[Verilog中的参数化建模](http://guqian110.github.io/pages/2014/07/09/parameterization_modeling_in_veriog.html" \o "Permalink to Verilog 中的参数化建模)

**Date**  Wed 09 July 2014 **Tags** [parameterization](http://guqian110.github.io/tag/parameterization.html)

<http://guqian110.github.io/pages/2014/07/09/parameterization_modeling_in_veriog.html>

和写软件程序一样，我们也希望 Verilog 的模块也可以重利用。要使模块可以重复利用，关键就在于避免硬编码(hard literal)，使模块参数化。

参数化建模的好处是可以使代码清晰，便于后续维护和修改。

Verilog 的参数化建模是有一定限制的，它的参数值是编译时计算的，不会引入任何实际的硬件电路。参数必须在编译时确定值。也就是说只能达到动态编译，固态运行，而非软件的动态编译，动态运行。

这主要是因为它是描述(Description)硬件的语言，而非软件设计(Design)语言。

比如一个计数器，我们可以设置一个参数来指定它的计数周期(动态编译)，但是这个计数周期在综合之后就是固定值了(固态运行)，不能在运行的时候动态地改为另外一个值(除非电路综合时同时产生了多个计数器，这种情况不算真正意义上的动态运行，而且也达不到真正意义上的动态运行，因为不可能把所有可能的计数器都实现了备用，耗费资源而且没有实际意义)。

参数化建模的主要目的是：

**提高模块的通用性，只需要修改参数，不用修改其他代码就可以适用于不同的环境中。**

总结一下我找到的资料，具体的参数化建模方法一共就 3 种：

1. `define 宏定义
2. parameter 模块参数化
3. `ifdef 等 条件编译

下面详细说明

Define Macro Substitution

`define 是编译器指令，功能是全局宏定义的文本代替。它类似于 C 语言中的 #define，用法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | // define  `define WORD\_REG reg [31:0]  // using  `WORD\_REG reg32; |

**Problem**

`define 定义的宏的作用域是全局的，这种机制会导致两个问题

1. 可能会有在不同文件中发生重定义的问题
2. 编译顺序有要求 file-order dependent，必须确保使用前，宏定义有效，所以每个使用到宏定义的源文件必须包含这个头文件，这会导致多重包含的问题。

**Solution**

1. 对于第一个问题，尽可能把所有的宏定义放在同一个头文件中，比如 "global\_define.vh"
2. 对于第二个问题，和 C++ 类似，头文件应该使用头文件保护符。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // global\_define.vh head file  `ifndef GLOBAL\_DEFINE\_VH  `define MAX = 8  `define SIZE = 4  // ...  `enif |

Guideline

1. 只有那些要求有全局作用域、并且在其他地方不会被修改的常量才用 define 来定义
2. 对于那些只限于模块内的常量，不要使用 define
3. 尽可能将所有的 define 都放在同一个文件中，然后在编译时先读取这个文件
4. 不要使用 `undef

Parameter

Parameter

应该避免硬编码设计 hard literal，使用参数 parameter 来代替。举个例子

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // use parameter  parameter SIZE = 8,  MAX = 10;  reg [SIZE - 1 : 0] din\_r;  // DO NOT use hard literal  reg [7 : 0] din\_r; |

Localparam

Verilog-2001 中添加了一个新的关键字 localparam，用来定义模块内部的、不能被其他模块修改的局部常量，概念类似于 C++ 中 class 的 protect 成员。

虽然 localparam 不能被外部模块修改，但是它可以用 parameter 来初始化。

|  |  |
| --- | --- |
| 1  2 | parameter N = 8;  localparam N1 = N - 1; |

Parameter Redefinition

在 Verilog-2001 出现之前，Verilog-1995 中只有两种方法实现参数重定义：

1. 使用 # 符号，顺序列表重定义
2. 使用 defparam

逐个讨论

1. Uisng #

**Syntax**

举个栗子，模块 myreg

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | module myreg (q, d, clk, rst);  parameter Trst = 1,  Tclk = 1,  SIZE = 4;  // ...  endmodule |

在上一层的模块中传递参数例化这个模块

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | module bad\_warpper (q, d, clk, rst)  // legal parameter passing  myreg #(1, 1, 8) r1(.q(q), .d(d), .clk(clk), .rst(rst) );  // illegal parameter passing  // myreg #(,,8) r1(.q(q), .d(d), .clk(clk), .rst(rst) );  endmodule |

**Pro**

虽然每次例化都要说明所有的参数值，但是比第二种方法好

**Con**

每次例化都要说明所有的参数值。

2. Using defparam

**Syntax**

|  |  |
| --- | --- |
| 1 | defparam path.name = value; |

比如在上面的例子中

|  |  |
| --- | --- |
| 1 | defparam r1.SIZE = 8; |

**Pro**

可以放在任何文件的任何地方，不用再重复没有修改的参数值

**Con**

因为 defparam 有这么 "强" 的功能，反而会导致一系列的问题

1. Hierarchical deparam.

比如顶层模块使用 defparam 修改子模块的参数，子模块中又使用 defparam 修改顶层模块要传递进来的参数，形成一个环，这样子可能导致综合时不提示错误，但是结果与预期不符。

1. Multiple defparams

在 单个文件 / 多个文件 中重复定义 defparam，会有微妙的问题，Verilog-1995 中没有定义这种现象，实际结果依赖于使用的综合工具。

因为 defparam 有这么多缺点，所以在 2001 年之前，Synopsys 是不支持 defparam 的，网上很多转载的博客都说 defparam 是不可综合的，实际上在后来，Synopsys 在压力之下添加了对其的支持。而我用 XST 也证明是支持 defparam 可综合。

综上原因，Verilog Standards Group (VSG) 倡议大家抵制使用 defparam，大神 Clifford E. Cummings 在论文中建议综合工具如果用户坚持使用 defparam 语句，必须添加以一个参数 +Iamstupid...

"The Verilog compiler found a defparam statement in the source code at (file\_name/line#). To use defparam statements in the Verilog source code, you must include the switch +Iamstupid on the command line which will degrade compiler performance and introduce potential problems but is bug-compatible with Verilog-1995 implementations. Defparam statements can be replaced with named parameter redefinition as define by the IEEE Verilog-2001 standard."

总结*一下，可以发现 Verilog-1995 中的两种方法都不怎么好，显然 VSG 也发现了这个问题，所以在 Verilog-2001 中，出现了第三种方法，并且墙裂推荐使用这种新方法。*

3. Using named parameter redefinition

**Syntax**

类似于模块例化时端口连接的方式，比如上例中只想改变 SIZE 的值

|  |  |
| --- | --- |
| 1 | myreg #(.SIZE(8)) r1(.q(q), .d(d), .clk(clk), .rst(rst) ); |

**Pro**

结合了前两种方法的有点，既显示说明了哪个参数值改变了，也将参数传递放在了实例化的语句中。这种方法是最干净的 (cleanest) 方法，不依赖于任何综合工具。

**Con**

貌似没有～

Guideline

1. 不要使用 defparam，应该使用 named parameter redefinition。

Example

1. clock cycle definition

因为时钟是一个设计中最基本的常量，它不会在随着模块变化，所以应该用 `define 来定义，并且将它放在顶层的头文件中。

1. FSM

在一个设计中可能有不止一个 FSM，而通常 FSM 有一些共同的状态名字，比如 IDLE、READY、READ、WRITE、ERROR、DONE 等，所以应该用 localparam 来定义这些常量。

Conditional Compilation

Verilog 的条件编译和 C 也十分类似。前面介绍 define 时，已经用到了条件编译中的 `ifdef。条件编译一共有 5 个关键字，分别是：

`ifdef `else `elsif `endif `ifndef

条件编译一般在以下情况中使用

1. 选择一个模块的不同部分
2. 选择不同的时序和结构
3. 选择不同的仿真激励

**Syntax**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // example1  `ifdef text\_macro  // do something  `endif  // example2  `ifdef text\_macro  // do something  `else  // do something  `endif  // example3  `ifdef text\_macro  // do something  `elsif  // do something  `else  // do something  `endif  // example4  `ifndef text\_macro  // do something  `else  // do something  `endif |

条件编译是一个非常好的技术，它可以帮助我们更好的管理代码。

举个栗子，比如我们写了一个程序，在 debug 阶段，在程序中添加了很多显示中间变量的语句，到最后 release 时，当然要去掉这些语句。最差的方法当然是删掉这些代码，但是如果以后我们还想 debug 时，又得手动写，而且时间长了，我们自己都记不清该加哪些语句了。稍微好点的方法是把它们注释起来，但是同样，时间长了，哪些该注释，那些不该注释又混淆了。最好的方法就是用条件编译。我们可以定义一个宏 DEBUG

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | `define DEBUG  // conditional compilation  `ifdef DEBUG  // debug  `else  // release  `endif |

这样，我们只需要选择是否注释第一行的宏定义就可快速在 debug 和 release 之间切换。

再比如在 Verilog 的模块中，针对不同的应用环境，我们要实现不同的模块，这时候也可以使用条件编译选择具体综合哪段代码。

Summary

总结一下，就是一下几点

**Guideline**

1. 只有那些要求有全局作用域、并且在其他地方不会被修改的常量才用 define 来定义
2. 对于那些只限于模块内的常量，不要使用 define
3. 尽可能将所有的 define 都放在同一个文件中，然后在编译时先读取这个文件
4. 不要使用 `undef
5. 不要使用 defparam，应该使用 named parameter redefinition。
6. 需要时使用条件编译

Reference

IEEE Std 1364-1995

IEEE Std 1364-2001

[New Verilog-2001 Techniques for Creating Parameterized Models](http://www.sunburst-design.com/papers/CummingsHDLCON2002_Parameters_rev1_2.pdf)

[(原创) 如何使用参数式模组? (SOC) (Verilog) (C/C++) (template)](http://www.cnblogs.com/oomusou/archive/2008/07/09/verilog_parameter.html)

[艾米电子 - 参数与常量，Verilog](http://blog.chinaaet.com/detail/14875)

[Verilog代码可移植性设计](http://www.eefocus.com/ilove314/blog/2012-03/231583_52a1d.html)