[**SIEMENS**](https://verificationacademy.com/)西门子

[AHomeC](https://verificationacademy.com/)[Verification Methodology CookbooksC](https://verificationacademy.com/cookbook)[UVM](https://verificationacademy.com/cookbook/uvm-universal-verification-methodology)

SystemVerilog性能指南

这些指南旨在使您能够识别可能影响测试台性能的编码习惯用法

[6](https://verificationacademy.com/topic/11e76ebf-1af0-3396-9193-00c43f615912/toggle-marked?page_path=cookbook/uvm-universal-verification-methodology/systemverilog-performance-guidelines)

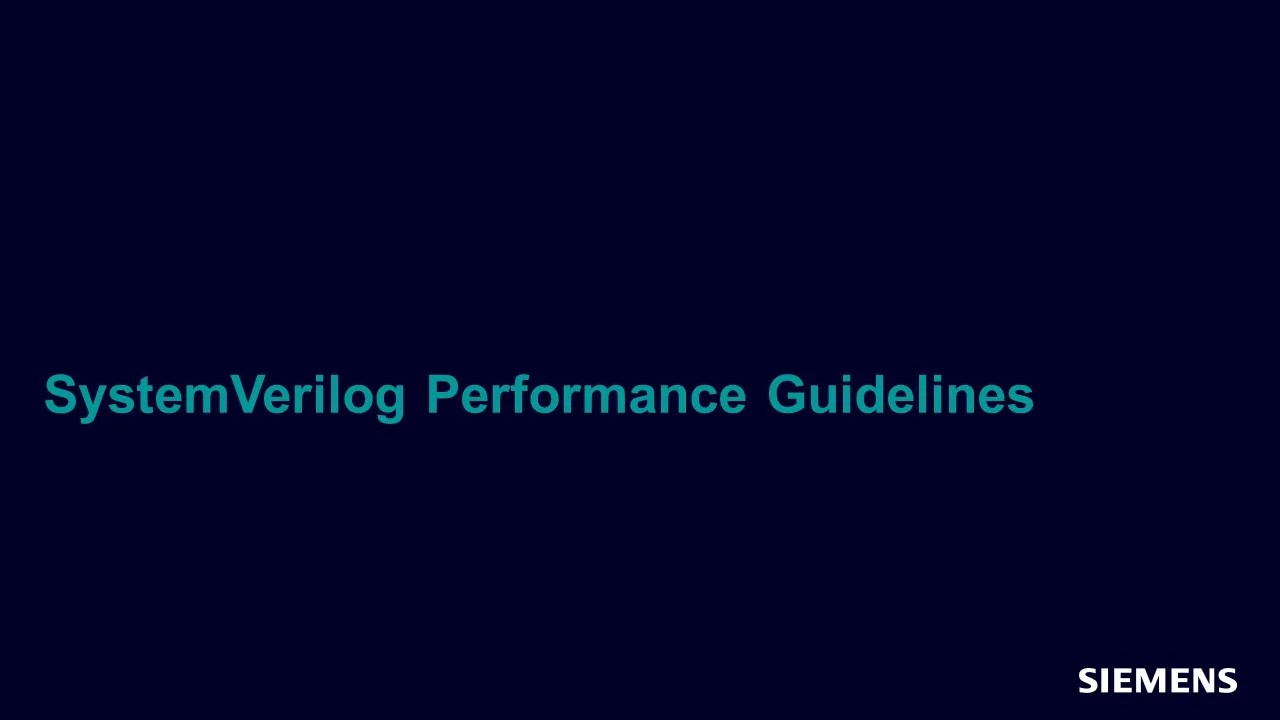
[UVM- 通用验证方法](https://verificationacademy.com/topics/uvm-universal-verification-methodology/)

**核查方法小组**

最**后更新于2014年3月**

O[**UVM**](https://verificationacademy.com/all-content?tag=UVM)**，**[**SystemVerilog**](https://verificationacademy.com/all-content?tag=SystemVerilog)**，**[**附录**](https://verificationacademy.com/all-content?tag=Appendix)**，**[**指南**](https://verificationacademy.com/all-content?tag=Guidelines)**，**[**性能**](https://verificationacademy.com/all-content?tag=Performance)**，**[**初学者**](https://verificationacademy.com/all-content?audience=Beginner)

[Q**标记查看**](https://verificationacademy.com/748ab351-eaaa-3238-b970-133fb7721c83/toggle-marked?page_path=cookbook/uvm-universal-verification-methodology/systemverilog-performance-guidelines)



请注意，这些指导原则中有许多与其他推荐的编码背道而驰

对业绩和方法需要之间的权衡的平衡看法

请练习

要做的。

虽然突出显示的一些代码结构可能会被编译器识别和优化，但由于支持调试、与PLI代码交互等的副作用，情况可能并不总是如此。因此，按照建议的方式重构代码几乎总是有好处的

SystemVerilog与C、C++和Java等主流软件语言有许多共同的特征，这里介绍的一些指导原则也与这些语言相关然而，SystemVerilog有一些独特的功能和短的时间，这可能会导致粗心的用户创建低性能和内存饥饿的代码没有意识到这一点。

使用代码分析工具可以使测试平台的性能调优变得更加容易代码概要可以识别代码中的“热点”，如果这些地方可以重构，那么测试平台几乎总是会得到改进。在没有分析工具的情况下，需要进行可视化代码检查，但这需要时间和精力。 这些指导原则旨在用于编码开始之前，以及根据代码分析或手动检查来审查代码。

代码分析

代码分析是一种自动化技术，可以在模拟运行期间使用，以使您了解测试平台代码中的“热点”位置。运行代码配置文件是一个运行时选项，如果可用，将记录在模拟器用户指南中。有关详细信息，请参阅Questa用户指南中的“性能分析和内存使用”一章

当您的测试平台代码达到了一个合理的成熟状态，并且您能够可靠地运行测试用例时，那么运行分析工具总是值得的。大多数代码分析器都是基于采样的;它们定期记录在给定的时间点哪些代码行是活动的，哪些过程调用正在进行 为了获得有统计意义的结果，它们需要运行足够长的时间来收集代码活动的代表性样本。

在没有性能问题的良好编写的测试平台中，采样的结果将是测试平台代码中的平坦分布。然而，如果分析显示测试平台的特定区域出现在不成比例的样本中，那么它通常指向该代码的潜在问题

分析是一种分析技术，结果将受到以下因素的影响：

正在分析的测试用例的特征

随机播种-导致测试台中不同级别的活动

测试工作台中的主导行为-测试工作台的某些区域可能只是做了

DUT编码样式

采样间隔

运行配置文件的模拟时间长度

在运行配置文件时，模拟中发生了什么

使用受约束的随机测试台，在分析报告时，总是值得运行具有不同种子的替代测试用例，因为这些可能会揭示不同的编码问题。

循环指南

环路性能取决于：

循环中的工作

在循环中进行的检查，以确定它是否应该处于活动状态

在循环中进行的工作应该保持在最低限度，并且对循环边界进行的检查应该具有最小的开销。下面是一些好的和坏的循环实践的例子

较低性能版本

// dynamic array, unknown size

int array[];

int total=0;

for(int i=0; i< array.size(); i++)begin

total+= array[i];

end

### 更高性能版本

// dynamic array, unknown size

int array[];

int array\_size;

int total=0;

array\_size= array.size();

for(int i=0; i< array\_size; i++)begin

total+= array[i];

end

在循环开始之前将变量设置为数组的大小，可以节省每次迭代时计算array.size（）的开销

较低性能版本

int decision\_weights[string];// Assoc

int case\_exponents[string];

int total\_weights;

foreach(decision\_weights[i])begin

total\_weights+= decision\_weights[i]\*

case\_exponents["high"];

end

更高性能版本

int decision\_weights[string];// Assoc

int case\_exponents[string];

int total\_weights;

int case\_exponent;

case\_exp= case\_exponents["high"]

foreach(decision\_weights[i])begin

total\_weights+= decision\_weights[i]\*

case\_exp;

end

对于较小的数组，foreach（）循环构造通常比for（inti=0;i<val>;i++）具有更高的性能

在每次循环迭代中查找关联数组中的指数值是不必要的，因为它可以在循环开始时查找

较低性能版本

int an\_array[50];

int indirection\_index;

int to\_find=42;

indirection\_index=-1;

// Look up an index via the array:

foreach(an\_array[i])begin

if(an\_array[i]== to\_find)begin

indirection\_index= i;

end

end

更高性能版本

int an\_array[50];

int indirection\_index;

int to\_find=42;

indirection\_index=-1;

// Look up an index via the array:

foreach(an\_array[i])begin

if(an\_array[i]== to\_find)begin

indirection\_index= i;

break;//break跳出

end

end

在本例中，在循环中搜索具有唯一条目的数组以获取给定值。在第二个示例中使用break会在找到匹配项后立即终止循环的计算

决策指南

当在逻辑或算术基础上做出决策时，有许多优化可以帮助提高性能：

短路逻辑表达式

一旦发现短路逻辑表达式的一个元素为假，就放弃对该表达式的求值使用短路逻辑表达有可能加快决策速度。

如果没有必要，对短路表达式中的项进行排序也可以避免昂贵的调用

一些例子：

AND求值，如果表达式的第一项为true，则跳过其余的

与

if(A && B && C)begin

// do something

end

于OR求值，如果表达式的第一项为true，则跳过其余求值

if(A || B || C)begin

// do something

End

如果表达式中的项具有不同的“费用”水平，则应排序这些项以首先计算最便宜的项：

较低性能版本

if(B.size() > 0) begin

if(B[$] == 42) begin

if(A) begin

// do something

end

end

end

更高性能版本

if(A && (B.size() > 0) && B[$] == 42) begin

// do something

end

如果廉价表达式A的结果为true，则不需要进行其他昂贵的条件测试

较低性能版本

if((A||B) && C)begin

// do something

End

更高性能版本

if(C && (A||B)) begin

// do something

end

一个稍微不太明显的变体，它节省了在C不为真时做出决定所需的计算。

重构逻辑决策逻辑

有时候，结合一些简单的布尔代数可以减少计算量。

较低性能版本

if((A && C) || (A && D)) begin

// do something

End

更高性能版本

if(A && (C || D)) begin

// do something

End

在上面的示例中，重构布尔条件删除了一个逻辑运算，使用A作为短路可能会减少活动决策逻辑

重构算术决策逻辑

记住重构算术项也可以导致优化。这不仅适用于决策逻辑，也适用于计算变量。

较低性能版本

if(((A\*B) - (A\*C))> E) begin

// do something

End

更高性能版本

if((A\*(B - C)) > E) begin

// do something

end

在上面的例子中，重构避免了乘法运算。

## 优先编码

如果您知道决策树中条件的相对频率，请将最频繁出现的条件移到树的顶部。这最常应用于case语句和嵌套if。

较低性能版本

// Case options follow the natural order:

case(char\_state)

START\_BIT: // do\_something to start tracking the char (once per word)

TRANS\_BIT: // do something to follow the char bit value (many times per word)

PARITY\_BIT: // Check parity (once per word, optional)

STOP\_BIT: // Check stop bit (once per word)

Endcase

更高性能版本

// case options follow order of likely occurrence:

case(char\_state)

TRANS\_BIT: // do something to follow the char bit value (many times per word)

START\_BIT: // do\_something to start tracking the char (once per word)

STOP\_BIT: // Check stop bit (once per word)

PARITY\_BIT: // Check parity (once per word, optional)

Endcase

大多数情况下，case语句在一次检查后退出，以保存进一步的比较。

较低性能版本

// ready is not valid most of the time

// read cycles predominate

//

if(write\_cycle) begin

if(addr inside {[2000:10000]}) begin

if(ready) begin

// do something

end

end

end

else if(read\_cycle) begin

if(ready) begin

// do something

end

end

更高性能版本

// ready is not valid most of the time

// read cycles predominate

//

if(ready) begin

if(read\_cycle) begin

// do something

end

else begin

if(addr inside {[2000:10000]}) begin

// do something

end

end

end

在这个例子的高性能版本中，如果ready无效，代码的其余部分就不会被计算。然后进行read\_cycle检查，这消除了对write\_cycle检查的需要

任务和函数调用指南

内联代码

在某些情况下，重构调用子例程方法的代码可能比使用子例程更好，这样方法的内容就可以展开并放在行内。如果子例程相对较短且具有多个参数，则尤其如此

任务和函数调用参数传递

在SystemVerilog中，将参数传递到任务和函数调用/从任务和函数调用传递参数是通过在任务或函数调用开始时复制变量，然后复制回方法执行期间所做的任何更改的结果来完成的 如果参数是复杂的变量类型（如字符串或数组），则这可能会成为相当大的开销，而替代方法是使用引用。

使用引用节省了复制的开销，但这确实意味着，由于变量在任务或函数中更新时不会复制到函数中，因此它也会在调用方法中更新避免此问题的一种方法是使变量成为const ref，从函数的角度来看，这有效地使其成为只读引用

### 较低性能版本

function void do\_it(input int q[$], input string name);

int m\_i;

string m\_s;

m\_s= name;

m\_i= q.pop\_front();

$display("string = %s, value = %0d", m\_s, m\_i);

q.push\_front(m\_i);

endfunction: do\_it

更高性能版本

function automatic void do\_it(ref int q[$], ref string name);

int m\_i;

string m\_s;

m\_s= name;

m\_i= q.pop\_front();

$display("string = %s, value = %0d", m\_s, m\_i);

q.push\_front(m\_i);

endfunction: do\_it

在较低性能版本的代码中，一个int队列和一个字符串被复制到函数中随着队列长度的增长，这变得越来越昂贵。在更高性能的版本中，int队列和字符串参数都是引用，这避免了复制操作并加快了函数的执行速度

课堂表现指引

在SystemVerilog中，类封装数据变量和对这些变量进行操作的方法。一个类可以被扩展以添加更多的变量，并添加或扩展现有的方法以提供新的功能。

所有这些便利性和功能都伴随着性能开销，可以通过以下准则最大限度地

避免不必要的对象构造

构造一个对象可能会有一个与之相关的开销。一般来说，尽量减少创建的对象数量

较低性能版本

// Function that returns an object handle

function bus\_object get\_next(bus\_state\_t bus\_state);

bus\_object bus\_txn = new();

if(bus\_state.status == active) begin

bus\_txn.addr = bus\_state.addr;

bus\_txn.opcode = bus\_state.opcode;

bus\_txn.data = bus\_state.data;

return bus\_txn;

end

return null;

endfunction: get\_next

更高性能版本

//

//

// Function that returns an object handle

//

function bus\_object get\_next(bus\_state\_t bus\_state);

bus\_object bus\_txn;

// Only construct the bus\_txn object if necessary:

if(bus\_state.status == active) begin

bus\_txn = new();

bus\_txn.addr = bus\_state.addr;

bus\_txn.opcode = bus\_state.opcode;

bus\_txn.data = bus\_state.data;

end

return bus\_txn;// Null handle if not active

endfunction: get\_next

没有必要构造总线事务对象，如果不构造，函数将返回空句柄

较低性能版本

taskhandle\_bus\_write; bus\_object write\_req=

bus\_object：：type\_id：：create（“write\_req”）;

write\_bus\_req\_fifo.get（write\_req）;

//do something with the write\_req

endtask：handle\_bus\_write

更高性能版本

task handle\_bus\_write;

bus\_object write\_req=

bus\_object::type\_id::create("write\_req");

write\_bus\_req\_fifo.get(write\_req);

// do something with the write\_req;

endtask: handle\_bus\_write

构造write\_req对象是多余的，因为它的句柄由bus\_write\_req\_fifo的get重新分配

直接变量赋值比set（）/get（）方法更快

调用一个方法来更新或检查一个变量比通过类层次结构路径直接访问要带来更高的开销

较低性能版本

// Class with access methods

class any\_thing;

int A;

functionvoid set\_A(int value);

A= value;

endfunction: set\_A

function int get\_A();

return A;

endfunction: get\_A

endclass: any\_thing

// code that uses this class

// and its access methods

any\_thing m;

int V;

initial begin

m=new();

V=1;

repeat(10) begin

m.set\_A(V);

V= V+ m.get\_A();

end

end

更高性能版本

// Class with access methods

class any\_thing;

int A;

function void set\_A(int value);

A= value;

endfunction: set\_A

function int get\_A();

return A;

endfunction: get\_A

endclass: any\_thing

// code that uses this class

// and makes direct assignments

any\_thing m;

int V;

initial begin

m=new();

V=1;

repeat(10) begin

m.A= V; //直接赋值

V= V+ m.A; //直接赋值

end

end

在类中使用数据变量的层次路径对其进行赋值比调用方法set（）/get（）更有效。但是，如果set（）/get（）方法不仅仅是一个简单的赋值，例如：类型转换或对提供的参数进行检查操作

**请注意：**这条准则是为了性能而制定的，它违反了普通的OOP准则，即类中的数据变量只能通过方法访问 使用直接访问方法来访问变量可以提高性能，但可能会使代码的可重用性降低，并且依赖于用户知道变量的名称和类型的假设。

避免方法链

在类中调用方法会带来开销，嵌套或链接方法调用会增加开销。当你实现或扩展一个类时，尽量减少涉及的方法的

较低性能版本

class mailbox\_e#(type T=int);

localmailbox#(T) mb;

// standard mailbox API

extern function new(int bound=0);

extern function int num();

extern task put(T item);

extern function int try\_put(T item);

extern task get(ref T item);

extern function int try\_get(ref T item);

extern function int try\_peek(ref T item);

// extended API

extern function void reset();

endclass: mailbox\_e

function mailbox\_e::new(int bound=0);

mb = new(bound);

endfunction

function int mailbox\_e::num();

return mb.num();

endfunction: num

task mailbox\_e::put(T item);

mb.put(item);

endtask: put

function int mailbox\_e::try\_put(T item);

return mb.try\_put(item);

endfunction: try\_put

task mailbox\_e::get(ref T item);

mb.get(item);

endtask:get

function int mailbox\_e::try\_get(ref T item);

return mb.try\_get(item);

endfunction: try\_get

function int mailbox\_e::try\_peek(ref T item);

return mb.try\_peek(item);

endfunction: try\_peek

function void mailbox\_e::reset();

T obj;

while (mb.try\_get(obj));

endfunction: reset

更高性能版本

class mailbox\_e # (type T = integer)

extends mailbox #(T);

extern function new(int bound=0);

// Flushes the mailbox:

extern function void reset();

endclass: mailbox\_e

function mb\_e::new(int bound=0);

super.new(bound);

endfunction

function void mb\_e::reset();

T obj;

while(try\_get(obj));

endfunction: reset

第二个实现直接扩展邮箱，避免了第一个示例中的额外层。

较低性能版本

class multi\_method;

int i;

function void m1();

m2();

endfunction: m1

function void m2();

m3();

endfunction: m2

function void m3();

i++;

endfunction: m3

endclass: multi\_method

更高性能版本

class multi\_method;

int i;

function void m1();

i++;

endfunction: m1

endclass: multi\_method

在第一个例子中，函数调用被实现为一个链，而第二个例子有一个单一的方法，将具有更高的性能。你的代码可能更复杂，但它可能有你可以展开的方法调用链

阵列指南

SystemVerilog有许多具有不同特性的数组类型，值得考虑哪种类型的数组最适合手头的任务下表

列出了考虑因素。

阵列类型 特征内存影响 性能影响

inta\_ray[7：0];

数组大小在编译时固定。索引是整数。

最少 数组索引是高效的。 搜索一个大的数组有一个开销。

int a\_ray[];

在模拟过程中确定/更改阵列大小。

索引是整数。

更少 的数组索引效率更高。管理规模很重要。

inta\_q[$];

使用模型为FIFO/LIFO类型存储自管理大小调整使用访问方法推送和弹出数据

更 高效的有序访问自管理调整最大限度地减少了性能影响

int a\_ray[string];

索引按定义的类型，而不是整数有方法帮助管理在编译时调整大小或不调整大小，随着使用而增长

稀疏存储或随机访问效率更高随着数据量的增加，效率会变得更低，但可以删除元素非整数索引可以提高抽象性

Messa

例如，使用关联数组而不是使用静态数组来对仅具有稀疏条目的大型存储器空间进行建模可能更有效。然而，如果关联数组由于条目的数量而变得很大，那么使用固定数组来对内存空间进行建模将变得更有效

使用关联数组默认值

在关联数组的某些应用中，可能会使用尚未添加到数组的索引进行访问，例如记分板稀疏存储器或访问项目的标签

当一个关联数组得到一个超出范围的访问时，默认情况下它会返回一个警告ge和一个未初始化的值。

为了避免这种情况，可以查询数组以确定索引是否存在，如果不存在，则不进行访问。如果使用默认变量语法，则可以避免此工作并提高性能：

阿尔瓦

较低性能版本

// Associative array declaration - no default value:

int aa[int];

if(aa.exists(idx)) begin

lookup = aa[idx];

end

更高性能版本

// Associative array declaration - setting the default to 0

int aa[int] = {default:0};

lookup = aa[idx];

避免工作

这里的基本原则是避免做某事，除非你不得不做。

这可以通过各种方式表现出来

不要随机化一个对象除非你需要不要构造一个对象除非你需

一旦你找到了你要找的东西，

最大限度地减少模拟中的字符串处理量-在UVM测试台中，这意味着使用`uvm\_info（）、`uvm\_warning（）、`uvm\_error（）、`uvm\_fatal（）宏来避免字符串操作，除非已激活适当级别的详细信息

约束性能指南

约束随机生成是SystemVerilog中最强大的功能之一然而，过度限制刺激的产生是非常容易的。在编写约束时，稍微考虑和计划一下，就可以使测试平台的性能有很大的不同

在类中编写受约束的随机代码时，应考虑以下因素

1. 最小化活动rand变量的数量-如果一个值可以从其他随机字段计算出来
2. 使用最少的数据类型-即位而不是逻辑，将向量宽度调整到所需的
3. 使用分层的类结构来打破随机化，使用短路决策树来最小化工作
4. 使用延迟随机化以避免不必要的随机化
5. 检查内联约束的重复使用-扩展类可能更有效
6. 避免在约束中使用算术运算符，尤其是 \*、/、%运算符
7. 蕴涵运算符是双向的，使用solvebefore强制执行前面项的概率
8. 使用pre-randomize（）方法预先设置或预先计算随机化过程中使用的状态变量
9. 使用post-randomize（）方法计算依赖于随机变量的变量值
10. 是否有一种替代的方式来编写约束，这意味着它不那么复杂？

说明这些要点的最佳方法是通过一个示例-请注意，上面的一些编号点在代码中作为注释引用

较低性能版本

class video\_frame\_item extends uvm\_sequence\_item;

typedef enum {live, freeze} live\_freeze\_t; // 2

typedef enum {MONO, YCbCr, RGB} video\_mode\_e; // 3

// Frame Packets will either be regenerated or repeated

// in the case of freeze.

rand live\_freeze\_t live\_freeze = live; // 1

int x\_pixels;

int y\_pixels;

rand int length; // 1

video\_mode\_e mode;

rand int data\_array []; // 2

// Constraints setting the data values

constraint YCbCr\_inside\_c {

foreach (data\_array[i]) data\_array[i] inside {[16:236]};

}

constraint RGB\_inside\_c {

foreach (data\_array[i]) data\_array[i] inside {[0:255]};

} c

onstraint MONO\_inside\_c {

foreach (data\_array[i]) data\_array[i] inside {[0:4095]};

} /

/ Constraints setting the size of the array

constraint YCbCr\_size\_c{

data\_array.size == (2\*length); // 6

}

constraint RGB\_size\_c{

data\_array.size == (3\*length); // 6

}

constraint MONO\_size\_c{

data\_array.size = =(length); // 6

} /

/ Frequency of live/freeze frames:

constraint live\_freeze\_dist\_c {

live\_freeze dist { freeze := 20, live := 80};

} /

/ Set the frame size in pixels

constraint calc\_length\_c {

length == x\_pixels \* y\_pixels; // 6

} /

/ UVM Factory Registration

`uvm\_object\_utils(video\_frame\_item)

// During freeze conditions we do not want to

// randomize the data on the randomize call.

// Set the randomize mode to on/off depending on

// whether the live/freeze value.

function void pre\_randomize(); // 8

if (live\_freeze == live) begin

this.data\_array.rand\_mode(1);

end

else begin

this.data\_array.rand\_mode(0);

end

endfunction: pre\_randomize

function void set\_frame\_vars(int pix\_x\_dim = 16,

int pix\_y\_dim = 16,

video\_mode\_e vid\_type = MONO);

x\_pixels = pix\_x\_dim;

y\_pixels = pix\_y\_dim;

// Default constraints are off

MONO\_inside\_c.constraint\_mode(0);

MONO\_size\_c.constraint\_mode(0);

YCbCr\_inside\_c.constraint\_mode(0);

YCbCr\_size\_c.constraint\_mode(0);

RGB\_inside\_c.constraint\_mode(0);

RGB\_size\_c.constraint\_mode(0);

mode = vid\_type;

case (vid\_type)

MONO : begin

this.MONO\_inside\_c.constraint\_mode(1);

this.MONO\_size\_c.constraint\_mode(1);

end

YCbCr: begin

this.YCbCr\_inside\_c.constraint\_mode(1);

this.YCbCr\_size\_c.constraint\_mode(1);

end

RGB : begin

this.RGB\_inside\_c.constraint\_mode(1);

this.RGB\_size\_c.constraint\_mode(1);

end

default : `uvm\_error(get\_full\_name(),

"!!!!No valid video format selected!!!\n\n", UVM\_LOW);

endcase

function new(string name = "video\_frame\_item");

super.new(name);

endfunction

endclass: video\_frame\_item

更高性能版本

typedef enum bit {live, freeze} live\_freeze\_t; // 2

typedef enum bit[1:0] {MONO, YCbCr, RGB} video\_mode\_e; // 2

class video\_frame\_item extends uvm\_sequence\_item;

// Frame Packets will either be regenerated or repeated

// in the case of freeze.

rand live\_freeze\_t live\_freeze= live; // 1

int length; // 1

video\_mode\_e mode;

bit [11:0] data\_array []; // 1, 2

constraint live\_freeze\_dist\_c {

live\_freeze dist { freeze :=20, live :=80};

}

// UVM Factory Registration

`uvm\_object\_utils(video\_frame\_item)

function void pre\_randomize(); // 8

if (live\_freeze == live) begin

case(mode)

YCbCr: begin

data\_array = new[2\*length];

foreach(data\_array[i]) begin

data\_array[i] = $urandom\_range(4095, 0);

end

end

RGB: begin

data\_array = new[3\*length];

foreach(data\_array[i]) begin

data\_array[i]= $urandom\_range(255, 0);

end

end

MONO: begin

data\_array = new[length];

foreach(data\_array[i]) begin

data\_array[i] = $urandom\_range(236, 16);

end

end

endcase

end

endfunction: pre\_randomize

function void set\_frame\_vars(int pix\_x\_dim = 16,

int pix\_y\_dim = 16,

video\_mode\_e vid\_type = MONO);

length = (pix\_x\_dim\* pix\_y\_dim); // 1, 6

mode = vid\_type;

endfunction: set\_frame\_vars

function new(string name = "video\_frame\_item");

super.new(name);

endfunction

endclass: video\_frame\_item

这两个代码片段在功能上是等效的，但在执行时间上有很大的差异重构后的代码进行了大量更改，从而大大加快

 在原始代码中，数组的大小是通过随机化两个变量来计算的-长度和数组大小。这是不必要的，因为视频帧是一个固定的大小，可以从类中的其他属性计算

数组的长度是使用约束内的乘法运算符计算的。在第一个示例中，数据数组的内容是由约束求解器计算的

在foreach（）循环中。 这是不必要的，并且对于较大的阵列来说是昂贵的。 由于这些值在可预测的范围内，因此它们可以在post\_randomize（）方法中生成。

默认情况下，枚举类型live\_freeze\_t和video\_mode\_e将具有底层整数类型，重构版本使用尽可能少的位类型。

 原始版本使用一组constraint\_mode（）和rand\_mode（）调用来控制随机化的工作方式，这通常不如编码约束以考虑状态条件有效。

实际上，最后一个示例中唯一的随机变量是live\_freeze位。

其他约束示例

较低性能版本

rand bit[31:0] addr;

constraint align\_addr\_c{

addr%4==0;

}

更高性能版本

rand bit[31:0] addr;

constraint align\_addr\_c{

addr[1:0]==0;

}

约束的第一个版本使用模运算符将最低两位设置为零，第二个版本直接这样做，避免了昂贵的算术运算。

较低性能版本

enum bit[3：0]{ADD，XOR，DIV，OR，AND，XOR，NAND，MULT}opcode\_e;

opcode\_eins;

constraintselect\_opcodes\_c{

insdist {ADD：=7，DIV：=7，MULT：=7};

}

更高性能版本

enum bit[3：0]{ADD，XOR，DIV，OR，AND，XOR，NAND，MULT}opcode\_e;

opcode\_eins;

constraintselect\_opcodes\_c{

ins inside {ADD, SUB, DIV, MULT};

}

这两个版本的约束在它们产生的结果中是等价的，但是第一个版本强制求解一个分布，这比将ins值限制在一个集合内要昂贵得多

Covergroup性能指南

覆盖组基本上是一组计数器，当采样值与bin过滤器匹配时，计数器会递增，保持性能的方法是尽可能节省覆盖组活动。covergroups的基本规则是管理样本箱的创建区块控制

每个覆盖点自动转换为覆盖点中采样的变量的每个可能值的一组bin或计数器这将等于2\*\*n个bin，其中n是变量中的位数，但这通常被SystemVerilogauto\_bins\_max变量限制为最多64个bin，以避免朴素编码的问题（想想32位int上的覆盖点会产生多少个bin）。

投资于覆盖组设计是值得的，创建产生有用信息的bin通常会减少使用的bin数量，这将有助于提高性能。覆盖群叉积项也有爆炸的可能，但有语法可以用来消除项。

较低性能版本

int[7：0]a;

int[7：0]b;

covergroup data\_cg;

A: coverpoint a; // 256 bins

B: coverpoint b; // 256 bins

A\_X\_B: cross A, B; // 65536 bins

endgroup: data\_cg

更高性能版本

covergroup data\_cg;

A:coverpoint a {

bins zero = {0}; // 1 bin

bins min\_zone[] = {[8'h01:8'h0F]}; // 15 bins

bins max\_zone[] = {[8'hF0:8'hFE]}; // 15 bins

bins max = {8'hFF}; // 1 bin

bins medium\_zone[16] = {[8'h10:8'hEF]}; // 16 bins

}

B: coverpoint b{

bins zero = {0};

bins min\_zone[] = {[8'h01:8'h0F]};

bins max\_zone[] = {[8'hF0:8'hFE]};

bins max = {8'hFF};

bins medium\_zone[16] = {[8'h10:8'hEF]};

}

A\_X\_B: cross A, B; // 2304 bins

endgroup: data\_cg

在第一个covergroup示例中，使用了默认值。如果没有max\_auto\_bins变量，A和B将有256个bin，交叉将有256\*256个bin，并且结果很难解释。当max\_auto\_bins设置为64时，A、B和 跨产品，这节省了性能，但使结果更难理解。

右侧的覆盖组示例创建了一些用户仓，这将理论仓的数量减少到A和B的48个仓和交叉的2304这提高了性能，并使结果更容易解释。

采样控制

覆盖组采样的一个常见错误是写入在固定事件（如时钟沿）采样的覆盖组，而不是在覆盖组中采样的值有效时采样覆盖组采样仅应在所需的测试台行为发生且覆盖组变量为稳定值时发生。仔细注意覆盖组抽样提高了所获得的结果的有效性，以及提高测试台的性能。

较低性能版本

int data;

bit active;

covergroup data\_cg @(posedge clk);

coverpoint data iff(valid == 1) {

bins a = {[0:4000]};

bins b = {[10000:100000]};

bins c = {[4001:4040]};

}

endgroup: data\_cg

更高性能版本

int data;

bit active;

covergroup data\_cg;

coverpoint data {

bins a = {[0:4000]};

bins b = {[10000:100000]};

bins c = {[4001:4040]};

}

endgroup: data\_cg

task update\_coverage;

forever begin

@(posedge clk);

if(valid) begin

data\_cg.sample();

end

end

endtask: update\_coverage

在第一个示例中，在时钟的上升沿对覆盖组进行采样，并且iff（有效）保护确定覆盖组中的bin是否递增，这意味着无论有效线的状态如何都对覆盖组进行采样。

在第二个例子中，内置的sample（）方法用于在设置了valid标志时对covergroup进行采样这将产生性能改进，特别是当valid很少为真时。

断言性能指南

SystemVerilog中的断言语法提供了一种非常简洁而强大的描述时态属性的方法然而，这种能力可能会影响性能。

这里有一些编写SystemVerilog断言的关键性能指南，它们也是

独特的触发

每次对属性进行采样时，都会检查启动属性计算的条件。 如果这个条件是不明确的，那么一个断言可能有多个正在进行的评估，这将可能导致错误的结果，并肯定会给模拟器带来更大的负载。

较低性能版本

property req\_rsp;

@(posedge clk);

req |=> (req & ~rsp)[\*2]

##1 (req && rsp)

##1 (~req && ~rsp);

endproperty: req\_rsp

更高性能版本

property req\_rsp;

@(posedge clk);

$rose(req) |=> (req & ~rsp)[\*2]

##1 (req && rsp)

##1 (~req && ~rsp);

endproperty: req\_rsp

在第一个例子中，每当req信号在逻辑1处被采样时，属性将被触发，这将导致断言的多个触发。在第二个示例中，该属性在req的上升沿触发，req是一个离散事件。

用于确保触发是唯一的其他策略是挑选唯一的事件，例如已知仅在时钟周期内有效的状态

安全VS活力

安全属性是具有时间界限的属性-例如req变为高电平后2个时钟，rsp应变为高电平。活跃度属性在时间上不受约束-例如。RSP将在REQ变高之后变高在编写断言时，重要的是要考虑正在进行的检查的生命周期，性能将受到保持在飞行中的断言的影响，因为它们完成时没有约束

大多数规范都应该为某些事情的发生定义某种时间限制，或者会有某种可以应用于属性的实际限制

较低性能版本

property req\_rsp;

@(posedge clk);

$(posedge req) |=>

(req & ~rsp)[\*1:2]

##1 (req && rsp)[->1] // Unbound condition - within any number of clocks

##1 (~req && ~rsp);

endproperty: req\_rsp

更高性能版本

property req\_rsp;

@(posedge clk);

$rose(req) |=>

(req & ~rsp)[\*1:4] // Bounds the condition to within 1-4 clocks

##1 (req && rsp)

##1 (~req && ~rsp);

endproperty: req\_rsp

断言守卫

断言可以使用iff（条件）保护构造来禁用这可确保仅在条件为true时对属性进行采样，这意味着可以使用

状态变量 这对于在重置期间或故意注入错误时过滤断言评估特别有用。

断言也可以使用系统任务$assertoff（）和$asserton（）来禁用，这些任务可以在SystemVerilog测试平台代码中以程序方式调用。 这些特性可用于管理整体性能，方法是在断言无效或不需要时取消激活断言。

较低性能版本

property req\_rsp;

@(posedge clk);

$(posedge req) |=>

(req & ~rsp)[\*1:2]

##1 (req && rsp)[->1]

##1 (~req && ~rsp);

endproperty: req\_rsp

更高性能版本

property req\_rsp;

// Disable if reset is active:

@(posedge clk) iff(!reset);

$rose(req) |=>

(req & ~rsp)[\*1:4]

##1 (req && rsp)

##1 (~req && ~rsp);

endproperty: req\_rsp

保持断言简单

编写断言时的一个常见错误是试图在一个属性中描述多个条件这总是会导致代码比实际需要的更复杂。

然后，如果断言失败，则需要进一步调试以找出失败的原因编写只检查协议一部分的属性更容易，当它们失败时，原因是显而易见的。

避免使用通过和失败消息

SystemVerilog断言语法允许用户添加pass和fail函数调用。避免使用

如果你的代码中有传递和失败的消息，字符串处理是昂贵的，如果断言失败，模拟器将自动生成一条消息

### 低性能版

REQ\_RSP:assert property(req\_rsp) begin

$display("Assertion REQ\_RSP passed at %t", $time);

else

$display("Error: Assertion REQ\_RSP failed at %t", $time);

End

更高性能版本

REQ\_RSP: assert property(req\_rsp) ;

请注意，即使留下一个空白开始...end for the pass子句会导致性能下降。

避免多个时钟

SystemVerilog断言可以使用多个时钟进行计时。这样做的规则相当复杂，并且乘法时钟序列的性能可能低于正常序列。

避免在断言中使用多个时钟，如果你发现自己写了一个时钟，那么这可能是你从错误的角度处理问题的标志



[先前](https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/systemverilog-guidelines)

[下](https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/uvm-guidelines)

**SystemVerilog性能指南** \*

最近关于SystemVerilog性能指南的论坛讨论

[**$system和DPI的性能比较**](https://verificationacademy.com/forums/t/performance-comparison-between-system-and-dpi/41727)

系统V...

[**在组合逻辑中使用非阻塞赋值**](https://verificationacademy.com/forums/t/using-nonblocking-assignments-in-combinational-logic/39414)

系统V...

[**按需网络研讨会录音：UVM编码指南：提示...**](https://verificationacademy.com/forums/t/on-demand-web-seminar-recording-uvm-coding-guidelines-tips-tricks-you-probably-didn-t-know/37427)

通知...

[**每个时钟周期对covergroup进行采样的开销**](https://verificationacademy.com/forums/t/overhead-of-sampling-a-covergroup-every-clock-cycle/32568)

系统V...

[**SV断言-捕获信号两个边沿之间的事件**](https://verificationacademy.com/forums/t/sv-assertions-capturing-an-event-in-between-two-edges-of-a-signal/32195)

系统V...

[问一个问题[](https://verificationacademy.com/forums/new-topic?tags=systemverilog-performance-guidelines)](https://verificationacademy.com/forums/new-topic?tags=systemverilog-performance-guidelines)

西门子

西门子数字工业软件

[4月](https://facebook.com/Siemens)[20](https://twitter.com/Siemens)[日](https://youtube.com/Siemens)

投资组合

[云](https://www.sw.siemens.com/en-US/digital-transformation/cloud/)

[设计、制造和PLM软件](https://plm.sw.siemens.com/en-US/)[电子设计自动化](https://eda.sw.siemens.com/en-US/)

[InsightsHub](https://plm.sw.siemens.com/en-US/insights-hub/)[菜单](https://www.mendix.com/)

**如何购买**[购买西门子](https://www.sw.siemens.com/en-US/buy/)[在线购买](https://www.dex.siemens.com/)

[合作伙伴](https://www.sw.siemens.com/en-US/partners/find-a-partner/)[学术](https://www.sw.siemens.com/en-US/academic/)[更新](https://www.sw.siemens.com/en-US/support-services/renewal/)

**门斯**

Si

[关于我们](https://www.sw.siemens.com/en-US/)[招贤纳士](https://www.sw.siemens.com/en-US/careers/)[社区](https://community.sw.siemens.com/s/)[活动](https://events.sw.siemens.com/en-US/)[领导力](https://www.sw.siemens.com/en-US/leadership/)[新闻与媒体](https://newsroom.sw.siemens.com/)[信任中心](https://www.sw.siemens.com/en-US/trust-center/)

接触

[VA -联系我们](https://verificationacademy.com/contact)[HLS -联系我们](https://hls.academy/contact)[PLM -联系我们](https://plm.sw.siemens.com/en-US/contact-plm/)[EDA -联系我们](https://resources.sw.siemens.com/en-US/contact-eda)[全球办事处](https://www.sw.siemens.com/en-US/office-locations/)[支持中心](https://support.sw.siemens.com/en-US?ref=footer)[提供反馈](https://webtac.industrysoftware.automation.siemens.com/feedback/)[报告盗版](https://www.sw.siemens.com/en-US/piracy-prevention/)

西门子2024

[使用条款](https://www.siemens.com/global/en/general/terms-of-use.html)[隐私政策](https://www.sw.siemens.com/en-US/privacy-policy/)[Cookie声明](https://www.siemens.com/global/en/general/cookie-notice.html)[DMCA](https://sw.siemens.com/en-US/dmca)[举报](https://www.siemens.com/global/en/company/about/compliance/reporting-channels.html)