

03 | 硬件语言筑基（一）：从硬件语言开启手写CPU之旅

2022-08-01 LMOS 来自北京

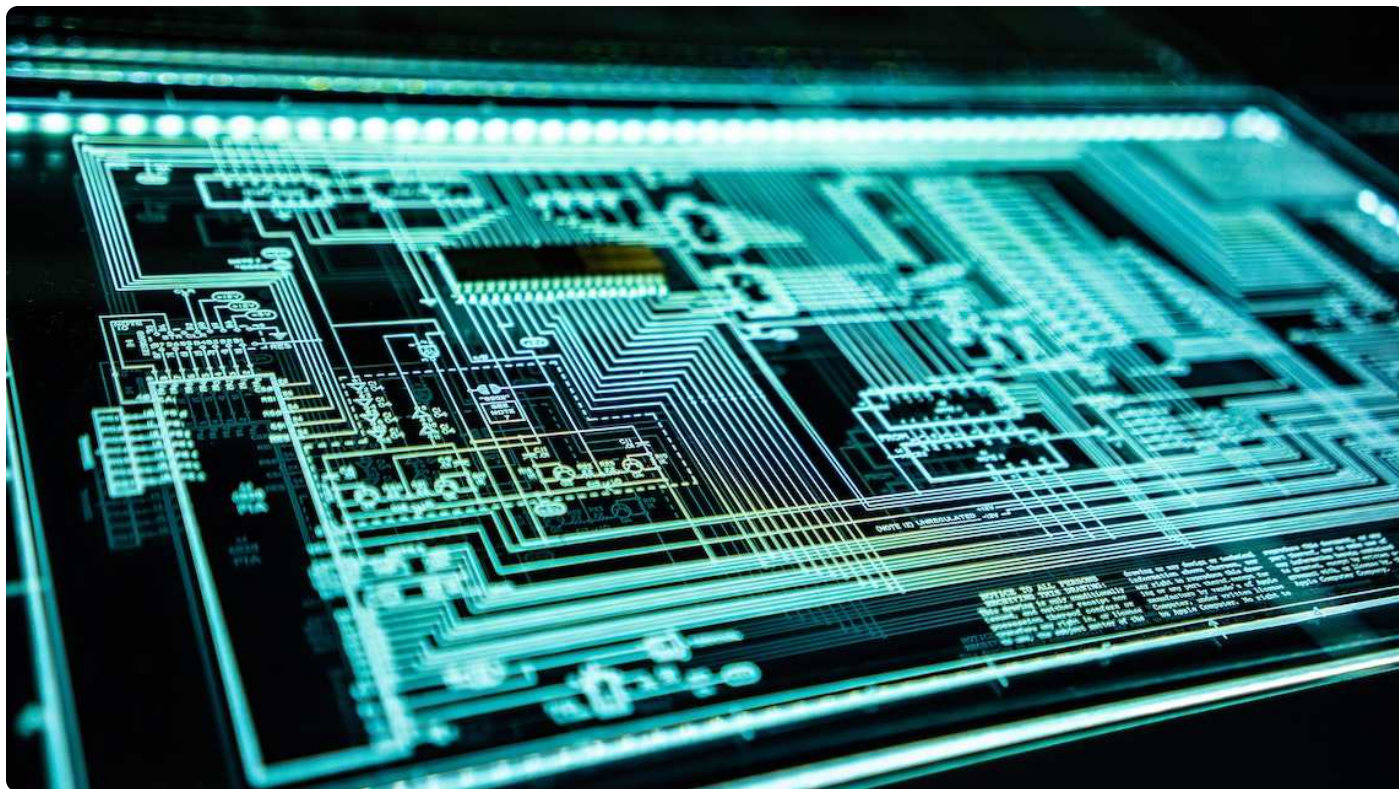


天下无鱼

<https://shikey.com/>

《计算机基础实战课》

[课程介绍 >](#)



讲述：陈晨

时长 15:18 大小 14.02M



你好，我是 LMOS。

我们都知道，自己国家的芯片行业被美国“吊打”这件事了吧？尤其是像高端 CPU 这样的芯片。看到相关的报道，真有一种恨铁不成钢的感觉。你是否也有过想亲自动手设计一个 CPU 的冲动呢？

万丈高楼从地起，欲盖高楼先打地基，芯片是万世之基，这是所有软件基础的开始，这个模块我会带你一起设计一个迷你 RISC-V 处理器（为了简单起见，我选择了最火热的 RISC-V 芯片）。哪怕未来你不从事芯片设计工作，了解芯片的工作机制，也对写出优秀的应用软件非常重要。

这个处理器大致是什么样子呢？我们将使用 Verilog 硬件描述语言，基于 RV32I 指令集，设计一个 32 位五级流水线的处理器核。该处理器核包括指令提取单元、指令译码单元、整型执行

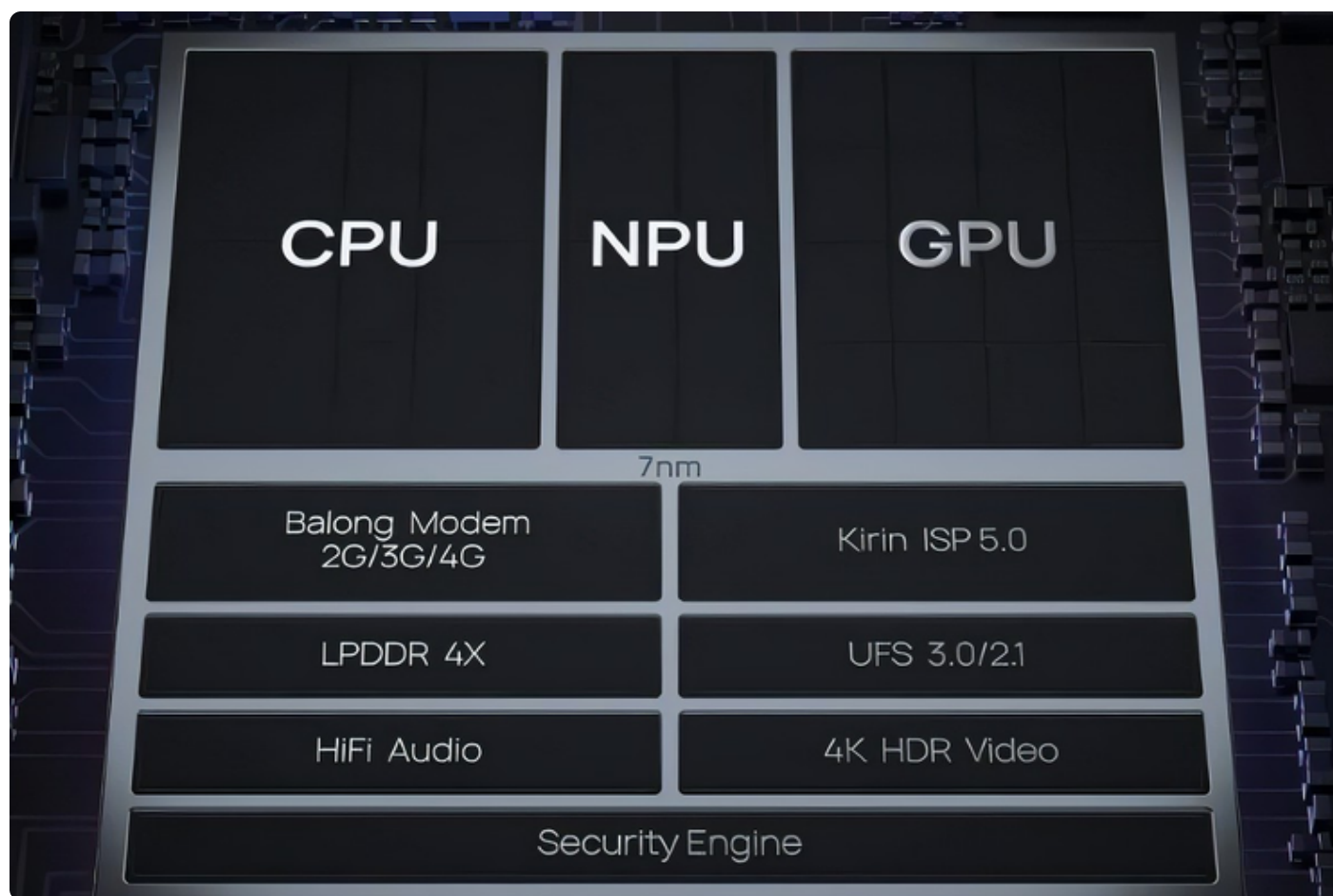
单元、访问存储器和写回结果等单元模块，支持运行大多数 RV32I 的基础指令。最后，我们还会编写一些简单汇编代码，放在设计出来的处理器上运行。

我会通过两节课的篇幅，带你快速入门 Verilog，为后续设计迷你 CPU 做好准备。这节课我们先来学习硬件描述语言基础，芯片内部的数字电路设计正是由硬件语言完成的。

一个芯片的内部电路是怎么样的？

作为开发，你日常最常用的编程语言是什么？也许是 C 语言、Java、Go、PHP.....这些高级编译语言吧。而硬件设计领域里，也有专门的硬件描述语言。为什么会出现专门的硬件描述语言呢？这还要先从芯片的内部结构说起。

一般情况下，你所接触到的处理器芯片，已经不是传统意义上的 CPU 了，比如在业界很有名的国产手机芯片华为麒麟 990 芯片。我把麒麟 990 的功能模块图贴在了后面，对照图片会更直观。这样一款芯片，包含了 CPU 核、高速缓存、NPU、GPU、DDR、PMU 等模块。



而在芯片设计时，根据不同模块的功能特点，通常把它们分为数字电路模块和模拟电路模块。

模拟电路还是像早期的半导体电路那样，处理的是连续变化的模拟信号，所以只能用传统的电路设计方法。而数字电路处理的是已经量化的数字信号，往往用来实现特定的逻辑功能，更容易被抽象化，所以就产生了专门用于设计数字电路的硬件描述语言。

硬件描述语言从发明到现在，已经有 20 多年历史。硬件描述语言可以让你更直观地去理解数字电路的逻辑关系，从而更方便地去设计数字电路。

现在业界的 IEEE 标准主要有 VHDL 和 Verilog HDL 这两种硬件描述语言。在高层次数字系统设计领域，大部分公司都采用 Verilog HDL 完成设计，我们后面的实现也会用到 Verilog。

千里之行，始于足下。在 Verilog 学习之前，我们需要先完成思路转换，也就是帮你解决这个问题：写软件代码和写硬件代码的最大区别是什么？搞明白了这个问题，你才能更好地领会 Verilog 语言的设计思想。

Verilog 代码和 C 语言、Java 等这些计算机编程语言有本质的不同，在可综合（这里的“可综合”和代码“编译”的意思差不多）的 Verilog 代码里，基本所有写出来的东西都对应着实际的电路。

所以，我们用 Verilog 的时候，必须理解每条语句实质上对应着什么电路，并且要从**电路的角度**来思考它为何要这样设计。而高级编程语言通常只要功能实现就行。

我再举几个例子来说明：声明变量的时候，如果指定是一个 reg，那么这个变量就有寄存数值的功能，可以综合出来一个实际的寄存器；如果指定是一段 wire，那么它就只能传递数据，只是表示一条线。在 Verilog 里写一个判断语句，可能就对应了一个 MUX（数据选择器），写一个 for 可能就是把一段电路重复好几遍。

最能体现电路设计思想的就是 always 块了，它可以指定某一个信号在某个值或某个跳变的时候，执行块里的代码。通过使用 Verilog 语言，我们就能完成芯片的数字电路设计工作了。没错，芯片前端设计工程师写 Verilog 代码的目的，就是**把一份电路用代码的形式表示出来，然后由计算机把代码转换为所对应的逻辑电路。**

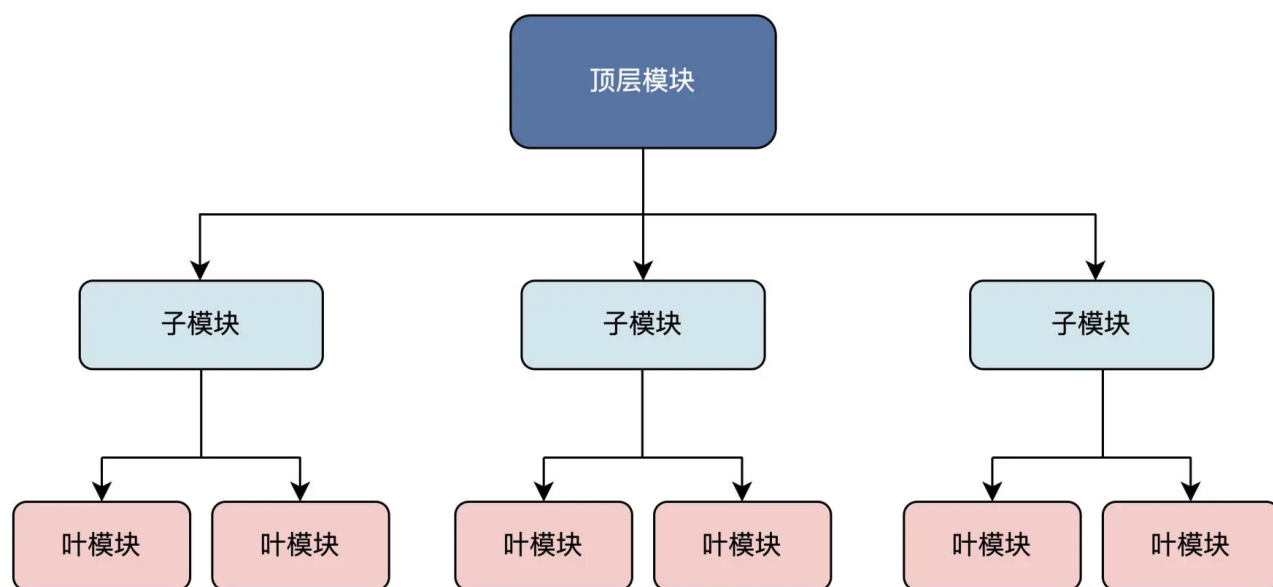
芯片如何设计？

说到这里你可能还有疑惑，听起来芯片设计也没那么复杂啊？其实这事儿说起来简单，但实践起来却相当复杂。接下来，我就说说，一个工业级的芯片在设计阶段大致会怎么规划。

在开始一个大的芯片设计时，往往需要先从整个芯片系统做好规划，在写具体的 Verilog 代码之前，把系统划分成几个大的基本的功能模块。之后，每个功能模块再按一定的规则，划分出下一个层次的基本单元。

这和 Verilog 语言的 module 模块化设计思想是一致的，上一层模块对下一层子模块进行例化，就像其他编程语言的函数调用一样。根据包含的子功能模块一直例化下去，最终就能形成 hierarchy 结构。

这种自顶向下的设计方法，可以用后面的树状结构图来表示：



极客时间

从上图我们也可以看出，Verilog 都是基于模块进行编写的，一个模块实现一个基本功能，大部分的 Verilog 逻辑语句都放在模块内部。

从一段代码入门 Verilog

说完语言思路和硬件的模块化设计，接下来，我带你学习一下 Verilog 的基本模块和逻辑语句是怎么写的。

很多 Verilog 初学者刚开始都是从一些基础知识慢慢去看，比如基本语法、数据类型、赋值语句、条件语句……总想着把 Verilog 的全部基础知识看完了，再开始动手写代码。

但是你有没有想过，这些详细的基础知识，一两天自然是看不完的。而当你坚持了一段时间把它看完，以为可以上手写代码的时候，又会发现前面的基本语句全都忘了。这样的学习方法并不可取，效果也不好，所以我换个方法带你入门。我们先不去罗列各种详细的基础知识，而是从学习一段代码开始。

我会以一个 4 位十进制计数器模块为例，让你对 Verilog 模块代码有更直观的认识，然后根据这段代码模块，给你讲讲 Verilog 语言基础。这里先把完整代码列出来，后面再详细拆解。

 复制代码

```
1 module counter(  
2     //端口定义  
3     input          reset_n, //复位端，低有效  
4     input          clk,      //输入时钟  
5     output [3:0]    cnt,      //计数输出  
6     output          cout     //溢出位  
7 );  
8  
9 reg [3:0]          cnt_r ;    //计数器寄存器  
10  
11 always@(posedge clk or negedge reset_n) begin  
12     if(!reset_n) begin        //复位时，计时归0  
13         cnt_r                <= 4'b0000 ;  
14     end  
15     else if (cnt_r==4'd9) begin //计时10个cycle时，计时归0  
16         cnt_r                <= 4'b0000;  
17     end  
18     else begin  
19         cnt_r                <= cnt_r + 1'b1 ; //计时加1  
20     end  
21 end  
22  
23 assign cout = (cnt_r==4'd9) ; //输出周期位  
24 assign cnt  = cnt_r ;         //输出实时计时器  
25  
26 endmodule
```

看了这段代码，也许你云里雾里，或者之前没接触过硬件语言，心里有点打鼓。不过别担心，入门硬件语言并不难，我们按照代码顺序依次来看。

模块结构

首先，让我们看一看这段代码的第一行和最后一行。没错，一个模块的定义是以关键字 `module` 开始，以 `endmodule` 结束。`module` 关键字后面跟的 `counter` 就是这个模块的名称。

看着有没有熟悉的感觉？你可能觉得，这个看着跟其他编程语言的函数定义也没多大区别吧？别急着下结论，再仔细看看接口部分，发现没有？这就和函数传入的参数很不一样了。

```
module counter(  
    // 接口部分  
    input          reset_n,  
    input          clk,  
    output [3:0]    cnt,  
    output         cout  
);  
.....          // 逻辑功能部分  
endmodule
```

Verilog 模块的接口必须要指定它是输入信号还是输出信号。

输入信号用关键字 **input** 来声明，比如上面第 4 行代码的 **input clk**；输出信号用关键字 **output** 来声明，比如代码第 5 行的 **output [3:0] cnt**；还有一种既可以输入，又可以输出的特殊端口信号，这种双向信号，我们用关键字 **inout** 来声明。

数据类型

前面我提到过，在可综合的 Verilog 代码里，基本所有写出来的东西都会对应实际的某个电路。而 Verilog 代码中定义的数据类型就能充分体现这一点。

```
parameter    SIZE = 2'b01;  
reg          [3:0]    cnt_r;  
wire [1:0]    cout_w;
```

比如上面代码的第 9 行，表示定义了位宽为 4 bit 的寄存器 **reg** 类型信号，信号名称为 **cnt_r**。

寄存器 **reg** 类型表示抽象数据存储单元，它对应的就是一种寄存器电路。**reg** 默认初始值为 X（不确定值），换句话说就是，**reg** 电路在上电之后，输出高电平还是低电平是不确定的，一般是在系统复位信号有效时，给它赋一个确定值。比如例子中的 **cnt_r**，在复位信号 **reset_n** 等于低电平时，就会给 **cnt_r** 赋“0”值。

`reg` 类型只能在 `always` 和 `initial` 语句中被赋值，如果描述语句是时序逻辑，即 `always` 语句中带有时钟信号，寄存器变量对应为触发器电路。比如上述定义的 `cnt_r`，就是在带 `clk` 时钟信号的 `always` 块中被赋值，所以它对应的是触发器电路；如果描述语句是组合逻辑，即 `always` 语句不带有时钟信号，寄存器变量对应为锁存器电路。

我们常说的电子电路，也叫电子线路，所以电路中的互连线是必不可少的。`Verilog` 代码用线网 `wire` 类型表示结构实体（例如各种逻辑门）之间的物理连线。`wire` 类型不能存储数值，它的值是由驱动它的元件所决定的。驱动线网类型变量的有逻辑门、连续赋值语句、`assign` 等。如果没有驱动元件连接到线网上，线网就默认为高阻态“Z”。

为了提高代码的可读性和可维护性，`Verilog` 还定义了一种参数类型，通过 `parameter` 来声明一个标识符，用来代表一个常量参数，我们称之为**符号常量**，即标识符形式的常量。这个常量，实际上就是电路中一串由高低电平排列组成的一个固定数值。

数值表达

说到数值，我们再了解一下 `Verilog` 中的数值表达。还是以前面的 4 位十进制计数器代码为例，我们定位到第 13 行代码：

```
cnt_r    <= 4'b0000;
```

这行代码的意思是，给寄存器 `cnt_r` 赋以 `4'b0000` 的值。

这个值怎么来的呢？其中的逻辑“0”低电平，对应电路接地（`GND`）。同样的，逻辑“1”则表示高电平，对应电路接电源 `VCC`。除此之外，还有特殊的“X”和“Z”值。逻辑“X”表示电平未知，输入端存在多种输入情况，可能是高电平，也可能是低电平；逻辑“Z”表示高阻态，外部没有激励信号，是一个悬空状态。

当然，为了代码的简洁明了，`Verilog` 可以用不同的格式，表示同样的数值。比如要表示 4 位宽的数值“10”，二进制写法为 `4'b1010`，十进制写法为 `4'd10`，十六进制写法为 `4'ha`。这里我需要特殊说明一下，数据在实际存储时还是用二进制，位宽表示储存时二进制占用宽度。

运算符

接下来我们看看 `Verilog` 的运算符，对于运算符，`Verilog` 和大部分的编程语言的表示方法是一样的。

比如算术运算符 `+` `-` `*` `/` `%`，关系运算符 `>` `<` `<=` `>=` `==` `!=`，逻辑运算符 `&&` `||` `!`（与或非），还有条件运算符 `?`，也就是 C 语言中的三目运算符。例如 `a?b:c`，表示 `a` 为真时输出 `b`，反之为 `c`。

但在硬件语言里，位运算符可能和一些高级编程语言不一样。其中包括 `~` `&` `|` `^`（按位取反、按位与，按位或，以及异或）；还有移位运算符，左移 `<<` 和右移 `>>`，在生成实际电路时，左移会增加位宽，右移位宽保存不变。

条件、分支、循环语句

还有就是条件语句 `if` 和分支语句 `case`，由于它们的写法和其它高级编程语言几乎一样，基本上你掌握了某个语言都能理解。

这里我们重点来对比不同之处，也就是用 Verilog 实现条件、分支语句有什么不同。用 `if` 设计的语句所对应电路是有优先级的，也就是多级串联的 MUX 电路。而 `case` 语句对应的电路是没有优先级的，是一个多输入的 MUX 电路。设计时，只要我们合理使用这两个语句，就可以优化电路时序或者节省硬件电路资源。

此外，还有循环语句，一共有 4 种类型，分别是 `while`，`for`，`repeat` 和 `forever` 循环。注意，循环语句只能在 `always` 块或 `initial` 块中使用。

过程结构

下面我们来说说过程结构，最能体现数字电路中时序逻辑的就是 `always` 语句了。`always` 语句块从 0 时刻开始执行其中的行为语句；每当满足设定的 `always` 块触发条件时，便再次执行语句块中的语句，如此循环反复。

因为 `always` 语句块的这个特点，芯片设计师通常把 `always` 块的触发条件，设置为时钟信号的上升沿或者下降沿。这样，每次接收到一个时钟信号，`always` 块内的逻辑电路都会执行一次。

前面代码例子第 11 行的 `always` 语句，就是典型的时序电路设计方法，有没有感觉到很巧妙？

```
always@(posedge clk or negedge rstn) begin
.....           // 逻辑语句
```


还有一种过程结构就是 **initial** 语句。它从 0 时刻开始执行，且内部逻辑语句只按顺序执行一次，多个 **initial** 块之间是相互独立的。理论上，**initial** 语句是不可以综合成实际电路的，多用于初始化、信号检测等，也就是在编写验证代码时使用。

到这里，在我看来比较重要的 **Verilog** 基础知识就讲完了，这门语言的知识脉络我也为你搭起了骨架。当然了，**Verilog** 相关知识远远不止这些。如果你对深入学习它很感兴趣，推荐你翻阅《**Verilog HDL 高级数字设计**》等相关资料拓展学习。

总结回顾

今天是芯片模块的第一节课，我们先了解了芯片的内部电路结构。一个芯片的内部电路往往分为数字电路模块和模拟电路模块。对于数字电路模块，可以使用 **Verilog** 硬件描述语句进行设计。

尽管 **Verilog** 这样的硬件语言你可能不大熟悉，但只要抓住本质，再结合代码例子建立知识脉络，学起来就能事半功倍。

要想熟悉硬件语言，我们最关键的就是做好思路转换。硬件语言跟高级编程语言本质的不同就是，使用 **Verilog** 的时候，必须理解每条语句实质上对应的什么电路，并且要从**电路的角度**来思考它为何要这样设计，而高级编程语言通常只要实现功能就行。

我再带你回顾一下，**Verilog** 语言和高级编程语言具体有哪些不同：

1.模块结构：**Verilog** 的模块结构和其他语言的函数定义不一样，它既可以有多个输入信号，也可以输出多个结果。而且，模块上的接口信号，必须要指定是输入信号和输出信号。

2.数据类型：跟我们在高级编程语言见到的变量类型相比，**Verilog** 定义的数据类型也有很大不同。**reg** 类型对应的是寄存器电路，**wire** 类型对应的是电路上的互连线，标识符对应的是一串固定的高低电平信号。

3.数据表达：**Verilog** 代码中的数据，本质上就是高低电平信号。“0”代表低电平，“1”代表高电平，不能确定高低电平的就用“X”来表示。

4.运算符: Verilog 中的大部分运算符和其他语言是一样的,但是要注意位操作运算符,它们对应的是每一位电平按指定逻辑跳变,还有移位操作,一定要注意移位信号的数据位宽。

5.条件、分支、循环语句: Verilog 中的条件 if 语句是有优先级的,而 case 语句则没有优先级,合理利用它们可以优化电路时序或节省硬件电路资源。循环语句则是把相同的电路重复好几遍。

6.过程结构: 这是实现时序电路的关键。我们可以利用 always 块语句设定一个时钟沿,用来触发相应逻辑电路的执行。这样,我们就可以依据时钟周期来分析电路中各个信号的逻辑跳变。而 initial 语句常在验证代码中使用,它可以从仿真的 0 时刻开始设置相关信号的值,并将这些值传输到待验证模块的端口上。



下节课,我会带你设计一个简单的电路模块,既能帮你复习今天学到的知识,还能通过实践体会一下代码是怎样生成电路的,敬请期待。


思考题

为什么很多特定算法，用 Verilog 设计并且硬件化之后，要比用软件实现的运算速度快很多？

欢迎你在留言区跟我交流互动，也推荐你把这节课分享给更多朋友。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 24 元

 生成海报并分享

 赞 11

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。


上一篇 02 | RISC特性与发展：RISC-V凭什么成为“半导体行业的Linux”？

下一篇 04 | 硬件语言筑基（二）：我们的代码是怎么生成具体电路的？

精选留言 (24)

 写留言



青玉白露 

2022-08-06 来自湖北

感觉大家很吃力，做了一个关于Verilog的初学者笔记，里面包含在线学习网站<https://zhuanlan.zhihu.com/p/550710744>

作者回复: 6666 感谢分享



 2



peter

2022-08-01 来自湖北

请教老师几个问题：

Q1：代码转化后的逻辑电路，只是电子版的“设计电路”，并不是实际的电路，对吗？

文中有这样一句：“把一份电路用代码的形式表示出来，然后由计算机把代码转换为所对应的逻辑电路。”，此处，“转换后的逻辑电路”并不是实际的电路，应该只是“设计电路”。

Q2：“cnt_r <=4'b000;”？ 此处应该是四个0吧，为什么写成三个0？笔误吗？

Q3：芯片中的模拟电路是用什么设计的？

Q4：芯片前端设计的代码，就是用verilog写的代码，对于一个芯片来说，一般规模多大？比如十万行代码？

作者回复: Q1：对

Q2：是笔误

Q3Q4：是的 只会更多

共 2 条评论 >

👍 5



LockedX

2022-08-01 来自湖北

老师，有个问题一直没有想明白

```
reg [3:0]      cnt_r;
```

```
wire [1:0]     cout_w;
```

reg和wire的声明，冒号前面是数字代表位宽，冒号后面的0含义是什么？

作者回复: 是0到3 4个bit 0到1 2个bit

共 2 条评论 >

👍 2



泡泡龙

2022-08-01 来自湖北

思考题我认为，软件实现要在通用的硬件上执行就势必要执行一些逻辑上的模拟或者转换。而专用的硬件则没有这些转换过程，并且可以针对性的优化执行路径，所以要快许多。

作者回复: 是的



👍 2



Geek_21cfda

2022-08-01 来自湖北

FPGA编程和今天讲得编程有什么区别吗？

作者回复: 有



👍 2



可爱因子1/n

2022-08-06 来自湖北

针对我自己的理解，打个比喻：在性能和单位时间以及油门相同的情况下，我开着一辆车的发动机的转速和一个直接裸的发动机的转速比较，裸的发动机的转速按常理来说应该会更快一

点，不知道我的比喻是否准确？

作者回复: 你说呢



爱酱大胜利

2022-08-04 来自湖北

思考题：

1.设计的目的 **CPU**属于“通用性”设计 一个**CPU**能运行这个算法也能运行那个算法 改改代码就可以 代价就是翻译过程因为通用性所带来的迁就 而硬件化的算法属于“专用性”设计 也就是说它只能做着一件事所以怎么设计方便怎么来 反正以后不用为了实现其他算法再改电路了 这也是软硬件实现的区别之一

2.运行效率 还是由于上一条的设计理念 导致软件的运行依赖于**CPU**实现的特定指令集 原本算法中一个硬件周期的动作可能被转化为多条指令 而每条指令可能会依赖多个运行周期 即使运行会得到优化 也远远比不上一个周期来的快 而且**CPU**为了通用性很可能不会将这种特殊性动作添加到一个不常用**CISC**里

作者回复: 66666



肖水平

2022-08-03 来自湖北

思考题：

根本原因就是专用芯片运算速度比**GPU**快，**GPU**运算速度比**CPU**快，而软件实现的算法是运行在**CPU**上的。

具体原因如下：

- 1、指令数多：软件实现的算法最终要编译成**CPU**可执行的机器指令，中间还会有很多控制指令；
- 2、**CPU**指令执行步骤多：**CPU**执行一条指令需要经过取指、译码、执行、访存、写回步骤；
- 3、多位运算需要拆分：比如**128**位的运算在**32/64**位**CPU**上执行编译后会拆分成多条指令；
- 4、将算法硬件化后可以直接执行运算，不需要通过指令来控制，多位运算也可以实现多位运算模块来直接运算；

总之软件实现的算法在**CPU**消耗的时钟数要多很多，而**CPU**也只擅长控制，不擅长运算。

作者回复: 是的 你学到了





2022-08-02 来自湖北

关于思考题：

我的理解是 Verilog更多面向的是硬件，通过它设计完成之后，可以直接参照其代码流程组合相关硬件，在针对特定算法的设计后，组合出来的就是该算法最“合身”的底层电路模型。

而对于一些软件语言的实现，更多的是业务层面的实现，业务实现之后还要面临“翻译”，翻译成计算机能够执行的指令集，这一步就已经开始出现了效率损耗，同时因为“翻译”动作，也就可能出现为了实现某一功能，底层指令可能要绕一些“弯路”才能实现，毕竟不能像Verilog那样，做到“完美贴切”的电路设计

作者回复: 对的



1



Geek_785f19

2022-08-01 来自湖北

请教老师：

平时做的是业务系统，对底层不熟悉，有兴趣多了解底层知识，学习本课程需要什么前置知识吗

作者回复: 不要什么前置 知识

共 2 条评论 >



1



小杰

2022-08-10 来自浙江

自己的理解。特定的算法，应该是依赖特定的实现，高级语言虽然能在数学上，看着像是一步一步在执行，实际最后的执行经过了编译器，这里其实已经不是一条语句对应cpu一条指令，然后不同cpu的执行可能也会有区别。专门的语言配合专门的cpu，来运行专门的算法，效率肯定是要高于上面使用，高级语言加商业化的通用cpu



天择

2022-08-05 来自湖北

对于一些特殊的算法，可以在硬件电路级别进行优化，充分利用不同电路的特点，而软件算法即使被编译器优化过，也被限制在写死的硬件电路上了。

作者回复: 嗯嗯





skyline

2022-08-04 来自湖北

“它从 0 时刻开始执行，且内部逻辑语句只按顺序执行一次，多个 initial 块之间是相互独立的“
问：如果内部是非阻塞赋值的话，岂不是内部也可以并行执行么？

作者回复: 嗯



肖水平

2022-08-03 来自上海

//可配置位宽和计数周期数的计数器

```
module counter #(
    parameter          WIDTH = 4, //计数器位宽
    parameter          COUNT = 10 //计数多少次输出周期位
)
(
    input              clk,      //输入时钟
    input              rstn,     //复位端，低有效
    output [WIDTH-1:0] cnt,      //计数输出
    output             cout      //输出周期位
);

reg [WIDTH-1:0]      cnt_r;      //计数器寄存器

always @ (posedge clk or negedge rstn) begin
    if (!rstn) begin           //复位时，计时归0
        cnt_r      <= 4'd0;
    end
    else if (cnt_r == (COUNT - 1)) begin //计时10个cycle时，计时归0
        cnt_r      <= 4'd0;
    end
    else begin
        cnt_r      <= cnt_r + 4'd1; //计时加1
    end
end

assign cout = (cnt_r == (COUNT - 1)); //输出周期位
assign cnt  = cnt_r;                  //输出实时计时器

endmodule
```



Geek_21cfda

2022-08-02 来自湖北

特定的算法需要大量特定的处理器的能力，通过硬件设计可以增加大量的算法所需的能力单元，如GPU就是CPU砍掉大量功能留下大量运算单元，所以GPU特别适合做图形渲染、深度学习模型运算

作者回复: 是的



墨撞的猪

2022-08-02 来自湖北

硬件设计是特定电路实现更符合项目，并且是真正的并行结构，软件是在特定的处理器下进行项目实现，顺序结构。效率远低于直接硬件设计实现。

作者回复: 是的



Leo

2022-08-02 来自湖北

软件执行，数据流和指令流需要在内存——cache——cpu之间流动，需要执行取指，译码，执行，回写等过程，并且还会出现cache miss的情况，增加时间开销。将算法定点化之后不涉及到以上的过程，所以效率大大提升！！

作者回复: 这是原因之一



Bryan

2022-08-02 来自湖北

思考题：盲猜因为软件需要经过操作系统管理的内存，会涉及到虚拟内存寻址等，而硬件则可以省掉这些开销

作者回复: 这是一种原因



wudishi



2022-08-02 来自湖北

目测因为硬件化之后可以利用到电路是天生并行的特性，而不像cpu需要一条条指令往下执行

作者回复: 嗯嗯 对的



砥砺前行

2022-08-02 来自广东

思考题是 类似于 FGPA和ASIC的区别吗 ASIC可以根据算法的逻辑采用最少的电路完成

