Verilog HDL的礼物

Verilog HDL扫盲文

akuei2学习笔记

黑金动力社区
www.heijin.org

目录

目录 02

第 0 章 VERILOG HDL 语言扫盲文 03
0.01 各种的 HDL 语言 03
0.02 HDL 语言的层次 03
0.03 RTL 级和组合逻辑级 04
0.04 VERILOG HDL 语言真的那么难掌握？ 05
0.05 高级语言和 VERILOG HDL 语言的区别 06
0.06 什么是 VERILOG HDL 语言的时序？ 07
0.07 VERILOG HDL 的综合语言 08
0.08 关于参考书和笔者的笔记 13
0.09 不要带偏见去学习 VERILOG HDL 语言 14
0.10 单文件主义 15
0.11 VERILOG HDL 语言结构简介 17
0.12 VERILOG HDL 语言使用规则（方法）简介 19
0.13 认识 RTL 级设计（建模） 23
0.14 过渡中，沉住气！朋友！ 25

总结 26

第 0 章：Verilog HDL 扫盲文

会翻开这本笔记的读者，估计你们都受够了参考书的“权威”，即使把厚厚的参考书都啃完了，发觉自己对 Verilog HDL 语言的理解还是“迷迷糊糊”。呵呵，笔者也是过来人，笔者当然清楚这样的心情，那种感觉真的是想“涅破了蛋蛋”。当读者还没有进入正章之前，笔者有责任帮读者们来个简单的扫盲扫盲。扫盲的目的有许多，第一是更进一步刷新读者对 Verilog HDL 语言的认识。第二则是可以清楚表达笔记所要讨论的范围。

0.1 各种的 HDL 语言

很多进入 FPGA 世界不久得朋友，第一个要学习当然是 HDL 语言，在网上流行的有 Verilog 和 VDL 这两个 HDL 语言。如果读者是 VDL HDL 语言的爱好者，那么读者可以立即把这本笔记关了。在笔者的眼中 VDL 太石板了，好像带着三角眼睛的中年女老师，对男学生都要求和尚头；对女生的裙子一定要长裙，这就是 VDL 给笔者的印象。笔者不是说它不好，只是笔者嫌它麻烦而已。反之 Verilog 却像是一个活泼而且爱捣蛋的小男孩，我们知道小男孩的思想很简单却很俏皮，我们要很难捉拿它。

网上有一个很常见的问题：“学习 VDL HDL 好？还是学习 Verilog HDL 好？”... 唉~朋友，很多问题的答案都是明显的。笔者会很好客的说：“来来来！Verilog HDL 很有趣也很好玩，不要理那个石板的 VDL HDL”。

“为什么笔记选择 Verilog HDL 语言？”

嗯 ... 这个问题笔者也很难回答，笔者是被特权同学带入这个世界的。当时学习的时候没有考虑那么多，但是后来发现到 Verilog HDL 语言有太多的潜能了，笔者不小心就陷入研究它的陷阱了。Verilog HDL 语言的语法和格式都比较随便，它没有 VDL HDL 语言那么严谨，可能是这个原因吧？事实上选择 VDL HDL 语言 也好，还是选择 Verilog HDL 语言也好，都是一些萝卜青菜的问题（各有所爱）。笔者自身不喜欢受限制太多，故 Verilog HDL 语言和笔记意气相投，所以笔者最终还是选择了它。

0.2 HDL 语言的层次

有一个很好笑的话题，老师常常都说 HDL 语言的层次是汇编语言和 C 语言的之间。假设汇编语言是低级语言，C 语言是高级语言，那么 HDL 语言既是不上又不小？啊哈哈哈，如果站在人类之中它亦是不男也不女。我们不需要为这个无聊的话题，浪费太多思考的时间。HDL 语言的英文全名是 Hardware Description Language ，中文译名就是硬件描述语言。事实上无论是汇编语言也好还是 C 语言也好，它们的作用就是用来控制处理器，反之 HDL 语言的作用只是用来建立一个硬件的模块而已。

打个比方，假设有一个 c51 单片机的串口硬件，我们可以使用汇编语言去控制它，我们也可以使用 C 语言去控制它。但是站在 HDL 语言的角度上，我们可以建立一个受控制的串口硬件模块，我们也可以建立一个不受控制（是自动的意思，而不是暴走的意思）的串口硬件模块，当然我们也可以用 HDL 去控制一个可以受控制的串口硬件模块。

从这一点，我们就可以看出 HDL 语言和汇编语言与 C 语言基本上就在不同的层次上的东东，我们又何为把它们来作比较呢？但是在一些标准上 HDL 语言却是硬件语言又是低级语言（凡是涉及硬件的通通都被打入低级语言的冷宫），总而言之 HDL 语言的层次就是很暧昧就是了。

实际上还有不同层次级的 HDL 语言，如 SystemVerilog 或者 SystemC。传言上它们都是系统级的 HDL 语言，相比之下 Verilog HDL 语言 和 VDL HDL 语言的层次都称为模块级。但是这些层次的区分一点也不重要，只要把 Verilog HDL 语言掌握得好，读者什么层次都可以实现。

0.3 RTL 级和组合逻辑级

笔者说太多废话了还是切入正题吧。虽说 Verilog HDL 语言是用来描述硬件，但是这些都是一些刻板的认识和标准而已。从笔者的眼中 Verilog HDL 语言建立的硬件模块可以分为有时钟源和无时钟源。

有时钟源的意思是需要时钟信号作为操作最基本消耗单位，硬件模块才能执行。无时钟源的意思就是不需要消费时钟信号，硬件模块也可以被执行。无时钟源最好的例子就是由组合逻辑建立的硬件模块，一些典型的设计如：硬件乘法器，硬件除法器等基本上都是由一些复杂的逻辑门组合，故它们根本就不需要时钟来操作，或者说它们只需要一个步骤又或者只需要一个时钟就可以完成任务。

打个比方，就如同人类吃饭，在无时钟源的情况下，放在餐桌上的食物一下子被吃光，而且瞬间食物被消化和被吸收（事实上这是不可能的）。反之有时钟源，就像我们需要心跳来提供身体一个节拍，每一个节拍都有固定的动作，先张口，送入饭，吃下饭，消化饭，然后吸收。

由此就有 RTL 级和组合逻辑级的建模之分。基本上 RTL 级建模的基本单元会是“寄存器”，然而组合逻辑级的建模基本单元就是逻辑门。单单用几行的文字来讲述 RTL 级和组合逻辑级的建模实在太抽象了，笔者聚个简单的例子：

```
module Add_Module( input [7:0]A, input [7:0]B, input [7:0]C, Output[15:0]);

    wire [7:0]_a = A;
    wire [7:0]_b = B;
    wire [7:0]_c = C;
```

```
    assign Output = _a + _b + _c;

endmodule
```

这是一个简单的组合逻辑级，建模而成的 3 路加法器。

```
module Add_module
( input CLK, input RSTn, input [7:0]A, input [7:0]B, input [7:0]C, Output[15:0]) ;

    reg [15:0]rTemp;

    alawys @ ( posedge CLK negedge RSTn )
        if( !RSTn )
            rTemp <= 4'0;
        else
            rTemp <= A + B + C;

    assign Output = rTemp;

endmodule
```

这是一个简单的 RTL 级，建模而成的 3 路加法器。

根据上面两个例子的三路加法器，一个是由组合逻辑级建模而成，另外一个则是由 RTL 级建模而成。组合逻辑级建模给人最直接的印象就是模块都不带“时钟信号”，反之 RTL 级建模的最大特征性，就是模块都会伴随“时钟信号”。当然，笔者不可能仅以“时钟信号”来区分组合逻辑级建模和 RTL 级建模。

在实际的学习中过于在意区分“什么是组合逻辑级建模，什么是 RTL 级建模”是对学习没有任何帮助的（不知道为什么很多参考书都很习注重区分），凡是有关 Verilog HDL 语言的建模它们都被需要。话虽如此，但是在这一本笔记里笔者还是比较注重 RTL 级的建模。

0.4 Verilog HDL 语言真的那么难掌握？

Verilog HDL 语言容易入门但是不容易掌握，估计这是所有学习 Verilog HDL 语言人们的心声。其实要掌握 Verilog HDL 语言是很简单的，笔者的秘诀就是“掌握 Verilog HDL 语言的思想”。在笔者眼中 Verilog HDL 语言的思想有两种，一种是建模和另一种是时序。建模是 Verilog HDL 语言的结构或者说是它的地基，然而时序是所有模块的活动记录。这些概念，目前的读者不明白也不要紧，笔者已经分为两个笔记来详谈了它们了。

笔者相信很多接触 FPGA 的朋友之前都有接触过单片机，在不知不觉之中，自然而然学习单片机的想法就主宰了学习 FPGA。笔者也是过来人，这样的心情笔者非常的了解。当我们再学习单片机的时候，很多人估计都是直接以 C 语言入门吧？笔者就先说说单片

机这个硬件吧：它们都是各大产商的产品，一些基本的硬件资源老早就已经嵌入在这个小小的单片机当中。余下，尤其说学习单片机还不如说我们学习如何控制单片机的寄存器更为贴切。一些硬件的发生，使用者可以完全不用知道，我们只要懂得如何配置控制该硬件的寄存器就可以了。

相比之下 FPGA 可以称为赤裸裸的乐高积木，如果读者要实现串口，读者就要自行建立串口硬件模块。要建立一个串口硬件模块，首先需要明白串口的操作原理，然后根据需要自定义和修改原理的发生。最后还要考虑这个串口硬件模块如何被控制？是否独立化它（见接口建模）？这些就是建模思路。

在软件方面，我们可以用 C 语言去配置单片机的寄存器和编辑单片机操作的逻辑。但是有一点请读者不要忘了，C 语言也是一个产品，它和 Verilog HDL 语言不同 C 语言在出生之后它就有自己的结构和自己一套的使用规则。最后根据单片机的产商需要，C 语言还可以进一步被自定义，这种现象不难看见，学习 c51 和学习 AVR 基本上就有两套的 C 语言用法。

反之 Verilog HDL 语言虽然有语法，但是 Verilog HDL 语言就没有自身的结构也没有自己一套的用法。在网上这种现象很普遍，读者会看到五花八门，百花齐放，各种各样的模块内容。很多时候，这些模块的内容只有设计者自己看懂而已，别人估计要花上几倍的精力才可以搞明白“它在干嘛”。

为此，如果要把 Verilog HDL 语言掌握好，第一要提供 Verilog HDL 语言的结构，第二要提供 Verilog HDL 语言一套用法（使用规则）。前者笔者是用建模来解决，后者笔者用“仿顺序操作”的想法来补充。

0.5 高级语言和 Verilog HDL 语言的区别

一些高级语言如 C 语言，组成操作行为的基本单位是“步骤”，举个例子：

```
步骤 1 sum1 = a + b + c;
步骤 2 sum2 = d + e + f;
```

在步骤 1 sum1 赋值与 a + b + c，然而在步骤 2 sum2 赋值与 d + e + f; 在使用者眼中，C 语言只要两个步骤就可以把操作完成。反之在使用者不可见之下，步骤 1 和步骤 2 总和所生成的指令估计会超过 8 个，然而单片机一个时钟只能执行一个指令而已，亦即要完成步骤 1 和步骤 2 至少需要超过 8 个时钟才能完成。

对于使用者来说，他们只是关心步骤而已，反而不会过于关心“一共被执行了多少指令”和“消耗了多少个时钟”。相反的对于 Verilog HDL 语言来说，尤其是 RTL 级的建模，时钟代表了模块执行所要消耗的单位。

```
    case( i )
```

```
    0:
    begin Sum1 <= a + b + c; Sum2 <= d + e + f; i <= i + 1'b1; end
```

在一个时钟之内 Sum1 和 Sum2 同时赋值与 a+b+c 与 d+e+f。

```
case( i )

    0:
    begin Sum1 <= a + b + c; i <= i + 1'b1; end

    1:
    begin Sum2 <= d + e + f; i <= i + 1'b1; end
```

在第一个时钟之内 Sum1 赋值与 a+b+c；在第二个时钟之内 Sum2 赋值与 d+e+f;

在上面两段简单的代码之中，Sum1 和 Sum2 可以在一个时钟内求得，又或者 Sum1 和 Sum2 可以分开两个时钟之内完成赋值。换句话说，理论上 Verilog HDL 语言在一个时钟之内可以完成“很多很多很多”的操作又可以完成单一操作，因为它的操作是由逻辑资源组成的，只要 FPGA 的逻辑资源允许的话，读者要在一个时钟内完成 1 万个 1 亿个操作都没有问题。这个事实也暴露了 Verilog HDL 语言是有并行的性质。

比较属性	C 语言	Verilog HDL 语言
最小单位	指令	逻辑资源
单个时钟可以执行的操作	一个指令（简单指令）	理论上是无限个操作 （并行操作的性质）
结构	有固定的结构	没有固定结构
使用规则	有固定的使用规则	没有规定使用规则
强度	软	偏硬

在笔者的眼中，总结上 C 语言和 Verilog HDL 语言之间的区别会是如上的图表。关于高级语言和 Verilog HDL 语言区别的内容笔者讨论到这里就好了，读者不要过于深入区分谁是谁，谁又不是谁，如此纠结对学习没有任何好处，更多认识，当读者们深入以后就会自然了解。

0.6 什么是 Verilog HDL 语言的时序？

时序是 Verilog HDL 语言的中心思想之二，在死板的教科书上时序图等于逻辑波形图，在某种程度上这样的解释没有任何错误。但是，这种的解释太过武断了，类似“鬈发的小孩都是顽皮的小孩（笔者是鬈发小孩）”的解释。时序也可以称为是“模块的活动记录”，又或者说：把每一个时钟中的模块活动记录都链接起来的话，就会形成俗称的“波

形图”。

从另一个角度来看的话，在长长的时序图里，包含了模块的活动规则，模块的沟通记录等一些贵重的信息。老实说，有关 Verilog HDL 语言与时序的概念，新手确实很难掌握。因为传统的印象中“时序”给人的概念是发生在物理上的，既有延迟，又有亚太事件（亚稳态）.... 等。虽然这也是时序的“一番”概念，但是这是发生在物理上而已，然而发生在 Verilog HDL 语言身上的时序是“理想”的，是不存在任何瑕疵。我们知道驱使（RTL 级）模块行动的最小单位就是时钟，即时发生延迟也只是延迟一个时钟，又或者 2~3 个时钟，绝对不会出现什么延迟 2ns~3ns （物理路径延迟）等的问题。

在笔者的眼中“时序”的思想非常简单，就是如何读懂波形图，如何把波形图和模块的内容作联系。但是很遗憾，在这一本笔记里笔者只是详谈建模而不讨论任何和“时序”有关的话题。原因很简单，就是“建模是 HDL 语言的所有”如果读者都搞不懂建模到底是什么一回事，模块的内容估计都会“惨不忍睹”。就算读者搞明白什么是时序（在不懂建模的概念之下）结局是什么意思也没有，这一点笔者不会骗你的 ...

取而代之笔者用“步骤”作为笔记的中心 ... 嗯~还有一点，笔者所设计的实例都是直接下载到开发板，运行并且观察结果。如果读者在仿真上遇到什么问题，千万不要来找笔者噢，呵呵~（如果读者无法很好掌握“时序”的概念，仿真会非常不好使的。）

更多有关“时序”的话题，笔者已经准备在另一本笔记里了。所以，笔者真的真的很有诚意的求求你们，先把建模学好再谈什么仿真 ....

0.7 Verilog HDL 的综合语言

Verilog HDL 语言有两个部分，是综合语言和验证语言。有关验证语言笔者就不谈了 ... 说了都感觉恶心，它已经伤了笔者很多的心。至于综合语言也就是建模最常用的，在这里笔者比较习惯称为它们建模语言。

综合语言常用的关键字不多,笔者就随便举例：

属性	关键字
模块建立	module ... endmodule
输入输出声明	input , output , inout
判断	if ... elseif ... else case() ... endcase
资源声明	reg , wire
基本操作用	always@() , posedge , negedge, assign , begin ... end
常量声明	parameter

至于综合语言常用的操作符也不多，笔者也随便举例：

属性	操作符
赋值	<= , =
逻辑判断	> , < , <= , >=, != , ==
下标	[]
位操作	{}, <<, >>
数学运算	+, - ， *，/, %
其他	?:(三目), *(声明组合逻辑用)

当读者看了笔者“随便举例”的综合语言以后，是不是会傻眼？不要怀疑，只要懂这些东西就已经足够了。事实上真正使得读者们产生疑惑感的是读者们手上用的参考书。这些砖家叫兽有的没的，为了“权威”什么事情都干，笔者没有损毁他们的意思，事实上很多参考书都是参考来又参考去，内容都是换汤不换药。更恶心的是这些参考书还添加了许多看不懂的内容，买的人往往都是最伤神和伤身的（钱包也一样伤 ）。不过也有一个例外就是夏教授写的书确实很经典。

我们先谈谈几个比较有趣的关键字和操作符：

例子 1 - reg 和 wire 的尴尬：

reg 和 wire 如果站在 RTL 级建模的角度上，reg 就是寄存器，作用是用来暂存内容，而且也提供操作空间；wire 就是连线，作用仅此而已。但是站在组合逻辑级建模上 reg 和 wire 已经是傻傻分不清楚了，举个例子：

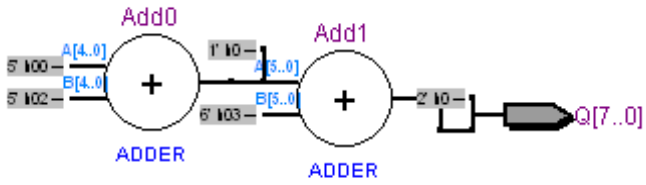
```
module omg_module ( output [7:0]Q );

    wire [3:0]A = 4'd0;
    wire [3:0]B = 4'd2;
    wire [3:0]C = 4'd3;

    assign Q = A + B + C;

endmodule
```

以上的一段代码，请问 wire 的作用是连线还是寄存内容了？呵呵，笔者没有说这样的使用方法有错呀。在这里笔者只是提出一个有趣的例子而已，对于一些初学者来说可能会非常的疑惑，尤其是那些习惯“面向对象”思想的人们 ...

```
module omg_module ( output [7:0]Q );

    reg [3:0]A;
    reg [3:0]B;
    reg [3:0]C;
```

```
    always @ ( * )
        begin
            A = 4'd0;
            B = 4'd2;
            C = 4'd3;
        end

    assign Q = A + B + C;

endmodule
```

如果笔者换成这样写法的话，是不是觉得更有道理呢？ always (*) 的使用暗示了这是一个组合逻辑，然而寄存器 A 的值是 4'd0, 寄存器 B 的值是 4'd2,寄存器 C 的值是 4'd3，Q 的驱动输出是寄存器 A,B,C 的值综合。



生成的 RTL 视图

更有趣得是，经过编译两段代码所生成的 RTL 视图都一样!?(⊙o⊙)？... 在这里，笔者是要告诉读者一个信息，当我们建模的时候“解读能力”的优先级往往都高过“内容精简性”。不要一度过于“贪小便宜”而把内容的解读能力跨下了，虽然这两个代码都没有错，而且结果一致。但是一个潜在的问题是，我们人类都是习惯把“东西分类”以便管理和理解，既然 wire 在字面上都是“连线”的意思了，就干脆把它当“连线”来使用 ...

例子 2 - always @ () 的多样性

1. always @ (posedge CLK or negedge RSTn) // 当 CLK 和 RSTn 变化的时候
2. always @ (*) // 什么时候都变化，亦即默认为组合逻辑
3. always @ (A) // 当 A 变化的时候

always @ () 的用法很多，但是用得最多的就是第 1 个和第 2 个。

关于第一个用法，表示了在 always @ ()之下的动作，对每一个 CLK 的上升沿或者 RSTn 的下降沿都被执行。笔者给出一些比较简单的例子：

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        Counter <= 8'd0;
    else
        Counter <= Counter + 1'b1;
```

这是一个简单的计数器实例，当 RSTn 产生下降沿 Counter 就清零，反之在每一个 CLK 的上升沿 Counter 就递增。always @() 这样的用法也给 RTL 级建模提供了基础，此外我们也可以经过自定义产生出另类的用法。

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        begin
            i <= 4'd0;
            .......
        end
    else
        case( i )

            0:
            .......

        endcase
```

上面一段代码是笔者最爱的“基于仿顺序操作想法”的基本“用法模板”，在后期里它可是大展拳脚。

```
always @ ( * ) A = 4'd9;   // 常数赋值

always @ ( * )   // 选择器
    if( Start_Sig[0] ) rQ = U1_Q;
    else if ( Start_Sig[1] ) rQ = U2_Q;
    else Q = 1'bx;
```

至于第二种 always @ () 的用法，笔者估计它是用得最“泛滥”了，尤其是在组合逻辑建模里，不过笔者使用的比较不多，除了“常数赋值”和“输出选择器”以外。

最后的第一种 always @()的用法，基本上只要懂得上面两种 always @ () 用法以后，自然而然也会了。

例子 3 - 最头疼的 = 和 <= 赋值

基本上要搞懂这两个赋值操作符号的作用，就必须把“时序”的概念搞懂先。一般上，

参考书只是告诉我们一个是非阻塞，一个是非阻塞 .... 说实话，当笔者把厚厚的参考书吃完以后，笔者完全还搞不懂究竟参考书都在说什么（砖家叫兽万岁 ）。如果从笔者的角度去理解的话，只有在时序的活动中才可以很清楚的看清它们的区别。宏观上，如同参考书中所说的一样；微观上，在时序中“=”是引发“即时事件”,“<=”则是引发“时间点事件”。

不过很可惜的是，在建模篇里笔者没有过多的讨论它们。一律有关 RTL 级活动的使用“<=”,一律有关组合逻辑级的活动都使用“=”。如果笔者在建模篇里过度涉及它们，笔记的目的就会本末倒置，这一点请读者们见谅。笔者已经在时序篇里准备好来讨论它们了。

*例子 4 - 要慎用的 */% 数学运算符*

当读者使用 * / 和 % 的数学运算符的时候，笔者请你们再三的三思（九思？呵呵！），因为使用它们的代价很大。如果读者所使用的 FPGA 有内嵌硬件乘法器又或者除法器的话，那么这些乘法器和除法器就会被消耗。相反的，如果读者说使用的 FPGA 是没有包含这些东西的话，资源逻辑的消耗是很大的。

一般上，如果是为了求出 *2 *4 *8 *16 又或者 /2 /4 /8 /16 笔者建议使用位操作的运算符，亦即 << 和 >> ，它们也可以求出同样的结果。如果想要求出的结果是不在 2^N 范围之内的话，读者还是求与其他的方法 ... 只有在用尽办法的时候才使用它们。至于 % 的数学运算符，它真的是一个罪恶，没有可以替代的第三方法 ....

在这里笔者要强调一下，笔者不是说不可以使用它们，笔者只是建议如果可以的话不要过度依赖它们，它们尽是一些逻辑资源的大食怪，不把资源吃光才怪。

0.8 关于参考书和读者的笔记

一般上参考书都是按“权威”的方式对书本的内容进行编辑，在这里“权威”不是指“最标准”而是指最大众化和出现最多次的风格。但是风格绝对不是 Verilog HDL 语言的中心，什么自顶向下设计？什么自底向上设计？什么一段式状态机？什么二段式状态机？什么三段式状态机？什么综合语言？什么验证语言？这一切的一切都不重要 .....

在笔者的眼中，参考书的风格过于死板了，而且也没有完全表达到 Verilog HDL 语言的中心位置。在这里，笔者不是有意去挑战一些已经被定下来 “几千年的传统”（使用标准），因为参考书的出发点都是把“FPGA”和“ HDL 语言”死死的绑在一起。换句话说，HDL 语言是 FPGA 的附属品，FPGA 才是中心。

事实上“HDL 语言是 HDL 语言”“FPGA 是 FPGA”各自都有自己的中心。笔者比较喜欢把 FPGA 看成是一个巨大的“乐高积木资源库”，HDL 语言是组合这些积木的工具。理论上工具是理想和完美的，工具只有用错方法的时候，如何“有效发挥这工具“这才是 Verilog HDL 语言的中心。

但是更多的时候，众多参考书所表达的是“怎样利用工具来完成目的”而不是“如何有效使用这个工具”。故参考书都以“权威”的形式，制定一些死沉沉标准，作为这个工具的使用标准。所以，很多新手都傻乎乎的认为这些“标准”是“好的”和“对的”，就这样不停的向墙角越转越深 ...

实际中悄悄好相反，HDL 语言的灵活度远远超过参考书的范围。笔者非常的建议，把参考书的内容读个明白就好，用不着过于执着那些“标准”。HDL 语言不像 C 语言那样，结构和使用方法那么严谨，HDL 语言非常的自由，自由到使用者也不知道如何泡制它。

所以说，一个模块有 10 种的编辑风格也不奇怪 ... 但是在 10 种之中，8~9 种的编辑风格都离不开那些“死沉沉的标准”。这些死沉沉的标准老早都不把 HDL 语言看成是理想和完美的工具了，这现象把工具的作用大打折扣。说实话，新手们要摆脱这些怨灵的纠缠真的很不容易 ... 当你打开某个参考书作为认识 HDL 语言入门书籍的时候，就像洗脑般“书中的内容不知不觉中主宰了你”。

但是，参考书也不是至于一文不值，有时候一些语法上的问题还得请教它们。在前一章节当中，笔者列出了一些常用的关键字和关键操作符，是不是比起读者从参考书中看到的还要少？读者是不是会觉得很困惑，到底要相信笔者好？还是要相信参考书好？笔者郑重的告诉读者，最好两方都不要相信，最重要的是明白其中的内容。最后发展出自己的一套风格。

笔者的风格非常要求“模块的解读能力”而且也非常接近 Verilog HDL 语言本身的特质。但是有一点不足的地方的是，笔者完全不遵守参考书说所的话。这也是很多读者在浏览笔者的笔记的时候会萌生出一种的感觉“怎么笔记的内容很特别”... 其实不是笔记很特别，而是读者老早已经习惯了参考书一套“死沉”的风格。这些之间的出入（区别），会使你们产生这样“很特别”的感觉。

在这里，笔者再强调一下 HDL 语言是一种非常自由的语言，它没有固定的结构，也没有固定的使用方法。参考书在乎“如何使用 HDL 语言去完成设计”，然而笔者比较在乎“如何有效使用 HDL 语言”。这两者之间都没有问题，不过是笔者的笔记比较偏向学习的口味，没有参考书那样死沉沉的气氛，比较愉快和轻松。

无论读者是浏览参考书也好，笔者的笔记也好，还是他人大大的笔记也好，不要过度把 HDL 语言看得太死，HDL 语言不适合这一套的。但是一些新手，尤其是上路一段时间的新手，笔者非常建议能静下心来好好阅读笔记中的每一章节。笔者不是按“项目的味道”去编辑笔记，而是按“学习的味道”去编辑笔记，内容都是在围绕“Verilog HDL 语言是什么？如何用好它？”之类的重点。

0.9 不要带偏见去学习 Verilog HDL 语言

某个声音：

“C 语言驱动的东西，既然用 Verilog HDL 语言去驱动!？省了吧 ...”
“Verilog HDL 语言能不能像 C 语言这样调用子程序？”
“Verilog HDL 很像 C 语言 ...”
“你怎么可以违背参考书 .... 怎么不使用状态机？”

上述的声音很容易在初学 Verilog HDL 语言的时候常常被听到。笔者还记得笔者在初学 Verilog HDL 语言的时候，印象最深刻的一句话就是“Verilog HDL 语言很像 C 语言，但是不要把 Verilog HDL 语言当成 C 语言”。这一句话一直在后期的学习中，给笔者许多的灵感。

在许多本笔记中，笔者常常说 C 语言和 Verilog HDL 语言是两个世界的居民，但是笔者又时不时借签 C 语言，这其中真的充满了很多矛盾 ... 很多新手都说 C 语言和 Verilog HDL 语言既然是不同的东西，自然而然 Verilog HDL 语言的作为和 C 语言的作为是 180° 相冲的。话句话说，Verilog HDL 语言可以应用的地方只适合“逻辑和底层设计”.... 不不不，这是天大的误会。

就这样，随之又产生“C 语言驱动的东西，既然用 Verilog HDL 语言去驱动”类似的声音。是谁规定 C 语言可以驱动的东西，Verilong HDL 语言就不能驱动？相反的，C 语言可以驱动的东西，如果读者也能使用 Verilog HDL 语言去驱动，那么这才是真正的学习。之所以会产生如此的声音，就如笔者在前几章节讲述的那样：

“Verilog HDL 语言的结构自由，使用方法也自由，自由到好像没有一样”

很多的设计都不包含结构和使用方法，只要设计可以发挥预期般的效果就 Okay ~ 如果读者明白了这个简单的道理，读者自然会明白自定义 Verilog HDL 语言的结构和使用方法是非常的重要和基础。很可惜呀，这一切的问题都要归咎于参考书，因为参考书从来

不考虑这一点，傻乎乎的读者就这样遭殃了。事实上 Verilog HDL 语言可以实现如同 C 语言那样调用子程序.. 但是问题就在于没有结构和没有使用方法，所以才会感觉困难而已。

笔者非常非常的建议，不要把参考书当着神来膜拜，即使是最常用的状态机，也有非常多的缺陷。这些缺陷在后期建模的时候，会如显的突出 ... 代码臃肿和解读能力下降等问题。不知道大伙有没有看过如同“蜘蛛网”的状态机的关系图？呵呵 ... 参考书所说的一切都不是绝对的，它们可供参考，但是不可供“迷信”。读者要使用状态机也好，还是不使用状态机也好，完全是自由。

Verilog HDL 语言的建模不是越复杂就越伟大，反之越直接的建模才是学习的方向。在这里，听笔者说：“当你放下偏见，你才可以接触到真理”，这简单的智慧在哪里都行得通，学习 Verilog HDL 语言也是这样一回事。

“放下一切对 Verilog HDL 语言学习的偏见吧，阿门”～ 呵呵！

0.10 单文件主义

单文件主义对于新手来说，某个程度上它是一个“伟大的主义" 但是又有很多人会受限这个“伟大的主义”。单文件主义就是，所有内容的设计都是在一个模块之内完成，这一点，有点像 C 语言中 main 那样，所有动作都在 main()函数中完成。单文件主义是新手都要经过的，当游走一段时间以后，慢慢的我们会发现这个主义的局限性。我们想要越过“它”，但是又不知道要如何往哪个方向 ... 这就是很多新手都会遇见的“瓶颈”。

在这里，笔者告诉读者们，能多快就有多快远离单文件主义。单文件主义基本上已是远离 Verilog HDL 语言的本质了，这是一个很危险的陷阱，如果读者不小心陷入，在接下来的学习路上是没有任何帮助的。这是一个事实，而不是笔记自身的想法。会养成单文件主义，多半的原因还是来源于参考书，这真的使笔者很无语 .....

笔者来说说为什么单文件主义远离了 Verilog HDL 语言的本质呢？Verilog HDL 语言，本质上是并行而且又有“面向对象”的味道。但是这“面向对象”的概念和 C++语言中的概念有所不同，然而它更接近现实中的“管理系统”（详解请看建模篇）。读者尝试想象，有没有可能一个系统的操作，没有部门，没有团队，没有小组？对，就是不可能。单文件主义恰恰好就是违反了这个简单的道理。

普通人类在理解上，都喜欢把东西分类，然后方便于管理。读者可以想象一个没有分类的系统，内容究竟到底有多乱吗？是一塌糊涂的乱！所以单文件主义的建模，最大的缺陷就是一个字“乱”，内容真的很乱。如果 Verilog HDL 语言是顺序语言的话，结果还不至于那样糟糕，相反的，Verilog HDL 语言却是并行性质的语言 ....

笔者举个简单的例子：

假设读者和女/男朋友约会，去法国餐厅吃晚餐。我们都知道法国料理有前菜，主菜，和点心之分。前菜简单可以是一杯开胃的 Lemon Tea ，又或者是清淡的蔬果料理。主菜可以是牛扒，鸡扒，海鲜还是野味。点心可以是一粒糖果，一杯雪糕，还是一碗“糖水”。在享受料理的途中，一位小生还可以为我们提供美丽的小提琴音乐（奢侈！真的很奢侈！好孩子不要这样做！为人太自私了）

进食动作			享受音乐动作
分类	料理	步骤	音乐分类
前菜	Lemon Tea 蔬果料理	1	烂漫并且忧伤的小提琴乐曲
主菜	牛扒 鸡扒 海鲜 野味	2	
点心	糖果 雪糕 糖水	3	

法国餐厅的进行全过程（有结构的系统操作）

笔者建立了一个简单的图表。在享受料理的过程中，主要有两个动作同时进行，亦即进食和享受音乐是并行发生的。由于享受音乐动作是全程发生的，所以内部用不着再细小分类了。反之在进食动作中，必须按照法式进食的规则，进食的规则有固定的步骤，就是先吃前菜，中吃主菜，后吃点心。这是一个有结构，并且是并行操作的全过程。

如果这个烂漫的约会失去了结构，但是有顺序结构支持，会是怎样的一个情形?

全部动作	步骤
先喝 Lemon Tea	1
听一下音乐	2
吃一下牛扒	3
听一下音乐	4
听一下音乐	5
吃一下蔬果	6
听一下音乐	7
吃一点雪糕	8
听一下音乐	9
.....	...

法国餐厅的进行全过程（失去结构但是有步骤的系统操作）

这是失去结构但是有顺序操作支持的烂漫约会 ... 过程虽然有点糟糕，但是不至于乱。但是会糟旁人的白眼，完全是破坏烂漫的气氛，差不多要被赶出去了。请读者想象一下，失去结构，又失去顺序操作支持的烂漫约会的过程到底会怎么样。笔者真的想象不出也画不出图表，这对烂漫的情侣估计只有被赶出去的份，真的很 OMG。

上文中所说的“没有分类”的烂漫约会，如果也“没有顺序操作”的支持 ... 会是一个非常糟糕的情形。因为这个系统操作（约会过程）没有结构的支持 ，这个情形也反映出了单文件主义的致命缺点。笔者有一句很经典的话：“解读能力差的模块是最糟糕的”，这一句话完全迎合单文件主义下所建立的模块。

0.11 Verilog HDL 语言结构简介

在前面，笔者已经说过 Verilog HDL 语言从出生以来，它的结构是非常自由，可以说自由到没有结构。所以学习 Verilog HDL 语言第一步就是要学会建立结构，建立结构用另一句话说就是学习建模。同样，笔者也说过 Verilog HDL 语言我们需要自定义自己的结构，在这里就拿笔者最爱的“低级建模”来做一些简单的扫盲。

低级建模被网友称为“自底向上”的设计方法，其实“自底向上”还是“自顶向下”都无所谓啦，最重要就是建模的思路。

低级建模的基本单元有：功能模块，控制模块，组合模块。

- 功能模块的内容包含了最基本的动作。
- 控制模块的内容包含了动作的控制步骤。
- 组合模块的内容包含了所有功能模块和控制模块之间的组合。

建模层次有：基础（模块）建模，仿顺序操作建模，接口建模，系统建模。

- 基础（模块）建模的内容包含了最小功能的模块。
- 仿顺序操作建模，这一个比较特别，主要是模仿了 C 语言中的函数。
- 接口建模的内容包含了一个已经封装完成的模块。
- 系统建模的内容包含了一个特定功能的模块。

举个例子，就拿串口来作个比方：

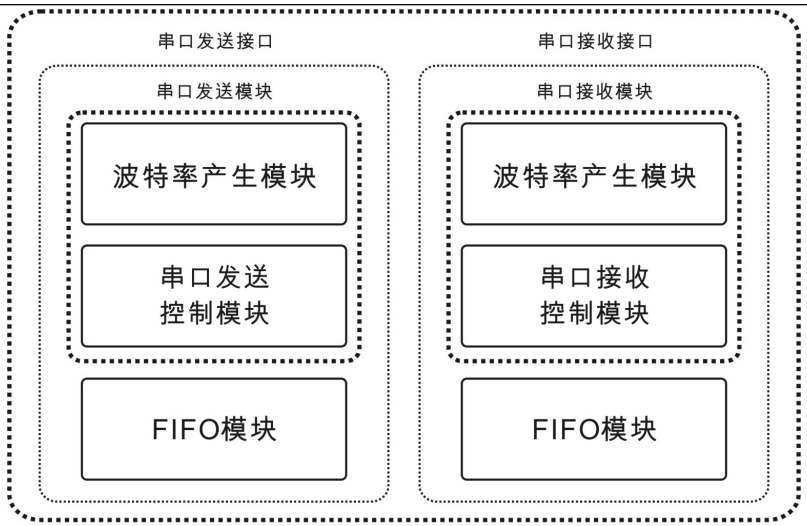


图 0.11 （串口系统建模的概念图）

在串口硬件模块（串口系统建模）里，分类了发送接口和接受接口。发送发送接口包含了 FIFO 模块，波特率产生模块和串口发送控制模块。串口接收串口包含了 FIFO 模块，波特率产生模块，串口接收控制模块。串口发送模块是组合了波特率产生模块和串口发送控制模块，串口接收模块是组合了波特率产生模块和串口接收控制模块。

串口系统建模之间的模块基本单元分类：

基本单元	模块
功能模块	波特率产生模块，FIFO 模块
控制模块	串口发送控制模块，串口接收控制模块
组合模块	串口发送模块，串口接收模块

串口系统建模之间的层次分类：

层次	建模
基本建模	串口发送模块，串口接收模块
接口建模	串口发送接口，串口接收接口
系统建模	串口系统

在这里笔者只是简介了笔者最爱的“低级建模”的结构分类而已，事实上每一个基本单元和每一个层次都有严谨的定义。建模是 Verilog HDL 语言的结构，越庞大的设计建模所带来的后期影响是读者/笔者远远都猜想不到的。故建模对于 Verilog HDL 语言来说是非常重要的基础。

好了笔者就不多话了，有关“低级建模”的笔记笔者已经老早准备好了。笔者在这里只是简单的表述一下 Verilog HDL 语言结构的概念而已，然而笔者需要再强调一下：Verilog HDL 语言的结构是自由的，然而笔者的“低级建模”是笔者自定义的结构而已。当然读者也可以建立自己的结构。

0.12 Verilog HDL 语言使用规则（方法）简介

“一种设计，超过 10 个编辑风格”这一句话不仅表示了一种设计有多种结构（很多设计都是单文件主义，或者不存在结构），而且也代表了设计中所使用的不同规则（方法）。但是不是所有设计都有固定的使用方法，这一点读者可以诚实的问自己：“自己的设计是否有一套的使用方法呢？”好吧~笔者就不故意为难了。

Verilog HDL 语言是一个非常自由的语言，故被使用最多的方法就是“自由方法”（没有-方法）呵呵~这是事实。当自定义自己的方法的时候，读者要问自己，自己所使用的方法是“注重什么”。换做笔者，笔者注重“模块解读能力”又或者说有一套方法可以提高模块的表达能力。故 Verilog HDL 语言的结构就是其中一环了，但是这还不够，当我们再开始设计的时候，我们还需要一套的“用法模板”。

在这里笔者向读者们推荐吧“基于仿顺序操作”的“用法模板”（到目前为止，笔者还没有给它固定的命名，如果读者有什么好想法就告诉笔者吧 ）。仿顺序操作是什么？就是利用 Verilog HDL 语言自身的特质去模仿一些顺序语言如 C 语言，故称为“仿顺序操作”。

仿顺序操作的思想是非常有潜力的，但是在建模篇里笔者仅是以“步骤”作为这个用法的入门认识。当读者对建模和时序有一定的认识以后，读者会进一步发现这个思想非常的“变态”（潜力的另一极端用词）。作为入门，仿顺序操作用法都是以“步骤”作为基本单位，这个概念和 C 语言的步骤没什么两样。

“步骤”也可以用“简易状态机”来理解，但是“状态机”的概念太死了，笔者不怎么喜欢，还是使用“步骤”来称呼比较情切：

仿顺序操作基本的用法模板：

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        begin
            i <= 4'd0;
            ....
        end
    else
        case( i )

            0:
            begin .... i <= i + 1'b1; end

            1:
            ......
```

```
        endcase
```

i 代表了“步骤的指向”至于 case ... endcase 之间是一个完成工作的经过。笔者举个简单的实例，就以流水灯来说话吧：

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        begin
            i <= 4'd0;
            rLED <= 4'b0000;
        end
    else
         case( i )

            0:
            begin rLED <= 4'b0001; i <=i + 1'b1; end

            1:
            if( Timer == T10MS ) i <= i + 1'b1; end

            2:
            begin rLED <= 4'b0010; i <=i + 1'b1; end

            3:
            if( Timer == T10MS ) i <= i + 1'b1; end

            4:
            begin rLED <= 4'b0100; i <=i + 1'b1; end

            5:
            if( Timer == T10MS ) i <= i + 1'b1; end

            6:
            begin rLED <= 4'b1000; i <=i + 1'b1; end

            7:
            if( Timer == T10MS ) i <= 4'd0; end

        endcase
```

在 case ... endcase 之间，步骤 i 等于 0,2,4,6 的时候是更新 LED（流水操作），步骤 i 等于 1,3,5,7 的时候是延迟 10ms。

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

```
    if( !RSTn )
        begin
            i <= 4'd0;
            rLED <= 4'b0000;
        end
    else
         case( i )

            0,2,4,6:
            begin rLED <= { rLED[0], rLED[3:1] }; i <=i + 1'b1; end

            1,3,5,7:
            if( Timer == T10MS ) i <= i + 1'b1; end

            8:
           begin i <= 4'd0; end

      endcase
```

此外，还可以把步骤 i 当成简单的循环。上面一段代码表达了，当步骤 i 等于 0,2,4,6 的时候就更新 rLED,。反之，当步骤 i 等于 1,3,5,7 的时候就延迟 10ms。在步骤 i 等于 8 的时候是步骤返回操作。

```
always @ ( posedge CLK or negedge RSTn )
    if( !RSTn )
        begin
            i <= 4'd0;
            rLED <= 4'b0001;
        end
    else
         case( i )

            0:
            begin rLED <= {rLED[0] , rLED[3:1]} ; i <=i + 1'b1; end

            1
            if( Timer == T10MS ) i <= i - 1'b1; end

      endcase
```

又或者更进一步的压缩步骤，使得代码更直接而且更节省资源。在上面一段代码当中，只有两个步骤，亦即步骤 0 和 1。步骤 0 是更新 rLED，步骤 1 是延迟 10ms，然而这两个步骤之间交互交替使而产生流水灯效果。

当然，步骤 i 的用法不仅而已，如果把“时序”的概念引入的话：

```
always @ ( posedge CLK or negedge RSTn )
      if( !RSTn )
          begin
               i <= 4'd0;
               Mper <= 8'd0;
               Mcand <= 8'd0;
               Sum <= 8'd0;
          end
      else
           case( i )
              0:
              begin Mper <= Mper_Sig; Mcand <= Mcand_Sig ; Sum <= 8'd0i <=i + 1'b1; end

              1
              if( Mcand == 0 ) i <= i + 1'b1;
              else Sum <= Sum + Mper; Mcand <= Mcand - 1'b1; end

             ......

         endcase
```

以上一段代码是简单的乘法运算，Mper 是乘数的暂存器，Mcand 是被乘数的暂存器，Sum 是累加空间。当步骤 i 等于 0 的时候初始化相关的寄存器，在步骤 i 等于 1 的时候执行乘法操作 ... 在这里读者也可这样说：“在 T0 的时候初始化相关的寄存器，在接下来的时钟执行乘法操作 ... ”

总结之下，这个用法可以伸缩的范围非常之大。除外，它所带来的好处也非常之多：

- 提供了 Verilog HDL 语言顺序操作的支持。
- 提高了模块的表达能力。
- 提供了仿顺序操作・建模的结构基础。。

但是它也带来一些限制：

- 不推荐嵌套 case ... endcase 和 if。
- 该用法不推荐出现过多在同一个模块中。

这些限制是笔者标记下来的，这之间和“低级建模”有多少关系。当然，如果读者不遵守的话也没有问题。显然这个用法也不是万能的，尤其是一些紧密的 RTL 级建模，如: VGA 驱动，它就显得无用武之地了。事实上这个用法到目前为止，笔者还在不停的研究当中，越深入学习它，就越发现它的潜能很深 ...

好了，之一章节笔者就介绍到那么多，自定义使用规则（用法）对于 Verilog HDL 语言来说是非常自由但也非常重要，没有固定的使用方法，模块的解读能力会大打折扣。这些简单的道理，也只有当读者不停深入学习 Verilog HDL 语言的时候自然就会明白。

0.13 认识 RTL 级设计（建模）

笔者差不多也要结束这一章 Verilog HDL 语言的扫盲文了，在结束之前笔者或多或少都有责任加强读者们对 RTL 级设计（建模）的认识。在前面，笔者曾经说过 RTL 级建模最受注意的特征就是“时钟”亦即 CLK 信号，要明白 CLK 的定义笔者就要乘坐时光机回到读学院的时候。

用凡人的话来说 CLK 代表了一个模块的心跳节拍，这个心跳节拍提供模块可以消耗的动力。但是 CLK 信号真正可以被模块所用到不是它的高电平又或者低电平，而是上升沿（低电平到高电平的变化）和下降沿（高电平到低电平的变化）。

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

在上面一段代码中的 always @ (posedge CLK) 表达了 always @ () 以下的内容在每一个 CLK 的上升沿发成操作：

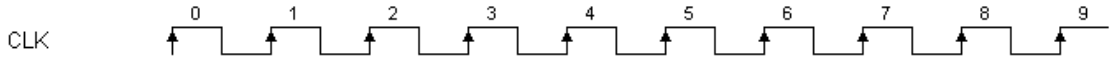


图 0.13a（CLK 的上升沿有效）

假设有一串长长而且连续的 CLK 就会产生最基本的时序图（没有 CLK 基本上是没有时序图），如图 0.13a 所示。

对于 RTL 级设计来说 CLK 是模块的心跳，没有心跳模块就不能活动，没有心跳就没有时序图。换另一句话说，构成 RTL 级最基本的设计需要“寄存器”为最小的建模单位，然后再加上模块可以活动的 CLK 信号。

```
always @ ( posedge CLK )
      Counter1 <= Counter1 + 1'b1;

always @ ( posedge CLK1/2 )
      Counter2 <= Counter2 + 1'b1;
```

上面有一段代码是 Counter1 + CLK 和 Counter2 + CLK1/2 促成最简单的 RTL 级设计。Counter1 在每一个 CLK 时钟内递增，然而 Counter2 在每一个 CLK1/2 时钟内递增。到

目前为止 Counter1 和 Counter2 还是独立关系。

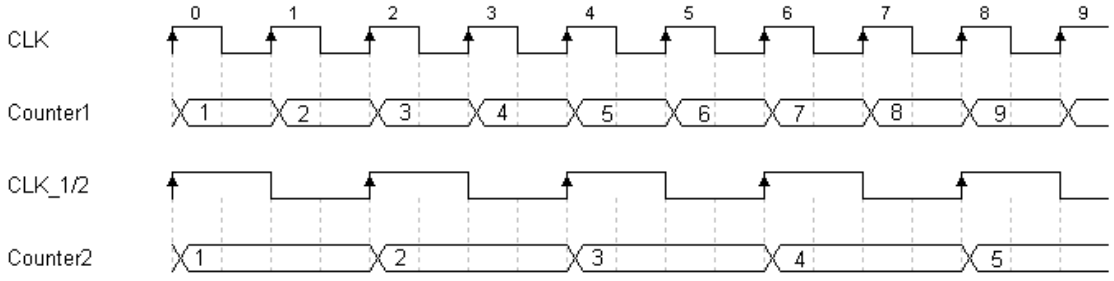


图 0.13b (Counter1 和 Counter2 产生的时序图)

图 0.13b 是 Counter1 + CLK 和 Counter2 + CLK1/2 产生的时序图。在这里 Counter2 所使用的频率是 Counter1 的一半。在每一个 CLK 的上升沿 Counter1 都递增，然而在每一个 CLK1/2 的上升沿 Counter2 都递增。

```
always @ ( posedge CLK )
    Counter1 <= Counter1 + 1'b1;

always @ ( posedge CLK1/2 )
    Counter2 <= Counter1;
```

假设笔者把 Counter1 和 Counter2 联系起“关系”的话，如上面的一段代码所述。又会产生怎样的时序图呢？

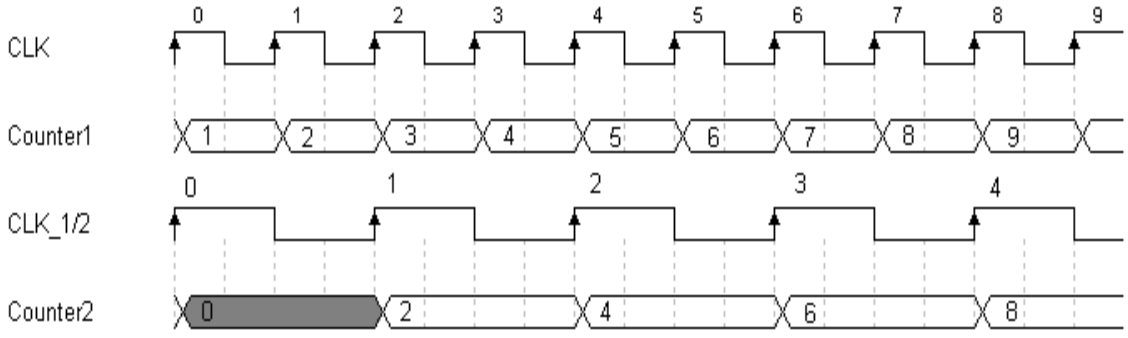


图 0.13c（Counter1 和 Counter2 建立关系以后产生的时序图）

图 0.13c 是 Counter1 和 Counter2 建立关系以后多产生的时序图。在 CLK 是 T0 的时候，CLK_1/2 也是在 T0。由于在 T0 之前 Counter1 什么也没有，所以 Counter 什么也读不到（一般上 0 为复位值）。在 CLK_1/2 是 T1，Counter2 尝试读取 Counter1 的“过去值”，结果 Counter2 读到值 2，所以在 CLK_1/2 的 T1，Counter2 的“未来值”是 2。类似的事情也发生在 CLK_1/2 的 T2，T3 和 T4 的时候。

在这里读者先不用管“过去值”和“未来值”的定义，这是笔者在时序篇里的专用词。读者需要焦距的是，每一次 Counter1 成功递增是发生在 CLK 的上升沿，然而 Counter2

每一次成功读取 Counter1 的值都是发生在 CLK_1/2 的上升沿。换句话说，CLK 的上升沿是触发 Counter1 递增，CLK_1/2 的上升沿是触发 Counter 读取 Counter1 的过去值。以上的内容就是 RTL 级设计最基本的思想。

至于组合逻辑级设计呢？在 Verilog HDL 语言中，如果我们把 Verilog HDL 语言看成是理想的语言，那么组合逻辑就可以直接无视被 CLK 的影响，因为组合逻辑取得的是即时的结果。

举个简单的组合逻辑级的设计：

```
always @ ( * )
    Sum = A + B + C
```

这是一个由组合逻辑所组成的简单 3 路加法器。

输入 A	输入 B	输入 C	即时结果 Sum = A + B + C
1	2	4	7
2	0	5	7
3	3	1	7
4	3	2	9

上面的图标表示了3路加法器求得的即时结果 ... 在这里CLK信号对于它来说已经再也不重要了。

假设读者不把 Verilog HDL 语言看成是“理想”的话，组合逻辑会产生“物理”上的延迟。但是笔者还是建议读者把 Verilog HDL 语言看成是一个理想的工具为好。换另一个角度来看的话，C 语言和 Verilog HDL 语言都是工具，难道 C 语言会产生“物理”上的延迟吗？此外这样的想法对于 Verilog HDL 语言的设计会带来很大的好处，尤其是看懂波形图（时序图）哪一环，效果会更加明显。

0.14 过渡中，沉住气！朋友！

到了这一章节笔者不知不觉又寂寞了，基本上有关 Verilog HDL 语言的扫盲文需要告一段落了。学习 Verilog HDL 语言不像学习一些高级语言，对于高级语言来说它们已经是完成品了，其外它们还有很多被隐藏的指令，这些好处无疑是减轻了学习者的负担。相反的 Verilog HDL 语言既是完成品，既不是完成品，就是因为它太自由了 ... 所以往往会让学习者感到疑惑，很疲惫和浮躁（我不学了！）。

学习 Verilog HDL 语言需要一段过渡期的，快则半年，普通则 1~2 年，慢则很多年。即使经过了过渡期这也不表示已经掌握 Verilog HDL 语言了。所以呀朋友，希望你们可以沉住气，“欲速则不达”这是老祖先的智慧，它非常适合用在学习 Verilog HDL 语言的路上。Verilog HDL 语言可以延长到的范围完完全全超过参考书的内容，对于笔者来说

也看不到尽头。

那些有学习单片机经验的朋友，最好不让学习单片机的思想主宰了你。就如笔者在前面所说的那样，Verilog HDL 语言既不是顺序语言而且也非常的自由，Verilog HDL 语言不像 C 语言那样有丰富的库支持，甚至库的概念也不适合用在它身上。但是即使它们是两个世界的居民，但是偶尔 Verilog HDL 语言可以在很多地方向 C 语言借签。相反的 C 语言就不能向 Verilog HDL 语言借签了。

此外 Verilog HDL 语言还有两大阵列，就是综合语言和验证语言，这更是给学习者雪上加霜。太多的学习者会困惑在这两种语言的中间，所谓的困惑是思路的困惑。在这里，笔者建议先无视验证语言，先把综合语言学好（综合语言也没有什么好学的，就如在前面章节笔者所举例的那样，关键字和操作符少得可怜），最重要还是掌握结构(建模)和使用规则（用法）。它们就像挥动着倚天剑和屠龙宝刀的招式，没有了这些招式倚天剑和屠龙宝刀不过是一件单纯的金属而已。至于验证语言，在未来有需要的时候再学也不迟。

当阅读他人模块的时候，不要过于转牛角尖的看懂他人的思路，只要明白其中的内容就好。最重要还是如何使用自己的结构和方法去建立他人的思路，从中读者会学得更多。这一点是绝对的事实。就算现在的读者没有能力建立自己的结构也没有关系，来日方长读者有的就是时间。如果读者很钟爱笔者所建立的结构和方法，笔者很乐意也很荣欣被应用（这也是笔者写笔记的初衷）。

最后的最后还是那么一句话，沉住气朋友，掌握 Verilog HDL 语言需要的不只是技术而已，最重要是那颗安静的心，安静的心会带读者乘风破浪，一方通行。此外记录笔记的习惯更为重要，向自己学习比起向他人学习更有学习的价值。如果有时间的话，就坐下来研究研究 Verilog HDL 语言这个怪家伙吧。

总结：

这一章终于到总结了，从第一章节到最后一个章节，从最简单的问题：“什么是 HDL 语言？”，直到一些困惑度极高的问题：“Verilog HDL 语言的结构？”，“Verilog HDL 语言的使用规则（用法）？”，甚至学习 Verilog HDL 语言该保持的心情，笔者也不放过。

不过这短短的二十几页，应该足够达到扫盲的作用了吧？虽然这一章的扫盲文，效果不足覆盖所有言内容，但是只要有效的开了一个入门的口子，对于接下来的路就简单许多。人们常说：“入门，入门”，学习要从入门开始，如果门都没有，又要如何进入呢？

笔者学习 Verilog HDL 语言也有一段时间了，对于入门的心得多少都有。所以笔者觉得比起如何入门，先知道个大概来得更重要。有一句笑话说过：“既然你要入门，你至少需要知道，你要进入那一扇门的形状？圆的还是方的？要是不清楚，就算有门在哪儿，你也进不了门，结果“进入门的门都没有”..... ”所以说，扫盲的作用是把目标建立出大概的轮廓 ... 不然真的进入门的门都没有~啊哈哈哈！好了好了，笔者也不多话了，是时候画上句号了。