Verilog语法基础讲解之参数化设计

在Verilog语法中，可以实现参数化设计。所谓参数化设计，就是在一个功能模块中，对于一个常量，其值在不同的应用场合需要设置为不同的置，则将此值在设计时使用parameter 关键字声明，那么在上层模块例化使用该功能模块时，可以根据具体需求重新配置该常量的值，从而实现不同应用场合对对应常量的灵活调整。

以下为使用Verilog设计的一个控制LED闪烁灯的模块代码：

|  |
| --- |
| 01 **module** counter**(**Clk**,**Rst\_n**,**led**);**  02  03 **input** Clk**;** //系统时钟,50M  04 **input** Rst\_n**;** //全局复位，低电平复位  05  06 **output** **reg** led**;** //led输出  07  08 **reg** **[**24**:**0**]**cnt**;** //定义计数器寄存器  09  10 //计数器计数进程  11 **always@(posedge** Clk **or** **negedge** Rst\_n**)**  12 **if(**Rst\_n **==** 1'b0**)**  13 cnt **<=** 25'd0**;**  14 **else** **if(**cnt **==** 25'd24\_999\_999**)**  15 cnt **<=** 25'd0**;**  16 **else**  17 cnt **<=** cnt **+** 1'b1**;**  18  19 //led输出控制进程  20 **always@(posedge** Clk **or** **negedge** Rst\_n**)**  21 **if(**Rst\_n **==** 1'b0**)**  22 led **<=** 1'b1**;**  23 **else** **if(**cnt **==** 25'd24\_999\_999**)**  24 led **<=** **~**led**;**  25 **else**  26 led **<=** led**;**  27  28 **endmodule** |

在此设计中，定义了一个名为cnt的计数器该cnt在输入时钟”Clk”的驱动下，每个时钟的上升沿执行一次自加操作，当计数到达24999999后，计数器清零，同时led驱动状态发生翻转。由于系统时钟为50M，即周期为20ns，cnt每计数25000000次，即500ms，驱动led状态翻转一次，从而可以实现控制led以1s为周期进行亮灭闪烁。在这个系统中，我们可以看到，如果我们希望将闪烁速度进行更改，例如改为每50ms让led翻转一次，则需要在代码中将”cnt **==** 25'd24\_999\_999”中25'd24\_999\_999这个常量修改为25'd24\_999\_99。

现在考虑另外一种情况，假如某个系统中有三个LED灯需要通过这种方式来进行驱动闪烁，每个LED灯的闪烁频率还不一样。现假设LED1闪烁频率为1s，LED2为0.1s，LED3为0.01秒。那么这三个LED驱动模块中，”25'd24\_999\_999”这个常量就需要分别修改为

LED1：”25'd24\_999\_999”

LED2：”25'd24\_999\_99”

LED3：”25'd24\_999\_9”

因此，必须独立的设计三个这样的模块，然后在上层模块中分别例化这三个模块，以实现需求的功能。

独立设计这样三个模块并分别例化，确实能够完全实现上述假设系统需求的功能，但是通过这种方式设计出来的模块，不具备通用性，如果要求发生变化，则还需要回到底层模块中去修改对应的常量，而底层模块中可能不止一个地方使用到了这个常量，例如上述代码中，第14行和第23行就分别用到了这个常量。如果模块中对此常量的使用次数较多，在修改的过程即增加了工作量，又极容易发生遗漏，从而导致错误。

而参数化设计的使用，则可以非常完美的解决这一问题。这里首先放上该设计使用参数化设计之后的该模块代码：

|  |
| --- |
| 01 **module** counter**(**Clk**,**Rst\_n**,**led**);**  02  03 **input** Clk**;** //系统时钟  04 **input** Rst\_n**;** //全局复位，低电平复位  05  06 **output** **reg** led**;** //led输出  07  08 **reg** **[**24**:**0**]**cnt**;** //定义计数器寄存器  09  10 **parameter** CNT\_MAX **=** 25'd24\_999\_999**;**  11  12 //计数器计数进程  13 **always@(posedge** Clk **or** **negedge** Rst\_n**)**  14 **if(**Rst\_n **==** 1'b0**)**  15 cnt **<=** 25'd0**;**  16 **else** **if(**cnt **==** CNT\_MAX**)**  17 cnt **<=** 25'd0**;**  18 **else**  19 cnt **<=** cnt **+** 1'b1**;**  20  21 //led输出控制进程  22 **always@(posedge** Clk **or** **negedge** Rst\_n**)**  23 **if(**Rst\_n **==** 1'b0**)**  24 led **<=** 1'b1**;**  25 **else** **if(**cnt **==** CNT\_MAX**)**  26 led **<=** **~**led**;**  27 **else**  28 led **<=** led**;**  29  30 **endmodule** |

这里，我们使用parameter在第16行声明了一个参数化常量”CNT\_MAX”，在第16行和第25行进行条件判断时，直接判断cnt的值是否与该参数值相等，而不再是直接写具体数字的方式。到这里大家就可以看到，通过这样一种方式，首先就能避免一个模块中多次使用该常量而对修改设计带来的隐患。因为当我们需要修改该常量的值以适应不同的应用要求时，直接修改这个parameter的值即可，从而可以避免因为该参数使用的地方过多而在修改时容易出现的遗漏的问题。

另外，在我们的系统中，假如也需要三个这样的模块来对应驱动三个LED，大家可能首先想到的就是将该代码复制得到三个文件，并将其中的parameter值修改为对应需要的值即可。此种方式虽然能够避免一个常量在模块中被多次使用，修改时容易发生的遗漏问题，但仍然还是需要存在三个独立的子模块，并未大大减轻设计工作量。其实，修改parameter有更加快捷的方式，这种方式不需要我们针对每一个需求分别复制设计代码并修改对应参数，只需要在上层模块例化该模块时，直接在例化的过程中修改该值即可。具体实现方法，这里以实例代码的形式进行讲解。

在上层模块中，例化子模块时直接修改其参数有两种语法结构，这里我们首先看第一种:

|  |
| --- |
| 01 **module** LED\_flicker**(**  02 Clk**,**  03 Rst\_n**,**  04 LED  05 **);**  06  07 **input** Clk**;**  08 **input** Rst\_n**;**  09 **output** **[**1**:**0**]**LED**;**  10  11 counter counter0**(**  12 **.**Clk**(**Clk**),**  13 **.**Rst\_n**(**Rst\_n**),**  14 **.**led**(**LED**[**0**])**  15 **);**  16  17 counter counter1**(**  18 **.**Clk**(**Clk**),**  19 **.**Rst\_n**(**Rst\_n**),**  20 **.**led**(**LED**[**1**])**  21 **);**  22  23 **defparam** counter0**.**CNT\_MAX **=** 24\_999\_99**;**  24 **defparam** counter1**.**CNT\_MAX **=** 24\_999\_9**;**  25  26 **endmodule** |

这里，在顶层模块中例化了两个counter模块，并分别取名为counter0和counter1。counter0的LED输出驱动LED[0]，counter1的LED输出驱动LED[1]，为了实现LED[0]以0.1秒的频率闪烁，LED[1]以0.01秒的频率闪烁，在第23行和24行分别使用”defparam”来重新定义CNT\_MAX的值。这里以第23行修改counter0中的CNT\_MAX的值来具体讲解此种语法的结构。

首先，使用关键字”defparam”来声明这里需要对某个参数的值进行重新定义，空格之后紧跟着的counter0为counter模块被例化的名字，counter0后紧随一个”.”，”.”后紧随着的就是counter模块中需要修改的参数的名字，然后使用”=”将新的值赋给改参数即可。这里使用模块名+”.”+参数名的方式，与C语言结构体中使用结构体成员的形式很类似，大家可以对比学习。通过这种方式，大家可以在不修改原有设计文件的情况下，例化后，在上层模块直接修改参数。虽然模块中设定了有默认值，但是使用defparam修改的值比原始设计文件中的值拥有更高的编译优先级。当使用defparam修改了原始文件中的参数值后，原始文件中的默认参数值即被忽略。

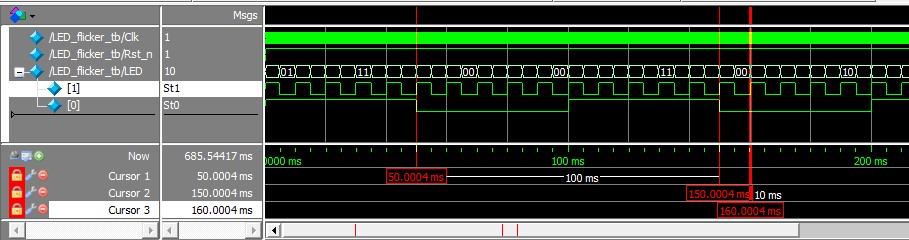
当我们设计完成后，就需要对设计的模块进行仿真，由于仿真执行所需花费的时间远远大于板级系统执行功能时需要的时间，因此仿真时，我们希望对设计模块中某些耗时比较长的设计在不影响理论正确性的前提下进行简单的修改，例如，上述counter模块中，实际仿真如果也按照24999999的计数值进行计数，则需要花费太多的仿真时间，严重影响设计效率，因此也可以在仿真时直接在testbench中使用defparam来对设计文件中的参数进行修改，从而降低仿真复杂度。上述LED\_flicker系统的testbench代码如下所示：

|  |
| --- |
| 01 `timescale 1ns**/**1ps  02 `define clk\_period 20  03  04 **module** LED\_flicker\_tb**;**  05  06 //source define  07 **reg** Clk**;**  08 **reg** Rst\_n**;**  09  10 //probe define  11 **wire** **[**1**:**0**]**LED**;**  12  13 //instant user module  14 LED\_flicker LED\_flicker0**(**  15 **.**Clk**(**Clk**),**  16 **.**Rst\_n**(**Rst\_n**),**  17 **.**LED**(**LED**)**  18 **);**  19  20 **defparam** LED\_flicker0**.**counter0**.**CNT\_MAX **=** 24**;**  21 **defparam** LED\_flicker0**.**counter1**.**CNT\_MAX **=** 24**;**  22  23 //generater clock  24 **initial** Clk **=** 1**;**  25 **always** **#(**`clk\_period**/**2**)**Clk **=** **~**Clk**;**  26  27 **initial** **begin**  28 Rst\_n **=** 1'b0**;**  29 **#(**`clk\_period **\*** 20 **+** 1**);**  30 Rst\_n **=** 1'b1**;**  31 **#(**`clk\_period **\*** 2000**);**  32 $stop**;**  33 **end**  34  35 **endmodule** |

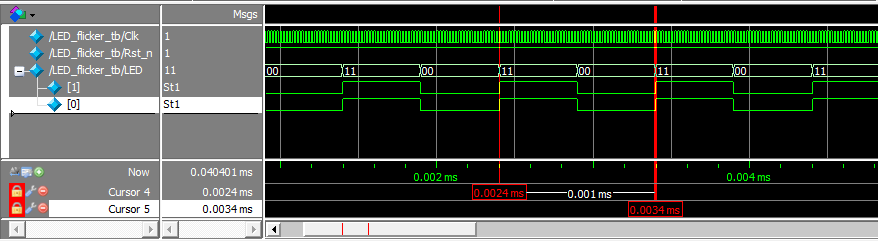
第20行和第21行，使用defparam对设计模块中的CNT\_MAX进行了重新定义，我们希望通过这种方式，来在仿真的时候，让计数值非常小，这样仿真便能很快的得出结果，通过这两句话大家也可以看到，结构体的形式能够实现多级调用。然而，当使用此testbench文件直接仿真LED\_flicker模块时，却并不能实现预期的效果。在modelsim中仿真时，报告如下信息：



虽然并不影响仿真的执行，然而从仿真执行的结果来看，这种在testbench中再次修改设计模块中参数的方式无法与实体模块中例化子模块后进行参数定义同时存在，将实体模块中使用defparam修改参数的部分屏蔽，则仿真文件中的defparam生效，实现了预期效果。下图分别为不屏蔽LED\_flicker中defparam内容和屏蔽LED\_flicker中defparam内容后的仿真结果，从图中可知，不屏蔽，LED[0]和LED[1]的翻转频率分别为100ms和10ms，结果与LED\_flicker中defparam定义的值一致，屏蔽掉后，LED[0]和LED[1]的翻转频率都是0.001ms，仿真结果与testench文件LED\_flicker\_tb中defparam定义的值一致。



不屏蔽LED\_flicker中defparam内容时仿真结果



屏蔽LED\_flicker中defparam内容时仿真结果

在实际使用中，我们往往希望实体设计模块中能够设定某参数值为符合实际板级运行要求的值，而在仿真中，为了节约仿真时间，我们又希望修改该参数值以配合仿真通过上面的例子我们发现，使用defparam语句难以同时满足实体设计和仿真的需求。那么有没有更好的方法来实现实体设计与仿真验证中对参数的修改能够共存且互不干扰呢？

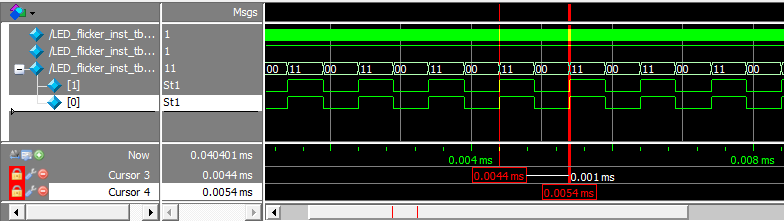
为了实现此功能，我们调整在实体模块中例化子模块时对参数的修改方式，上面我们介绍的是使用defparam语句重新定义子模块参数内容。接下来我们使用在模块例化时，直接对参数进行例化的方式来修改子模块中的参数值。使用参数例化方式修改参数值的LED闪烁灯模块代码如下所示：

|  |
| --- |
| 01 **module** LED\_flicker\_inst**(**  02 Clk**,**  03 Rst\_n**,**  04 LED  05 **);**  06  07 **input** Clk**;**  08 **input** Rst\_n**;**  09 **output** **[**1**:**0**]**LED**;**  10  11 counter  12 **#(**  13 **.**CNT\_MAX**(**24\_999\_99**)**  14 **)**  15 counter0**(**  16 **.**Clk**(**Clk**),**  17 **.**Rst\_n**(**Rst\_n**),**  18 **.**led**(**LED**[**0**])**  19 **);**  20  21 counter  22 **#(**  23 **.**CNT\_MAX**(**24\_999\_9**)**  24 **)**  25 counter1**(**  26 **.**Clk**(**Clk**),**  27 **.**Rst\_n**(**Rst\_n**),**  28 **.**led**(**LED**[**1**])**  29 **);**  30  31 **endmodule** |

这里，第11行到19行为例化counter0模块的代码。这里对于子模块中的参数，使用了和端口相类似的形式进行修改。通过这种方式，在例化每个功能模块的时候，直接将所需修改参数的值也同时通过例化的方式修改了。

下面对这种例化方式的代码结构进行简单介绍:

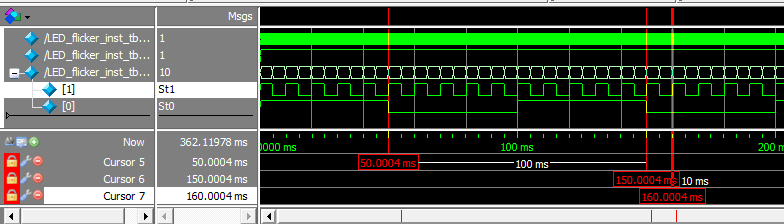
首先，第11行为原始设计模块的名字，第12行和14行为一对小括号，对参数例化的内容就在此括号中进行。注意11号，即括号前面需要使用”#”符号来声明这是对参数进行例化。括号内，参数例化形式与端口例化形式一致，使用”.”+原始参数名+新值的方式来修改新的值。通过这种方式，能够避开对于同一个参数多次使用defparam来重定义值而引发的冲突。使用上述testench文件仿真LED\_flicker\_inst模块的仿真结果如下图所示：



使用defparam重新修改CNT\_MAX值的仿真结果

可以看到，LED[0]和LED[1]的翻转频率都是0.001ms，为通过testbench修改后的结果。

为了验证使用这种例化的方式是否能够达到预期的效果，这里将testbench中defparam内容屏蔽，再来观察仿真结果即可，屏蔽掉testbench文件中defparam内容后仿真结果如下图所示：



屏蔽testbench文件中defparam内容后仿真

可以看到，屏蔽掉testbench中队CNT\_MAX值的修改内容后，LED[0]和LED[1]的翻转频率分别为100ms和10ms，仿真结果就与实际设计值一致。但是在我们实际板级使用的时候，testbench中的值并不会对实际设计内容产生影响，因此，即使不屏蔽testbench中对参数值的修改，也不会影响板级设计结果。

因此我们可以总结得出，为了同时保证实际逻辑设计和仿真验证时对参数的修改能够共存，互不影响，在实体模块设计中，使用参数例化的方式修改被例化模块中的参数。在仿真验证用的testbench中，使用defparam语句来修改对应的参数。

通过本教程内容，讲解了在Verilog中使用参数化设计的方法，展示了参数化设计在实际使用时候的巨大便利。介绍了两种修改参数值的方法，并通过实际的例子展示了两种修改方式的特点，并最终给出了能够同时兼容实际逻辑设计和仿真验证的参数修改方法。希望大家以后。在设计自己的代码的时候，多使用这种参数化的设计方式。

如有更多问题，欢迎加入芯航线FPGA技术支持群：472607506

小梅哥

2015年11月7日于芯航线电子工作室