这里请注意，FIFO\_Write\_Data 是直接由 FIFO\_Read\_Data 驱动（56 行）。在 43~47 行，主要是将数据写入 tx\_interface.v 的 FIFO 内。

最后 i 被赋予 0，以示重复上述的动作。

*rx\_tx\_interface\_demo.v*

```

1. module rx_tx_interface_demo
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input RX_Pin_In,
7.     output TX_Pin_Out
8. );
9.
10.    ****
11.
12.    wire [7:0]FIFO_Read_Data;
13.    wire Empty_Sig;
14.
15.    rx_interface U1
16.    (
17.        .CLK( CLK ),
18.        .RSTn( RSTn ),
19.        .RX_Pin_In( RX_Pin_In ),           // input - from top
20.        .Read_Req_Sig( Read_Req_Sig ),    // input - from U2
21.        .FIFO_Read_Data( FIFO_Read_Data ), // output - to U2
22.        .Empty_Sig( Empty_Sig )          // output - to U2
23.    );
24.
```

```
25.      *****/
26.
27.      wire Read_Req_Sig;
28.      wire [7:0]FIFO_Write_Data;
29.      wire Write_Req_Sig;
30.
31.      inter_control_module U2
32.      (
33.          .CLK( CLK ),
34.          .RSTn( RSTn ),
35.          .Empty_Sig( Empty_Sig ),           // input - from U1
36.          .FIFO_Read_Data( FIFO_Read_Data ), // input - from U1
37.          .Read_Req_Sig( Read_Req_Sig ),     // output - to U1
38.          .Full_Sig( Full_Sig ),           // input - from U3
39.          .FIFO_Write_Data( FIFO_Write_Data ), // output - to U3
40.          .Write_Req_Sig( Write_Req_Sig )    // output - to U3
41.      );
42.
43.      *****/
44.
45.      wire Full_Sig;
46.
47.      tx_interface U3
48.      (
49.          .CLK( CLK ),
50.          .RSTn( RSTn ),
51.          .Write_Req_Sig( Write_Req_Sig ),    // input - from U2
52.          .FIFO_Write_Data( FIFO_Write_Data ), // input - from U2
53.          .Full_Sig( Full_Sig ),           // output - to U2
54.          .TX_Pin_Out( TX_Pin_Out )        // output - to top
55.      );
56.
57.      *****/
58.
59. endmodule
```

实验十八演示说明:

演示的结果会是：当读者从上位机发送数据，该数据会经过 FPGA 然后传回至上位机显示。所以笔者就说该演示，实际上是没有意义的 ...

完成后的扩展图：

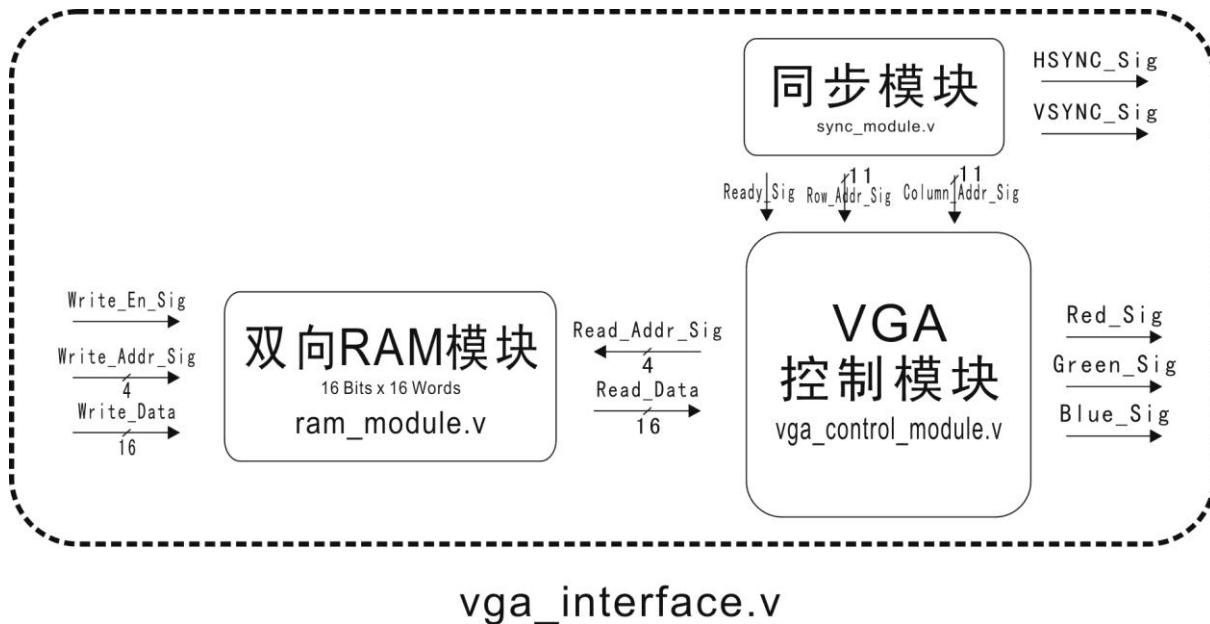


### 实验十八演示结论：

该演示说白了，就是演示 FIFO 的调用。在实验十六至十八，FIFO 几乎扮演着很重要的角色。FIFO 用来缓冲数据，该目的本身是没有什么亮点，但是 FIFO 可以使某个模块封装后，拥有“**独立性**”的特性，这才是重点。

## 5.6 实验十九：VGA 封装

在笔者还没有开始写这本笔记的时候，笔者和大众初学者一样，都喜欢在网络上找资料。有一篇论文“[基于 FPGA 的 VGA 接口](#)”，笔者很感兴趣，但是论文始终是论文，论文的东西都是用来毕业，瞎了笔者的狗眼。笔者就在那个时候突发奇想：“[有没有什么的办法，以最小的条件，来实验该论文中的内容呢？](#)”



上图是组合模块 `vga_interface.v`，里面包含了“[同步模块](#)”（用于配置显示标准，和驱动 VGA），“[VGA 控制模块](#)”（用于控制图像信息），还有一个双向 RAM 模块。是不是觉得很疑惑：为什么多了一个 RAM 模块出来。

在实验九中，VGA 控制模块读取的图片信息是来自 ROM 模块。相反的实验十九的图像信息是来自 RAM。在完成 VGA 接口之前，我们必须设定一些参数。

图像分辨率	16 x 16
图像颜色	点阵
图像显示位置	X = 3 , Y = 2
显示标准	800 x 600 x 60Hz

显示标准，关于这个参数，在 3.4 章中有详细的介绍，这里就不重复了。图像分辨率，说简单点就是一副图像信息的大小。图像颜色，点阵的意思就是黑和白。图像显示位置，是指一副图信息开始显示的位置，X = 3 Y = 2，表示图像在屏幕的坐标 (3, 2) 显示。

*sync\_module.v*

```
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. reg isReady;
42.
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. assign Column_Addr_Sig = isReady ? Count_H - 11'd216 : 11'd0; // Count from 0;
62. assign Row_Addr_Sig = isReady ? Count_V - 11'd27 : 11'd0; // Count from 0;
63.
64. *****/
65.
66. endmodule
```

该 sync\_module.v 是支持 800 x 600 x 60Hz 的显示标准（需要 40Mhz 的时钟源）。

### vga\_control\_module.v

```
1. module vga_control_module
2. (
3.     CLK, RSTn,
4.     Ready_Sig, Column_Addr_Sig, Row_Addr_Sig,
5.     Ram_Data, Ram_Addr,
6.     Red_Sig, Green_Sig, Blue_Sig
7. );
8.     input CLK;
9.     input RSTn;
10.
11.    input Ready_Sig;
```

```
12.      input [10:0]Column_Addr_Sig;
13.      input [10:0]Row_Addr_Sig;
14.
15.      input [15:0]Ram_Data;
16.      output [3:0]Ram_Addr;
17.
18.      output Red_Sig;
19.      output Green_Sig;
20.      output Blue_Sig;
21.
22.      *****/
23.
24.      reg [4:0]m;
25.
26.      always @ ( posedge CLK or negedge RSTn )
27.          if( !RSTn )
28.              m <= 5'd0;
29.          else if( Ready_Sig && Row_Addr_Sig > 1 && Row_Addr_Sig < 18 )
30.              m <= Row_Addr_Sig[4:0] - 5'd2;
31.          else
32.              m <= 5'd0;
33.
34.      reg [4:0]n;
35.
36.      always @ ( posedge CLK or negedge RSTn )
37.          if( !RSTn )
38.              n <= 5'd0;
39.          else if( Ready_Sig && Column_Addr_Sig > 2 && Column_Addr_Sig < 19 )
40.              n <= Column_Addr_Sig[4:0] - 5'd3;
41.          else
42.              n <= 5'd0;
43.
44.      *****/
45.
46.      reg isSize;
47.
48.      always @ ( posedge CLK or negedge RSTn )
49.          if( !RSTn )
50.              isSize <= 1'b0;
51.          else if( ( Row_Addr_Sig > 1 && Row_Addr_Sig < 18 ) &&
52.                  ( Column_Addr_Sig > 2 && Column_Addr_Sig < 19 ) )
53.              isSize <= 1'b1;
54.          else
55.              isSize <= 1'b0;
```

```

56.
57.      ****
58.
59.      assign Ram_Addr = m[3:0];
60.
61.      assign Red_Sig = Ready_Sig && isSize ? Ram_Data[ 5'd15 - n ] : 1'b0;
62.      assign Green_Sig = Ready_Sig && isSize ? Ram_Data[ 5'd15 - n ] : 1'b0;
63.      assign Blue_Sig = Ready_Sig && isSize ? Ram_Data[ 5'd15 - n ] : 1'b0;
64.
65.      ****
66.
67.
68. endmodule

```

29行的“Row\_Addr\_Sig > 1”表示了Y开始显示位置是2，而39行的“Column\_Addr\_Sig > 2”表示了X开始显示位置是3。我们知道图像的面积是 $16 \times 16$ ，所以29行的if条件说明了图像有效高度（行）。同样的在39行的if条件说明了图像有效的长度（列）。

在24行和34行的寄存器m和n，是用计数有效的行（30行）和列（40行），并且用于列寻址和行寻址（61~63行）。（有效的行是从Row\_Addr\_Sig > 1开始计数，有效的列是从Column\_Addr\_Sig > 2开始计数）

在46~54行的isSize标志寄存器是用来确定“一副图像的显示框”，亦即“图像显示有效范围”，在51~52行定义了“图像显示有效”的条件。这也说明了一副 $16 \times 16$ 的图像有效显示在Y=2和X=3之后和Y=17和X=18之前。

59行表示m行寻址等价于Ram\_Addr地址。61~63行，是列寻址，同是点阵操作。由于该VGA接口的颜色支持参数是“黑白”的缘故，所以61~63行都是同样的点阵操作。

*ram\_module.v*



关于RAM，有电子背景的同学都知道它是什么。RAM和ROM不同的是，RAM支持访问（读和写操作），然而ROM只支持读操作。一般上RAM有分为单端口，双向端口，三向端口。双向端口，有分为真双向端口（True），和假双向端口（Simple），真双向端口有写入时钟和读取时钟，然而假双向端口读写共用一个时钟。为了方便的建模，在

---

这里我们使用假双向端口 RAM。

在这里，我们先要考虑 vga\_interface.v 支持的图像分辨率，亦即  $16 \times 16$ 。所以 RAM 所需要的储存空间是  $16\text{Bits} \times 16\text{Words}$ 。RAM 和 FIFO 一样，要访问 RAM 的时候都需要拉高 xx\_En\_Sig 信号。由于 RAM 包含 16 Bits 所以 Write\_Data 和 Read\_Data，皆是 16 位的位宽。当然 16 Words 表示了 xx\_Addr\_Sig 是 4 位的位宽。

一个普遍存在与双向端口 RAM 的问题是“[同时读写冲突](#)”。一般双向端口 RAM 的芯片都内置仲裁逻辑。“[仲裁逻辑](#)”的设计比较复杂，我们使用另一种方法，就是“[访问优先级](#)”。

在这里，我们可以这样定义：写操作的优先级高于读操作。

所以呀，使用 QuartusII 自建的双向端口 RAM 不怎么适合。换一句话，我们必须手动创建包含“[访问优先级](#)”的“[假双向端口 RAM](#)”。

```

1. module ram_module
2. (
3.     input CLK,
4.     input RSTn,
5.     input Write_En_Sig,
6.     input [3:0]Write_Addr_Sig,
7.     input [15:0]Write_Data,
8.     input [3:0]Read_Addr_Sig,
9.     output [15:0]Read_Data
10.
11. );
12.
13. ****
14.
15. (* ramstyle = "no_rw_check , m4k" , ram_init_file = "ram_initial_file.mif" *) reg [15:0] RAM[15:0];
16.
17. ****
18.
19. reg [15:0]rData;
20.
21. always @ ( posedge CLK or negedge RSTn )
22.     if( !RSTn )
23.         rData <= 16'd0;
24.     else if( Write_En_Sig )
25.         RAM[ Write_Addr_Sig ] <= Write_Data;
```

---

```
26.         else
27.             rData <= RAM[ Read_Addr_Sig ];
28.
29.             ****
30.
31.             assign Read_Data = rData;
32.
33.             ****
34.
35. endmodule
```

第 1~11 行是 RAM 模块的输入输出口。在 15 行，声明了 16Bits x 16 Words 的储存器。如果以 Altera 的 FPGA 为例，尝试回想一下，FPGA 都内置了片上资源 m4k。当我们声明储存器的时候，我们可以指定它由 m4k 组成。

```
(* ram_tyle = m4k *)
```

当然我们也可以指定储存器由逻辑资源组成：

```
(* ram_tyle = logic *)
```

在前面，笔者已经说过：“双向端口的 RAM 存在访问冲突的问题”。当 QuartusII 在综合的时候，由于 m4k 资源本身的特性，并不适合“写时读”（read-during-write），亦即“访问冲突”。虽然，我们手动为 RAM 模块添加了访问优先级，故可避免实际的“访问冲突”问题。但是 Quartus II 的综合器是一个大笨蛋，没有提示它：“不用关心 m4k 资源的访问冲突问题”，综合器是不知道的，在编译的时候会一直 Warning！因此：

```
(* ram_style = no_rw_check *)
```

还有一点就是关于储存器初始化的问题：

还记得在建立 ROM 的时候，笔者都为 ROM 建立一个 .mif，然后对 ROM 初始化。同样的道理，RAM 储存器在创建的时候也必须初始化。该 RAM 模块初始化的信息，是实验十之五之中的“第一副小绿人”。

```
(* ram_init_file = ram_initial_file.mif *)
```

21~27 行是 RAM 模块的访问优先级逻辑。在 24 行表示了写操作比读操作拥有更高的优先级。当 Write\_En\_Sig 拉高的时候，对 RAM 储存器执行写操作。一旦 Write\_En\_Sig 被拉低“读操作”作为 RAM 储存器的默认状态。

有一重点必须注意就是 rData 寄存器是用来暂存读数据。为了避免“[当写操作执行时 Read\\_Data 失去驱动源](#)”。

*vga\_interface.v*

```
1. module vga_interface
2. (
3.     input RSTn,
4.
5.     input Write_En_Sig,
6.     input [3:0]Write_Addr_Sig,
7.     input [15:0]Write_Data,
8.
9.     input VGA_CLK, // 40MHz
10.    output VSYNC_Sig,
11.    output HSYNC_Sig,
12.    output Red_Sig,
13.    output Green_Sig,
14.    output Blue_Sig
15.
16. );
17.
18. *****/
19.
20. wire Ready_Sig;
21. wire [10:0]Column_Addr_Sig;
22. wire [10:0]Row_Addr_Sig;
23.
24. sync_module U1
25. (
26.     .CLK( VGA_CLK ), // input - from top
27.     .RSTn( RSTn ),
28.     .VSYNC_Sig( VSYNC_Sig ), // output - to top
29.     .HSYNC_Sig( HSYNC_Sig ), // output - to top
30.     .Ready_Sig( Ready_Sig ), // output - to U2
31.     .Column_Addr_Sig( Column_Addr_Sig ), // output - to U2
32.     .Row_Addr_Sig( Row_Addr_Sig ) // output - to U2
33. );
34.
35. *****/
36.
37. wire [3:0]Read_Addr_Sig;
38.
```

```
39.    vga_control_module U2
40.    (
41.        .CLK( VGA_CLK ),           // input - from top
42.        .RSTn( RSTn ),           //
43.        .Ready_Sig( Ready_Sig ),   // input - from U1
44.        .Column_Addr_Sig( Column_Addr_Sig ), // input - from U1
45.        .Row_Addr_Sig( Row_Addr_Sig ), // input - from U1
46.        .Ram_Data( Read_Data ),    // output - to U3
47.        .Ram_Addr( Read_Addr_Sig ), // output - to U3
48.        .Red_Sig( Red_Sig ),      // output - top
49.        .Green_Sig( Green_Sig ),   // output - top
50.        .Blue_Sig( Blue_Sig )     // output - top
51.    );
52.
53.    /*************************************************************************/
54.
55.    wire [15:0]Read_Data;
56.
57.    ram_module U3
58.    (
59.        .CLK( VGA_CLK ),           // input - from top
60.        .RSTn( RSTn ),           //
61.        .Write_En_Sig( Write_En_Sig ), // input - from top
62.        .Write_Addr_Sig( Write_Addr_Sig ), // input - from top
63.        .Write_Data( Write_Data ),    // input - from top
64.        .Read_Addr_Sig( Read_Addr_Sig ), // input - from U2
65.        .Read_Data( Read_Data )      // output - to U2
66.    );
67.
68.    /*************************************************************************/
69.
70. endmodule
```

VGA 接口的时钟源是由全局时钟源经过倍频后输入的（9 行）。

### 实验十九说明：

读者是不是觉得实验十九和实验九相比，简直是莫名其妙对吧？在这里笔者稍微区分一下实验九和实验十九的不同。实验九顾名思义就是 VGA 模块，它所使用的显示方法是同步的。换句话说，也就是图片信息处理和 VGA 显示驱动都是在同样的时间下。当然也可以这样想，图片信息是由 VGA 模块本身提供的，又或者图片信息早已固定并且存在。

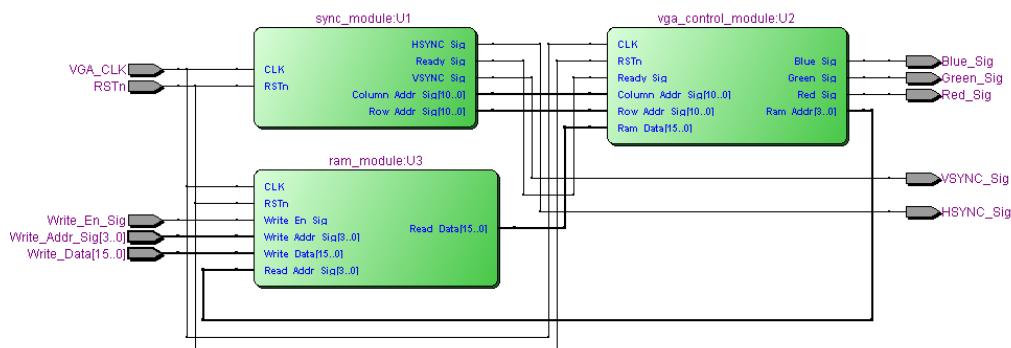
实验十九的 VGA 接口，恰好是与实验九的 VGA 显示模块相反。它所使用的显示方法是异步的。也就是说，图像信息处理和 VGA 显示驱动是在不同的时间下完成。看得简单点就是，图像信息是由外部提供。

在实验十九中的 VGA 接口，图像信息是暂存在 RAM 模块里，而且 RAM 模块里边的图像信息是由上一层模块写入。读者可能会产生这样一个问题：“[假设 vga\\_control\\_module.v 在读取 RAM，地址 0 的信息的时候，上一层模块同时对 RAM，地址 0 执行写操作 ...](#)”。事实上，由于访问优先级逻辑的关系，当发生“[访问冲突](#)”的时候，vga\_control\_module.v 读取的是 上一次的 rData（上一次从 RAM 中读出的数据）信息。

读者可能又会问：“[当发生访问冲突的时候，rData 暂存的数据当然不是当前的显示数据，图像显示既不是出现错误？](#)”。你知道吗？LCD 显示技术为了消除 LCD 显示残影的问题，在指定间隔时的时间里，都会插入若干“[黑帧](#)”（全黑图像信息）。如果 LCD 插入“[黑帧](#)”，已不是屏幕忽然间全黑，然后又恢复，又全黑，又恢复 .... 呵呵！事实上，人体的视觉是很笨重和迟钝的，这些“[短暂](#)”的“[黑帧](#)”人眼是察觉不出来的。这也使得 LCD 消除残影的方法。

同样的道理，以  $800 \times 600 \times 60\text{Hz}$  为显示标准。如果在 1 秒内出现 1~16 个残缺的帧，人眼是不会察觉到，因为该显示标准时每秒 60 帧图像。用动画的话来说，每秒 8 帧产生拖尾效果，每秒 24 帧产生专业动画效果，每秒 40 帧对于人眼来说是“[瞬间](#)”的效果。在 40 帧之间，其中有 1~2 个帧是损毁的帧，人眼是不会察觉的。

完成后的扩展图：



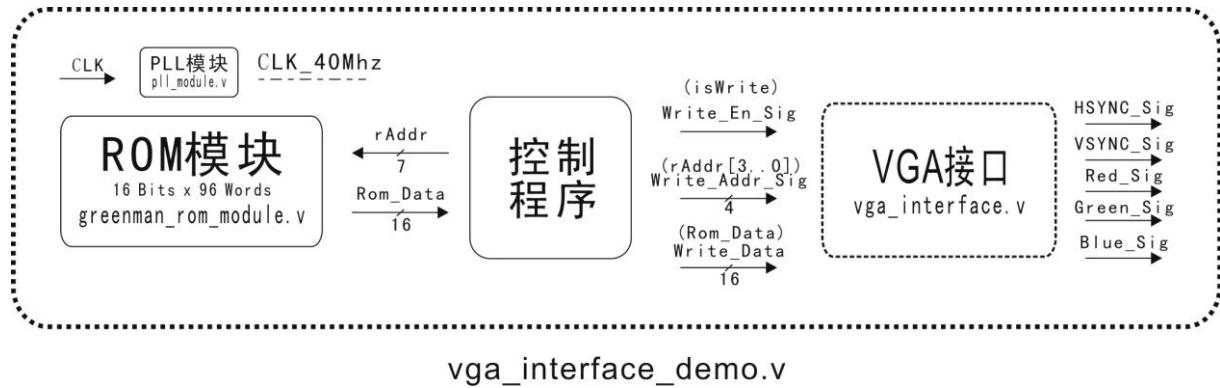
实验十九结论：

实验九和实验十九最大的差别就是图像信息处理的方法。前者是由模块内部提供的，后者是由上一层模块写入。此外实验十九当 vga\_control\_module.v 对 ram\_module.v 读取信息的时候，如果在同一个时间，上一层模块对 vga\_interface.v 的 ram\_module.v 执行

写操作，故会发生“[访问冲突](#)”。为了避免这个问题，笔者对 ram\_module.v 加入了访问优先级逻辑，访问优先级定义为：写操作优先级高于读操作。

还有有关“[实验十九说明](#)”中提及种种“[显示问题](#)”会在“[实验十九演示](#)”中演示。

## 实验十九演示：小绿人请加油



上图的 `vga_interface_demo.v` 组合模块表示了，控制程序从 ROM 模块读取图片信息，然后写入 VGA 接口。该 ROM 模块是基于实验九之五的 `greenman_rom_module.v`，里边包含了 6 副  $16 \times 16$  的图片信息。控制程序每隔 250ms 写入不同的信息至 VGA 接口，在屏幕上会出现小绿人的动画。

`vga_interface_demo.v`

```

1. module vga_interface_demo
2. (
3.     input CLK,
4.     input RSTn,
5.     output VSYNC_Sig,
6.     output HSYNC_Sig,
7.     output Red_Sig,
8.     output Blue_Sig,
9.     output Green_Sig
10. );
11.
12. *****/
13.
14. wire CLK_40Mhz;
15.
16. pll_module u1
17. (
18.     .inclk0(CLK),
19.     .c0(CLK_40Mhz)

```

## Verilog HDL 那些事儿 – 建模篇

```
20.      );
21.
22.      /************************************************************************/
23.
24.      parameter T1MS = 16'd39999;
25.
26.      /************************************************************************/
27.
28.      reg [15:0]C1;
29.
30.      always @ ( posedge CLK_40Mhz or negedge RSTn )
31.          if( !RSTn )
32.              C1 <= 15'd0;
33.          else if( C1 == T1MS )
34.              C1 <= 15'd0;
35.          else if( isCount )
36.              C1 <= C1 + 1'b1;
37.          else
38.              C1 <= 15'd0;
39.
40.      /************************************************************************/
41.
42.      reg [9:0]CMS;
43.
44.      always @ ( posedge CLK_40Mhz or negedge RSTn )
45.          if( !RSTn )
46.              CMS <= 10'd0;
47.          else if( CMS == rTimes )
48.              CMS <= 10'd0;
49.          else if( C1 == T1MS )
50.              CMS <= CMS + 1'b1;
51.
52.      /************************************************************************/
53.
54.      reg [6:0]Y;
55.
56.      always @ ( posedge CLK_40Mhz or negedge RSTn )
57.          if( !RSTn )
58.              Y <= 7'd0;
59.          else
60.              case( i )
61.
62.                  0 : Y <= 7'd0;
63.                  2 : Y <= 7'd16;
```

```
64.          4 : Y <= 7'd32;
65.          6 : Y <= 7'd48;
66.          8 : Y <= 7'd64;
67.         10: Y <= 7'd80;
68.
69.      endcase
70.
71.  *****/
72.
73.  reg [3:0]i;
74.  reg [6:0]rAddr;
75.  reg [4:0]X;
76.  reg isWrite;
77.  reg isCount;
78.  reg [9:0]rTimes;
79.
80.  always @ ( posedge CLK_40Mhz or negedge RSTn )
81.    if( !RSTn )
82.      begin
83.        i <= 4'd0;
84.        rAddr <= 7'd0;
85.        X <= 5'd0;
86.        isWrite <= 1'b0;
87.        isCount <= 1'b0;
88.        rTimes <= 10'd100;
89.      end
90.    else
91.      case ( i )
92.
93.        0, 2, 4, 6, 8, 10:
94.          if( X == 16 ) begin rAddr <= 7'd0; X <= 5'd0; isWrite <= 1'b0; i <= i + 1'b1; end
95.          else begin rAddr <= Y + X; X = X + 1'b1; isWrite <= 1'b1;end
96.
97.        1, 3, 5, 7, 9, 11:
98.          if( CMS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
99.          else begin rTimes <= 10'd250; isCount <= 1'b1; end
100.
101.        12:
102.          i <= 4'd0;
103.
104.      endcase
105.
106.  *****/
107.
```

```
108.  
109.     wire [15:0]Rom_Data;  
110.  
111.     greenman_rom_module U2  
112.     (  
113.         .clock( CLK_40Mhz ),  
114.         .address( rAddr ),  
115.         .q( Rom_Data )  
116.     );  
117.  
118.     /*****  
119.  
120.     vga_interface U3  
121.     (  
122.         .RSTn( RSTn ),  
123.         .Write_En_Sig( isWrite ),  
124.         .Write_Addr_Sig( rAddr[3:0] ),  
125.         .Write_Data( Rom_Data ),  
126.         .VGA_CLK( CLK_40Mhz ),  
127.         .VSYNC_Sig( VSYNC_Sig ),  
128.         .HSYNC_Sig( HSYNC_Sig ),  
129.         .Red_Sig( Red_Sig ),  
130.         .Green_Sig( Green_Sig ),  
131.         .Blue_Sig( Blue_Sig )  
132.     );  
133.  
134.     /*****  
135.  
136. endmodule
```

16~20 行实例化了 pll 模块, pll 模块将全局时钟倍频至 40Mhz。24 行是 1ms 常量的声明, 然而 28~50 行建立了 1ms 定时器 (28~38 行) 和 ms 级计数器 (42~50 行)。54~69 行表示了 6 副图像信息在 ROM 模块的起始地址。

在 73~104 行就是该组合模块的核心功能, 在 93~95 行表示了 - 当步骤为 0, 2, 4, 6, 8, 10 时, 对 vga\_interface.v 的 RAM 模块写入不同的图像信息。在 97~99 行是 250ms 的延迟, 亦即每一副图像信息的停留时间。

109~116 行实例化了 greenman\_rom\_module.v。在 120~132 行实例化了 vga\_interface.v。一开始的时候 vga\_interface.v 内部的 RAM 模块会执行初始化该储存信息, 亦即作为默认图像信息。当 vga\_interface\_demo.v 进入步骤 0, 在同一时间, 在 62 行会对 Y 寄存器赋值与 ROM 模块的第一副图像信息的起始地址, 亦即地址 0。在 94~95 行 isWrite 被拉高, 以示对 vga\_interface.v 的 RAM 模块写入使能 (123 行)。

Y 寄存器作为图像信息在 ROM 模块的起始地址，X 寄存器作为每一副图像信息的行寻址地址，然而 rAddr 作为 X 寄存器和 Y 寄存器的总和，同时也是作为 ROM 模块的（驱动）地址信号（114 行）。

在 94 行中，x 会递增至 0~15，已对一副图像信息的行寻址（Addr 寄存器的 前 4 位 [3..0]，同时作为 vga\_interface.v 的写入地址-124 行）。还有一点必须注意的是，vga\_interface.v 的写入数据是由 ROM 模块的 Rom\_Data 驱动（125 行）。直到 x 等于 16（95 行），亦即一幅图像的行寻址已经结束，rAddr，X，和 isWrite 寄存器被赋予 0 值。i 递增以示下一个步骤。

在 97~99 行，步骤 1, 3, 5, 7, 9, 11 是延迟 250ms 的操作。rTimes 寄存作为延迟 ms 的计数值（98 行），rTime 赋值与 250，亦即是延迟 250ms。当 isCount 被拉高的时候（99 行），定时器和计数器都会开始工作（28~50 行）。直到 ms 级的计数计数到 250 为止（98 行），isCount 被拉低，i 递增以示下一个步骤。

上述步骤的执行大致如下：

- (一) 写入第 0 副图像信息至 vga\_interface.v，延迟 250ms。
- (二) 写入第 1 副图像信息至 vga\_interface.v，延迟 250ms。
- (三) 写入第 2 副图像信息至 vga\_interface.v，延迟 250ms。
- (四) 写入第 3 副图像信息至 vga\_interface.v，延迟 250ms。
- (五) 写入第 4 副图像信息至 vga\_interface.v，延迟 250ms。
- (六) 写入第 5 副图像信息至 vga\_interface.v，延迟 250ms。
- (七) 重复执行步骤 1~6。

实验十九演示说明：

在演示中，读者会看到由六副图像信息每隔 250ms 不停的循环和切换而产生的小绿人动画。实际上，每秒 60 帧的图像信息中，有几幅图像是“**崩坏**”，由于人类的视觉迟钝的关系察觉不出来。

实验十九演示结论：

不知道读者还记得“**接口的定义**”吗？就是“**最后的工程**”和“**独立性**”。在实验十九的演示中，VGA 接口是一个独立的个体，它以每秒显示 60 帧图像信息而工作。当我们对 VGA 接口调用的时候，我们不需要顾及 VGA 接口的内部是如何工作，反之我们只要关心如何对 VGA 接口 写入图像信息即可。如实验十九演示中那样，为了实现小绿人动画，我们在上一层模块中，每隔 250ms 写入不同的图像信息。

和在实验九之五不同的是，我们为了实现小绿人的动画效果，必须考虑由 sync\_module.v

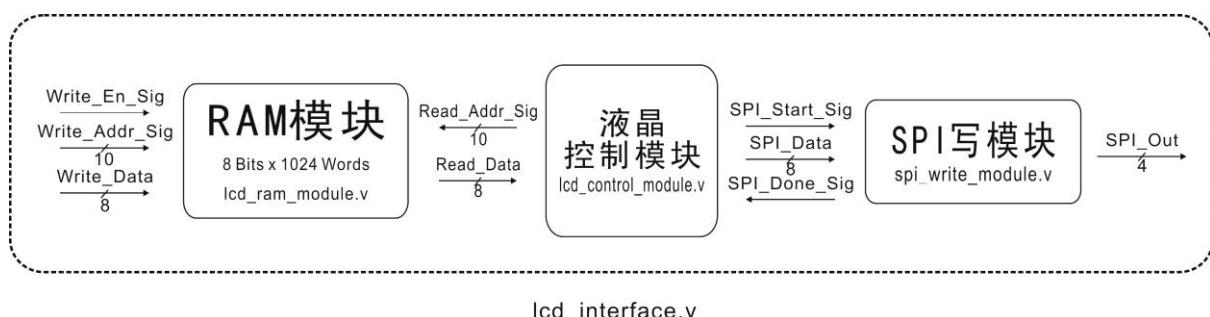
输出的 Frame\_Sig 信号。以计数 Frame\_Sig 信号的方式去实现小绿人动画。类似的方法大大的压缩了灵活性。原理上 vga\_interface.v 的显示原理，适合任何显示接口，因为人类的肉眼对画面的“[瞬间切换速度](#)”不敏感。

## 5.7 实验二十：LCD（12864）封装

在这里笔者先提及一些重点。从 5.1 章开始，读者是否已经发觉到，笔者对封装建模，使用了许多第二章至第四章的建模基础。与其说，封建建模涉及了许多基础建模，还不如说建模基础是为了后期建模才存在。这一点笔者一开始就一直强调“**低级建模**”是为了后期的建模做好准备。当然“**封装建模**”也不是最后的建模。实际上“**封装建模**”也是“**低级建模**”的一环而已。因为“**封装建模**”一样也是为后期的建模在做准备。

（那么下一个“后期”又是什么？读者先猜猜吧 ... ）

继 5.6 章后（VGA 接口），这一篇同样是有关显示的封装。无论是 VGA 接口 还是 LCD 接口（LCD 封装），原理上都是一样，不同的只是液晶显示驱动的方法而已。



lcd\_interface.v

上图是 lcd\_interface.v 的组合模块。它包含了液晶控制模块，RAM 模块，和 SPI 写模块。和 VGA 接口一样，RAM 模块包含了图像信息，而且 RAM 模块也添加了访问优先级的逻辑。SPI 写模块和实验 12 是一模一样。至于液晶控制模块就有点特殊了。

液晶控制模块包含了对液晶的“**初始化控制**”和“**绘图控制**”的功能以外，还能自动的从 RAM 模块读取图像信息。此外，对于 lcd\_interface.v 的调用，我们只要针对 lcd\_interface.v 的 RAM 模块 写入图像信息即可。

lcd\_interface.v 的功能大致如下：

- (一) 初起的时候，液晶控制模块对液晶初始化。RAM 模块本身也自行初始化。
- (二) 每隔一段时间，液晶控制模块就会从 RAM 模块读取图像信息，然后利用这些信息来驱动液晶的显示。

在封装之前，需要考虑几个参数的配置：

液晶扫描频率	40Hz
图像分辨率	64 x 128

液晶的扫描频率 40Hz，亦即每 25ms 为液晶写入一副 64 x 128 大小分辨率的图像。

*lcd\_ram\_module.v*

```
1. module lcd_ram_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_En_Sig,
7.     input [9:0]Write_Addr_Sig,
8.     input [7:0]Write_Data,
9.
10.    input [9:0]Read_Addr_Sig,
11.    output [7:0]Read_Data
12. );
13.
14. /*************************************************************************/
15.
16. (* ramstyle = " no_rw_check, m4k ", ram_init_file = " pikachu.mif " *) reg [7:0] RAM [1023:0];
17.
18. reg [7:0]rData;
19.
20. always @ ( posedge CLK or negedge RSTn )
21.     if( !RSTn )
22.         rData <= 8'd0;
23.     else if( Write_En_Sig )
24.         RAM[ Write_Addr_Sig ] <= Write_Data;
25.     else
26.         rData <= RAM[ Read_Addr_Sig ];
27.
28. /*************************************************************************/
29.
30. assign Read_Data = rData;
31.
32. /*************************************************************************/
33.
34. endmodule
```

在 16 行，声明了该 8 Bits x 1024 Words 的储存空间，是由 m4k 资源组成。而且还提示 Quartus II 的综合器无视“**写时读**”的问题。该储存器是由 pika\_ani.mif 文件初始化。

*spi\_write\_module.v*

```
1. module spi_write_module
2. (
3.     CLK, RSTn,
4.     Start_Sig,
5.     SPI_Data,
6.     Done_Sig,
7.     SPI_Out
8. );
9.
10.    input CLK;
11.    input RSTn;
12.    input Start_Sig;
13.    input [9:0]SPI_Data;
14.    output Done_Sig;
15.    output [3:0]SPI_Out; // [3]CS [2]A0 [1]CLK [0]DO
16.
17.    *****/
18.
19.    parameter T0P5US = 4'd9;
20.
21.    *****/
22.
23.    reg [3:0]Count1;
24.
25.    always @ ( posedge CLK or negedge RSTn )
26.        if( !RSTn )
27.            Count1 <= 4'd0;
28.        else if( Count1 == T0P5US )
29.            Count1 <= 4'd0;
30.        else if( Start_Sig )
31.            Count1 <= Count1 + 1'b1;
32.        else
33.            Count1 <= 4'd0;
34.
35.    *****/
36.
37.    reg [4:0]i;
38.    reg rCLK;
39.    reg rDO;
40.    reg isDone;
41.
```

## Verilog HDL 那些事儿 – 建模篇

```
42.      always @ ( posedge CLK or negedge RSTn )
43.          if( !RSTn )
44.              begin
45.                  i <= 5'd0;
46.                  rCLK <= 1'b1;
47.                  rDO <= 1'b0;
48.                  isDone <= 1'b0;
49.              end
50.          else if( Start_Sig )
51.              case( i )
52.
53.                  5'd0, 5'd2, 5'd4, 5'd6, 5'd8, 5'd10, 5'd12, 5'd14:
54.                      if( Count1 == T0P5US ) begin rCLK <= 1'b0;rDO <= SPI_Data[ 7 - ( i >> 1 ) ]; i <= i + 1'b1; end
55.
56.                  5'd1, 5'd3, 5'd5, 5'd7, 5'd9, 5'd11, 5'd13, 5'd15 :
57.                      if( Count1 == T0P5US ) begin rCLK <= 1'b1; i <= i + 1'b1; end
58.
59.                  5'd16:
60.                      begin isDone <= 1'b1; i <= i + 1'b1; end
61.
62.                  5'd17:
63.                      begin isDone <= 1'b0; i <= 5'd0; end
64.
65.              endcase
66.
67.          /*****
68.
69.          assign Done_Sig = isDone;
70.          assign SPI_Out = { SPI_Data[9], SPI_Data[8], rCLK, rDO };
71.
72.          *****/
73.
74.      endmodule
```

基本上和实验十二的 `spi_write_module.v` 是完全一样。

### *lcd\_control\_module.v*

```
1.  module lcd_control_module
2.  (
3.      input CLK,
```

```
4.      input RSTn,
5.
6.      input [7:0]Read_Data,
7.      output [9:0]Read_Addr_Sig,
8.
9.      input SPI_Done_Sig,
10.     output SPI_Start_Sig,
11.     output [9:0]SPI_Data
12. );
13.
14. *****/
15.
16. parameter T25MS = 21'd1_249_999; // 40Hz  50M*0.025-1=1_249_999
17.
18. *****/
19.
20. reg [20:0]C1;
21.
22. always @ ( posedge CLK or negedge RSTn )
23.   if( !RSTn )
24.     C1 <= 21'd0;
25.   else if( C1 == T25MS )
26.     C1 <= 21'd0;
27.   else
28.     C1 <= C1 + 1'b1;
29.
30. *****/
31.
32. reg [1:0]isStart;
33.
34. always @ ( posedge CLK or negedge RSTn )
35.   if( !RSTn )
36.     isStart <= 2'b10;
37.   else if( C1 == T25MS )
38.     isStart <= 2'b01;
39.   else if( isDone )
40.     isStart <= 2'b00;
41.
42. *****/
43.
44. reg [5:0]i;
45. reg [9:0]rData;
46. reg [7:0]x;
47. reg [3:0]y;
```

## Verilog HDL 那些事儿 – 建模篇

```
48.      reg isSPI_Start;
49.      reg isDone;
50.
51.      always @ ( posedge CLK or negedge RSTn )
52.          if( !RSTn )
53.              begin
54.                  i <= 6'd0;
55.                  rData <= { 2'b11, 8'h2f };
56.                  x <= 8'd0;
57.                  y <= 4'd0;
58.                  isSPI_Start <= 1'b0;
59.                  isDone <= 1'b0;
60.              end
61.          else if( isStart[1] ) // Initial Function
62.              case( i )
63.
64.                  0:
65.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
66.                      else begin rData <= { 2'b00, 8'haf }; isSPI_Start <= 1'b1; end
67.
68.                  1:
69.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
70.                      else begin rData <= { 2'b00, 8'h40 }; isSPI_Start <= 1'b1; end
71.
72.                  2:
73.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
74.                      else begin rData <= { 2'b00, 8'ha6 }; isSPI_Start <= 1'b1; end
75.
76.                  3:
77.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
78.                      else begin rData <= { 2'b00, 8'ha0 }; isSPI_Start <= 1'b1; end
79.
80.                  4:
81.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
82.                      else begin rData <= { 2'b00, 8'hc8 }; isSPI_Start <= 1'b1; end
83.
84.                  5:
85.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
86.                      else begin rData <= { 2'b00, 8'ha4 }; isSPI_Start <= 1'b1; end
87.
88.                  6:
89.                      if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end
90.                      else begin rData <= { 2'b00, 8'ha2 }; isSPI_Start <= 1'b1; end
91.
```

```
92.          7:  
93.          if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
94.          else begin rData <= { 2'b00, 8'h2f }; isSPI_Start <= 1'b1; end  
95.  
96.          8:  
97.          if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
98.          else begin rData <= { 2'b00, 8'h24 }; isSPI_Start <= 1'b1; end  
99.  
100.         9:  
101.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
102.         else begin rData <= { 2'b00, 8'h81 }; isSPI_Start <= 1'b1; end  
103.  
104.         10:  
105.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
106.         else begin rData <= { 2'b00, 8'h24 }; isSPI_Start <= 1'b1; end  
107.  
108.         11:  
109.         begin rData <= { 2'b11, 8'h2f }; isDone <= 1'b1; i <= i + 1'b1; end  
110.  
111.         12:  
112.         begin isDone <= 1'b0; i <= 6'd0; end  
113.  
114.       endcase  
115.     else if( isStart[0] ) // Draw Function  
116.       case( i )  
117.  
118.         // Setting Y address ( row of lcd )  
119.         0, 4, 8, 12, 16, 20, 24, 28:  
120.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
121.         else begin rData <= { 2'b00, 4'hb, y }; isSPI_Start <= 1'b1; end  
122.  
123.         // Setting X address ( column for each row ) [7..4]  
124.         1, 5, 9, 13, 17, 21, 25, 29:  
125.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
126.         else begin rData <= { 2'b00, 4'h1, 4'h0 }; isSPI_Start <= 1'b1; end  
127.  
128.         // Setting X address ( column for each row ) [3..0]  
129.         2, 6, 10, 14, 18, 22, 26, 30:  
130.         if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; i <= i + 1'b1; end  
131.         else begin rData <= { 2'b00, 4'h0, 4'h0 }; isSPI_Start <= 1'b1; end  
132.  
133.         // Column filling 128 times  
134.         3, 7, 11, 15, 19, 23, 27, 31:  
135.         if( x == 8'd128 ) begin y <= y + 1'b1; x <= 8'd0; i <= i + 1'b1; end
```

```
136.           else if( SPI_Done_Sig ) begin isSPI_Start <= 1'b0; x <= x + 1'b1; end
137.           else begin rData <= { 2'b01, Read_Data }; isSPI_Start <= 1'b1; end
138.
139.           32:
140.           begin rData <= { 2'b11, 8'd0 };y <= 4'd0; isDone <= 1'b1; i <= i + 1'b1;end
141.
142.           33:
143.           begin isDone <= 1'b0; i <= 6'd0; end
144.
145.       endcase
146.   /*****
147.
148.   assign Read_Addr_Sig = x + (y << 7);
149.
150.   assign SPI_Start_Sig = isSPI_Start;
151.   assign SPI_Data = rData;
152.
153.   *****/
154.
155. endmodule
```

( ⊙o⊙ )哇，代码那么长！不要被吓到！该模块主要是基于“命令式仿顺序操作”的液晶模块而已，其中还添加了定时器。

在 16 行定义了 25ms 的常量。20~28 行是 25ms 的定时器。

我们知道“**仿顺序操作**”的模块都有一个特征，就是“**不被使能不工作**”，“**完成工作就报告**”。但是为了使液晶控制模块有独立性的能力（自己使能自己），结果添加一个“**定时使能**”的功能。在 32~40 行的定时器就是充当这样的角色。isStart 寄存器（32 行）的位宽是 2 位，亦即该液晶控制模块拥有 2 个功能。在初始状态 isStart 被复位为 2'b10。

在 51~145 行就是液晶控制模块的核心部分。61~114 行是 initial\_module.v 的部分，然而该功能被使能是在 isStart[1]，亦即 isStart 寄存器最高位被拉高的时候才发生。这也就是说，lcd\_interface.v 初始化的时候，51~145 的“**initial function**”（液晶初始化功能）就被执行。

在同一个时间 20~28 行的定时器也开始计数。但是在定时器完成计数之前，在 109 行，产生了“**完成反馈**”，亦即“**initial function**”已经执行完毕。此时在 39 行，if 条件成立 isStart 被清零。

115~145 行是“**draw function**”（液晶绘图功能）。该功能会发生在，当 isStart[0]，isStart 寄存器的最低位被拉高的时候。每隔 25ms 的时间在 20~28 行的定时器都会产生定时，isStart 的最低位都会被拉高。换句话说，每隔 25ms “**draw function**”就会被执行。

当“draw function”完成后（140行），就会产生一个“完成反馈”。在同一个时间39行的if条件就会成立，isStart会被清零。

在148行的Read\_Addr\_Sig信号是作为“RAM模块”读取的寻址信号。

在前面，笔者显示了该lcd\_interface.v的扫描频率是40Hz。如果换做公式来表达的话：

$$\begin{aligned} T &= 1 / F \\ &= 1 / 40\text{Hz} \\ &= 25 \text{ ms} \end{aligned}$$

这也是20~28行的定时器要每隔25ms产生一次定时的原因。因为每隔25ms，isStart寄存器的最低位就会被拉低，然后“draw function”就会被执行。换句话说，定时器的存在是为了充当“[仿顺序操作](#)”模块的“Start\_Sig”信号。当然也可以这样说“Start\_Sig和Done\_Sig都是发生在液晶控制模块的内部”（液晶控制模块自己自动使能自己）。

#### *lcd\_interface.v*

```

1. module lcd_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Write_En_Sig,
7.     input [9:0]Write_Addr_Sig,
8.     input [7:0]Write_Data,
9.
10.    output [3:0]SPI_Out // [3]CS [2]A0 [1]SCLK [0]SDA
11. );
12.
13. ****
14.
15. wire [7:0]Read_Data;
16.
17. lcd_ram_module U1
18. (
19.     .CLK( CLK ),
20.     .RSTn( RSTn ),
21.     .Write_En_Sig( Write_En_Sig ),           // input - from top
22.     .Write_Addr_Sig( Write_Addr_Sig ),       // input - from top
23.     .Write_Data( Write_Data ),               // input - from top
24.     .Read_Addr_Sig( Read_Addr_Sig ),         // input - from U2

```

```
25.      .Read_Data( Read_Data )          // output - to U2
26. );
27.
28. /*****
29.
30.     wire [9:0]Read_Addr_Sig;
31.     wire SPI_Start_Sig;
32.     wire [9:0]SPI_Data;
33.
34. lcd_control_module U2
35. (
36.     .CLK( CLK ),
37.     .RSTn( RSTn ),
38.     .Read_Data( Read_Data ),           // input - from U1
39.     .Read_Addr_Sig( Read_Addr_Sig ), // output - to U1
40.     .SPI_Done_Sig( SPI_Done_Sig ),   // input - from U3
41.     .SPI_Start_Sig( SPI_Start_Sig ), // output - to U3
42.     .SPI_Data( SPI_Data )          // output - to U3
43. );
44.
45. *****/
46.
47.     wire SPI_Done_Sig;
48.
49. spi_write_module U3
50. (
51.     .CLK( CLK ),
52.     .RSTn( RSTn ),
53.     .Start_Sig( SPI_Start_Sig ),      // input - from U2
54.     .SPI_Data( SPI_Data ),          // input - from U2
55.     .Done_Sig( SPI_Done_Sig ),      // output - to U2
56.     .SPI_Out( SPI_Out )           // output - to top
57. );
58.
59. *****/
60.
61. endmodule
```

该组合模块和 "图形" 基本上都是相似的，自己看着办吧。

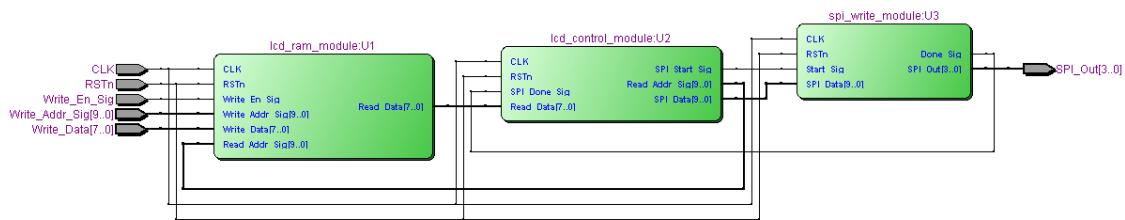
实验二十说明:

如果实验十二和实验二十相比较，实验二十的 RAM 模块替代了实验十二的 ROM 模块作为图像信息的储存器。SPI 写模块没有任何改变。实验二十的液晶控制模块使用了“**命令仿顺序操作**”的方法来整合了实验十二的 initial\_module.v 和 draw\_module.v。

此外液晶控制模块在初始化的时候，会对液晶资源执行初始化的操作。然后每个 25ms，液晶控制模块都会从 RAM 模块读取图像信息，用于液晶的显示驱动。

所以说 vga\_interface.v 和 lcd\_interface.v 有许多相同的地方。基本上它们都是使用同一个显示原理，就是提高扫描频率，利用肉眼的弱点，时而产生动态扫描的效果。

完成的扩展图：

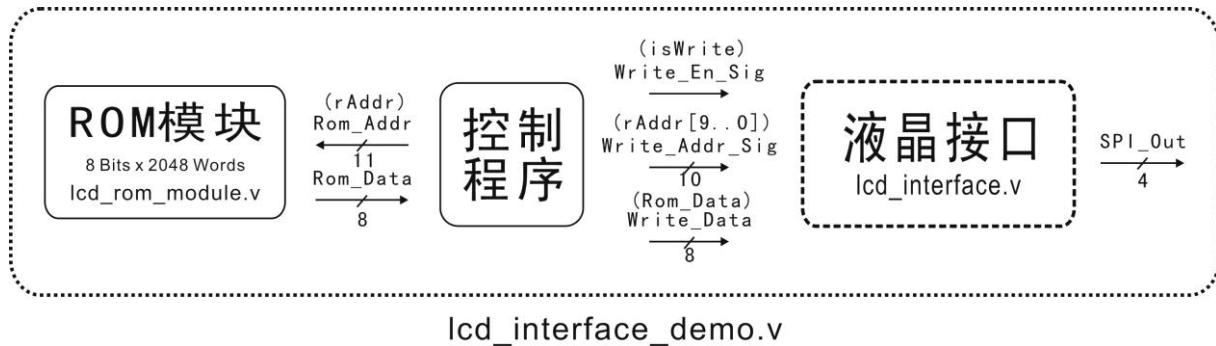


实验二十结论：

调用 lcd\_interface.v 模块如同对 RAM 模块写入信息。

## 实验二十演示：

这一章主要是演示 lcd\_interface.v 如何调用。



在组合模块 lcd\_interface\_demo.v 中 ROM 模块的储存空间是 8 Bits x 2048 Words，我们知道一副分辨率 64 x 128 的图像所占的储存空间是 8 Bits x 1024 Words，该 ROM 模块亦即包含两副图像信息。



皮卡丘动作 1



比卡丘动作 2

就是以上这只小可爱的两幅图像信息。换句话说，该演示是要实现动画。

lcd\_interface\_demo.v 的大致操作如下：

- (一) 从 ROM 模块读第一副图像信息，然后写入 lcd\_interface.v 的 RAM 模块里。
- (二) 延迟大约 500ms
- (三) 从 ROM 模块读第一副图像信息，然后写入 lcd\_interface.v 的 RAM 模块里。
- (四) 延迟大约 500ms
- (五) 重复步骤 1~4。

lcd\_interface\_demo.v

```
1. module lcd_interface_demo
2. (
```

```
3.      input CLK,
4.      input RSTn,
5.
6.      output [3:0]SPI_Out
7.  );
8.
9.  /***** */
10.
11. parameter T1MS = 16'd49999;
12.
13. /***** */
14.
15. reg [15:0]C1;
16.
17. always @ ( posedge CLK or negedge RSTn )
18.     if( !RSTn )
19.         C1 <= 16'd0;
20.     else if( C1 == T1MS )
21.         C1 <= 16'd0;
22.     else if( isCount )
23.         C1 <= C1 + 1'b1;
24.     else
25.         C1 <= 16'd0;
26.
27. /***** */
28.
29. reg [9:0]CMS;
30.
31. always @ ( posedge CLK or negedge RSTn )
32.     if( !RSTn )
33.         CMS <= 10'd0;
34.     else if( CMS == rTimes )
35.         CMS <= 10'd0;
36.     else if( C1 == T1MS )
37.         CMS <= CMS + 1'b1;
38.
39. /***** */
40.
41. reg [1:0]Z;
42.
43. always @ ( posedge CLK or negedge RSTn )
44.     if( !RSTn )
45.         Z <= 2'd0;
46.     else
```

## Verilog HDL 那些事儿 – 建模篇

```
47.         case( i )
48.
49.             0: Z <= 2'd0;
50.             2: Z <= 2'd1;
51.
52.         endcase
53.
54.     /***** */
55.
56.     reg [3:0]i;
57.     reg [10:0]rAddr;
58.     reg [9:0]rTimes;
59.     reg isWrite;
60.     reg isCount;
61.     reg [8:0]X;
62.     reg [3:0]Y;
63.
64.     always @ ( posedge CLK or negedge RSTn )
65.         if( !RSTn )
66.             begin
67.                 i <= 4'd0;
68.                 rAddr <= 11'd0;
69.                 rTimes <= 10'd100;
70.                 isWrite <= 1'b0;
71.                 isCount <= 1'b0;
72.                 X <= 9'd0;
73.                 Y <= 4'd0;
74.             end
75.         else
76.             case( i )
77.
78.                 0, 2:
79.                     if( Y == 8 ) begin Y <= 4'd0; i <= i + 1'b1; isWrite <= 1'b0; end
80.                     else if( X == 128 ) begin X <= 8'd0; Y <= Y + 1'b1; end
81.                     else begin rAddr = X + ( Y << 7 ) + ( Z << 10 ); X <= X + 1'b1; isWrite <= 1'b1; end
82.
83.                 1, 3:
84.                     if( CMS == rTimes ) begin isCount <= 1'b0; i <= i + 1'b1; end
85.                     else begin isCount <= 1'b1; rTimes <= 10'd500; end
86.
87.                 4:
88.                     i <= 4'd0;
89.
90.             endcase
```

```

91.
92.      *****/
93.
94.      wire [7:0]Rom_Data;
95.
96.      lcd_rom_module U1
97.      (
98.          .clock( CLK ),
99.          .address( rAddr ),
100.         .q( Rom_Data )
101.     );
102.
103.    *****/
104.
105.    lcd_interface U2
106.    (
107.        .CLK( CLK ),
108.        .RSTn( RSTn ),
109.        .Write_En_Sig( isWrite ),
110.        .Write_Addr_Sig( rAddr[9:0] ),
111.        .Write_Data( Rom_Data ),
112.        .SPI_Out( SPI_Out )
113.    );
114.
115.    *****/
116.
117. endmodule

```

在 11 行声明了 1ms 的常量。15~25 行是 1ms 的定时器，然而 29~37 行是秒级的计数器。在 94~101 行实例化了 ROM 模块，而且在 105~113 行实例化了 lcd\_interface.v。

64~90 行是该模块的控制程序。X 寄存器计数列填充 (61 行)，Y 寄存器计数行切换 (62 行)，Z 寄存器控制图像切换 (41 行)。那么图像寻址地址的表达式是  $X + (Y \ll 7) + (Z \ll 10)$ 。 $X \ll 7$  表示了一行有 128 长度， $Z \ll 10$  表示了一副图像有 1024 的长度。

在 79~81 行表示了，从 ROM 模块 读取图像信息至 lcd\_interface.v 的 RAM 模块的操作。84~85 行是延迟 500ms 的操作。

当步骤 0 的时候 (78 行)，Z 的值是 0 (49 行)，也就是第一副图像信息被选择，然后在 79~81 行，会将 ROM 模块 0~1023 的图像信息写入 lcd\_interface.v 的 RAM 模块。当完成 8 次的 128 次列填充，79 行的 if 条件就会成立，i 会递增以示下一个步骤。(注意，在 79~81 行的操作的期间 isWrite 一直被拉高，也就是说 lcd\_interface.v 的 Write\_En\_Sig 也是一直被拉高 - 81 行。)

当步骤等于 1 的时候（83 行），会产生 500ms 的延迟效果。然后 i 会递增，以示下一个步骤。

当步骤 2 的时候（78 行），Z 的值是 1（50 行），也就是说第二幅图像信息被选中。接下来的动作，和步骤 0 一样，只是图像信息不同而已。

当步骤等于 3 的时候（83 行），同样也会产生 500ms 的延迟效果。i 递增 ...

步骤 4（87 行）将 i 清零，以示重复执行步骤 0~3。（在这里有一个重点，就是在 110 行，Write\_Addr\_Sig 信号是由 rAddr[9..0] 驱动。）

实验二十演示说明：

没有什么特别的。对 lcd\_interface.v 的调用如同对 RAM 模块写入信息而已。

实验二十结论说明：

嗯！封装以后的 lcd\_interface.v，调用的工作都非常方便了。

## 5.8 实验二十一：RTC 接口

在 5.1 章中，笔者说过“**每一件硬件资源的封装，都有自己的考虑**”。

`key_interface.v` 考虑了“**5 个同样功能按键**”。

`smg_interface.v` 考虑了“**6 位数码管的显示**”。

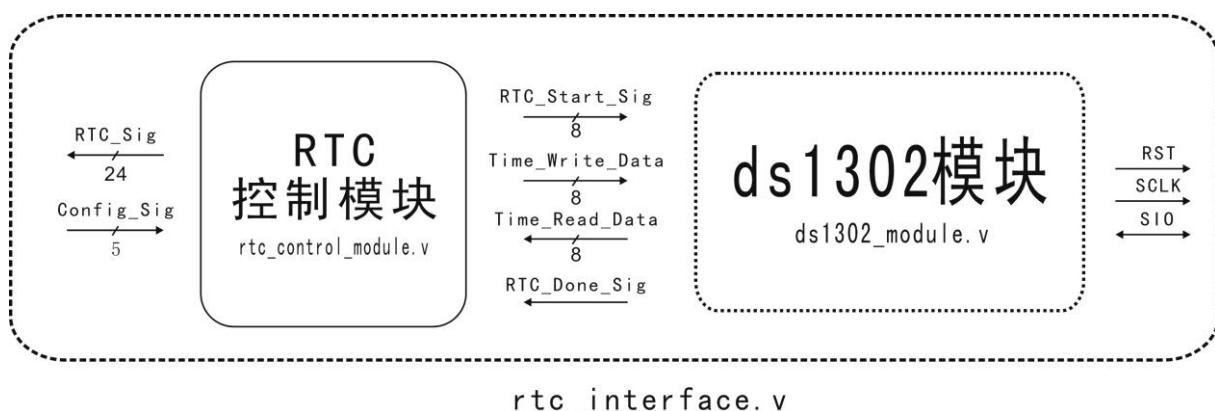
`beep_interface.v`, `ps2_interface.v` 考虑了“**利用 FIFO 成为独立化的接口**”。

`tx_interface.v`, `rx_interface.v` 和上述蜂鸣器接口与 PS2 接口考虑同样的事情。

`vga_interface.v`, `lcd_interface.v` 考虑了“**利用 RAM 成为独立化的接口**”。

当然 DS1302 模块也离不开“**封装**”，但是 DS1302 模块的封装会比较特别，因为它驱动的对象是 RTC 芯片。如果只是“**单纯的驱动**”而已，那么 ds1302 模块已经是却却有余。

我们要封装 DS1302 模块成为 RTC 接口当然有我们自己的“**考虑**”。笔者的想法很简单，笔者只要为 RTC 芯片 加入“**驱动**”，“**配置**”，“**输出**”这三个功能即可。



上图是组合模块 `rtc_interface.v`，该组合模块包含了“**驱动**”的 ds1302 模块，“**显示**”和“**配置**”的 RTC 控制模块。RTC 控制模块扮演着“**配置**”和“**显示**”的同时，它也扮演着控制 ds1302 模块的角色。在 4.3 章中（实验十三）我们知道 ds1302 模块它可以支持 8 种命令，然而我们可以基于这些基础，来决定“**配置**”的设计方案。

我们先来复习 ds1302 模块 可以支持的 8 种命令：

RTC_Start_Sig[ 7..0 ]	
位命令	功能
1000_0000	关闭写保护
0100_0000	变更时寄存器
0010_0000	变更分寄存器
0001_0000	变更秒寄存器

0000_1000	开启写保护
0000_0100	读取时寄存器
0000_0010	读取分寄存器
0000_0001	读取秒寄存器

RTC 控制模块除了“驱动”以外，还有“输出”这个工作。的 RTC\_Sig 信号它包含了 24 位位宽，然而位的分配如下：

RTC_Sig[23..0]		
时间分配	位分配	时间
时钟[23..16]	[23..20]	时钟-十位
	[19..16]	时钟-个位
分钟[15..8]	[15..12]	分钟-十位
	[11..8]	分钟-个位
秒钟[7..0]	[7..4]	秒钟-十位
	[3..0]	秒钟-个位

接下来的“配置”设计方案和 Config\_Sig 信号有关。Config\_Sig 包含 5 位位宽，然而位分配如下：

Config_Sig[4..0]	
位分配	意义
[4]	进入配置模式 退出配置模式
[3]	+1 操作
[2]	-1 操作
[1]	向左切换 (时 <= 分 <= 秒)
[0]	向右切换 (时 => 分 => 秒)

Config\_Sig 信号的每一个“位”都对“高脉冲敏感”，换句话说，如果某“位”接收“一个高脉冲”就有“一次性的操作”。假设 Config\_Sig[4]接收一个高脉冲就“进入配置模式”。然后隔一段时间之后 Config\_Sig[4] 再接收一个高脉冲就会“退出配置模式”。

笔者先大致的说明一下 RTC 接口的功能：

（在这里笔者需要强调一点，在这里所谓的“时钟”不是我们生活概念上的“时钟”，而是“时分秒”中的时间单位的“时-钟”。）

在一开始的时候 RTC 接口 初始化 DS1302 芯片，将时钟，分钟，秒钟都配置为 00。然后 RTC 接口，会从 00-00-00 开始计时。换句话说，在初始的状态 RTC\_Sig 的输出是 24'h00\_00\_00。

假设 Config\_Sig[4] 接收一个高脉冲，DS1302 芯片就停止计时 RTC\_Sig 输出也会停止更新，这时候 RTC 接口就进入配置模式。

在配置模式中“**配置时钟**”作为进入配置后的默认配置选项。如果 Config\_Sig[3] 接收一个高脉冲，当前的“**时钟值**”就会递增。反之，如果 Config\_Sig[2] 接收一个高脉冲，当前的“**时钟值**”就会递减。“**时钟值**”最大的值是 23，最小值是 00。

假设我要更动“**分钟**”，Config\_Sig[0] 就要接收一个高脉冲，从“**配置时钟**”向右切换“**配置分钟**”。和“**配置时钟**”同样的原理。如果此时 Config\_Sig[3] 接收一个高脉冲，当前的“**分钟值**”就会递增，反之 Config\_Sig[2] 接收一个高脉冲会使得当前的“**分钟值**”递减。“**分钟值**”最大的值是 59，最小值是 00。

如果接下来我要配置“**秒钟**”，Config\_Sig[0] 就要接收一个高脉冲。如果接下来笔者又要配置“**时钟**”，Config\_Sig[1] 就要接收一个高脉冲。“**秒钟值**”的最大值是 59，最小值是 00。

在这里有一点必须注意的是：

在“**配置模式**”中，时钟值的更动会更新 RTC\_Sig 的输出。假设笔者在当前的配置模式中，笔者将“**时钟值**”更新为 23，那么 RTC\_Sig 的输出 更新为 24'h23\_00\_00。

“**配置时钟**”是“**切换**”的最左边，“**配置秒钟**”是“**切换**”的最右边，然而“**配置分钟**”是“**切换**”的中间。也就是说：

**时钟 <=> 分钟 <=> 秒钟**

最后，假设笔者的最终决定是 12 - 20 - 12 这时候 RTC\_Sig 的输出是 24'h12\_20\_12。然后笔者退出配置模式，那么 Config\_Sig[4] 需要接收一个高脉冲，RTC 接口就会从“**配置模式**”退出至“**正常模式**”。当返回“**正常模式**”，RTC 接口会从 12-20-12 开始计时。

*rtc\_control\_module.v*

```

1. module rtc_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [4:0]Config_Sig, // [4]Middle [3]Up [2]Down [1]Left [0]Right
7.
8.     input RTC_Done_Sig,
9.     output [7:0]RTC_Start_Sig,
10.
11.    input [7:0]Time_Read_Data,
12.    output [7:0]Time_Write_Data,
```

## Verilog HDL 那些事儿 - 建模篇

```
12.  
13.      output [23:0]RTC_Sig  
14. );  
15.
```

3~13 行是 `rtc_control_module.v` 的输入输出定义。

```
16.      /*****  
17.  
18.      reg isConfig;  
19.  
20.      always @ ( posedge CLK or negedge RSTn )  
21.          if( !RSTn )  
22.              isConfig <= 1'b0;  
23.          else if( Config_Sig[4] )  
24.              isConfig <= ~isConfig;  
25.  
26.      *****/
```

在第 18 行定义了 `isConfig` 的标志寄存器。`isConfig` 的默认值是逻辑 0，也就是说在初始化的状态下，RTC 接口会进入“正常模式”（22 行）。如果 `Config_Sig[4]` 接收到一个高脉冲 `isConfig` 的值就会介于 0~1 之间切换（23~24），亦即 `isConfig` 逻辑 0 代表“正常模式”，逻辑 1 代表“配置模式”。

```
27.  
28.      reg [3:0]i;  
29.      reg [7:0]rData;  
30.      reg [7:0]Hour;  
31.      reg [7:0]Min;  
32.      reg [7:0]Sec;  
33.      reg [7:0]Temp;  
34.      reg [7:0]Comp;  
35.      reg [3:0]Go;  
36.      reg [7:0]isStart;  
37.  
38.      always @ ( posedge CLK or negedge RSTn )  
39.          if( !RSTn )  
40.              begin  
41.                  i <= 4'd0;  
42.                  rData <= 8'd0;  
43.                  Hour <= 8'd0;  
44.                  Min <= 8'd0;  
45.                  Sec <= 8'd0;  
46.                  Temp <= 8'd0;
```

```

47.          Comp <= 8'd0;
48.          Go <= 4'd0;
49.          isStart <= 8'd0;
50.      end

```

28~50 行是核心部分有关的寄存器声明和寄存器初始化。rData 值作为 Time\_Write\_Data 的驱动。Hour , Min, Sec 是作为“**时分秒种**”的暂存器。Temp 和 Comp 只作为“**时间值**”递增和递减操作的暂存器。Go 寄存器是步骤 i 的返回指示。isStart 是作为使能 ds1302 模块的命令寄存器。所有寄存器的初始化都是清零状态。

```

51. else
52.     case( i )
53.
54.     /***** // Initial
55.
56.     0: // write unprotect
57.     if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
58.     else begin isStart <= 8'b1000_0000; rData <= 8'd0; end
59.
60.     1: // initial hour
61.     if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
62.     else begin isStart <= 8'b0100_0000; rData <= Hour; end
63.
64.     2: // initial min
65.     if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
66.     else begin isStart <= 8'b0010_0000; rData <= Min; end
67.
68.     3: // initial sec and start clock
69.     if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
70.     else begin isStart <= 8'b0001_0000; rData <= Sec; end
71.

```

在一开始的时候，RTC 时钟写进入初始化状态：

步骤 0（56~58 行）关闭 DS1302 芯片的写保护。步骤 1（60~62 行）是初始化 DS1302 芯片的“**时钟值**”。步骤 2（64~66 行）是初始化 DS1302 芯片的“**分钟值**”。步骤 3（68~70 行）是初始化 DS1302 芯片的“**秒钟值**”，同时也是启动 DS1302 芯片开始计数。所以说，初始化状态的“**时间**”是 24'h00\_00\_00。

```

72.     /***** // Normal Status
73.
74.     4:
75.     if( isConfig ) begin i <= 4'd8; end
76.     else i <= i + 1'b1;

```

## Verilog HDL 那些事儿 – 建模篇

```
77.  
78.      5: // Read hour  
79.      if( RTC_Done_Sig ) begin Hour <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end  
80.      else isStart <= 8'b0000_0100;  
81.  
82.      6: // Read min  
83.      if( RTC_Done_Sig ) begin Min <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end  
84.      else isStart <= 8'b0000_0010;  
85.  
86.      7: // Read sec  
87.      if( RTC_Done_Sig ) begin Sec <= Time_Read_Data; isStart <= 8'd0; i <= 4'd4; end  
88.      else isStart <= 8'b0000_0001;  
89.
```

步骤 4~7，是正常模式。当进入步骤 4（74~76），if 条件就会先判断，isConfig 是否被拉高？如果 isConfig 不被拉高，就进入下一个步骤。步骤 5 是执行“[读时钟](#)”的命令，步骤 6 是执行“[读分钟](#)”的操作，步骤 7 是读秒钟的操作。最后会返回步骤 4。换句话说步骤 4~7 是正常模式的循环。但是，一旦步骤 4 中 if 条件成立的话，就会进入“[配置模式](#)”。

```
90.  ****// Pre-config  
91.  
92.  8: // Set off clock and initial "Temp"  
93.  if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end  
94.  else begin Temp <= Hour; isStart <= 8'b0001_0000; rData <= 8'b1000_0000; end
```

步骤 8 是预配置，也就是说在关闭 ds1302 芯片计时的同时，更新 Temp 寄存器为 Hour 寄存器的值，这一点很重要，因为下一个步骤的操作需要。（在 DS1302 秒寄存器的最高位写入 1，亦即关闭计时）。可能读者会问“[在为 DS1302 的秒寄存器写入 8'b1000\\_0000，的时候，会不会破坏当前秒寄存器的值呢？](#)”。秒寄存器的值在“[正常模式](#)”时已经被暂存在 Sec。

```
95.  
96.  ****// Config status  
97.  
98.  9: // Config hour  
99.  if( !isConfig ) i <= 4'd1;  
100. else if( Config_Sig[3] ) begin Temp <= Hour; Comp <= 8'h23; Go <= 4'd9; i <= 4'd12; end  
101. else if( Config_Sig[2] ) begin Temp <= Hour; Go <= 4'd9; i <= 4'd15; end  
102. else if( Config_Sig[0] ) begin Temp <= Min; i <= i + 1'b1; end  
103. else Hour <= Temp;  
104.  
105. 10: // Config min
```

```

106. if( !isConfig ) i <= 4'd1;
107. else if( Config_Sig[3] ) begin Temp <= Min; Comp <= 8'h59; Go <= 4'd10; i <= 4'd12; end
108. else if( Config_Sig[2] ) begin Temp <= Min; Go <= 4'd10; i <= 4'd15; end
109. else if( Config_Sig[1] ) begin Temp <= Hour; i <= i - 1'b1; end
110. else if( Config_Sig[0] ) begin Temp <= Sec; i <= i + 1'b1; end
111. else Min <= Temp;
112.
113. 11:// Config sec
114. if( !isConfig ) i <= 4'd1;
115. else if( Config_Sig[3] ) begin Temp <= Sec; Comp <= 8'h59; Go <= 4'd11; i <= 4'd12; end
116. else if( Config_Sig[2] ) begin Temp <= Sec; Go <= 4'd11; i <= 4'd15; end
117. else if( Config_Sig[1] ) begin Temp <= Min; i <= i - 1'b1; end
118. else Sec <= Temp;
119.

```

步骤 9~11 是配置模式。在 103 行读者是否看见这样一句代码，将 Temp 的值赋予 Hour 寄存器。读者尝试想象，如果在预配置至下（步骤 8），没有为 Temp 赋予 Hour 的值。那么当进入配置模式，无疑 103 行的代码会被执行，Temp 的初值为 0，Hour 的值已不是被破坏了？此外 111 行，118 行有类似的作用 ...

步骤 9 是“[配置时钟](#)”，在 99 行的 if 条件先判断 isConfig 是否为逻辑 0，如果是就退出配置模式，如果不是就处于“配置时钟”的状态。103 行的代码，会作为默认一直被执行着。

当 Config\_Sig[3] 接收一个高脉冲，亦即“[时钟值递增](#)”的操作。Temp 会暂存 Hour 的值，然后 Comp 寄存器陪暂存 Hour 的最大值，也就是 8'h23。Go 寄存器指示着步骤 9，然后 i 寄存器被赋予 4'd12，该表示下一个执行步骤为 12。

```

120. ****// Increase
121.
122. 12:
123. if( Temp < Comp ) begin Temp <= Temp + 1'b1; i <= i + 1'b1; end
124. else begin Temp <= Comp; i <= Go; end
125.
126. 13:
127. if( Temp[3:0] > 4'd9 ) begin Temp <= { Temp[7:4] + 1'b1 , 4'd0 }; i <= i + 1'b1; end
128. else i <= i + 1'b1;
129.
130. 14:
131. if( Temp[7:4] > Comp[7:4] ) begin Temp <= Comp; i <= Go; end
132. else i <= Go;
133.

```

步骤 12~14 是“[值递增](#)”操作。在 123 行 if 条件会先判断，如果 Temp 的值小于 Comp

的值（如果当前的时钟值小于最大的时钟值的话），Temp 会递增。然后会进入步骤 13。否则的话 Temp 的值会赋予 Comp 的值（当前时钟值赋予最大的时钟值），然后返回 Go 指示的步骤（如果当前是“[配置时钟](#)”，就会返回“[配置时钟的步骤](#)”，这也是为什么在 100 行，Go 会指向当前的执行步骤）。

步骤 13 是进位操作，在 127 行 if 条件会判断 Temp 的个位（时钟的个位）是否大于 9，如果“是”就执行进位操作，然后 i 递增以示下一个步骤。否则 i 也会递增以示下一个步骤。

步骤 14，在 131 行的 if 条件会判断 Temp 的十位（当前“[时钟值](#)”的十位）是否大于 Comp 的十位（“[时钟值](#)”最大值的十位），如果“是”（亦即“[当前时钟值](#)”大于“[时钟值最大值](#)”）Temp 就赋予 Comp 的值（当前“[时钟值](#)”赋予“[时钟值最大值](#)”）。然后 i 赋予 Go 的值，以示返回“[配置时钟](#)”的步骤，亦即步骤 9。否则，同样 i 会赋予 Go 的值，返回步骤 9。

```
95.  
96. //***** // Config status  
97.  
98. 9: // Config hour  
99. if( !isConfig ) i <= 4'd1;  
100. else if( Config_Sig[3] ) begin Temp <= Hour; Comp <= 8'h23; Go <= 4'd9; i <= 4'd12; end  
101. else if( Config_Sig[2] ) begin Temp <= Hour; Go <= 4'd9; i <= 4'd15; end  
102. else if( Config_Sig[0] ) begin Temp <= Min; i <= i + 1'b1; end  
103. else Hour <= Temp;  
104.  
105. 10: // Config min  
106. if( !isConfig ) i <= 4'd1;  
107. else if( Config_Sig[3] ) begin Temp <= Min; Comp <= 8'h59; Go <= 4'd10; i <= 4'd12; end  
108. else if( Config_Sig[2] ) begin Temp <= Min; Go <= 4'd10; i <= 4'd15; end  
109. else if( Config_Sig[1] ) begin Temp <= Hour; i <= i - 1'b1; end  
110. else if( Config_Sig[0] ) begin Temp <= Sec; i <= i + 1'b1; end  
111. else Min <= Temp;  
112.  
113. 11: // Config sec  
114. if( !isConfig ) i <= 4'd1;  
115. else if( Config_Sig[3] ) begin Temp <= Sec; Comp <= 8'h59; Go <= 4'd11; i <= 4'd12; end  
116. else if( Config_Sig[2] ) begin Temp <= Sec; Go <= 4'd11; i <= 4'd15; end  
117. else if( Config_Sig[1] ) begin Temp <= Min; i <= i - 1'b1; end  
118. else Sec <= Temp;  
119.
```

在步骤 9 时，如果 Config\_Sig[2] 接收一个高脉冲，Temp 就会暂存 Hour 的值（Temp 暂存当前的“[时钟值](#)”），然后 Go 寄存器指向当前步骤，亦即步骤 9。i 寄存器被赋予 4'd15，也就是说下一个步骤会进入步骤 15。

```

134.   **** // Decrease
135.
136.   15:
137.   if( Temp[3:0] > 0 ) begin Temp[3:0] <= Temp[3:0] - 1'b1; i <= Go; end
138.   else if( Temp[3:0] == 0 && Temp[7:4] > 0 ) begin Temp <= { Temp[7:4] - 1'b1, 4'd9 }; i <= Go; end
139.   else begin Temp <= 8'd0; i <= Go; end
140.

```

步骤 15 是递减操作。在 137 行 if 条件会先判断 Temp 的个位（当前“时钟值”的个位）大于 0？如果是 Temp 的个位就会递减（当前“时钟值”的个位递减 1）。然后 i 寄存器会指向 Go 的值，亦即返回步骤 9，返回“配置时钟”。

如果 137 行的 if 不成立，就会判断 138 行的 if 条件：Temp 的个位等于 0 的同时 Temp 的十位又大于 0（当前“时钟值”的个位等于 0，而且当前“时钟值”的十位大于 0），就会将 Temp 的十位递减，Temp 的个位赋予 4'd9（将当前“时钟值”的十位递减 1，当前“时钟值”的个位赋予 9）。最后 i 寄存器会指向返回 Go 的值，亦即步骤 9。

如果 137 行和 138 行的 if 条件不成立（139 行），即表示 Temp 的值（当前的“时钟值”是 8'h00）。i 寄存器会指向返回 Go 的值，亦即步骤 9。

```

95.
96.   **** // Config status
97.
98.   9: // Config hour
99.   if( !isConfig ) i <= 4'd1;
100.  else if( Config_Sig[3] ) begin Temp <= Hour; Comp <= 8'h23; Go <= 4'd9; i <= 4'd12; end
101.  else if( Config_Sig[2] ) begin Temp <= Hour; Go <= 4'd9; i <= 4'd15; end
102.  else if( Config_Sig[0] ) begin Temp <= Min; i <= i + 1'b1; end
103.  else Hour <= Temp;
104.
105.  10: // Config min
106.  if( !isConfig ) i <= 4'd1;
107.  else if( Config_Sig[3] ) begin Temp <= Min; Comp <= 8'h59; Go <= 4'd10; i <= 4'd12; end
108.  else if( Config_Sig[2] ) begin Temp <= Min; Go <= 4'd10; i <= 4'd15; end
109.  else if( Config_Sig[1] ) begin Temp <= Hour; i <= i - 1'b1; end
110.  else if( Config_Sig[0] ) begin Temp <= Sec; i <= i + 1'b1; end
111.  else Min <= Temp;
112.
113.  11: // Config sec
114.  if( !isConfig ) i <= 4'd1;
115.  else if( Config_Sig[3] ) begin Temp <= Sec; Comp <= 8'h59; Go <= 4'd11; i <= 4'd12; end
116.  else if( Config_Sig[2] ) begin Temp <= Sec; Go <= 4'd11; i <= 4'd15; end
117.  else if( Config_Sig[1] ) begin Temp <= Min; i <= i - 1'b1; end

```

```
118.    else Sec <= Temp;  
119.
```

在 103 行的这段代码很重要，因为无论是递增操作，或者是递减操作。最后操作的结果（Temp 值）都要更新于 Hour 寄存器。

如果 Config\_Sig[0] 接收一个高脉冲（102 行），亦即是向右边切换，换句话说就是从“[时钟配置](#)”向右切换到“[分钟配置](#)”。此时 Temp 的值必须更新为 Min 的值。和 103 行同样的道理。当 Config\_Sig[0] 就收一个高脉冲，步骤 9 会递增至步骤 10。

在步骤 10（105~111 行），无疑在 111 行的代码会被执行，如果在 102 行 Temp 值没有被更新为 Min 的值，不难想象得到 Min 寄存器的值与 Hour 寄存器的值是一样的，这显然是严重的 BUG。

“[分钟配置](#)”和“[时钟配置](#)”的“[递增操作](#)”或者“[递减操作](#)”都是大同小异。Config\_Sig[3] 如果被触发，就会进入步骤 12，亦即“[递增操作的步骤](#)”。Config\_Sig[2] 被触发就会进入步骤 15 的“[递减操作](#)”。

不同之处是：Temp 暂存的再也不是 Hour 而是 Min 的值，Comp 的最大值是 8'h59，Go 的寄存器指向“[分钟配置](#)”的步骤。

如果 Config\_Sig[0] 接收一个高脉冲，i 会递增，“[分钟配置](#)”向右切换至“[秒钟配置](#)”（108 行），Temp 会暂存 Sec 的值。如果 Config\_Sig[1] 接收一个高脉冲，i 会递减，“[分钟配置](#)”向左切换至“[时钟配置](#)”（109 行），Temp 会暂存 Hour 的值。

在步骤 9(98~103 行)没有 Config\_Sig[1]，在步骤 11(113~118 行)没有 Congfig\_Sig[0]。这也表示“[时钟配置 <=> 分钟配置 <=> 秒钟配置](#)”，配置模式会在这 3 个时间配置之间切换。换句话说“[时钟配置](#)”是“[向左切换的最边](#)”，然而“[秒钟配置](#)”是“[向右切换的最边](#)”这一个事实。

```
120.    /***** // Increase  
121.  
122.    12:  
123.    if( Temp < Comp ) begin Temp <= Temp + 1'b1; i <= i + 1'b1; end  
124.    else begin Temp <= Comp; i <= Go; end  
125.  
126.    13:  
127.    if( Temp[3:0] > 4'd9 ) begin Temp <= { Temp[7:4] + 1'b1 , 4'd0 }; i <= i + 1'b1; end  
128.    else i <= i + 1'b1;  
129.  
130.    14:  
131.    if( Temp[7:4] > Comp[7:4] ) begin Temp <= Comp; i <= Go; end  
132.    else i <= Go;  
133.
```

```
134.   /***** // Decrease
135.
136.   15:
137.   if( Temp[3:0] > 0 ) begin Temp[3:0] <= Temp[3:0] - 1'b1; i <= Go; end
138.   else if( Temp[3:0] == 0 && Temp[7:4] > 0 ) begin Temp <= { Temp[7:4] - 1'b1, 4'd9 }; i <= Go; end
139.   else begin Temp <= 8'd0; i <= Go; end
```

在步骤 12~14 的递增操作和在步骤 15 的递增操作，在某种程度上，可以把它们看成为“[递增函数和递减函数](#)”。根据一些常有的设计方法，我们必定会建立一个“[时钟递增](#)”，“[分钟递增](#)”，“[秒钟递增](#)”，“[时钟递减](#)”，“[分钟递减](#)”和“[秒钟递减](#)”等的多个操作步骤，这无疑是会消耗许多的逻辑资源。

在某程度的根本上“[时钟递增](#)”，“[分钟递增](#)”，“[秒钟递增](#)”，“[时钟递减](#)”，“[分钟递减](#)”和“[秒钟递减](#)”等步骤，都可以共用一个“[递增步骤](#)”和“[递减步骤](#)”。但是问题就在于“[参数传递](#)”和“[参数返回](#)”是有关“[代码概念](#)”的操作。我们知道 Verilog HDL 语言，是“[硬件描述语言](#)”，它没有“[代码的概念](#)”…

这时候我们必须把思路往后推移。笔者还记得自己在学习“[微处理器](#)”的时候（在笔者的心目中微处理器是悲剧，和单片机是不同的东西），为了使两个值相加，必须将两个值载入“[操作空间](#)”，然后使用指令使它们相加。如果使用这个思路反映到步骤 12~14 和步骤 15 的“[递增递减操作](#)”。寄存器 Temp，寄存器 Comp，和寄存器 Go 等就有所谓的“[操作空间](#)”的意义。

```
98.    9: // Config hour  
99.    if( !isConfig ) i <= 4'd1;
```

```
105.   10: // Config min  
106.  if( !isConfig ) i <= 4'd1;
```

```
113.   11: // Config sec  
114.  if( !isConfig ) i <= 4'd1;
```

在步骤 9, 10, 11 期间，如果 99, 106, 114 行的 if 条件判断到 isConfig 被拉低的话。估计只有一件事情要发生，那就是“[退出配置模式](#)”。

```
72.    /***** // Normal Status  
73.  
74.    4:  
75.    if( isConfig ) begin i <= 4'd8; end  
76.    else i <= i + 1'b1;
```

我们知道要进入“[配置模式](#)” isConfig 必须是逻辑 1，然后在“[通常模式](#)”中，如果步骤 4 的 75 行 if 条件检测到，才会进入“[预配置模式](#)”（步骤 8），执行预设置的操作。当“[退出配置模式](#)”之后 i 寄存器会赋值为 4'd1，亦即表示返回步骤 1。

```
54.    /***** // Initial  
55.  
56.    0: // write unprotect  
57.    if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end  
58.    else begin isStart <= 8'b1000_0000; rData <= 8'd0; end
```

```

59.
60.    1: // initial hour
61.    if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
62.    else begin isStart <= 8'b0100_0000; rData <= Hour; end
63.
64.    2: // initial min
65.    if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
66.    else begin isStart <= 8'b0010_0000; rData <= Min; end
67.
68.    3: // initial sec and start clock
69.    if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
70.    else begin isStart <= 8'b0001_0000; rData <= Sec; end
71.

```

步骤 1，是初始化 Hour，但是关键点就是在 62 行的 rData <= Hour。同样的步骤 2 的 66 行和步骤 3 的 70 行都是同样的意义。就是把配置后的 Hour，Min，Sec 作为输入数据，然后调用 DS1302 模块的命令，针对 DS1302 芯片的时寄存器，分寄存器和秒寄存器执行更新。

```

72.    ****// Normal Status
73.
74.    4:
75.    if( isConfig ) begin i <= 4'd8; end
76.    else i <= i + 1'b1;
77.
78.    5: // Read hour
79.    if( RTC_Done_Sig ) begin Hour <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end
80.    else isStart <= 8'b0000_0100;
81.
82.    6: // Read min
83.    if( RTC_Done_Sig ) begin Min <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end
84.    else isStart <= 8'b0000_0010;
85.
86.    7: // Read sec
87.    if( RTC_Done_Sig ) begin Sec <= Time_Read_Data; isStart <= 8'd0; i <= 4'd4; end
88.    else isStart <= 8'b0000_0001;
89.

```

最后步骤 i 的流程也会进入 4~7 之间，也就是说 从“[配置模式的退出](#)”后会进入步骤 1 执行时间值更新的操作，然后会进入“[普通模式](#)”。

```

140.
141.        ****
142.

```

## Verilog HDL 那些事儿 – 建模篇

```
143.         endcase
144.
145. ****
146.
147.     assign RTC_Start_Sig = isStart;
148.     assign Time_Write_Data = rData;
149.     assign RTC_Sig = { Hour , Min , Sec };
150.
151. ****
152.
153.
154. endmodule
```

在148行 RTC\_Start\_Sig 由 isStart 命令寄存器驱动。在149行 Time\_Write\_Data 由 rData 寄存器驱动。在150行 RTC\_Sig 由 Hour, Min, Sec 寄存器联合驱动。

```
1. module rtc_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [4:0]Config_Sig, //[4]Middle [3]Up [2]Down [1]Left [0]Right
7.
8.     input RTC_Done_Sig,
9.     output [7:0]RTC_Start_Sig,
10.    input [7:0]Time_Read_Data,
11.    output [7:0]Time_Write_Data,
12.
13.    output [23:0]RTC_Sig
14. );
15.
16. ****
17.
18. reg isConfig;
19.
20. always @ ( posedge CLK or negedge RSTn )
21.     if( !RSTn )
22.         isConfig <= 1'b0;
23.     else if( Config_Sig[4] )
24.         isConfig <= ~isConfig;
25.
26. ****
27.
```

```
28.      reg [3:0]i;
29.      reg [7:0]rData;
30.      reg [7:0]Hour;
31.      reg [7:0]Min;
32.      reg [7:0]Sec;
33.      reg [7:0]Temp;
34.      reg [7:0]Comp;
35.      reg [3:0]Go;
36.      reg [7:0]isStart;
37.
38.      always @ ( posedge CLK or negedge RSTn )
39.          if( !RSTn )
40.              begin
41.                  i <= 4'd0;
42.                  rData <= 8'd0;
43.                  Hour <= 8'd0;
44.                  Min <= 8'd0;
45.                  Sec <= 8'd0;
46.                  Temp <= 8'd0;
47.                  Comp <= 8'd0;
48.                  Go <= 4'd0;
49.                  isStart <= 8'd0;
50.              end
51.          else
52.              case( i )
53.
54.                  /***** // Initial
55.
56.                  0: // write unprotect
57.                      if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
58.                      else begin isStart <= 8'b1000_0000; rData <= 8'd0; end
59.
60.                  1: // initial hour
61.                      if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
62.                      else begin isStart <= 8'b0100_0000; rData <= Hour; end
63.
64.                  2: // initial min
65.                      if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
66.                      else begin isStart <= 8'b0010_0000; rData <= Min; end
67.
68.                  3: // initial sec and start clock
69.                      if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
70.                      else begin isStart <= 8'b0001_0000; rData <= Sec; end
71.
```

## Verilog HDL 那些事儿 – 建模篇

```
72.      **** // Normal Status
73.
74.      4:
75.      if( isConfig ) begin i <= 4'd8; end
76.      else i <= i + 1'b1;
77.
78.      5: // Read hour
79.      if( RTC_Done_Sig ) begin Hour <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end
80.      else isStart <= 8'b0000_0100;
81.
82.      6: // Read min
83.      if( RTC_Done_Sig ) begin Min <= Time_Read_Data; isStart <= 8'd0; i <= i + 1'b1; end
84.      else isStart <= 8'b0000_0010;
85.
86.      7: // Read sec
87.      if( RTC_Done_Sig ) begin Sec <= Time_Read_Data; isStart <= 8'd0; i <= 4'd4; end
88.      else isStart <= 8'b0000_0001;
89.
90.      **** // Pre-config
91.
92.      8: // Set off clock and initial "Temp"
93.      if( RTC_Done_Sig ) begin isStart <= 8'd0; i <= i + 1'b1; end
94.      else begin Temp <= Hour; isStart <= 8'b0001_0000; rData <= 8'b1000_0000; end
95.
96.      **** // Config status
97.
98.      9: // Config hour
99.      if( !isConfig ) i <= 4'd1;
100.     else if( Config_Sig[3] ) begin Temp <= Hour; Comp <= 8'h23; Go <= 4'd9; i <= 4'd12; end
101.     else if( Config_Sig[2] ) begin Temp <= Hour; Go <= 4'd9; i <= 4'd15; end
102.     else if( Config_Sig[0] ) begin Temp <= Min; i <= i + 1'b1; end
103.     else Hour <= Temp;
104.
105.    10: // Config min
106.    if( !isConfig ) i <= 4'd1;
107.    else if( Config_Sig[3] ) begin Temp <= Min; Comp <= 8'h59; Go <= 4'd10; i <= 4'd12; end
108.    else if( Config_Sig[2] ) begin Temp <= Min; Go <= 4'd10; i <= 4'd15; end
109.    else if( Config_Sig[1] ) begin Temp <= Hour; i <= i - 1'b1; end
110.    else if( Config_Sig[0] ) begin Temp <= Sec; i <= i + 1'b1; end
111.    else Min <= Temp;
112.
113.    1: // Config sec
114.    if( !isConfig ) i <= 4'd1;
115.    else if( Config_Sig[3] ) begin Temp <= Sec; Comp <= 8'h59; Go <= 4'd11; i <= 4'd12; end
```

```

116.         else if( Config_Sig[2] ) begin Temp <= Sec; Go <= 4'd11; i <= 4'd15; end
117.         else if( Config_Sig[1] ) begin Temp <= Min; i <= i - 1'b1; end
118.         else Sec <= Temp;
119.
120.         /***** // Increase
121.
122.         12:
123.             if( Temp < Comp ) begin Temp <= Temp + 1'b1; i <= i + 1'b1; end
124.             else begin Temp <= Comp; i <= Go; end
125.
126.         13:
127.             if( Temp[3:0] > 4'd9 ) begin Temp <= { Temp[7:4] + 1'b1 , 4'd0 }; i <= i + 1'b1; end
128.             else i <= i + 1'b1;
129.
130.         14:
131.             if( Temp[7:4] > Comp[7:4] ) begin Temp <= Comp; i <= Go; end
132.             else i <= Go;
133.
134.         **** // Decrease
135.
136.         15:
137.             if( Temp[3:0] > 0 ) begin Temp[3:0] <= Temp[3:0] - 1'b1; i <= Go; end
138.             else if( Temp[3:0] == 0 && Temp[7:4] > 0 ) begin Temp <= { Temp[7:4] - 1'b1, 4'd9 }; i <= Go; end
139.             else begin Temp <= 8'd0; i <= Go; end
140.
141.
142.         ****/
143.
144.     endcase
145.
146.     ****/
147.
148.     assign RTC_Start_Sig = isStart;
149.     assign Time_Write_Data = rData;
150.     assign RTC_Sig = { Hour , Min , Sec };
151.
152.     ****/
153.
154.
155. endmodule

```

这是完成的代码，好好的浏览一番吧。

*rtc\_interface.v*

```
1. module rtc_interface
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [4:0]Config_Sig,
7.
8.     output [23:0]RTC_Sig,
9.
10.    output RST,
11.    output SCLK,
12.    inout SIO
13. );
14.
15. ****
16.
17.     wire [7:0]RTC_Start_Sig;
18.     wire [7:0]Time_Write_Data;
19.
20.     rtc_control_module U1
21.     (
22.         .CLK( CLK ),
23.         .RSTn( RSTn ),
24.         .Config_Sig( Config_Sig ),           // input - from top
25.         .RTC_Done_Sig( RTC_Done_Sig ),      // input - from U2
26.         .RTC_Start_Sig( RTC_Start_Sig ),    // output - to U2
27.         .Time_Read_Data( Time_Read_Data ), // input - from U2
28.         .Time_Write_Data( Time_Write_Data ),// output - to U2
29.         .RTC_Sig( RTC_Sig )                // output - to top
30.     );
31.
32. ****
33.
34.     wire RTC_Done_Sig;
35.     wire [7:0]Time_Read_Data;
36.
37.     ds1302_module U2
38.     (
39.         .CLK( CLK ),
40.         .RSTn( RSTn ),
```

```

41.      .Start_Sig( RTC_Start_Sig ),           // input - from U1
42.      .Done_Sig( RTC_Done_Sig ),           // output - to U1
43.      .Time_Write_Data( Time_Write_Data ), // input - from U1
44.      .Time_Read_Data( Time_Read_Data ),  // output - to U1
45.      .RST( RST ),                      // output - to top
46.      .SCLK( SCLK ),                    // output - to top
47.      .SIO( SIO )                      // inout - with top
48.  );
49.
50.
51.
52. endmodule

```

rtc\_interface.v 组合模块和“图形”是一模一样。

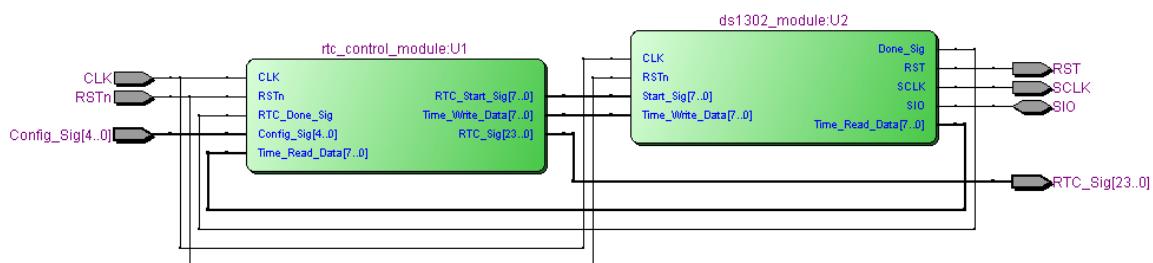
### 实验二十一说明:

说实话 RTC 接口的封装有一点难度。RTC 接口的“驱动”是由 DS1302 模块负责但是被 RTC 控制模块控制着。然而“显示”和“配置”等功能却是由 RTC 控制模块负责。

对于 RTC 接口的配置控制，我们只要知道如何调用 Config\_Sig 信号。Config\_Sig 信号的“每一位”都有“配置的用意”。笔者不得不承认“配置”在设计方面，确实有一点难度。但是困难归困难，为了未来，就要克服。

在这里读者只要明白了“RTC 控制模块如何操作”，自然而然会明白“RTC 控制模块的设计思路”。

完成后的扩展图：



### 实验二十一结论:

这一章实验主要是讲解如何为“DS1302 芯片”执行封装。

## 总结：

第五章终于完结了，在这里笔者来个简单的总结：

接口建模中所谓的“**最后工程**”是针对某个硬件“**有考虑**”的封装。

说道“**封装**”，我们必须考虑-经过“**封装**”后的模块都有“**独立性**”的特质。所以“**封装**”的模块会很有效的将 Verilog HDL 语言的特性显带出来。因为经过“**封装**”以后的模块，都能“**独立**”的运行起来，这也好比似“**并行**”的概念。

经过第五章的洗礼，读者们是不是领悟到任何一章内容都是在为下一章做好准备。在第二章中，笔者就提及过：“低级建模是模仿管理系统”的一种建模方法，一个大部分是由许多小部分组成，如果换成另外一个角度去想象：

- 每一个简单的功能模块，可以看似一位员工。
- 每一个简单的控制模块，可以看似一位领导。
- 每一个简单的组合模块，可以看似一组小组。
- 每一个组合模块再组合起来，可以看似一个大组。
- 每一个大组为某种“**目的**”存在，如果有独立运行的能力，就可以成为部门（接口）。
- 最后由许多接口组合起来，就成为一个“**系统**”。

这就是“**低级建模**”最基本思路。

低级建模对于每一个模块都有区分“**身份**”。这好比员工是员工，领导是领导，小组是小组，大组是大组，部门是部门，各个都有自己的特征。

无论是什么样子管理系统，员工和员工之间，领导和员工之间，部门经理和领导之间，必须和谐共处。这好比是“**低级建模**”中的“**代码风格**”。从实验一到实验二一，读者可能会发现到笔者所使用的“**代码风格**”都是清一色，笔者会很厚脸皮的告诉你，“**这是低级建模的固有代码结构**”。

很多读者一开始的时候可能会把“**步骤 i**”看成状态机。在创建“**低级建模**”的初头，笔者老是觉得“**典型的状态机**”用法实在是太麻烦了。如果是小代码量的建模，那么“**典型状态机**”是没有问题。但是遇见“**多级建模**”或者“**多状态**”的时候“**典型的状态机**”就是“**见鬼**”了。

笔者索性就建立自己的“**风格**”，在“**低级建模**”里笔者使用“**步骤**”来取代“**状态机**”。果真这个决定是对的。这样的设定，给笔者在后期的实验带来许多方便。在网上有一位网友很可爱的为笔者的笔记“**Verilog HDL 建模技巧 - 低级建模之仿顺序操作·思路篇**”评价：

“俺觉得，每一个有仿顺序操作结构的模块，都可以理解成为一个子状态机 ... ”

他的理解没有一丝错误，不过是在理解上有不同的立场。这也难怪，因为当时笔者在写这一本笔记的时候，正是“**低级建模**”的创建初头“**步骤**”的使用还没有成熟。

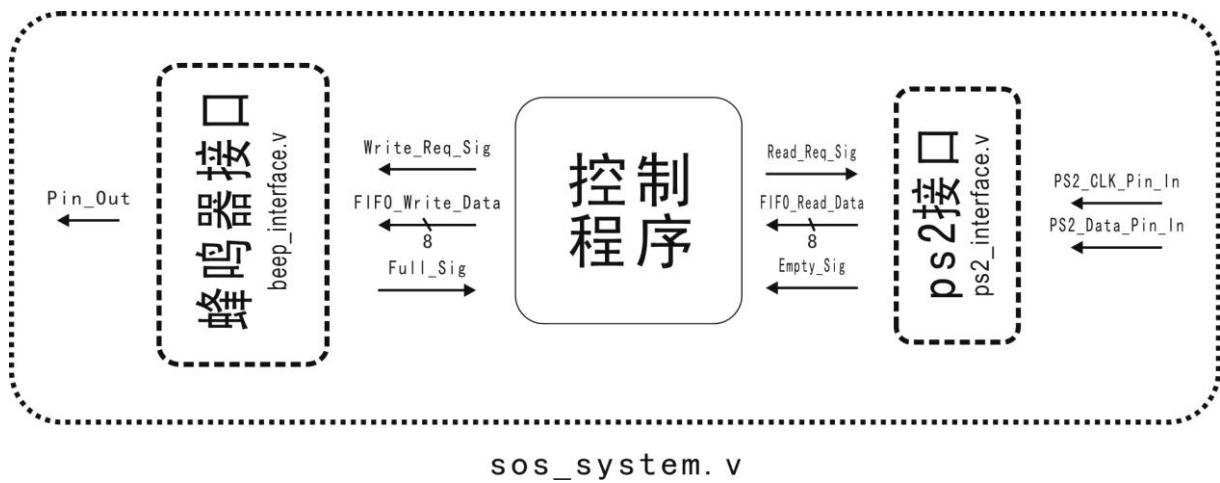
话说远了 ... 下一章就是大团圆的一章笔记了。在进入下一章之前，请确保读者本身已经理解，掌握“**低级建模是什么**”这一重点。如果还没有很好的理解“**低级建模时什么**”，请务必加强对“**低级建模**”的理解和掌握。

学习最重要不是要得到好成绩，也不是学习速度，而是掌握学习的重点 .....

# 第六章：低级建模 - 系统建模

在第五章中，我们为各个资源和模块进行封装。在第五章的结尾笔者留了这样一个问题“[模块的封装是为什么在做准备？](#)”然而这一章就是答案，模块的封装就是为“[系统建模](#)”作准备。在现实中，一个系统是由几个部门组成。然而在低级建模中，一个简单的系统可以由几个接口和控制模块建立而成。

## 6.1 实验二十二：SOS 系统



上图是 SOS 系统的“[图形](#)”。说得简单点，SOS 系统就是 PS2 接口和蜂鸣器接口再加上控制程序组合而成。该控制程序的工作就是从 PS2 接口的 FIFO 里，将数据移入蜂鸣器接口的 FIFO 里。系统功能也很单纯，当笔者在键盘上按下 S，它就会产生 S 摩斯码。如果笔者在键盘上按下 O，它就会产生 O 摩斯码。

*sos\_system.v*

```

1. module sos_system
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input PS2_CLK_Pin_In,
7.     input PS2_Data_Pin_In,
8.

```

```
9.      output Pin_Out
10. );
11.
12. ****
13.
14. reg [2:0]i;
15. reg isRead;
16. reg isWrite;
17.
18. always @ ( posedge CLK or negedge RSTn )
19.     if( !RSTn )
20.         begin
21.             i <= 3'd0;
22.             isRead <= 1'b0;
23.             isWrite <= 1'b0;
24.         end
25.     else
26.         case( i )
27.
28.             0:
29.                 if( !Empty_Sig ) i <= i + 1'b1;
30.
31.             1:
32.                 begin isRead <= 1'b1; i <= i + 1'b1; end
33.
34.             2:
35.                 begin isRead <= 1'b0; i <= i + 1'b1; end
36.
37.             3:
38.                 if( !Full_Sig ) i <= i + 1'b1;
39.
40.             4:
41.                 begin isWrite <= 1'b1; i <= i + 1'b1; end
42.
43.             5:
44.                 begin isWrite <= 1'b0; i <= 3'd0; end
45.
46.
47.         endcase
48.
49. ****
50.
51. wire Empty_Sig;
52. wire [7:0]FIFO_Read_Data;
```

```
53.
54.     ps2_interface U1
55.     (
56.         .CLK( CLK ),
57.         .RSTn( RSTn ),
58.         .PS2_CLK_Pin_In( PS2_CLK_Pin_In ),      // input - from top
59.         .PS2_Data_Pin_In( PS2_Data_Pin_In ),    // input - from top
60.         .Read_Req_Sig( isRead ),                // input - from code
61.         .Empty_Sig( Empty_Sig ),                // output - to code
62.         .FIFO_Read_Data( FIFO_Read_Data )      // output - to U2
63.     );
64.
65.     *****/
66.
67.     wire Full_Sig;
68.
69.     beep_interface U2
70.     (
71.         .CLK( CLK ),
72.         .RSTn( RSTn ),
73.         .Write_Req_Sig( isWrite ),              // input - from code
74.         .FIFO_Write_Data( FIFO_Read_Data ),    // input - from U1
75.         .Full_Sig( Full_Sig ),                // output - to code
76.         .Pin_Out( Pin_Out )                  // output - to top
77.     );
78.
79.     *****/
80.
81. endmodule
```

实验二十二说明:

好笨蛋的系统 .... 汗! 这怎么看都和实验十八的演示（串口接口演示）很相似 ...

实验二十二将结论:

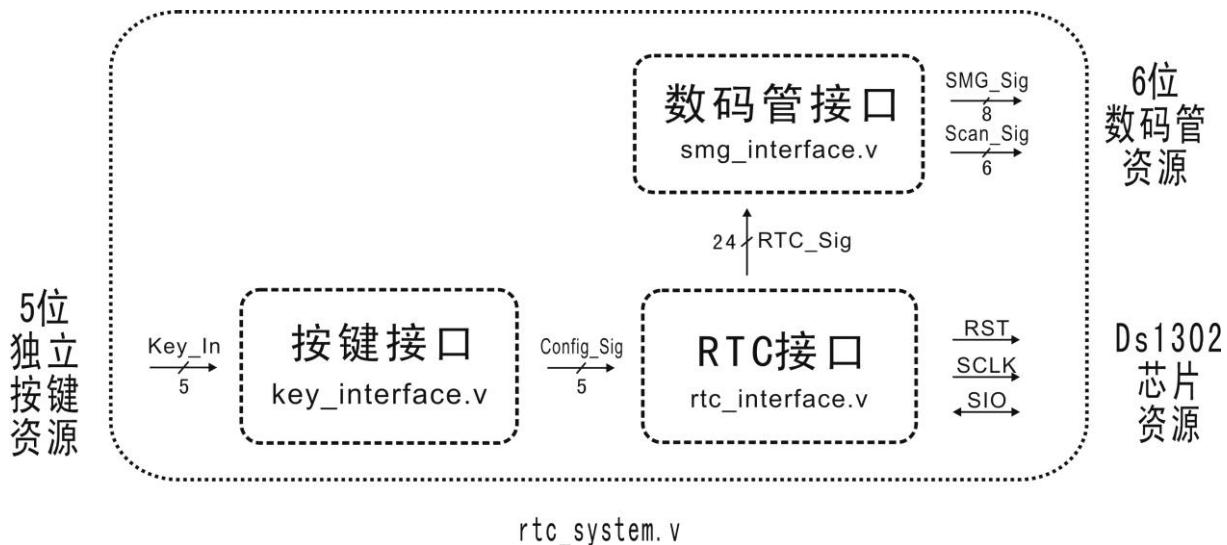
这个实验是给读者有一个“**系统**”简单的概念。实际上很多初学者都被众多的“**参考书**”，  
的“**系统**”的概念搞得有许多误会。笔者以前也是如此，在笔者的概念中“**系统**”是“**非常庞大，非常复杂**”，其实是笔者想太多了。在 Verilog HDL 的世界里“**系统**”可以小得简单，大得复杂，最重要是设计的定义。

(⊙o⊙)! 下载程序到黑金开发板后, 就努力敲键盘的 S 键和 O 键, 尽量求救吧。

## 6.2 实验二十三：RTC 系统

在实验二十三，我们将两个接口，蜂鸣器接口和 PS 接口组成 SOS 系统。这一章我们将使用按键接口，RTC 接口和数码管接口组成 RTC 系统。

在这里笔者不得不提及，在第五章的末段（5.8 章），笔者对于 RTC 接口的演示有所保留。由于 RTC 接口的演示涉及到“[系统建模](#)”，笔者才有如此的举动。所以说，这一章实验读者可以把它看成是 RTC 接口的演示实验 ...



上图 `rtc_system.v` 组合模块显示了，该 RTC 系统建成需要各种接口。“[按键接口](#)”是 5 位独立按键资源的输入接口，然而“[按键接口](#)”的输出信号作为“[RTC 接口](#)”Config\_Sig 信号的驱动。“[RTC 接口](#)”的输出信号 RTC\_Sig 作为“[数码管接口](#)”的驱动。此外，“[RTC 接口](#)”也驱动着 DS1302 芯片。在某种程度看来“[RTC 接口](#)”扮演着中枢的角色。

`rtc_system.v`

```

1. module rtc_system
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [4:0]Key_In,      // Key interface
7.
8.     output [7:0]SMG_Data, // SMG interface
9.     output [5:0]Scan_Sig,

```

```
10.
11.     output RST,           // RTC interface
12.     output SCLK,
13.     inout SIO
14. );
15.
16. ****
17.
18. wire [4:0]Key_Out;
19.
20. key_interface U1
21. (
22.     .CLK( CLK ),
23.     .RSTn( RSTn ),
24.     .Key_In( Key_In ),    // input - from top
25.     .Key_Out( Key_Out )  // output - to U2
26. );
27.
28. ****
29.
30. wire [23:0]RTC_Sig;
31.
32. rtc_interface U2
33. (
34.     .CLK( CLK ),
35.     .RSTn( RSTn ),
36.     .Config_Sig( Key_Out ), // input - from U1
37.     .RTC_Sig( RTC_Sig ),   // output - to U3
38.     .RST( RST ),          // output - to top
39.     .SCLK( SCLK ),         // output - to top
40.     .SIO( SIO )            // inout - with top
41. );
42.
43. ****
44.
45. smg_interface U3
46. (
47.     .CLK( CLK ),
48.     .RSTn( RSTn ),
49.     .Number_Sig( RTC_Sig ), // input - from U2
50.     .SMG_Data( SMG_Data ), // output - to top
51.     .Scan_Sig( Scan_Sig )  // output - to top
52. );
53.
```

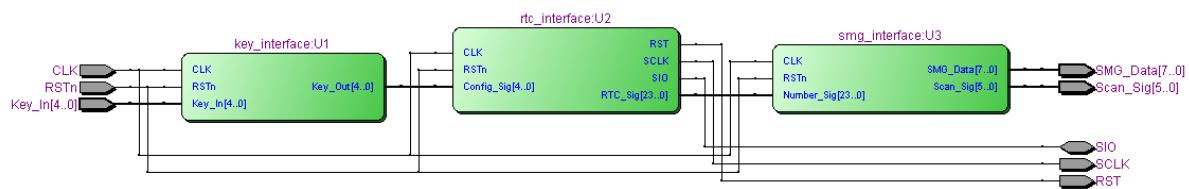
```
54.      ****  
55.  
56. endmodule
```

哦 ... 这个组合模块基本上和“[图形](#)”一样，自己看着办吧。

实验二十三说明:

如果读者不明白这个系统如何运作，就赶紧复习 按键接口，数码管接口 和 RTC 接口吧。

完成后的扩展图:

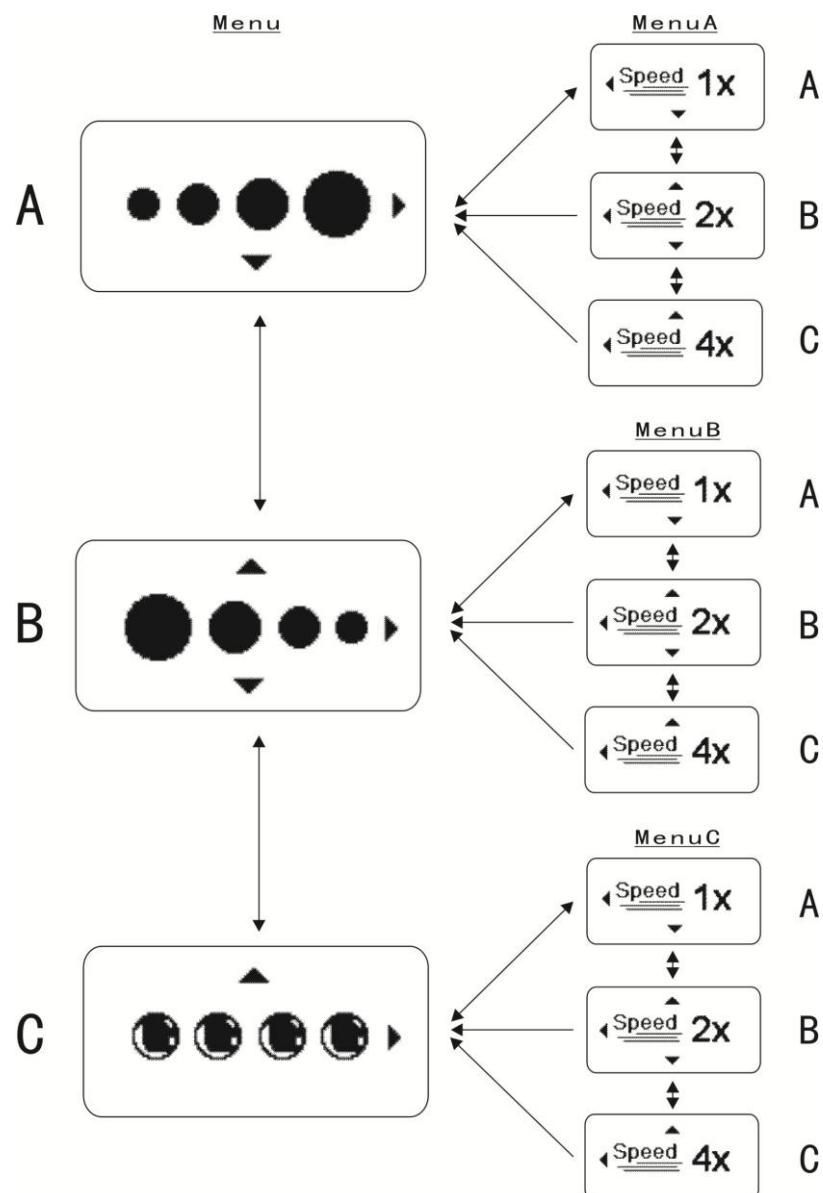


实验二十三结论:

读者是否以为“[系统建模](#)”只是将几个接口东拼西凑而已？下一章就是大团圆了，笔者就来点刺激，给读者不一样的“[系统建模](#)”的概念。

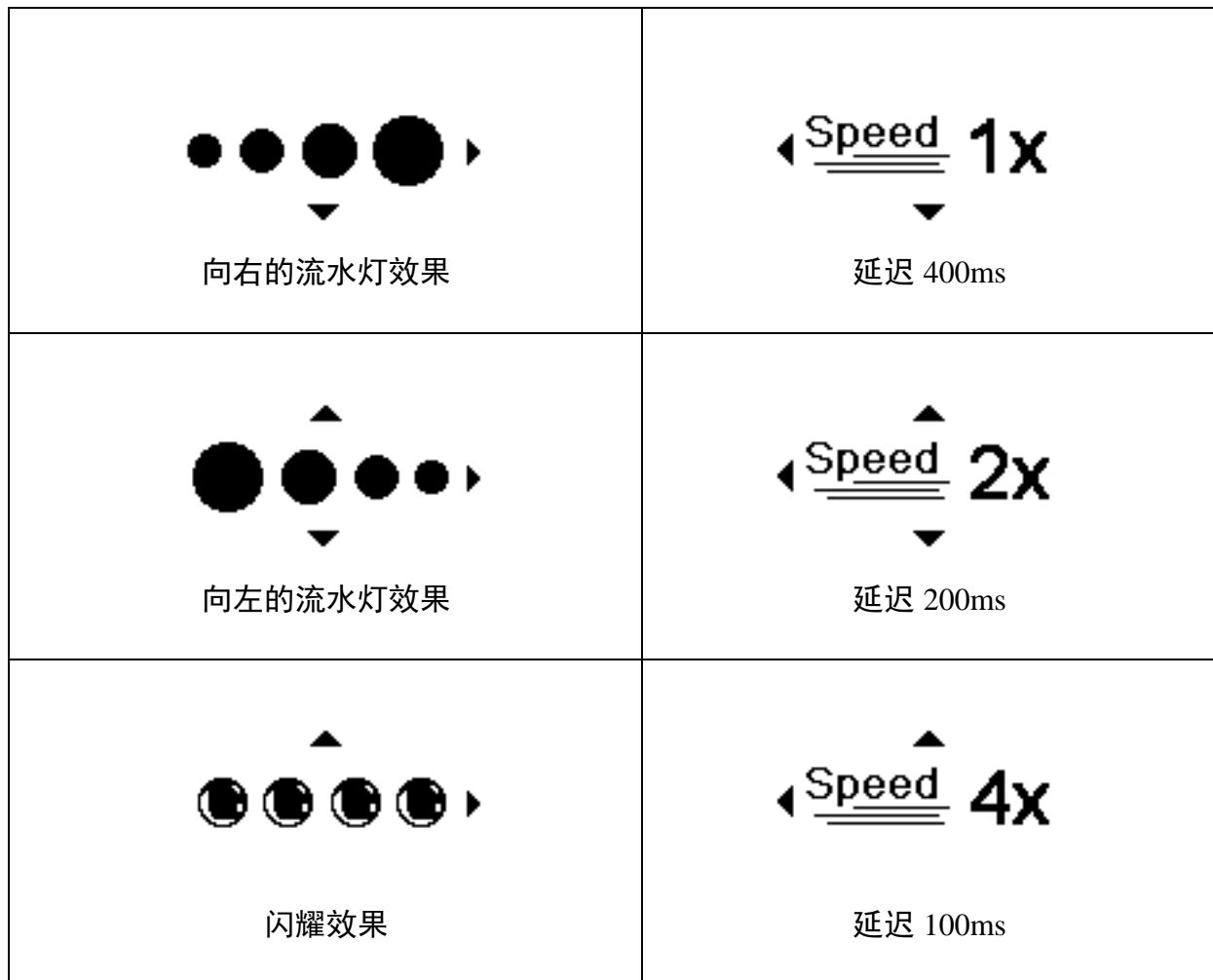
### 6.3 实验二十四：GUI 系统

终于写到笔记的尾声了，在 6.1 章和 6.2 章，笔者所建立的系统都是由几个接口东拼西凑组合而成，那些只是“[系统建模](#)”一个简单概念而已。在这一章笔者用另一种概念，一种更接近“[系统建模](#)”的实例。同时间我们也探讨“[接口](#)”对“[系统建模](#)”的重要性。这一章我们要建立的是简易的“[GUI 系统](#)”（GUI - 顾名思义就是“[图形接口](#)”想知道更详细的就查维基百科吧）。



(GUI 系统的目录)

上图是“[GUI 系统](#)”的层次关系。Menu 代表主目录，MenuA，MenuB 和 MenuC 代表子目录。然而每一个目录的图像都有代表的意义：



当然，每副图像的“箭头”也不是花瓶，图像中“箭头”表示了“[目录与子目录之间切换的关系](#)”和“[同目录中不同选项切换的关系](#)”。引一个例子来讲，“向右流水灯效果”的图像可以向右切入“延迟 400ms”，然而“延迟 400ms”可以向下切入“延迟 200ms”。

“[GUI 系统](#)”主要的功能如下：

在主目录 Menu 有 3 个选项，亦即“向右的流水灯效果”，“向左的流水灯效果”和“闪耀效果”。然而每一个 Menu 的选项，还包含各自的子目录，每一个子目录都有 3 个选项，亦即“延迟 400ms”，“延迟 200ms”，“延迟 100ms”（延迟的意义上就是效果的延迟时间）。

很简单吧？但是好戏在后头。

“GUI 系统”有一个经典的难题就是“[目录指针](#)”。当我们从目录或者选项之中发生更换“[目录指针](#)”都要一一追踪。想到“[指针](#)”读者一定会联想到 C 语言的“[变量指针](#)”，“[函数指针](#)”和“[结构体指针](#)”等。

在前面笔者就强调过，Verilog HDL 语言是硬件描述语言，而不是高级语言，它没有“[代码的结构和特性](#)”。但是 Verilog HDL 语言有一个很强大的东西，就是“[位操作](#)”，我们只要稍微的下功夫一番，就会完成“[目录 Flag](#)”。

我们先假设 Menu 有“[三个选项](#)”，我们可以这样作：

```
reg [2:0]Menu; // 建立一个寄存器表示该目录 Flag
```

```
Menu[2] = 选项 A 的 Flag // 向右的流水灯效果的选项  
Menu[1] = 选项 B 的 Flag // 向左的流水灯效果的选项  
Menu[0] = 选项 C 的 Flag // 闪耀效果的选项
```

假设，默认的选项是“向右的流水灯效果”，那么 Menu 寄存器经初始化过后的赋值是 3'b100。再假设，笔者从当前的“向右的流水灯效果”向下切换至“向左的流水灯效果” Menu 寄存器的值表示 2'b010；

同样的道理，我们可以为每一个 Menu 选项的子目录创建一个子“[目录 Flag](#)”：

```
reg [2:0]MenuA; // MenuA 的目录 Flag  
reg [2:0]MenuB; // MenuB 的目录 Flag  
reg [2:0]MenuC; // MenuC 的目录 Flag
```

```
MenuA[2] = 选项 A 的 Flag // 向右流水灯效果的“400ms 延迟”选项  
MenuA[1] = 选项 B 的 Flag // 向右流水灯效果的“200ms 延迟”选项  
MenuA[0] = 选项 C 的 Flag // 向右流水灯效果的“100ms 延迟”选项  
MenuB[2] = 选项 A 的 Flag // 向左流水灯效果的“400ms 延迟”选项  
MenuB[1] = 选项 B 的 Flag // 向左流水灯效果的“200ms 延迟”选项  
MenuB[0] = 选项 C 的 Flag // 向左流水灯效果的“100ms 延迟”选项  
MenuC[2] = 选项 A 的 Flag // 闪耀效果的“400ms 延迟”选项  
MenuC[1] = 选项 B 的 Flag // 闪耀效果的“200ms 延迟”选项  
MenuC[0] = 选项 C 的 Flag // 闪耀效果的“100ms 延迟”选项
```

最后我们建立一个 Menu\_Sig 信号将所有“[目录 Flag](#)”整合起来，成为“[目录路径](#)”：

```
output [11:0]Menu_Sig;
```

```
assign Menu_Sig = { Menu, MenuA, MenuB, MenuC };
```

Menu_Sig[11..0]	
{ Menu, MenuA, MenuB, MenuC }	选项
12'b100_000_000_000	向右的流水灯效果的选项
12'b010_000_000_000	向左的流水灯效果的选项
12'b001_000_000_000	闪耀效果的选项
12'b100_100_000_000	向右的流水灯效果“延迟 400ms”的选项
12'b100_010_000_000	向右的流水灯效果“延迟 200ms”的选项
12'b100_001_000_000	向右的流水灯效果“延迟 100ms”的选项
12'b010_000_100_000	向左的流水灯效果“延迟 400ms”的选项
12'b010_000_010_000	向左的流水灯效果“延迟 200ms”的选项
12'b010_000_001_000	向左的流水灯效果“延迟 100ms”的选项
12'b001_000_000_100	闪耀效果“延迟 400ms”的选项
12'b001_000_000_010	闪耀效果“延迟 200ms”的选项
12'b001_000_000_001	闪耀效果“延迟 100ms”的选项

为了更好的表达每一个选项和每一个目录的路径, 故笔者就建立图表。假设笔者进入“向右的流水灯效果”的“延迟 400ms”的选项那么 Menu\_Sig 信号的表达会是如此:

```
12'b100_100_000_000
```

从中我们看到 Menu (Menu\_Sig[11:9]) 的 A 项被设置, 我们知道“[目录路径](#)”从 Menu 的 A 项开始开始。然后我们又知道 MenuA ( Menu\_Sig[8:6] ) 的 A 项被设置, 那么我们可以这样结论: “[目录路径](#)是从 Menu 的 A 项开始, 然后到 MenuA 的 A 项结束”。亦即, 从“向右的流水灯效果的选项”切入“向右的流水灯效果“延迟 400ms”的选项”。

---

讨论完了 GUI 系统的目录结构, 接下来我们要讨论的问题就是“[配置](#)”。“[GUI 系统](#)”主要是由“[上下左右](#)”四个信号来配置。

Config_Sig[4..0]	
分配	功能
Config_Sig[4]	Enter (保留)
Config_Sig[3]	上
Config_Sig[2]	下
Config_Sig[1]	左
Config_Sig[0]	右

虽说 Config\_Sig 有五位, 但是 GUI 系统的目录切换真正被使用到的仅是 Config\_Sig[3..0] Config\_Sig[4] 被保留作为其他用途。

和 5.8 章一样 Config\_Sig 中的每一位都对 “[高脉冲敏感](#)”。

在这里笔者假设一个例子：“GUI 系统” 经初始化过后 “向右流水灯效果” 是默认选项。这时候笔者只有两个切换的选择：

(一) Config\_Sig[2] 接收一个高脉冲，从 “向右流水灯效果” 选项，向下切换至 “向左流水灯效果” 选项。

(二) Config\_Sig[0] 接收一个高脉冲，就会切入 “向右流水灯效果” 的子目录选项。

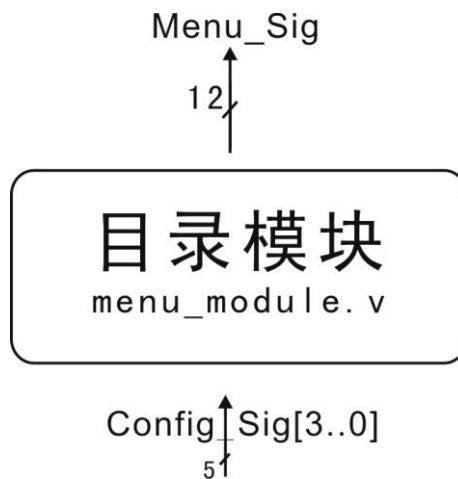
笔者再假设一个情况，如果 “向右流水灯效果” 的 “400ms 延迟” 作为开始选项，那么：

(一) Config\_Sig[2] 接收一个高脉冲，从 “向右流水灯效果” 的 “400ms 延迟” 选项，向下切换至 “向右流水灯效果” 的 “200ms 延迟” 选项。

(二) Config\_Sig[1] 接收一个高脉冲，从 “向右流水灯效果” 的 “400ms 延迟” 选项（子目录）退回 “向右流水灯效果” 选项（目录）。

至于目录从哪里来又切换至那里去，读者就浏览 “GUI 系统的目录” 吧。

*menu\_module.v*



关于 *menu\_module.v* 到底要它属于 “[控制模块](#)” 还是 “[功能模块](#)”，笔者也曾经纠结过。但是笔者还是给它定位 “[功能模块](#)”，实际上这个模块的功能也很单纯，就是根据 Config\_Sig 信号的配置如何，就产生怎样 Menu\_Sig。

*menu\_module.v* 主要的功能就是跟踪 “[目录路径](#)” 而已。也就是说 “GUI 系统”的 “[目录路径](#)” 会因为 Config\_Sig 信号而产生变化，然而这个模块只是跟踪，然后更改 Menu\_Sig。具体的功能还是直接看代码比较强。

```
1. module menu_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [3:0]Config_Sig,    // [4]Enter [3]Move_Up [2]Move_Down [1]Left [0]Right
7.     output [11:0]Menu_Sig   // [11:9]Menu [8:6]MenuA [5:3]MenuB [2:0]MenuC
8. );
9.
10. ****
11.
12. reg [3:0]i;
13. reg [2:0]Menu;
14. reg [2:0]MenuA;
15. reg [2:0]MenuB;
16. reg [2:0]MenuC;
17.
18. always @ ( posedge CLK or negedge RSTn )
19.     if( !RSTn )
20.         begin
21.             i <= 4'd0;
22.             Menu <= 3'b100;
23.             MenuA <= 3'b000;
24.             MenuB <= 3'b000;
25.             MenuC <= 3'b000;
26.         end
27.     else
28.         case( i )
29.
30.         **** // Main Menu
31.
32.         **** // MenuA
33.
34.         0:
35.         if( Config_Sig[2] ) begin Menu <= 3'b010; i <= 4'd1; end
36.         else if( Config_Sig[0] ) begin MenuA <= 3'b100; i <= 4'd3; end
37.
38.         **** // MenuB
39.
40.         1:
41.         if( Config_Sig[3] ) begin Menu <= 3'b100; i <= 4'd0; end
42.         else if( Config_Sig[2] ) begin Menu <= 3'b001; i <= 4'd2; end
```

```
43.         else if( Config_Sig[0] ) begin MenuB <= 3'b100; i <= 4'd6; end
44.
45.
46.         /***** // MenuC
47.
48.         2:
49.             if( Config_Sig[3] ) begin Menu <= 3'b010; i <= 4'd1; end
50.             else if( Config_Sig[0] ) begin MenuC <= 3'b100; i <= 4'd9; end
51.
52.             /***** // Child MenuA
53.
54.             /***** // MenuAA
55.
56.         3:
57.             if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
58.             else if( Config_Sig[2] ) begin MenuA <= 3'b010; i <= 4'd4; end
59.
60.             /***** // MenuAB
61.
62.         4:
63.             if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
64.             else if( Config_Sig[3] ) begin MenuA <= 3'b100; i <= 4'd3; end
65.             else if( Config_Sig[2] ) begin MenuA <= 3'b001; i <= 4'd5; end
66.
67.
68.             /***** // MenuAC
69.
70.         5:
71.             if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
72.             else if( Config_Sig[3] ) begin MenuA <= 3'b010; i <= 4'd4; end
73.
74.             /***** // Child MenuB
75.             /***** // MenuBA
76.
77.         6:
78.             if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end
79.             else if( Config_Sig[2] ) begin MenuB <= 3'b010; i <= i + 1'b1; end
80.
81.             /***** // MenuBB
82.
83.         7:
84.             if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end
85.             else if( Config_Sig[3] ) begin MenuB <= 3'b100; i <= i - 1'b1; end
86.             else if( Config_Sig[2] ) begin MenuB <= 3'b001; i <= i + 1'b1; end
```

```
87.  
88.      **** // MenuBC  
89.  
90.      8:  
91.      if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end  
92.      else if( Config_Sig[3] ) begin MenuB <= 3'b010; i <= i - 1'b1; end  
93.  
94.      **** // Child MenuC  
95.      **** // MenuCA  
96.  
97.      9:  
98.      if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end  
99.      else if( Config_Sig[2] ) begin MenuC <= 3'b010; i <= i + 1'b1; end  
100.  
101.     **** // MenuCB  
102.  
103.      10:  
104.      if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end  
105.      else if( Config_Sig[3] ) begin MenuC <= 3'b100; i <= i - 1'b1; end  
106.      else if( Config_Sig[2] ) begin MenuC <= 3'b001; i <= i + 1'b1; end  
107.  
108.     **** // MenuCC  
109.  
110.      11:  
111.      if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end  
112.      else if( Config_Sig[3] ) begin MenuC <= 3'b010; i <= i - 1'b1; end  
113.  
114.     ****  
115.  
116.      endcase  
117.  
118.     ****  
119.  
120.      assign Menu_Sig = { Menu, MenuA, MenuB, MenuC };  
121.  
122.     ****  
123.  
124.endmodule
```

在 12~16 行是核心功能中所使用的寄存器。Menu 是主目录 Flag 的寄存器，MenuA 是 Menu 的 A 项的子目录 Flag 寄存器，其他的 MenuB 和 MenuC 都是大同小异。在这里有一点必须注意，在“[GUI 系统](#)”初始化的时候 Menu 的 A 项作为默认选项，所以在 22 行 Menu 寄存器的值初始化为 3'b100。

30~50 行就是主目录 Menu，根据 Config\_Sig 信号产生的结果。初头会进入步骤 0，亦即主目录 Menu 的 A 项，在 35 行是向下切换的动作（Menu 寄存器赋值为 3'b010，进入步骤 1），36 行是切入子目录的动作(Menu 寄存器清零，MenuA 寄存器赋值 3'b100，进入步骤 3），亦即进入子目录后，子目录的 A 项作为默认。步骤 1 和 2 分别是 Menu 的 B 项和 C 项。主目录 Menu 项与项之间的切换都根据“GUI 系统”的“目录结构”。

步骤 1（40 行），如果 41 行成立的话，就会切换回 Menu 的 A 项（Menu 寄存器赋值为 3'b100，返回步骤 0）。如果 42 行成立，就会切换到 Menu 的 C 项（Menu 寄存器赋值为 3'b001，进入步骤 2）。如果 43 行成立，就会切入 Menu 的 B 项的子目录（MenuB 寄存器赋值为 3'b100，进入步骤 6）亦即进入子目录后，子目录的 A 项作为默认。

步骤 2（48 行），如果 49 行成立的话，就会切换回 Menu 的 B 项（Menu 寄存器赋值为 3'b010）。如果 50 行成立的话，就会切入 Menu 的 C 项的子目录（MenuC 寄存器赋值为 3'b100，进入步骤 9）亦即进入子目录后，子目录的 A 项作为默认。

```

52.          ****// Child MenuA
53.
54.          ****// MenuAA
55.
56.          3:
57.          if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
58.          else if( Config_Sig[2] ) begin MenuA <= 3'b010; i <= 4'd4; end
59.
60.          ****// MenuAB
61.
62.          4:
63.          if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
64.          else if( Config_Sig[3] ) begin MenuA <= 3'b100; i <= 4'd3; end
65.          else if( Config_Sig[2] ) begin MenuA <= 3'b001; i <= 4'd5; end
66.
67.
68.          ****// MenuAC
69.
70.          5:
71.          if( Config_Sig[1] ) begin MenuA <= 3'b000; i <= 4'd0; end
72.          else if( Config_Sig[3] ) begin MenuA <= 3'b010; i <= 4'd4; end
73.
```

在 36 行如果 if 条件成立的话，Menu 寄存器会保存父目录的 Flag，然后 MenuA 寄存器会设置该 A 项的 Flag。换句话说从 Menu 的 A 项切入的话，就会进入 Menu 的 A 项的子目录 MenuA 的 A 项（MenuA 的 A 项作为进入该目录后的默认选项），亦即进入步骤 3。步骤 3~5（56~72 行）是目录 MenuA 的选项。

步骤 3（56 行）就是 MenuA 的 A 项。如果 58 行成立就会切换至 MenuA 的 B 项（MenuA

寄存器会赋值与 3'b010，会进入步骤 4)，如果 57 行成立就会切出至父目录 Menu，然而根据 Menu 的跟踪，会返回 Menu 的 A 项（MenuA 寄存器清零，会返回步骤 0）。

步骤 4（62 行）是 MenuA 的 B 项，如果 63 行成立会切出至父目录（MenuA 寄存器清零，返回步骤 0），亦即 Menu 的 A 项。如果 64 行成立，就会切回 MenuA 的 A 项（MenuA 寄存器赋值为 3'b100，返回步骤 3）。如果 65 行成立，会切换 MenuA 的 C 项（MenuA 寄存器赋值为 3'b001，进入步骤 5）。

步骤 5（70 行）是 MenuA 的 C 项。如果 71 行成立的话就会切出至父目录（MenuA 寄存器清零，返回步骤 0）如果 72 行成立的话就会切回 MenuA 的 B 项（MenuA 寄存器赋值为 3'b010，返回步骤 4）。

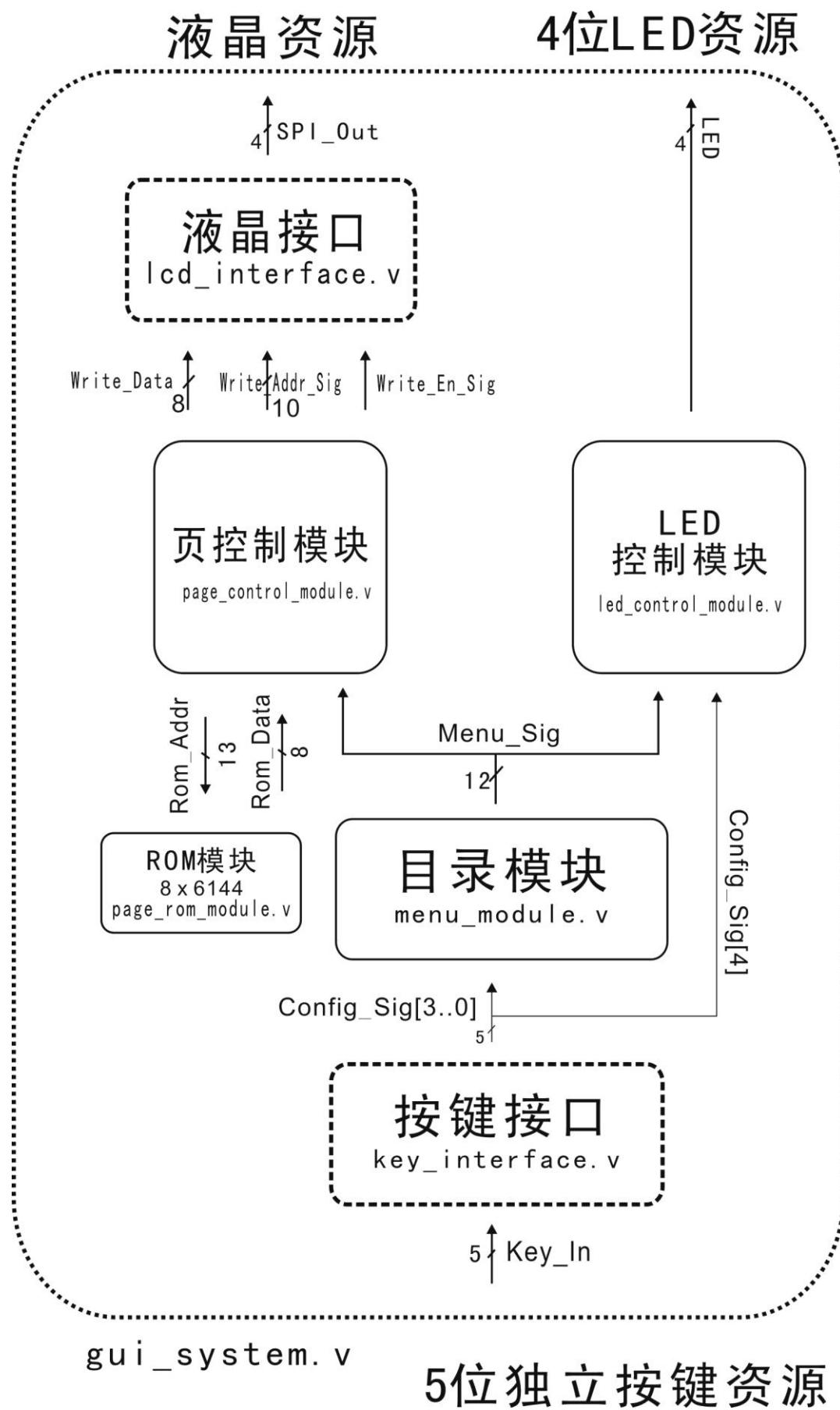
```
74.          ****// Child MenuB
75.          ****// MenuBA
76.
77.          6:
78.          if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end
79.          else if( Config_Sig[2] ) begin MenuB <= 3'b010; i <= i + 1'b1; end
80.
81.          ****// MenuBB
82.
83.          7:
84.          if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end
85.          else if( Config_Sig[3] ) begin MenuB <= 3'b100; i <= i - 1'b1; end
86.          else if( Config_Sig[2] ) begin MenuB <= 3'b001; i <= i + 1'b1; end
87.
88.          ****// MenuBC
89.
90.          8:
91.          if( Config_Sig[1] ) begin MenuB <= 3'b000; i <= 4'd1; end
92.          else if( Config_Sig[3] ) begin MenuB <= 3'b010; i <= i - 1'b1; end
93.
94.          ****// Child MenuC
95.          ****// MenuCA
96.
97.          9:
98.          if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end
99.          else if( Config_Sig[2] ) begin MenuC <= 3'b010; i <= i + 1'b1; end
100.
101.         ****// MenuCB
102.
103.        10:
104.        if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end
105.        else if( Config_Sig[3] ) begin MenuC <= 3'b100; i <= i - 1'b1; end
```

```
106.           else if( Config_Sig[2] ) begin MenuC <= 3'b001; i <= i + 1'b1;   end
107.
108.           /***** // MenuCC
109.
110.           11:
111.           if( Config_Sig[1] ) begin MenuC <= 3'b000; i <= 4'd2; end
112.           else if( Config_Sig[3] ) begin MenuC <= 3'b010; i <= i - 1'b1; end
113.
114.           *****/
115.
116.       endcase
117.
118.   *****/
119.
120. assign Menu_Sig = { Menu, MenuA, MenuB, MenuC };
121.
122. *****/
123.
124.endmodule
```

Menu 的 B 项的子目录 MenuB (77~92 行) 和 Menu 的 C 项的子目录 MenuC (97~112 行), 与 Menu 的 A 项的子目录 MenuA (56~72 行) 的操作都是大同小异, 笔者就不多罗嗦了 (再这样写下去, 笔者会患上焦急症候群 ... (○o○))。

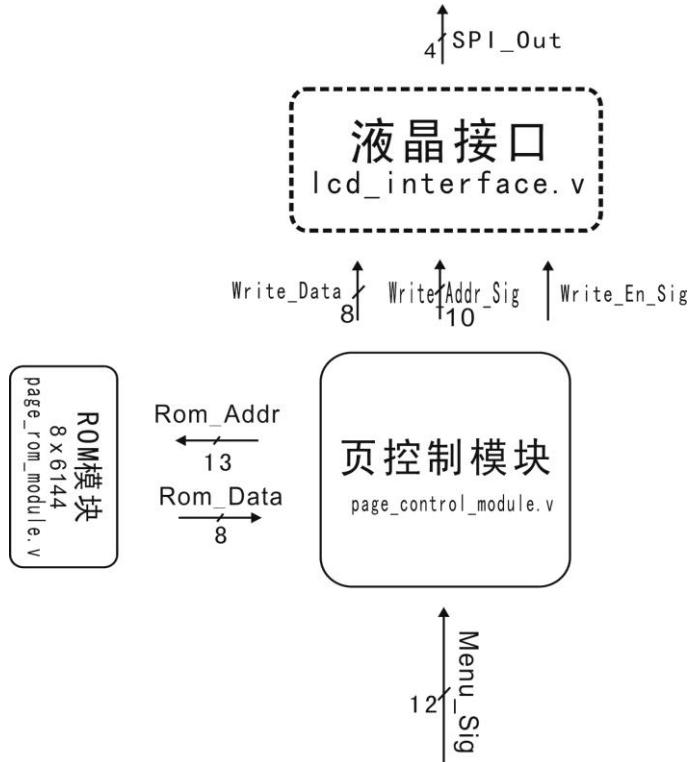
120 行是 Menu\_Sig 的输出, 该信号是由 Menu 寄存器, MenuA 寄存器, MenuB 寄存器和 MenuC 寄存, 按顺序联合驱动。

在这里我们可以证实一点, menu\_module.v 的工作是依据 Config\_Sig 信号对目录结构的影响来跟踪 “[目录路径](#)”。然而这个 “目录路径” 的可视信号便是 Menu\_Sig 信号。



上图是“**GUI 系统**”的全图形（不要被吓到），笔者会慢慢解释的。

“**目录模块**”就如前面说所那样，它是跟踪“**GUI 系统**”的“**目录路径**”，该模块只需要 Config\_Sig[3..0]，然而随着 Config\_Sig[3..0]的更动，信号 Menu\_Sig 也会随着更改。然后 Menu\_Sig 信号分别驱动“**页控制模块**”和“**LED 控制模块**”，我们先看左半部分：



上面的“**图形**”和实验二十演示（LCD 接口演示实验）非常相似吧。ROM 模块所拥有的空间是 8 Bits x 6144 Words，亦即这个 ROM 模块储存了 6 x 8 Bits x 1024 Words，也就说它包含了 6 副 8 Bits x 1024 Words 的图像信息。

地址 0~1023 是“向右流水灯效果”的图像信息。

地址 1024~2047 是“向左流水灯效果”的图像信息。

地址 2048~3071 是“闪耀效果”的图像信息。

地址 3072~4095 是“400ms 延迟”的图像信息。

地址 4096~5119 是“200ms 延迟”的图像信息。

地址 5120~6143 是“100ms 延迟”的图像信息。

“**页控制模块**”的主要功能就是根据 Menu\_Sig 信号，从 ROM 模块读取不同的图像信息写入液晶接口。

假设 Menu\_Sig 是 12'b100\_000\_000\_000。那么，地址 0~1023 “**向右流水灯效果**”的图像信息会被写入 LCD 接口。

*page\_control\_module.v*

```
1. module page_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [11:0]Menu_Sig,
7.
8.     input [7:0]Rom_Data,
9.     output [12:0]Rom_Addr,
10.
11.    output Write_En_Sig,
12.    output [9:0]Write_Addr_Sig,
13.    output [7:0]Write_Data
14. );
15.
16. //***** // Check out Menu_Sig
17.
18. reg [11:0]F1;
19. reg [11:0]F2;
20.
21. always @ ( posedge CLK or negedge RSTn )
22.     if( !RSTn )
23.         begin
24.             F1 <= 12'd0;
25.             F2 <= 12'd0;
26.         end
27.     else
28.         begin
29.             F1 <= Menu_Sig;
30.             F2 <= F1;
31.         end
32.
33. //***** // Menu check out and changing picture
34.
35. reg [2:0]Z;
36.
37. always @ ( posedge CLK or negedge RSTn )
38.     if( !RSTn )
39.         Z <= 3'd0;
40.     else if( F1 != F2 )
```

```
41.
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
58.
59.     /***** // Main function
60.
61.     reg i;
62.     reg [12:0]rAddr;
63.     reg [7:0]X;
64.     reg [3:0]Y;
65.     reg isWrite;
66.
67.     always @ ( posedge CLK or negedge RSTn )
68.         if( !RSTn )
69.             begin
70.                 i <= 1'd1;
71.                 rAddr <= 13'd0;
72.                 X <= 8'd0;
73.                 Y <= 4'd0;
74.                 isWrite <= 1'b0;
75.             end
76.         else
77.             case( i )
78.
79.                 0: // if Menu_Sig changing
80.                 if( F1 != F2 ) i <= i + 1'b1;
81.
82.                 1: // Draw Function
83.                 if( Y == 8 ) begin isWrite <= 1'b0; Y <= 4'd0; i <= 1'd0; end
84.                 else if( X == 128 ) begin X <= 8'd0; Y <= Y + 1'b1; end
```

```
85.           else begin rAddr <= ( X + (Y << 7) + (Z << 10) ); isWrite <= 1'b1; X <= X + 1'b1;end
86.
87.       endcase
88.
89.   /***** */
90.
91.   assign Rom_Addr = rAddr;
92.   assign Write_En_Sig = isWrite;
93.   assign Write_Addr_Sig = rAddr[9:0];
94.   assign Write_Data = Rom_Data;
95.
96.   /***** */
97.
98. endmodule
```

3~13 行 page\_control\_module.v 的输入输出接口。

16~32 行这一行代码和 detect\_module.v 很相识，但是我们不是要检测电平的变化，而是要检测“Menu\_Sig”的变化。当 Menu\_Sig 产生变化的时候 上一个时间的 Menu\_Sig 和下一个时间的 Menu\_Sig 的值是不一样，然而 F1 寄存器暂存下一个时间的 Menu\_Sig，F2 寄存器则暂存 上一个时间的 Menu\_Sig。

当我们检测 Menu\_Sig 是否发生变化的时候，可以这样表达：

```
if( F1 != F2 ) // Menu_Sig 发生变化
    .... // 执行语句
else      // Menu_Sig 没有发生变化
    .... // 执行语句
```

在“[液晶接口实验演示](#)”中，我们知道 Z 寄存器是用来“[表达图像的切换](#)”。在 42~57 行是根据不同“Menu\_Sig”的结果，切换不同的图像。也就是说“不同的 Menu\_Sig 值，都有不同 Z 值”（Z 值是图像信息的偏移量值）。

Z 值	图像信息	Z 值	图像信息
0	“向右流水灯效果” 图像信息	3	“延迟 400ms” 图像信息
1	“向左流水灯效果” 图像信息	4	“延迟 200ms” 图像信息
2	“闪耀效果” 图像信息	5	“延迟 100ms” 图像信息

在 40 行表示了“[当 Menu\\_Sig 产生变化，就根据 Menu\\_Sig 的值，更新 Z 寄存器的值](#)”。

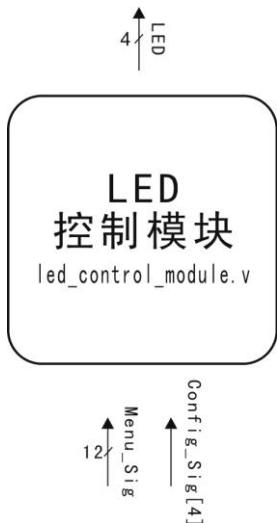
61~87 行就是该控制模块的功能。ROM 模块的空间是 0 ~ 6143，所以驱动用的 rAddr 寄存器的位宽是 13 位（62 行）。X 寄存是用来计数列填充（63 行），Y 寄存器是行计数（64 行）。

初始化的时候步骤 i 被初始化为 1（70 行），目的是为液晶资源写入“[默认选项的图像信息](#)”。初始化的时候由于 Z 值是 0，所以“[向右流水灯效果](#)”的图像信息作为启动系统的默认选项。

79 行的步骤 0，是用来检测“[Menu\\_Sig 是否发生变化](#)”？如果 Menu\_Sig 发生变化步骤 i 就递增以示下一个步骤（80 行）。

82 行步骤 1 是绘图操作，该 85 行中的  $rAddr \leq (X + Y \ll 7 + Z \ll 10)$  表达式，是图像信息寻址的表达式，笔者就不重复了，如果笔者有不明白的地方请复习 5.7 章的实验二十演示。91~94 行是该控制模块的输出驱动。

*led\_control\_module.v*



左图是 LED 控制模块的图形，该控制模块会根据不同的 Menu\_Sig 产生不同的 LED 效果。然而该控制模块不像 page\_control\_module.v 那样，在 Menu\_Sig 产生变化的瞬间，输出也会产生变化。每当 Menu\_Sig 产生变化，如果 Config\_Sig[4] 没有接收一个高脉冲，LED 的输出效果是不会更新的。Config\_Sig[4] 在位分配的意义上正是“Enter”的作用。说简单点，如果“[Enter](#)”没有被执行，LED 的效果也不会更新。

```

1. module led_control_module
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input Config_Sig,
7.
8.     input [11:0]Menu_Sig,
9.
10.    output [3:0]LED

```

## Verilog HDL 那些事儿 – 建模篇

```
11. );
12.
13. *****/
14.
15. parameter T1MS = 16'd49_999;
16.
17. *****/ // 1ms timer
18.
19. reg [15:0]Count1;
20.
21. always @ ( posedge CLK or negedge RSTn )
22.     if( !RSTn )
23.         Count1 <= 16'd0;
24.     else if( Count1 == T1MS )
25.         Count1 <= 16'd0;
26.     else if( isCount )
27.         Count1 <= Count1 + 1'b1;
28.     else if( !isCount )
29.         Count1 <= 16'd0;
30.
31. *****/ // 1ms counter
32.
33. reg [8:0]Count_MS;
34.
35. always @ ( posedge CLK or negedge RSTn )
36.     if( !RSTn )
37.         Count_MS <= 9'd0;
38.     else if( Count_MS == rTimes )
39.         Count_MS <= 9'd0;
40.     else if( Count1 == T1MS )
41.         Count_MS <= Count_MS + 1'b1;
42.
43. *****/ // Menu checkout and change variable
44.
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;

```

## Verilog HDL 那些事儿 – 建模篇

```
99.          end
100.         else if( Config_Sig ) // if press down Enter key...
101.           begin
102.             LED_Mode <= Mode;
103.             rTimes <= Delay;
104.           end
105.
106.         /***** // LED mode
107.
108.         reg [3:0]rLED;
109.         reg isCount;
110.
111.       always @ ( posedge CLK or negedge RSTn )
112.         if ( !RSTn )
113.           begin
114.             rLED <= 4'b1000;
115.             isCount <= 1'b0;
116.           end
117.         else
118.           case( LED_Mode )
119.
120.             3'b100: // Move to right
121.               if( Count_MS == rTimes ) begin rLED <= { rLED[0] , rLED[3:1] }; isCount <= 1'b0; end
122.               else isCount <= 1'b1;
123.
124.             3'b010: // Move to Left
125.               if( Count_MS == rTimes ) begin rLED <= { rLED[2:0] , rLED[3] }; isCount <= 1'b0; end
126.               else isCount <= 1'b1;
127.
128.             3'b001: // Reverse
129.               if( Count_MS == rTimes ) begin rLED <= ~rLED; isCount <= 1'b0; end
130.               else isCount <= 1'b1;
131.
132.           endcase
133.
134.         /*****
135.
136.       assign LED = rLED;
137.
138.     /*****
139.
140.   endmodule
```

第 15 行是 1ms 的常量定义。在 19~29 行是 1ms 的定时器。33~41 行是 1ms 的计数器。

45~58 行是用来暂存上一个时间的 Menu\_Sig 和下一个时间的 Menu\_Sig，和 page\_control\_module 的 16~31 行是同样的道理。

第 62 行的 Mode 寄存器是用来暂存 LED 的效果值。3'b100 表示向右流水灯效果，3'b010 表示向左流水灯效果，3'b001 表示闪耀效果。

63 行的 Delay 寄存器是用来暂存延迟的值。

在 70 行，如果 if 条件成立 (Menu\_Sig 产生变化)，在 72~87 行 Mode 的寄存器和 Delay 寄存器的值，会根据 Menu\_Sig 不同的值都会产生变化。

举个例子 12'b001\_000\_000\_001 表示了“[闪耀效果](#)”的“[延迟 100ms](#)”的选项。那么 Mode 的值是 3'b001 和 Delay 的值是 100ms。

Mode 寄存器和 Delay 寄存器只是“[用来暂存某值](#)”而已。真正被用到的寄存器是 LED\_Mode 和 rTimes。在初始化的情况下 LED\_Mode 的初值是 3'b100，亦即“[向右流水灯效果](#)”，rTimes 的初值是 400。如果 Enter 键被按下 (Config\_Sig[4]接收一个高脉冲) LED\_Mode 赋值于 Mode 值，rTimes 赋值于 Delay 值 (100~104 行)。

108~132 行是该控制模块的主要功能。在 118 行，会根据 LED\_Mode 的值产生不一样的效果。会根据不同的 rTimes 值产生不一样的延迟。

*gui\_system.v*

```

1. module gui_system
2. (
3.     input CLK,
4.     input RSTn,
5.
6.     input [4:0]Key_In,
7.
8.     output [3:0]SPI_Out,
9.
10.    output [3:0]LED
11. );
12.
13.    ****
14.
15.    wire [4:0]Key_Out;
16.
17.    key_interface U1
18.    (

```

```
19.      .CLK( CLK ),
20.      .RSTn( RSTn ),
21.      .Key_In( Key_In ),    // input - from top
22.      .Key_Out( Key_Out ) // output - to U2
23. );
24.
25. ****
26.
27. wire [11:0]Menu_Sig;
28.
29. menu_module U2
30. (
31.      .CLK( CLK ),
32.      .RSTn( RSTn ),
33.      .Config_Sig( Key_Out[3:0] ),      // input - from U1
34.      .Menu_Sig( Menu_Sig )          // output - to U4 and U6
35. );
36.
37. ****
38.
39. wire [7:0]Rom_Data;
40.
41. page_rom_module U3
42. (
43.      .clock( CLK ),
44.      .address( Rom_Addr ),    // input - from U4
45.      .q( Rom_Data )          // output - to U4
46. );
47.
48. ****
49.
50. wire [12:0]Rom_Addr;
51. wire Write_En_Sig;
52. wire [9:0]Write_Addr_Sig;
53. wire [7:0]Write_Data;
54.
55. page_control_module U4
56. (
57.      .CLK( CLK ),
58.      .RSTn( RSTn ),
59.      .Menu_Sig( Menu_Sig ),      // input - from U2
60.      .Rom_Data( Rom_Data ),    // input - from U3
61.      .Rom_Addr( Rom_Addr ),    // output - to U3
62.      .Write_En_Sig( Write_En_Sig ), // output - to U5
```

```

63.      .Write_Addr_Sig( Write_Addr_Sig ),      // output - to U5
64.      .Write_Data( Write_Data )                  // output - to U5
65. );
66.
67. ****
68.
69. lcd_interface U5
70. (
71.     .CLK( CLK ),
72.     .RSTn( RSTn ),
73.     .Write_En_Sig( Write_En_Sig ),           // input - from U4
74.     .Write_Addr_Sig( Write_Addr_Sig ),       // input - from U4
75.     .Write_Data( Write_Data ),               // input - from U4
76.     .SPI_Out( SPI_Out )                   // output - to top
77. );
78.
79. ****
80.
81. led_control_module U6
82. (
83.     .CLK( CLK ),
84.     .RSTn( RSTn ),
85.     .Config_Sig( Key_Out[4] ),             // input - from U1
86.     .Menu_Sig( Menu_Sig ),                // input - from U2
87.     .LED( LED )                         // output - to top
88. );
89.
90. endmodule

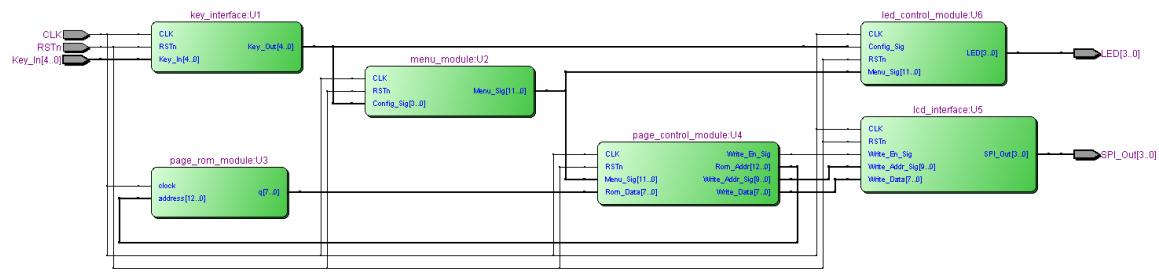
```

该组合模块和“[图形](#)”是大同小异，自己看着办吧。

[实验二十四说明:](#)

该“[GUI 系统](#)”在“显示”方面的设计比较简单，就是“[一个事件一副图像](#)”，“[一个事件一个效果](#)”

完成后的扩展图：



### 实验二十四结论：

看吧！这一章的实验再也不是由几个接口东拼西凑成为一个系统，而是“**接口模块在全部系统中只占一个重要部分而已**”。虽然实验二十四充其量是一个简易的“**GUI 系统**”而已，但是这个实验已经将系统的概念表达得很清楚了。

## 总结：

笔记终于写完了，从第一章开始到第五章，所有的实验，所有的内容都是在为第六章作准备。

“[系统建模](#)”比起“[基础建模](#)”或者“[封装（接口建模）](#)”不是同一个等级的东西。因为“[系统建模](#)”的建模量不是一般的多，而是非常多。如果没有建模技巧的支持要完成“[系统建模](#)”是一件苦差事。

所以呀，“[系统建模](#)”作为“[低级建模](#)”结束的一页，是再适合不过了。就如笔者在前面说所的，“[前期的建模是为后期的建模作准备](#)”。显然“[系统建模](#)”不可能是后期建模的最后一站，在“[系统建模](#)”的后面还有更后期的建模。但是，那是什么？笔者也不知道 ...

笔者只知道一个事实“[当读者有本事走到这里，完成所有试验，明白什么是系统建模，读者就已经了解什么是低级建模](#)”。在笔者的眼里“[系统建模](#)”是“[低级建模](#)”的综合练习，因为要完成“[系统建模](#)”必须掌握好“[低级建模](#)”的所有基础。

最后笔者补充一些重点：

笔者的建模技巧-低级建模，不仅是关心“如何建模”而已？笔者非常执着与“代码风格”或者“代码结构”。从实验的开始到实验的结束，笔者都是使用同样的“代码风格”。要如何维护好自己的代码风格，并不是这本笔记讨论的范围，但是笔者很建议读者可以参考笔者的“[代码风格](#)”。

读者呀，到这里为止是不是已经触碰了笔者心目中的“低级建模”？“一个大东西是需要许多的小东西不停的组成和不停的组合。在组合的过程要相互尊重（明白模块之间的性质），相互协调（不同性质的模块之间的调用），相互支持（一层接一层的组合）...”

## 结束语

终于把这本笔记编辑完毕了，编辑笔记的过程真是辛酸但是又是真实。编写这本笔记的初头笔者也是重新从零开始的。说实话，笔者在编辑这本笔记之前水平很低，但是当笔者掌握了“建模技巧”之后，跳跃式的进步。读者们相不相信，就见仁见智。

话说 3 个月的时间说长不长说短不短，悄悄好是四份之一年，但是这一段时间对于笔者来说是绝对真实而且值得的。当这本笔记完成之际，笔者仿佛又多了解了 Verilog HDL + FPGA 的世界。Verilog HDL + FPGA 的世界是深不可测，如果以笔者的话来说，笔者也仅是了解到冰山一角而已。但是这一步的踏出，笔者发现了新大陆。

好了，笔者不再罗嗦了。笔者真心希望读者们可以借与这本笔记重新去认识 Verilog HDL + FPGA 的世界。Verilog HDL + FPGA 的世界一点也不可怕，而且多姿多彩，只是我们在学习的路上，忽然间迷失而已，只要重新思考，重新出发，就会发现这个世界的不同。

在这里，可能读者们会产生这样的问题：“下一站的学习旅程，我应该从哪里开始？”笔者不能断定什么，但是笔者可以建议以下的几个选择：

- 一、了解功能仿真和验证（你会了解系统级的硬件描述语言）。
- 二、了解时序分析（你会了解寄存器级的世界）。

三、了解 NIOS II ( 你会了解软核 )。

四、继续走 Verilog HDL 的道路。

笔者的选择是肯定的-第四点，因为笔者从这本笔记从新了解到 Verilog HDL 语言的很强大，所以笔者想要更了解它。事实上“建模篇”只是用 Verilog HDL 语言塑造一个模糊的东西( 形状 )而已 ,它还是为成品 ,接下来的路 ,笔者还要继续细化这个模糊的东西 .....  
( Verilog HDL 那些事儿 - 时序篇 ) 呱~~(~\_~)b

如果读者对笔者的学习笔记很感兴趣的话 , 笔者很欢迎读者到笔者的博客下载。如果读者想对笔者说什么 , 就去社区留言吧。

博客 : <http://blog.ednchina.com/akuei2>

社区 : <http://www.heijin.com>