Mentor Graphics® | Verification Academy

Cookbook

Online Methodology Documentation from the Mentor Graphics Verification Methodology Team

Contact VMDOC@mentor.com

http://verificationacademy.com

SV/Guidelines

Mentor Graphics SystemVerilog Guidelines

SystemVerilog Do's

- Use a consistent coding style - see guidelines
- Use a descriptive typedef for variables
- Use an end label for methods, classes and packages
- Use `includes to compile classes into packages
- Define classes within packages
- Define one class per file
- Only `include a file in one package
- Import packages to reference their contents
- Check that $cast() calls complete successfully
- Check that randomize() calls complete succesfully
- Use if rather than assert to check the status of method calls
- Wrap covergroups in class objects
- Only sample covergroups using the sample() method
- Label covergroup coverpoints and crosses

SystemVerilog Don'ts

- Avoid `including the same class in multiple locations
- Avoid placing code in $unit
- Avoid using associative arrays with a wildcard index
- Avoid using #0 delays
- Don't rely on static initialization order

The SystemVerilog coding guidelines and rules in this article are based on Mentor's experience and are designed to steer users away from coding practices that result in SystemVerilog that is either hard to understand or debug.

Please send any suggestions, corrections or additions to ?subject=SV/Guidelines vmdoc@mentor.com [1]

General Coding Style

Although bad coding style does not stop your code from working, it does make it harder for others to understand and makes it more difficult to maintain. Take pride in writing well-ordered and uniformly formatted code.

1.1 Guideline: Indent your code with spaces

Use a consistent number of spaces to indent your code every time you start a new nested block, 2 or 3 spaces is recommended. Do not use tabs since the tab settings vary in different editors and viewers and your formatting may not look as you intended. Many text editors have an indenting mode that automatically replaces tabs with a defined number of spaces.

1.2 Guideline: Only one statement per line

Only have one declaration or statement per line. This makes the code clearer and easier to understand and debug.

Recommended	**Not Recommended**
`// Variable definition:`	`// Variable definition:`
`logic enable;`	`logic enable, completed, in_progress;`
`logic completed;`	
`logic in_progress;`	`// Statements:`
	`if(enable == 0) in_progress = 1; else in_progress = 0;`
`// Statements:`	
`// (See next Guideline for the use of begin-end pairs`	
`// with conditional statements)`	
`//`	
`if(enable == 0)`	
`  in_progress = 1;`	
`else`	
`  in_progress = 0;`	

1.3 Guideline: Use a begin-end pair to bracket conditional statements

This helps make it clear where the conditional code begins and where it ends. Without a begin-end pair, only the first line after the conditional statement is executed conditionally and this is a common source of errors.

Recommended	**Not Recommended**
`// Both statements executed conditionally:`	`if(i > 0)`
`if(i > 0) begin`	`  count = current_count;`
`  count = current_count;`	`  target = current_target; // This statement is executed unconditionally`
`  target = current_target;`	
`end`	

1.4 Guideline: Use parenthesis in Boolean conditions

This makes the code easier to read and avoids mistakes due to operator precedence issues.

Recommended

```
// Boolean or conditional expression
if((A==B) && (B > (C*2)) || (B > ((D**2)+1))) begin
  ...
end
```

Not Recommended

```
// Boolean or conditional expression
if(A==B && B > C*2 || B > D**2+1) begin
  ...
end
```

1.5 Guideline: Keep your code simple

Avoid writing tricky and hard to understand code, keep it simple so that it is clear what it does and how so that others can quickly understand it in case a modification is required.

1.6 Guideline: Keep your lines to a reasonable length

Long lines are difficult to read and understand, especially if you need to scroll the editor to the right to read the end of the line. As a guideline, keep your line length to around 80 characters, break the line and indent at logical places.

Not Recommended

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  mbus_seq_item rhs_;

  if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_compare", "cast failed, check type compatability");
      return 0;
  end
  do_compare = super.do_compare(rhs, comparer) && (MADDR == rhs_.MADDR)
&& (MWDATA == rhs_.MWDATA) && (MREAD == rhs_.MREAD) &&
(MOPCODE == rhs_.MOPCODE) && (MPHASE == rhs_.MPHASE) && (MRESP ==
rhs_.MRESP) &&  (MRDATA == rhs_.MRDATA);
endfunction: do_compare
```

Recommended

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  mbus_seq_item rhs_;

  if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_compare", "cast failed, check type compatability");
      return 0;
  end
  do_compare = super.do_compare(rhs, comparer) &&
               (MADDR == rhs_.MADDR) &&
               (MWDATA == rhs_.MWDATA) &&
               (MREAD == rhs_.MREAD) &&
               (MOPCODE == rhs_.MOPCODE) &&
               (MPHASE == rhs_.MPHASE) &&
               (MRESP == rhs_.MRESP) &&
               (MRDATA == rhs_.MRDATA);
endfunction: do_compare
```

1.7 Guideline: Use lowercase for names, using underscores to separate fields

This makes it clearer what the name is, as opposed to other naming styles such as CamelCase which are harder to read.

Recommended	Not Recommended
`axi_fabric_scoreboard_error`	`AxiFabricScoreboardError`

1.8 Guideline: Use prefix_ and _postfix to delineate name types

Use prefixes and postfixes for name types to help differentiate between variables. Pre and post fixes for some common variable types are summarised in the following table:

prefix/postfix	Purpose
_t	Used for a type created via a typedef
_e	Used to indicate a enumerated type
_h	Used for a class handle
_m	Used for a protected class member (See guideline 2.2)
_cfg	Used for a configuration object handle
_ap	Used for an analysis port handle
_group	Used for a covergroup handle

1.9 Guideline: Use a descriptive typedef when declaring a variable instead of a built-in type

This makes the code clearer and easier to understand as well as easier to maintain. An exception is when the built-in type keyword best describes the purpose of the variable's type.

```
// Descriptive typedef for a 24 bit audio sample:
typedef bit[23:0] audio_sample_t;
```

1.10 Guideline: Use the end label for classes, functions, tasks, and packages

This forces the compiler to check that the name of the item matches the end label which can trap cut and paste errors. It is also useful to a person reading the code.

```
// Using end labels
package my_pkg;

  //...
  class my_class;
```

```
    // ...
    function void my_function();
    //...
    endfunction: my_function

    task my_task;
    // ...
    endtask: my_task

  endclass: my_class

endpackage: my_pkg
```

1.11 Guideline: Comment the intent of your code

Add comments to define the intent of your code, don't rely on the users interpretation. For instance, each method in a class should have a comment block that specifies its input arguments, its function and its return arguments.

This principle can be extended to automatically generate html documentation for your code using documentation tools such as **NaturalDocs.**

Class Names and Members

2.1 Guideline: Name classes after the functionality they encapsulate

Use classes to encapsulate related functionality. Name the class after the functionality, for instance a scoreboard for an Ethernet router would be named "router_scoreboard".

2.2 Guideline: Private class members should have a m_ prefix

Any member that is meant to be private should be named with a 'm_' prefix, and should be made local or protected. Any member that will be randomized should not be local or protected.

2.3 Guideline: Declare class methods using extern

This means that the class body contains the method prototypes and so users only have to look at this section of the class definition to understand its functionality.

```
// Descriptive typedefs:
typedef logic [31:0] raw_sample_t;
typedef logic [15:0] processed_sample_t

// Class definition illustrating the use of externally defined methods:
class audio_compress;

rand int iteration_limit;
rand bit valid;
rand raw_sample_t raw_audio_sample;
rand processed_sample_t processed_sample;

// function: new
// Constructor - initializes valid
extern function new();

// function: compress_sample
// Applies compression algorithm to raw sample
// inputs: none
// returns: void
extern function void compress_sample();

// function: set_new_sample
// Set a new raw sample value
// inputs:
//   raw_sample_t new_sample
// returns: void
extern function void set_new_sample(raw_sample_t new_sample);

endclass: audio_compress

function audio_compress::new();
  valid = 0;
  iteration_limit = $bits(processed_sample_t);
endfunction

function void audio_compress::compress_sample();
  for(int i = 0; i < iteration_limit; i++) begin
    processed_sample[i] = raw_audio_sample[((i*2)-1):(i*2)];
  end
  valid = 1;
endfunction: compress_sample

function void audio_compress:set_new_sample(raw_sample_t new_sample);
  raw_audio_sample = new_sample;
  valid = 0;
endfunction: set_new_sample
```

Files and Directories

The following guidelines concern best practices for SystemVerilog files and directories.

File Naming

3.1 Guideline: Use lower case for file and directory names

Lower case names are easier to type.

3.2 Guideline: Use .sv extensions for compile files, .svh for `include files

The convention of using the .sv extension for files that are compiled and .svh for files that get included makes it easier to sort through files in a directory and also to write compilation scripts.

For instance, a package definition would have a .sv extension, but would reference `included .svh files:

3.3 Guideline: `include .svh class files should only contain one class and be named after that class

This makes it easier to maintain the code, since it is obvious where the code is for each class.

3.4 Guideline: Use descriptive names that reflect functionality

File names should match their content. The names should be descriptive and use postfixes to help describe the intent - e.g. _pkg, _env, _agent etc.

`include versus import

3.5 Rule: Only use `include to include a file in one place

The `include construct should only be used to include a file in just one place. `include is typically used to include .svh files when creating a package file.

If you need to reference a type or other definition, then use 'import' to bring the definition into scope. Do not use `include. The reason for this is that type definitions are scope specific. A type defined in two scopes using the same `include file are not recognised as being the same. If the type is defined in one place, inside a package, then it can be properly referenced by importing that package.

An exception to this would be a macro definition file such as the 'uvm_macros.svh' file.

Directory Names

Testbenches are constructed of SystemVerilog UVM code organized as **packages**, collections of verification IP organized as packages and a description of the hardware to be tested. Other files such as C models and documentation may also be required. Packages shoudl be organized in a hierarchy.

3.6 Guideline: Each package should have its own directory

Each package should exist in its own directory. Each of these package directories should have one file that gets compiled - a file with the extension .sv

Each package should have at most one file that may be included in other code. This file may define macros.

```
abc_pkg.sv
abc_macros.svh
```

For a complex package (such as a UVC) that may contain tests, examples and documentation, create subdirectories:

```
abc_pkg/examples
abc_pkg/docs
abc_pkg/tests
abc_pkg/src/abc_pkg.sv
```

For a simple package the subdirectories may be omitted

```
abc_pkg/abc_pkg.sv
```

Sample File Listing

```
./abc_pkg/src
./abc_pkg/src/abc_pkg.sv

./abc_pkg/src/abc_macros.svh

./abc_pkg/src/abc_env.svh
./abc_pkg/src/abc_interface.sv

./abc_pkg/src/abc_driver.svh
./abc_pkg/src/abc_monitor.svh
./abc_pkg/src/abc_scoreboard.svh

./abc_pkg/src/abc_sequence_item.svh
./abc_pkg/src/abc_sequencer.svh
./abc_pkg/src/abc_sequences.svh

./abc_pkg/docs/
./abc_pkg/docs/abc_user_guide.docx

./abc_pkg/tests/
./abc_pkg/tests/......

./abc_pkg/examples/
./abc_pkg/examples/a/....
./abc_pkg/examples/b/....
./abc_pkg/examples/c/....

./testbench1/makefile
./testbench1/tb_env.sv
./testbench1/tb_top.sv
./testbench1/test.sv
```

Using Packages

3.7 Rule: Import packages to reference their contents

When you use a function or a class from a package, you import it, and `include any macro definitions.

If you `include the package source, then you will be creating a new namespace for that package in every file that you `include it into, this will result in type matching issues.

```
import abc_pkg::*;
  `include "abc_macros.svh"
```

3.8 Rule: When compiling a package, use +incdir+ to reference its source directory

To compile the package itself, you use a +incdir to reference the source directory. Make sure that there are no hardcoded paths in the path string for the `included file.

```
vlog +incdir+$ABC_PKG/src abc_pkg.sv
```

To compile code that uses the package, you also use a +incdir to reference the source directory if a macro file needs to be `included.

```
vlog +incdir+$ABC_PKG/src tb_top.sv
```

To compile the packages, and the testbench for the example:

```
vlib work

# Compile the Questa UVM Package (for UVM Debug integration)
vlog +incdir+$QUESTA_UVM_HOME/src \
  $QUESTA_UVM_HOME/src/questa_uvm_pkg.sv

# Compile the VIP (abc and xyz)
vlog +incdir+../abc_pkg/src \
  ../abc_pkg/src/abc_pkg.sv
vlog +incdir+../xyz_pkg/src \
  ../xyz_pkg/src/xyz_pkg.sv

# Compile the DUT (RTL)
vlog ../dut/dut.sv

# Compile the test
vlog +incdir+../test_pkg/src \
  ../test_pkg/src/test_pkg.sv

# Compile the top
vlog tb_top.sv

# Simulate
vsim -uvm=debug -coverage +UVM_TESTNAME=test \
  -c tb_top -do "run -all; quit -f"
```

SystemVerilog Language Guidelines

4.1 Rule: Check that $cast() has succeeded

If you are going to use the result of the cast operation, then you should check the status returned by the $cast call and deal with it gracefully, otherwise the simulation may crash with a null pointer.

Note that it is not enough to check the result of the cast method, you should also check that the handle to which the cast is made is not null. A cast operation will succeed if the handle from which the cast is being done is null.

```
// How to check that a $cast has worked correctly
function my_object get_a_clone(uvm_object to_be_cloned);
  my_object t;

  if(!$cast(t, to_be_cloned.clone()) begin
    `uvm_error("get_a_clone", "$cast failed for to_be_cloned")
  end
  if(t == null) begin
    `uvm_fatal("get_a_clone", "$cast operation resulted in a null handle, check to_be_cloned handle")
  end

  return t;
endfunction: get_a_clone
```

4.2 Rule: Check that randomize() has succeeded

If no check is made the randomization may be failing, meaning that the stimulus generation is not working correctly.

```
// Using if() to check randomization result
if(!seq_item.randomize() with {address inside {[o:32'hF000_FC00]};}) begin
  `uvm_error("seq_name", "randomization failure, please check constraints")
end
```

4.3 Rule: Use if rather than assert to check the status of method calls

Assert results in the code check appearing in the coverage database, which is undesired. Incorrectly turning off the action blocks of assertions may also produce undesired results.

Constructs to be Avoided

The SystemVerilog language has been a collaborative effort with a long history of constructs *borrowed* from other languages. Some constructs have been improved upon with newer constructs, but the old constructs remain for backward compatibility and should be avoided. Other constructs were added before being proven out and in practice cause more problems than they solve.

4.4 Rule: Do not place any code in $unit, place it in a package

The compilation unit, $unit, is the scope outside of a design element (package, module, interface, program). There are a number of problems with timescales, visibility, and re-usability when you place code in $unit. Always place this code in a package.

4.5 Guideline: Do not use associative arrays with a wildcard index[*]

A wildcard index on an associative array is an un-sized integral index. SystemVerilog places severe restrictions on other constructs that cannot be used with associative arrays having a wildcard index. In most cases, an index type of [int] is sufficient. For example, a foreach loop requires a fixed type to declare its iterator variable.

```
string names[*]; // cannot be used with foreach, find_index, ...
string names[int];
...
foreach (names[i])
  $display("element %0d: %s",i,names[i]);
```

4.6 Guideline: Do not use #0 procedural delays

Using a #0 procedural delay, sometimes called a delta delay, is a sure sign that you have coded incorrectly. Adding a #0 just to get your code working usually avoids one race condition and creates another one later.Often, using a non-blocking assignment (<=) solves this class of problem.

4.7 Guideline: Avoid the use of the following language constructs

A number of SystemVerilog language constructs should be avoided altogether:

Construct	Reason to avoid
checker	Ill defined, not supported by Questa
final	Only gets called when a simulation completes
program	Legacy from Vera, alters timing of sampling, not necessary and potentially confusing

Coding Patterns

Some pieces of code fall into well recognized patterns that are know to cause problems

4.8 Rule: Do not rely on static variable initialization order, initialize on first instance.

The ordering of static variable initialization is undefined. If one static variable initialization requires the non-default initialized value of another static variable, this is a race condition. This can be avoided by creating a static function that initializes the variable on the first reference, then returns the value of the static variable, instead of directly referencing the variable.

```
typedef class A;
typedef class B;
A a_top=A::get_a();
B b_top=B::get_b();
class A;
  static function A get_a();
    if (a_top == null) a_top =new();
    return a_h;
  endfunction
endclass : A
```

```
class B;
  A a_h;
  protected function new;
    a_h = get_a();
  endfunction
  static function B get_b();
    if (b_top == null) b_top =new();
    return b_top;
  endfunction
endclass : B
```

Covergroups

4.9 Guideline: Create covergroups within wrapper classes

Covergroups have to be constructed within the constructor of a class. In order to make the inclusion of a covergroup within a testbench conditional, it should be wrapped within a wrapper class.

4.10 Guideline: Covergroup sampling should be conditional

Build your covergroups so that their sample can be turned on or off. For example use the 'iff' clause of covergroups.

```
// Wrapped covergroup with sample control:
class cg_wrapper extends uvm_component;

logic[31:0] address;
bit coverage_enabled

covergroup detail_group;
  ADDRESS:  coverpoint addr iff(coverage_enabled) {
    bins low_range = {[0:32'h0000_FFFF]};
    bins med_range = {[32'h0001_0000:32'h0200_FFFF]};
    bins high_range = {[32'h0201_0000:32'h0220_FFFF]};
  }
// ....
endgroup: detail_group

function new(string name = "cg_wrapper", uvm_component parent = null);
  super.new(name, parent);
  // Construct covergroup and enable sampling
  detail_group = new();
  coverage_enabled = 1;
endfunction

// Set coverage enable bit - allowing coverage to be enabled/disabled
function void set_coverage_enabled(bit enable);
  coverage_enabled = enable;
endfunction: set_coverage_enabled

// Get current state of coverage enabled bit
function bit get_coverage_enabled();
  return coverage_enabled;
endfunction: get_coverage_enabled

// Sample the coverage group:
function void sample(logic[31:0] new_address);
```

```
  address = new_address;
  detail_group.sample();
endfunction: sample
```

Coverpoint sampling may not be valid in certain situations, for instance during reset.

```
// Using iff to turn off unnecessary sampling:

// Only sample if reset is not active
coverpoint data iff(reset_n != 0) {
  // Only interested in high_end values if high pass is enabled:
  bins high_end = {[10000:20000]} iff(high_pass);
  bins low_end = {[1:300]};
  }
```

Collecting Coverage

4.11 Guideline: Use the covergroup sample() method to collect coverage

Sample a covergroup by calling the sample routine, this allows precise control on when the sampling takes place.

4.12 Rule: Label coverpoints and crosses

Labelling coverpoints allows them to be referenced in crosses and easily identified in reports and viewers.

```
payload_size_cvpt: coverpoint ...
```

Labelling crosses allows them to be easily identified

```
payload_size_X_parity: cross payload_size_cvpt, parity;
```

4.13 Guideline: Name your bins

Name your bins, do not rely on auto-naming.

```
bin minimum_val = {min};
```

4.14 Guideline: Minimize the size of the sample

It is very easy to specify large numbers of bins in covergroups through autogeneration without realising it. You can minimise the impact of a covergroup on simulation performance by thinking carefully about the number and size of the bins required, and by reducing the cross bins to only those required.

Other SystemVerilog Guidelines Documents

- Stu Sutherlands' SystemVerilog for Design [2]
- Chris Spear's SystemVerilog for Verification [3]
- Doulos' SystemVerilog Golden Reference Guide [4]
- Adam Erickson's Are Macros Evil? DVCon 2011 Best Paper [5]

SV/PerformanceGuidelines

These guidelines are aimed at enabling you to identify coding idioms that are likely to affect testbench performance. Please note that a number of these guidelines run counter to other recommended coding practices and a balanced view of the trade off between performance and methodology needs to be made.

Whilst some of the code structures highlighted might be recognized and optimized out by a compiler, this may not always be the case due to the side effects of supporting debug, interactions with PLI code and so on. Therefore, there is almost always a benefit associated with re-factoring code along the lines suggested.

SystemVerilog shares many common characteristics with mainstream software languages such as C, C++ and Java, and some of the guidelines presented here would be relevant to those languages as well. However, SystemVerilog has some unique capabilities and short-comings which might cause the unwary user to create low performance and memory hungry code without realising it.

Tuning the performance of a testbench is made much easier the use of code profiling tools. A code profile can identify 'hot-spots' in the code, and if these places can be refactored the testbench is almost invariably improved. In the absence of a profiling tool, visual code inspection is required but this takes time and concentration. These guidelines are intended to be used before coding starts, and for reviewing code in the light of code profiling or by manual inspection.

Code Profiling

Code profiling is an automatic technique that can be used during a simulation run to give you an idea of where the 'hot-spots' are in the testbench code. Running a code profile is a run time option, which if available, will be documented in the simulator user guide. See the "Profiling Performance and Memory Use" chapter in the Questa User Guide for more information.

When your testbench code has reached a reasonable state of maturity and you are able to reliably run testcases, then it is always worth running the profiling tool. Most code profilers are based on sampling; they periodically record which lines of code are active and which procedural calls are in progress at a given point in time. In order to get a statistically meaningful result, they need to be run for a long enough time to collect a representative sample of the code activity.

In a well written testbench with no performance problems, the outcome of the sampling will be a flat distribution across the testbench code. However, if the analysis shows that a particular area of the testbench is showing up in a disproportionate number of samples then it generally points to a potential problem with that code.

Profiling is an analysis technique and the results will be affected by:

- The characteristics of the testcase(s) that are being analysed
- Random seeding - causing different levels of activity in the testbench

- Dominant behaviour in your testbench - some areas of the testbench may simply be doing more work
- DUT coding style
- The sample interval
- The length of the simulation time that the profile is run for
- What is going on in the simulation whilst the profile is being run

With constrained random testbenches it is always worth running through alternative testcases with different seeds whilst analysing the profiling report since these may throw light on different coding issues.

Loop Guidelines

Loop performance is determined by:

- The work that goes on within the loop
- The checks that are made in the loop to determine whether it should be active or not

The work that goes on within the loop should be kept to a minimum, and the checks made on the loop bounds should have a minimum overhead. Here are some examples of good and bad loop practices:

Lower Performance Version

```
// dynamic array, unknown size
int array[];
int total = 0;

for(int i = 0; i < array.size(); i++) begin
  total += array[i];
end
```

Higher Performance Version

```
// 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
```

Setting a variable to the size of the array before the loop starts saves the overhead of calculating the array.size() on every iteration.

Lower Performance Version

```
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
```

Higher Performance Version

```
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
```

The foreach() loop construct is typically higher performance than for(int i = 0; i < <val>; i++) for smaller arrays.

The lookup of the exponent value in the associative array on every loop iteration is uneccessary, since it can be looked up at the beginning of the loop.

Lower Performance Version

```
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
```

Higher Performance Version

```
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;
  end
end
```

In this example, an array with unique entries is being searched within a loop for a given value. Using break in the second example terminates the evaluation of the loop as soon as a match is found.

Decision Guidelines

When making a decision on a logical or arithmetic basis there are a number of optimisations that can help improve performance:

Short-circuit logic expressions

The evaluation of a short circuit logic expression is abandoned as soon as one of its elements is found to be false. Using a short-circuit logic express has the potential to speed up a decision. Ordering the terms in a short-circuit expression can also avoid an expensive call if it is not neccessary. Some examples:

With an AND evaluation, if the the first term of the expression is untrue, the rest of the evaluation is skipped:

```
if(A && B && C) begin
  // do something
end
```

With an OR evaluation, if the first term of the expression is true, then the rest of the evaluation is skipped:

```
if(A || B || C) begin
  // do something
end
```

If the terms in the expression have a different level of "expense", then the terms should be ordered to compute the least expensive first:

Lower Performance Version

```
if(B.size() > 0) begin
  if(B[$] == 42) begin
    if(A) begin
        // do something
    end
  end
end
```

Higher Performance Version

```
if(A && (B.size() > 0) && B[$] == 42) begin
  // do something
end
```

If the inexpensive expression A evaluates untrue, then the other expensive conditional tests do not need to be made.

Lower Performance Version

```
if((A||B) && C) begin
  // do something
end
```

Higher Performance Version

```
if(C && (A||B)) begin
  // do something
end
```

A slightly less obvious variant, which saves the computation required to arrive at a decision if C is not true.

Refactoring logical decision logic

Sometimes a little bit of boolean algebra combined with some short-circuiting can reduce the computation involved.

lower performance code

```
if((A && C) || (A && D)) begin
 // do something
end
```

higher performance code

```
if(A && (C || D)) begin
 // do something
end
```

In the above example, refactoring the boolean condition removes one logical operation, using A as a short-circuit potentially reduces the active decision logic

Refactoring arithmetic decision logic

Remembering to refactor arithmetic terms can also lead to optimisations. This does not just apply to decision logic, but also to computing variables.

Lower Performance Version

```
if(((A*B) - (A*C)) > E) begin
  // do something
end
```

Higher Performance Version

```
if((A*(B - C)) > E) begin
  // do something
end
```

In the above example, refactoring avoids a multiplication operation.

Priority encoding

If you know the relative frequency of conditions in a decision tree, move the most frequently occurring conditions to the top of the tree. This most frequently applies to case statements and nested ifs.

Lower Performance Version

```
// 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
```

Higher Performance Version

```
// 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
```

Most of the time, the case statement exits after one check saving further comparisons.

Lower Performance Version

```
// 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
```

Higher Performance Version

```
// 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
```

In the higher performance version of this example, if ready is not valid, the rest of the code does not get evaluated. Then the read_cycle check is made, which removes the need for the write_cycle check.

Task and Function Call Guidelines

In-Lining Code

In some situations it may be better to re-factor code that is calling sub-routine methods so that the contents of the method are unrolled and put in-line rather than using the sub-routine. This will be particularly true if the sub-routine is relatively short and has multiple arguments.

Task And Functional Call Argument Passing

In SystemVerilog, passing arguments to/from task and function calls is done by making a copy of the variable at the start of the task or function call and then copying back the result of any changes made during the execution of the method. This can become quite an overhead if the arguments are complex variable types such as strings or arrays, and the alternative is to use a reference. Using a reference saves the overhead of the copies but it does mean that since the variable is not copied into the function if it is updated in the task or function, then it is also updated in the calling method. One way to avoid this issue is to make the variable a const ref, this effectively makes it a read only reference from the point of view of the function.

Lower Performance Version

```
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
```

Higher Performance Version

```
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
```

```
In the lower performance version of the code, a queue of ints and a string are
copied into the function. As the queue grows in length, this becomes increasingly
expensive. In the higher performance version, both the int queue and the string
arguments are references, this avoids the copy operation and speeds up the execution
 of the function.
```

Class Performance Guidelines

In SystemVerilog, a class encapsulates data variables and methods that operate on those variables. A class can be extended to add more variables and add to or extend the existing methods to provide new functionality. All of this convenience and functionality comes with a performance overhead which can be minimised by the following guidelines:

Avoid Uneccessary Object Construction

Constructing an object can have an overhead associated with it. As a general rule, try to minimise the number of objects created.

Lower Performance Version

```
//
// 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
```

Higher Performance Version

```
//
// 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
```

```
It is not necessary to construct the bus transaction object, the function will
return a null handle if it is not constructed.
```

Lower Performance Version

```
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
```

Higher Performance Version

```
task handle_bus_write;
  bus_object write_req;


  write_bus_req_fifo.get(write_req);
  // do something with the write_req;

endtask: handle_bus_write
```

Constructing the write_req object is redundant since its handle is re-assigned by the get from the bus_write_req_fifo.

Direct Variable Assignment Is Faster Than set()/get() Methods

Calling a method to update or examine a variable carries a higher overhead than direct access via the class hierarchical path.

Lower Performance Version

```
// 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 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
```

Higher Performance Version

```
// 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;
    V = V + m.A;
  end
end
```

Making an assignment to the data variable within the class using its hierarchical path is more efficient than calling a method to set()/get() it. However, if the set()/get() method does more than a simple assignment - e.g. a type conversion or a checking operation on the arguments provided, then the method approach should be used.

Note that: this guideline is for performance and flouts the normal OOP guideline that data variables within a class should only be accessible via methods. Using direct access methods to get to variables improves performance, but comes at the potential cost of making the code less reuseable and relies on the assumption that the user knows the name and type of the variable in question.

Avoid Method Chains

Calling a method within a class carries an overhead, nesting or chaining method calls together increases the overhead. When you implement or extend a class try to minimise the number of levels of methods involved.

Lower Performance Version

```
class mailbox_e #(type T = int);
  local mailbox #(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
```

Higher Performance Version

```
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
```

The second implementation extends the mailbox directly and avoids the extra layer in the first example.

Lower Performance Version ➔ **Higher Performance Version**

```
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
```

```
class multi_method; int i;

function void m1();
  i++;
endfunction: m1

endclass: multi_method
```

In the first example, a function call is implemented as a chain, whereas the second example has a single method and will have a higher performance. Your code may be more complex, but it may have method call chains that you could unroll.

Array Guidelines

SystemVerilog has a number of array types which have different characteristics, it is worth considering which type of array is best suited to the task in hand. The following table summarises the considerations.

Array Type	Characteristics	Memory Impact	Performance Impact
Static Array int a_ray[7:0];	Array size fixed at compile time. Index is by integer.	Least	Array indexing is efficient. Search of a large array has an overhead.
Dynamic Array int a_ray[];	Array size determined/changed during simulation. Index is by integer.	Less	Array indexing is efficient. Managing size is important.
Queues int a_q[$];	Use model is as FIFO/LIFO type storage Self-managed sizing Uses access methods to push and pop data	More	Efficient for ordered accesses Self managed sizing minimises performance impact
Associative arrays int a_ray[string];	Index is by a defined type, not an integer Has methods to aid management Sized or unsized at compile time, grows with use	More	Efficient for sparse storage or random access Becomes more inefficient as it grows, but elements can be deleted Non-integer indexing can raise abstraction

For example, it may be more efficient to model a large memory space that has only sparse entries using an associative array rather than using a static array. However, if the associative array becomes large because of the number of entries then it would become more efficient to implement to use a fixed array to model the memory space.

Use Associative Array Default Values

In some applications of associative arrays there may be accesses using an index which has not been added to the array, for instance a scoreboard sparse memory or a tag of visited items. When an associative array gets an out of range access, then by default it returns an warning message together with an uninitialized value. To avoid this scenario, the array can be queried to determine if the index exists and, if not, the access does not take place. If the default variable syntax is used, then this work can be avoided with a performance improvement:

Lower Performance Version

```
// Associative array declaration - no default value:
int aa[int];

if(aa.exists(idx)) begin
  lookup = aa[idx];
end
```

Higher Performance Version

```
// Associative array declaration - setting the default to 0
int aa[int] = {default:0};

lookup = aa[idx];
```

Avoiding Work

The basic principle here is to avoid doing something unless you have to. This can manifest itself in various ways:

- Don't randomize an object unless you need to
- Don't construct an object unless you need to
- Break out of a loop once you've found what you're looking for
- Minimise the amount of string handling in a simulation - in UVM testbenches this means using the `uvm_info(), `uvm_warning(), `uvm_error(), `uvm_fatal() macros to avoid string manipulation unless the right level of verbosity has been activated

Constraint Performance Guidelines

Constrained random generation is one of the most powerful features in SystemVerilog. However, it is very easy to over constrain the stimulus generation. A little thought and planning when writing constraints can make a big difference in the performance of a testbench. Always consider the following when writing constrained random code in classes:

1. Minimize the number of active rand variables - if a value can be calculated from other random fields then it should be not be rand
2. Use minimal data types - i.e. bit instead of logic, tune vectors widths to the minimum required
3. Use hierarchical class structures to break down the randomization, use a short-circuit decision tree to minimize the work
4. Use late randomization to avoid unnecessary randomization
5. Examine repeated use of in-line constraints - it may be more efficient to extend the class
6. Avoid the use of arithmetic operators in constraints, especially *, /, % operators
7. Implication operators are bidirectional, using solve before enforces the probability distribution of the before term(s)
8. Use the pre-randomize() method to pre-set or pre-calculate state variables used during randomization
9. Use the post-randomize() method to calculate variable values that are dependent on random variables.
10. Is there an alternative way of writing a constraint that means that it is less complicated?

The best way to illustrate these points is through an example - note that some of the numbered points above are referenced as comments in the code:

Lower Performance Version:

```
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]};
}
constraint 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_report_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
```

Higher Performance Version:

```
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
```

The two code fragments are equivalent in functionality, but have a dramatic difference in execution time. The re-factored code makes a number of changes which speed up the generation process dramatically:

- In the original code, the size of the array is calculated by randomizing two variables - length and array size. This is not necessary since the video frame is a fixed size that can be calculated from other properties in the class.
- The length of the array is calculated using a multiplication operator inside a constraint
- In the first example, the content of the data array is calculated by the constraint solver inside a foreach() loop. This is unnecessary and is expensive for larger arrays. Since these values are within a predictable range they can be generated in the post_randomize() method.
- The enum types live_freeze_t and video_mode_e will have an underlying integer type by default, the refactored version uses the minimal bit types possible.
- The original version uses a set of constraint_mode() and rand_mode() calls to control how the randomization works, this is generally less effective than coding the constraints to take state conditions into account.
- In effect, the only randomized variable in the final example is the live_freeze bit.

Other Constraint Examples

Lower Performance Version

```
rand bit[31:0] addr;
constraint align_addr_c {
  addr%4 == 0;
}
```

Higher Performance Version

```
rand bit[31:0] addr;
constraint align_addr_c {
  addr[1:0] == 0;
}
```

The first version of the constraint uses a modulus operator to set the lowest two bits to zero, the second version does this directly avoiding an expensive arithmetic operation

Lower Performance Version

```
enum bit[3:0] {ADD, SUB, DIV, OR, AND, XOR, NAND, MULT} opcode_e;
opcode_e ins;

constraint select_opcodes_c {
  ins dist {ADD:=7, SUB:=7, DIV:=7, MULT:=7};
}
```

Higher Performance Version

```
enum bit[3:0] {ADD, SUB, DIV, OR, AND, XOR, NAND, MULT} opcode_e;
opcode_e ins;

constraint select_opcodes_c {
  ins inside {ADD, SUB, DIV, MULT};
}
```

The two versions of the constraint are equivalent in the result they produce, but the first one forces a distribution to be solved which is much more expensive than limiting the ins value to be inside a set.

Covergroup Performance Guidelines

Covergroups are basically sets of counters that are incremented when the sampled value matches the bin filter, the way to keep performance up is be as frugal as possible with the covergroup activity. The basic rules with covergroups are to manage the creation of the sample bins and the sampling of the covergroup.

Bin Control

Each coverpoint automatically translates to a set of bins or counters for each of the possible values of the variable sampled in the coverpoint. This would equate to 2**n bins where n is the number of bits in the variable, but this is typically limited by the SystemVerilog auto_bins_max variable to a maximum of 64 bins to avoid problems with naive coding (think about how many bins a coverpoint on a 32 bit int would produce otherwise). It pays to invest in covergroup design, creating bins that yield useful information will usually reduce the number of bins in use and this will help with performance. Covergroup cross product terms also have the potential to explode, but there is syntax that can be used to eliminate terms.

Lower Performance Version

```
bit[7:0] a;
bit[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
```

Higher Performance Version

```
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
```

In the first covergroup example, the defaults are used. Without the max_auto_bins variables in place, there would be 256 bins for both A and B and 256*256 bins for the cross and the results are difficult to interpret. With max_auto_bins set to 64 this reduces to 64 bins for A, B and the cross product, this saves on performance but makes the results even harder to understand. The right hand covergroup example creates some user bins, which reduces the number of theoretical bins down to 48 bins for A and B and 2304 for the cross. This improves performance and makes the results easier to interpret.

Sample Control

A common error with covergroup sampling is to write a covergroup that is sampled on a fixed event such as a clock edge, rather than at a time when the values sampled in the covergroup are valid. Covergroup sampling should only occur if the desired testbench behavior has occurred and at a time when the covergroup variables are a stable value. Careful attention to covergroup sampling improves the validity of the results obtained as well as improving the performance of the testbench.

Lower Performance Version **Higher Performance Version**

```
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
```

```
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
```

In the first example, the covergroup is sampled on the rising edge of the clock and the iff(valid) guard determines whether the bins in the covergroup are incremented or not, this means that the covergroup is sampled regardless of the state of the valid line. In the second example, the built-in sample() method is used to sample the covergroup ONLY when the valid flag is set. This will yield a performance improvement, especially if valid is infrequently true.

Assertion Performance Guidelines

The assertion syntax in SystemVerilog provides a very succinct and powerful way of describing temporal properties. However, this power comes with the potential to impact performance. Here are a number of key perforamnce guidelines for writing SystemVerilog assertions, they are also good general assertion coding guidelines

Unique Triggering

The condition that starts the evaluation of a property is checked every time it is sampled. If this condition is ambiguous, then an assertion could have multiple evaluations in progress, which will potentially lead to erroneous results and will definitely place a greater load on the simulator.

Lower Performance Version **Higher Performance Version**

```
property req_rsp;
  @(posedge clk);
  req |=>
  (req & ~rsp)[*2]
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

```
property req_rsp;
  @(posedge clk);
  $rose(req) |=>
  (req & ~rsp)[*2]
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

In the first example, the property will be triggered every time the req signal is sampled at a logic 1, this will lead to multiple triggers of the assertion. In the second example, the property is triggered on the rising edge of req which is a discrete event. Other strategies for ensuring that the triggering is unique is to pick unique events, such as states that are known to be only valid for a clock cycle.

Safety vs Liveness

A safety property is one that has a bound in time - e.g. 2 clocks after req goes high, rsp shall go high. A liveness property is not bound in time - e.g. rsp shall go high following req going high. When writing assertions it is important to consider the life-time of the check that is in progress, performance will be affected by assertions being kept in flight because there is no bound on when they complete. Most specifications should define some kind of time limit for something to happen, or there will be some kind of practical limit that can be applied to the property.

Lower Performance Version

```
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
```

Higher Performance Version

```
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
```

Assertion Guards

Assertions can be disabled using the iff(condition) guard construct. This makes sure that the property is only sampled if the condition is true, which means that it can be disabled using a state variable. This is particularly useful for filtering assertion evaluation during reset or a time when an error is deliberately injected. Assertions can also be disabled using the system tasks $assertoff() and $asserton(), these can be called procedurally from within SystemVerilog testbench code. These features can be used to manage overall performance by de-activating assertions when they are not valid or not required.

Lower Performance Version

```
property req_rsp;
  @(posedge clk);
  $(posedge req) |=>
  (req & ~rsp)[*1:2]
   ##1 (req && rsp)[->1]
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Higher Performance Version

```
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
```

Keep Assertions Simple

A common mistake when writing assertions is to try to describe several conditions in one property. This invariably results in code that is more complex than it needs to be. Then, if the assertion fails, further debug is required to work out why it failed. Writing properties that only check one part of the protocol is easier to do, and when they fail, the reason is obvious.

Avoid Using Pass And Fail Messages

SystemVerilog assertion syntax allows the user to add pass and fail function calls. Avoid the use of pass and fail messages in your code, string processing is expensive and the simulator will automatically generate a message if an assertion fails.

Lower Performance Version

```
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
```

Higher Performance Version

```
REQ_RSP: assert property(req_rsp) ;
```

Note that even leaving a blank begin ... end for the pass clause causes a performance hit.

Avoid Multiple Clocks

SystemVerilog assertions can be clocked using multiple clocks. The rules for doing this are quite complex and the performance of a multiply clocked sequence is likely to lower than a normal sequence. Avoid using multiple clocks in an assertion, if you find yourself writing one, then it may be a sign that you are approaching the problem from the wrong angle.

UVM/Guidelines

Mentor Graphics UVM Guidelines

UVM Do's

- Define classes within packages
- Define one class per file
- Use factory registration macros
- Use message macros
- Manually implement do_copy(), do_compare(), etc.
- Use sequence.start(sequencer)
- Use start_item() and finish_item() for sequence items
- Use the uvm_config_db API
- Use a configuration class for each agent
- Use phase objection mechanism
- Use the phase_ready_to_end() func
- Use the run_phase() in transactors
- Use the reset/configure/main/shutdown phases in tests

UVM Don'ts

- Avoid `including a class in multiple locations
- Avoid constructor arguments other than name and parent
- Avoid field automation macros
- Avoid uvm_comparer policy class
- Avoid the sequence list and default sequence
- Avoid the sequence macros (`uvm_do)
- Avoid pre_body() and post_body() in a sequence
- Avoid explicitly consuming time in sequences
- Avoid set/get_config_string/_int/_object()
- Avoid the uvm_resource_db API
- Avoid callbacks
- Avoid user defined phases
- Avoid phase jumping and phase domains (for now)
- Avoid raising and lowering objections for every transaction

The UVM library is both a collection of classes and a methodology for how to use those base classes. UVM brings clarity to the SystemVerilog language by providing a structure for how to use the features in SystemVerilog. However, in many cases UVM provides multiple mechanisms to accomplish the same work. This guideline document is here to provide some structure to UVM in the same way that UVM provides structure to the SystemVerilog language.

Mentor Graphics has also documented pure SystemVerilog Guidelines as well. Please visit the SV/Guidelines article for more information.

Class Definitions

1.1 Rule: Define all classes within a package with the one exception of Abstract/Concrete classes.

Define all classes within a package. Don't `include class definitions haphazardly through out the testbench. The one exception to defining all classes within a package is Abstract/Concrete classes. Having all classes defined in a package makes it easy to share class definitions when required. The other way to bring in class definitions into an UVM testbench is to try to import the class wherever it is needed. This has potential to define your class multiple times if the class is imported into two different packages, modules, etc. If a class is `included into two different scopes, then SystemVerilog states that these two classes are different types.

Abstract/Concrete classes are the exception because they must be defined within a module to allow them to have access to the scope of the module. The Abstract/Concrete class is primarily used when integrating verification IP written in Verilog or VHDL.

1.2 Rule: Define one class per file and name the file <CLASSNAME>.svh.

Every class should be defined in its own file. The file should be named <CLASSNAME>.svh. The file should then be included in another file which defines a package. All files included into a single package should be in the same directory. The package name should end in _pkg to make it clear that the design object is a package. The file that contains the class definition should not contain any import or include statements. This results in a file structure that looks like this:

example_agent/ <-- Directory containing Agent code

example_agent_pkg.sv
- example_item.svh
- example_config.svh
- example_driver.svh
- example_monitor.svh
- example_agent.svh
- example_api_seq1.svh
- reg2example_adapter.svh

With that list of files, the example_pkg.sv file would look like this:

```
// Begin example_pkg.sv file
`include "uvm_macros.svh"
package example_pkg;
  import uvm_pkg::*;
  import another_pkg::*;

  //Include any transactions/sequence_items
  `include "example_item.svh"
  //Include any configuration classes
  `include "example_config.svh"

  //Include any components
  `include "example_driver.svh"
  `include "example_monitor.svh"
  `include "example_agent.svh"

  //Include any API seqeunces
  `include "example_api_seq1.svh"

  //Include the UVM Register Layer Adapter
  `include "reg2example_adapter.svh"

endpackage : example_pkg
// End example_pkg.sv file
```

If one of the files that defines a class (example_driver.svh) contains a package import statement in it, it would be just like the import statement was part of the example package. This could result in trying to import a package multiple times which is inefficient and wastes simulation time.

1.3 Rule: Call super.new(name [,parent]) as the first line in a constructor.

Explicitly pass the required arguments to the super constructor. That way the intent is clear. Additionally, the constructor should only contain the call to super.new() and optionally any calls to new covergroups that are part of the class. The calls to new the covergroups must be done in the constructor according to the SystemVerilog LRM. All other objects should be built in the build_phase() function for components or the beginning of the body() task for sequences.

1.4 Rule: Don't add extra constructor arguments other than name and parent.

Extra arguments to the constructor will at worst case result in the UVM Factory being unable to create the object that is requested. Even if the extra arguments have default values which will allow the factory to function, the extra arguments will always be the default values as the factory only passes along the name and parent arguments.

Factory

The UVM Factory provides an easy, effective way to customize an environment without having to extend or modify the environment directly. To make effective use of the UVM Factory and to promote as much flexibility for reuse of code as possible, Mentor Graphics recommends following guidelines. For more information, refer to the Factory article.

2.1 Rule: Register and create all classes with the UVM Factory.

Registering all classes and creating all objects with the UVM Factory maximizes flexibility in UVM testbenches. Registering a class with the UVM Factory carries no run-time penalty and only slight overhead for creating an object via the UVM Factory. Classes defined by UVM are registered with the factory and objects created from those classes should be created using the UVM Factory. This includes classes such as the uvm_sequencer.

2.2 Rule: Import packages that define classes registered with the UVM Factory.

This guideline may seem obvious, but be sure to import all packages that have classes defined in them. If the package is not imported, then the class will not be registered with the UVM Factory. The most common place that this mistake is made is not importing the package that contains all the tests into the top level testbench module. If that test package is not imported, then UVM will not understand the definition of the test the call to run_test() attempts to create the test object.

2.3 Guideline: When creating an object with the UVM Factory, match the object's handle name with the string name passed into the create() call.

UVM builds an object hierarchy which is used for many different functions including UVM Factory overrides and configuration. This hierarchy is based on the string name that is passed into the first argument of every constructor. Keeping the handle name and this string hierarchy name the same will greatly aid in debug when the time comes. For more info, visit the Testbench/Build article.

Macros

The UVM library contains many different types of macros. Some of them do a very small, well defined job and are very useful. However, there are other macros which may save a small bit of time initially, but will ultimately cost more time down the road. This extra time is consumed in both debug and run time. For more information on this topic, please see the MacroCostBenefit article.

Factory Registration Macros

3.1 Rule: Use the `uvm_object_utils(), `uvm_object_param_utils(), `uvm_component_utils() and `uvm_component_param_utils() factory registration macros.

The factory registration macros provide useful, well defined functionality. As the name implies, these macros register the UVM object or component with the UVM factory which is both necessary and critical. These macros are `uvm_object_utils(), `uvm_object_param_utils(), `uvm_component_utils() and `uvm_component_param_utils(). When these macros are expanded, they provide the type_id typedef (which is the factory registration), the static get_type() function (which returns the object type), the get_object_type() function (which is not static and also returns the object type) and the create() function. Notice that the `uvm_sequence_utils() and `uvm_sequencer_utils() macros which also register with the factory are not recommended. Please see the starting sequences section for more information.

Message Macros

3.2 Guideline: Use the UVM message macros which are `uvm_info(), `uvm_warning(), `uvm_error() and `uvm_fatal().

The UVM message macros provide a performance savings when used. The message macros are `uvm_info(), `uvm_warning(), `uvm_error() and `uvm_fatal(). These macros ultimately call uvm_report_info, uvm_report_warning, etc. What these macros bring to the table are a check to see if a message would be filtered before expensive string processing is performed. They also add the file and line number to the message when it is output.

Field Automation Macros

3.3 Guideline: Do write your own do_copy(), do_compare(), do_print() , do_pack() and do_unpack() functions. Do not use the field automation macros.

The field automation Macros on the surface look like a very quick and easy way to deal with data members in a class. However, the field automation macros have a very large hidden cost. As a result, Mentor Graphics does not use or recommend using these macros. These macros include `uvm_field_int(), `uvm_field_object(), `uvm_field_array_int(), `uvm_field_queue_string(), etc. When these macros are expanded, they result in hundreds of lines of code. The code that is produced is not code that a human would write and consequently very difficult to debug even when expanded. Additionally, these macros try to automatically get information from the UVM resource database. This can lead to unexpected behavior in a testbench when someone sets configuration information with the same name as the field being registered in the macro.

Mentor Graphics does recommend writing your own do_copy(), do_compare(), do_print(), do_pack() and do_unpack() methods. Writing these methods out one time may take a little longer, but that time will be made up when running your simulations. For example, when writing your own do_compare() function, two function calls will be executed to compare (compare() calls do_compare()) the data members in your class. When using the macros, 45 function calls are executed to do the same comparison. Additionally, when writing do_compare(), do not make use of the uvm_comparer policy class that is passed in as an argument. Just write your comparison to return a "0" for a mis-compare and a "1" for a successful comparison. If additional information is needed, it can be displayed in the do_compare() function using the `uvm_info(), etc. macros. The uvm_comparer policy class adds a significant amount of overhead and doesn't support all types.

Sequences

UVM contains a very powerful mechanism for creating stimulus and controlling what will happen in a testbench. This mechanism, called sequences, has two major use models. Mentor Graphics recommends starting sequences in your test and using the given simple API for creating sequence items. Mentor Graphics does not recommend using the sequence list or a default sequence and does not recommend using the sequence macros. For more information, please refer to the Sequences article.

Starting Sequences

4.1 Rule: Do start your sequences using sequence.start(sequencer).

To start a sequence running, Mentor Graphics recommends creating the sequence in a test and then calling sequence.start(sequencer) in one of the run phase tasks of the test. Since start() is a task that will block until the sequence finishes execution, you can control the order of what will happen in your testbench by stringing together sequence.start(sequencer) commands. If two or more sequences need to run in parallel, then the standard SystemVerilog fork/join(_any, _none) pair can be used. Using sequence.start(sequencer) also applies when starting a child sequence in a parent sequence. You can also start sequences in the reset_phase(), configure_phase(), main_phase() and/or the shutdown_phase(). See Phasing below

Using Sequence Items

4.2 Rule: Do create sequence items with the factory. Do use start_item() and finish_item(). Do not use the UVM sequence macros (`uvm_do, etc.).

To use a sequence item in a sequence Mentor Graphics recommends using the factory to create the sequence item, the start_item() task to arbitrate with the sequencer and the finish_item() task to send the randomized/prepared sequence item to the driver connected to the sequencer.

Mentor Graphics does not recommend using the UVM sequence macros. These macros include the 18 macros which all begin with `uvm_do, `uvm_send, `uvm_create, `uvm_rand_send. These macros hide the very simple API of using sequence.start() or the combination of start_item() and finish_item() . Instead of remembering three tasks, 18 macros have to be known if they are used and instances still exist where the macros don't provide the functionality that is needed. See the MacroCostBenefit article for more information.

pre_body() and post_body()

4.3 Guideline: Do not use the pre_body() and post_body() tasks that are defined in a sequence.

Do not use the pre_body() and post_body() tasks that are defined in a sequence. Depending on how the sequence is called, these tasks may or may not be called. Instead put the functionality that would have gone into the pre_body() task into the beginning of the body() task. Similarly, put the functionality that would have gone into the post_body() task into the end of the body() task.

Sequence Data Members

4.4 Guideline: Make sequence input variables rand, output variables non-rand.

Making input variables rand allows for either direct manipulation of sequence control variables or for easy randomization by calling sequence.randomize(). Data members of the sequence that are intended to be accessed after the sequence has run (outputs) should not be rand as this just would waste processor resources when randomizing. See the Sequences/API article for more info.

Time Consumption

4.5 Rule: Sequences should not explicitly consume time.

Sequences should not have explicit delay statements (#10ns) in them. Having explicit delays reduces reuse and is illegal for doing testbench accelleration in an emulator. For more information on considerations needed when creating an emulation friendly UVM testbench, please visit the Emulation article.

Virtual Sequences

4.6 Guideline: When a virtual sequencer is used, virtual sequences should check for null sequencer handles before executing.

When a virtual sequencer is used, a simple check to ensure a sequencer handle is not null can save a lot of debug time later.

Phasing

UVM introduces a graph based phasing mechanism to control the flow in a testbench. There are several "build phases" where the testbench is configured and constructed. These are followed by "run-time phases" which consume time running sequences to cause the testbench to produce stimulus. "Clean-up phases" provide a place to collect and report on the results of running the test. See Phasing for more information.

5.1 Guideline: Transactors should only implement the run_phase(). They should not implement any of the other time consuming phases.

Transactors (drivers and monitors) are testbench executors. A driver should process whatever transactions are sent to it from the sequencer and a monitor should capture the transactions it observes on the bus regardless of the when the transactions occur. See Phasing/Transactors for more information.

5.2 Guideline: Avoid the usage of reset_phase(), configure_phase(), main_phase(), shutdown_phase() and the pre_/post_ versions of those phases.

The reset_phase(), configure_phase(), main_phase() or shutdown_phase() phases will be removed in a future version of UVM. Instead to provide sync points (which is what the new phases essentially add), use normal fork/join statements to run sequences in parallel or just start sequences one after another to have sequences run in series. This works because sequences are blocking. For more advanced synchronization needs, uvm_barriers, uvm_events, etc. can be used.

5.3 Guideline: Avoid phase jumping and phase domains (for now).

These two features of UVM phasing are still not well understood. Avoid them for now unless you are a very advanced user.

5.4 Guideline: Do not use user defined phases.

Do not use user defined phases. While UVM provides a mechanism for adding a phase in addition to the defaults (build_phase, connect_phase, run_phase, etc.), the current mechanism is very difficult to debug. Consider integrating and reusing components with multiple user defined phases and trying to debug problems in this scenario.

Configuration

6.1 Rule: Use the uvm_config_db API to pass configuration information. Do not use set/get_config_object(), set/get_config_string() or set/get_config_int() as these are deprecated. Also do not use the uvm_resource_db API.

UVM provides a mechanism for higher level components to configure lower level components. This allows a test to configure what a testbench will look like and how it will operate for a specific test. This mechanism is very powerful, but also can be very inefficient if used in an incorrect way. Mentor Graphics recommends using only the uvm_config_db API as it allows for any type and uses the component hierarchy to ensure correct scoping. The uvm_config_db API should be used to pass configuration objects to locations where they are needed. It should not be used to pass integers, strings or other basic types as it much easier for a name space collision to happen when using low level types.

The set/get_config_*() API should not be used as it is deprecated. The uvm_resource_db API should not be used due to quirks in its behavior.

6.2 Rule: Do create configuration classes to hold configuration values.

To provide configuration information to agents or other parts of the testbench, a configuration class should be created which contains the bits, strings, integers, enums, virtual interface handles, etc. that are needed. Each agent should have its own configuration class that contains every piece of configuration information used by any part of the agent. Using the configuration class makes it convenient and efficient to use a single uvm_config_db #(config_class)::set() call and is type-safe. It also allows for easy extension and modification if required.

6.3 Guideline: Don't use the configuration space for frequent communication between components.

Using the resource database for frequent communication between components is an expensive way to communicate. The component that is supposed to receive new configuration information would have to poll the configuration space which would waste time. Instead, standard TLM communication should be used to communicate frequent information changes between components. TLM communication does not consume any simulation time other than when a transaction is being sent.

Infrequent communication such as providing a handle to a register model is perfectly acceptible.

6.4 Rule: Pass virtual interface handles from the top level testbench module into the testbench by using the uvm_config_db API.

Since the uvm_config_db API allows for any type to be stored in the UVM resource database, this should be used for passing the virtual interface handle from the top level testbench module into the UVM test. This call should be of the form uvm_config_db #(virtual bus_interface)::set(null, "uvm_test_top", "bus_interface", bus_interface); Notice the first argument is null which means the scope is uvm_top. The second argument now lets us limit the scope of who under uvm_top can access this interface. We are limiting the scope to be the top level uvm_test as it should pull the virtual interface handles out of the resource database and then add them to the individual agent configuration objects as needed.

Coverage

7.1 Guideline: Place covergroups within wrapper classes extended from uvm_object.

Covergroups are not objects or classes. They can not be extended from and also can't be programatically created and destroyed on their own. However, if a covergroup is wrapped within a class, then the testbench can decide at run time whether to construct the coverage wrapper class.

Please refer to the SystemVerilog Guidelines for more on general covergroup rules/guidelines.

End of Test

8.1 Rule: Do use the phase objection mechanism to end tests. Do use the phase_ready_to_end() function to extend time when needed.

To control when the test finishes use the objection mechanism in UVM. Each UVM phase has an argument passed into it (phase) of type uvm_phase. Objections should be raised and dropped on this phase argument. For more information, visit the End of Test article.

8.2 Rule: Only raise and lower objections in a test. Do not raise and lower objections on a transaction by transaction basis.

In most cases, objections should only be raised and lowered in one of the time consuming phases in a test. This is because the test is the main controller of what is going to happen in the testbench and it therefore knows when all of the stimulus has been created and processed. If more time is needed in a component, then use the phase_ready_to_end() function to allow for more time. See Phasing/Transactors for more information.

Because there is overhead involved with raising and dropping an objection, Mentor Graphics recommends against raising and lowering objections in a driver or monitor as this will cause simulation slowdown due to the overhead involved.

Callbacks

9.1 Guideline: Do not use callbacks.

UVM provides a mechanism to register callbacks for specific objects. This mechanism should not be used as there are many convoluted steps that are required to register and enable the callbacks. Additionally callbacks have a non-negligible memory and performance footprint in addition to potential ordering issues. Instead use standard object oriented programming (OOP) practices where callback functionality is needed. One OOP option would be to extend the class that you want to change and then use the UVM factory to override which object is created. Another OOP option would be create a child object within the parent object which gets some functionality delegated to it. To control which functionality is used, a configuration setting could be used or a factory override could be used.

MCOW (Manual Copy on Write)

10.1 Guideline: If a transaction is going to be modified by a component, the component should make a copy of it before sending it on.

The general policy for TLM 1.0 (blocking put/get ports, analysis_ports, etc.) in SV is MCOW ("moocow"). This stands for manual copy on write. Once a handle has passed across any TLM port or export, it cannot be modified. If you want to modify it, you must manually take a copy and write to that. In C++ and other languages, this isn't a problem because a 'const' attribute would be used on the item and the client code could not modify the item unless it was copied. SV does not provide this safety feature, so it can be very dangerous.

Sequences and TLM 2.0 connections don't follow the same semantic as TLM 1.0 connections and therefore this rule doesn't apply.

Command Line Processor

11.1 Rule: Don't prefix user defined plusargs with "uvm_" or "UVM_".

User defined plusargs are reserved by usage by the UVM committee and future expansion. For more information, see the UVM/CommandLineProcessor article.

11.2 Guideline: Do use a company and/or group prefix to prevent namespace overlap when defining user defined plusargs.

This allows for easier sharing of IP between groups and potentially companies as it is less likely to have a collision with a plusarg name. For more information, see the UVM/CommandLineProcessor article.

UVM/Performance Guidelines

Although the UVM improves verification productivity, there are certain aspects of the methodology that should be used with caution, or perhaps not at all, when it comes to performance and scalability considerations.

During the simulation run time of a UVM testbench there are two distinct periods of activity. The first is the set of UVM phases that have to do with configuring, building and connecting up the testbench component hierarchy, the second is the run-time activity where all the stimulus and analysis activity takes place. The performance considerations for both periods of activity are separate.

These performance guidelines should be read in conjunction with the other methodology cookbook guidelines, there are cases where judgement is required to trade-off performance, re-use and scalability concerns.

UVM Testbench Configuration and Build Performance Guidelines

The general requirement for the UVM testbench configuration and build process is that it should be quick so that the run time phases can get going. With small testbenches containing only a few components - e.g. up to 10, the build process should be short, however when the testbench grows in size beyond this then the overhead of using certain features of the UVM start to become apparent. With larger testbenches with 100s, or possibly 1000s of components, the build phase will be noticeably slower. The guidelines in this section apply to the full spectrum of UVM testbench size, but are most likely to give most return as the number of components increases.

Avoid auto-configuration

Auto-configuration is a methodology inherited from the OVM where a component's configuration variables are automatically set to their correct values from variables that have been set up using set_config_int(), set_config_string(), uvm_config_db #(..)::set() etc at a higher level of the component hierarchy. In order to use auto-configuration, field macros are used within a component and the super.build_phase() method needs to be called during the build_phase(), the auto-configuration process then attempts to match the fields in the component with entries in the configuration database via a method in uvm_component called apply_config_settings(). From the performance point of view, this is VERY expensive and does not scale.

Lower Performance Version

```
class my_env extends uvm_component;
```

```
bit has_axi_agent;
bit has_ahb_agent;
string system_name;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

// Required for auto-configuration
`uvm_component_utils_begin(my_env)
  `uvm_field_int(has_axi_agent, UVM_DEFAULT)
  `uvm_field_int(has_ahb_agent, UVM_DEFAULT)
  `uvm_field_string(system_name, UVM_DEFAULT)
`uvm_component_utils_end

function new(string name = "my_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
  super.build_phase(phase); // Auto-configuration called here
  if(has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
  `uvm_info("build_phase", $sformatf("%s built", system_name))
endfunction: build_phase

endclass: my_env
```

Higher Performance Version

```
class my_env extends uvm_component;

my_env_config cfg;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

`uvm_component_utils(my_env)

function new(string name = "my_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
  // Get the configuration, note class variables not required
  if(!uvm_config_db #(my_env_config)::get(this, "", "my_env_config", cfg)) begin
    `uvm_error("build_phase", "Unable to find my_env_config in uvm_config_db")
  end
  if(cfg.has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(cfg.has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
  `uvm_info("build_phase", $sformatf("%s built", cfg.system_name))
endfunction: build_phase

endclass: my_env
```

The recommended practice is not to use field macros in a component, and to not call super.build_phase() if the class you are extending is from a UVM component base class such as uvm_component. Even then, when a component does not have a build_phase() method implementation, the default build_phase() from the uvm_component base class will be called which will attempt to do auto-configuration. In UVM 1.1b, a fix was added that stops the apply_config_settings() method from continuing if there are no field macros in the component, this speeds up component build, but it is more efficient to avoid this method being called altogether.

Minimize the use of the uvm_config_db

The uvm_config_db is a database, as with any database it takes longer to search as it grows in size. The uvm_config_db is based on the uvm_resource and the uvm_resource_db classes. The uvm_resource_db uses regular expressions and the component hierarchy strings to make matches, it attempts to check every possible match and then returns the one that is closest to the search, this is expensive and the search time increases exponentially as the database grows. Therefore, the uvm_config_db should be used sparingly, if at all. This also applies to the set/get_config_xxx() methods since they in turn are based on the uvm_config_db.

Use configuration objects to pass configuration data to components

One way to minimize the number of uvm_config_db entries is to group component configuration variables into a configuration object. That way only one object needs to be set() in the uvm_config_db. This has reuse benefits and is the recommended way to configure reusable verification components such as agents.

Lower Performance Version

```
class static_test extends uvm_test;

// Test that builds an env containing an AXI agent

virtual axi_if AXI; // Used by the AXI agent

// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  uvm_config_db #(virtual axi_if)::set(this, "env.axi_agent*", "v_if", AXI);
  uvm_config_db #(uvm_active_passive_enum)::set(
                    this, "env.axi_agent*", "is_active", UVM_ACTIVE);
  uvm_config_db #(int)::set(this, "env.axi_agent*", "max_burst_size", 16);
  // Other code

endfunction: build_phase

endclass: static_test

// The AXI agent:
```

```
class axi_agent extends uvm_component;

// Configuration parameters:
virtual axi_if AXI;
uvm_active_passive_enum is_active;
int max_burst_size;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  if(!uvm_config_db #(uvm_active_passive_enum)::get(this,
                                      "", "is_active", is_active)) begin
    `uvm_error("build_phase", "is_active not found in uvm_config_db")
  end
  if(!uvm_config_db #(int)::get(
                    this, "", "max_burst_size", max_burst_size)) begin
    `uvm_error("build_phase", "max_burst_size not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  monitor.AXI = AXI;
  if(is_active == UVM_ACTIVE) begin
    driver.AXI = AXI;
    driver.max_burst_size = max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

Higher Performance Version

```
// Additional agent configuration class:
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;

function new(string name = "axi_agent_config");
  super.new(name);
endfunction

endclass: axi_agent_config

class static_test extends uvm_test;

// Test that builds an env containing an AXI agent
```

```
axi_agent_config axi_cfg; // Used by the AXI agent

// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  axi_cfg = axi_agent_config::type_id::create("axi_cfg");
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI",
                                           axi_cfg.AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  axi_cfg.is_active = UVM_ACTIVE;
  axi_cfg.max_burst_size = 16;
  uvm_config_db #(axi_agent_config)::set(this,
                     "env.axi_agent*", "axi_agent_config", axi_cfg);
  // Other code

endfunction: build_phase

endclass: static_test

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object not
                                        found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  monitor.AXI = cfg.AXI;
  if(cfg.is_active == UVM_ACTIVE) begin
    driver.AXI = cfg.AXI;
    driver.max_burst_size = cfg.max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

The higher performance version of the example uses one uvm_config_db #(...)::set() call and two get() calls, compared with three set() and four get() calls in the lower performance version. There are also just two uvm_config_db entries compared to four. With a large number of components, this form of optimization can lead to a considerable performance boost.

Minimize the number of uvm_config_db #(...)::get() calls

The process of doing a get() from the uvm_config_db is expensive, and should only be used when really necessary. For instance, in an agent, it is only really necessary to get() the configuration object at the agent level and to assign handles to the sub-components from there. It is an unecessary overhead to have separate get() calls inside the driver and monitor components.

Lower Performance Version

```
// Agent configuration class - configured and set() by the test class
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;

function new(string name = "axi_agent_config");
  super.new(name);
endfunction

endclass: axi_agent_config

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                    "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                            not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

endclass: axi_agent

// The axi monitor:
class axi_monitor extends uvm_component;

axi_if AXI;
axi_agent_config cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                     "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                              not found in uvm_config_db")
  end
  AXI = cfg.AXI;
endfunction: build_phase

endclass: axi_monitor

// The axi driver:
class axi_monitor extends uvm_component;

axi_if AXI;
int max_burst_size;
axi_agent_config cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                       "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                              not found in uvm_config_db")
  end
  AXI = cfg.AXI;
  max_burst_size = cfg.max_burst_size;
endfunction: build_phase

endclass: axi_driver
```

Higher Performance Version

```
// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                       "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                               not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

// Direct assignment to the monitor and driver variables
// from the configuration object variables.
function void connect_phase(uvm_phase phase);
  monitor.AXI = cfg.AXI;
  if(cfg.is_active == UVM_ACTIVE) begin
    driver.AXI = cfg.AXI;
    driver.max_burst_size = cfg.max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

```
// The axi_monitor and axi_driver are implemented without
// a uvm_config_db #()::get()
```

The higher performance version has two fewer calls to the uvm_config_db #()::get() method, which when multiplied by a large number of components can lead to a performance improvement.

Use specific strings with the uvm_config_db set() and get() calls

The regular expression algorithm used in the search attempts to get the closest match based on the UVM component's position in the testbench hierarchy and the value of the key string. If wildcards are used in either the set() or get() process, then this adds ambiguity to the search and makes it more expensive. For instance, setting the context string to "*" means that the entire component hierarchy will be searched for uvm_config_db settings before a result is returned.

Lower Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "*", "*_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "*_config", cfg)) begin
  `uvm_error(...)
end
```

Higher Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "env.sb", "sb_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "sb_config", cfg)) begin
  `uvm_error(...)
end
```

In the higher performance version of this code, the scope is very specific and will only match on the single component for a single key, this cuts downs the search time in the uvm_config_db

Minimise the number of virtual interface handles passed via uvm_config_db from the TB module to the UVM environment

Using the testbench package to pass virtual interface handles

One common application of the uvm_config_db is to use it to pass virtual interfaces into the testbench. An alternative is to use a package which is a common namespace for the top level testbench module and the UVM test class. This helps minimize the number of entries in the uvm_config_db.

Lower Performance Version

```
// In the top level testbench module:
module top_tb;

import uvm_pkg::*;
import test_pkg::*;

// Instantiate the static interfaces:
axi_if AXI();
ddr2_if DDR2();

// Hook up to DUT ....

// UVM initial block:
initial begin
  uvm_config_db #(virtual axi_if)::set("uvm_test_top", "", "AXI", AXI);
  uvm_config_db #(virtual ddr2_if)::set("uvm_test_top", "", "DDR2", DDR2);
  run_test();
end

endmodule: top_tb

// In the test class - inside test package that imports tb_if_pkg
class static_test extends uvm_component;

axi_agent_config axi_cfg;
ddr2_agent_config ddr2_cfg;

function void build_phase(uvm_phase phase);
  axi_cfg = axi_agent_config::type_id::create("axi_cfg");
  if(!uvm_config_db #(virtual axi_if)::get(this,
                                           "", "AXI", axi_cfg.AXI)) begin
    `uvm_error("build_phase", "AXI virtual interface
                                              not found in uvm_config_db")
  end
  // Rest of configuration ....
  ddr2_cfg = ddr2_agent_config::type_id::create("ddr2_cfg");
  if(!uvm_config_db #(virtual ddr2_if)::get(this,
                                         "", "DDR2", ddr2_cfg.DDR2)) begin
    `uvm_error("build_phase", "DDR2 virtual interface
                                              not found in uvm_config_db")
  end
  // ...
endfunction: build_phase

endclass: static_test
```

Higher Performance Version

```
// Testbench interface package:
package tb_if_pkg;

virtual axi_if AXI;
virtual ddr2_if DDR2;

endpackage: tb_if_pkg;

// In the top level testbench module:
module top_tb;

import uvm_pkg::*;
import test_pkg::*;
import tb_if_pkg::*; // Contains virtual interface handles

// Instantiate the static interfaces:
axi_if AXI();
ddr2_if DDR2();

// Hook up to DUT ....

// UVM initial block:
initial begin
  tb_if_pkg::AXI = AXI;
  tb_if_pkg::DDR2 = DDR2;
  run_test();
end

endmodule: top_tb

// In the test class - inside test package that imports tb_if_pkg
class static_test extends uvm_component;

axi_agent_config axi_cfg;
ddr2_agent_config ddr2_cfg;

function void build_phase(uvm_phase phase);
  axi_cfg = axi_agent_config::type_id::create("axi_cfg");
  axi_cfg.AXI = tb_if_pkg::AXI;
  // Rest of configuration
  ddr2_cfg = ddr2_agent_config::type_id::create("ddr2_cfg");
  ddr2_cfg.DDR2 = tb_if_pkg::DDR2;
  // ...
endfunction: build_phase

endclass: static_test
```

The second, higher performance, example shows how to use a shared package to pass the virtual interface handles from the top level testbench module to the UVM test class. In this case this approach saves a pair of uvm_config_db set() and get() calls, and also eliminates an entry from the uvm_config_db for each virtual interface handle. When dealing with large numbers of virtual interfaces, this can result in a substantial performance improvement. If there is an existing package that is used to define other testbench configuration parameters such as bus field widths then it can be extended to include the virtual interface handles.

Consolidate the virtual interface handles into a single configuration object

An alternative to using a package to pass the virtual interface handles from the testbench top level module to the UVM test, is to create a single configuration object that contains all the virtual interface handles and to make the virtual interface assignments in the top level module before setting the configuration object in the uvm_config_db. This reduces the number of uvm_config_db entries used for passing virtual interface handles down to one.

```
// Virtual interface configuration object:
class vif_handles extends uvm_object;
`uvm_object_utils(vif_handles)

virtual axi_if AXI;
virtual ddr2_if DDR2;

endclass: vif_handles

// In the top level testbench module:
module top_tb;

import uvm_pkg::*;
import test_pkg::*;

// Instantiate the static interfaces:
axi_if AXI();
ddr2_if DDR2();

// Virtual interface handle container object:
vif_handles v_h;

// Hook up to DUT ....

// UVM initial block:
initial begin
  // Create virtual interface handle container:
  v_h = vif_handles::type_id::create("v_h");
  // Assign handles
  v_h.AXI = AXI;
  v_h.DDR2 = DDR2;
  // Set in uvm_config_db:
  uvm_config_db #(vif_handles)::set("uvm_test_top", "", "V_H", vh);
  run_test();
end

endmodule: top_tb
```

Pass configuration information through class hierarchical references

The ultimate in minimizing the use of the uvm_config_db is not to use it at all. It is perfectly possible to pass handles to configuration objects through the class hierarchy at build time. Using handle assignments is the most efficient way to do this.

Lower Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  uvm_config_db #(env_config_object)::set(this, "env", "env_cfg", env_cfg);
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(env_config_object)::get(this,
                                             "", "env_cfg", cfg)) begin
    `uvm_error(...)
  end
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
  uvm_config_db #(axi_agent_config)::set(this,
                                "axi", "axi_agent_config", cfg.axi_cfg);
  // Also for the DDR2 agent:
  ddr2 = ddr2_agent::type_id::create("ddr2", this);
  uvm_config_db #(ddr2_agent_config)::set(this,
                             "ddr2", "ddr2_agent_config", cfg.ddr2_cfg);
  // etc
endfunction: build_phase
```

Higher Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  // Assign the env configuration object handle directly:
  env.cfg = env_cfg;
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
  // Assign axi agents configuration handle directly:
  axi.cfg = cfg.axi_cfg;
  // Also for the DDR2 agent:
  ddr2 = ddr2_agent::type_id::create("ddr2", this);
  ddr2.cfg = cfg.ddr2_cfg;
  // etc
endfunction: build_phase
```

The higher performance example avoids the use of the uvm_config_db altogether, providing the ultimate configuration and build performance enhancement. The impact of using this approach is that it requires the assignments to be chained together; that it requires the agent code to test for a null config object handle before attempting to get the configuration object handle; and that any stimulus hierarchy needs to take care of getting handles to testbench resources such as register models.

Another major consideration with this direct approach to the assignment of configuration object handles is that if VIP is being re-used, it may well be implemented with the expectation that its configuration object will be set in the uvm_config_db. This means that there may have to be some use of the uvm_config_db to support the reuse of existing VIP.

Minimize the use of the UVM Factory

The UVM factory is there to allow UVM components or objects to be overriden with derived objects. This is a powerful technique, but whenever a component is built a lookup has to be made in a table to determine which object type to construct. If there are overrides, this lookup becomes more complicated and there is a performance penalty. Try to manage the factory overrides used to reduce the overhead of the lookup.

UVM Testbench Run-Time Performance Guidelines

The guidelines presented here represent areas of the UVM which have been seen to cause performance issues during the run-time phases of the testbench, they are mostly concerned with stimulus generation. There are other SystemVerilog coding practices that can also followed to enhance run-time performance and these are described in the SystemVerilog Performance Guidelines article.

Avoid polling the uvm_config_db for changes

Do not use the uvm_config_db to communicate between different parts of the testbench, for instance by setting a new variable value in one component and getting it inside a poll loop in another. It is far more efficient for the two components to have a handle to a common object and to reference the value of the variable within the object.

Lower Performance Version

```
// In a producer component setting the value inside a loop:
int current_id = 0;

forever begin
  // Lots of work making a transfer occur
  // Communicate the current id:
  uvm_config_db #(int)::set(null, "*", "current_id", current_id);
  current_id++;
end

// In a consumer component looking out for the current_id value
int current_id;

forever begin
   uvm_config_db #(int)::wait_modified(this, "*", "current_id");
   if(!uvm_config_db #(int)::get(this,
                           "", "current_id", current_id)) begin
     `uvm_error( ....)
   end
   // Lots of work to track down a transaction with the current_id
end
```

Higher Performance Version

```
// Config object containing current_id field:
packet_info_cfg pkt_info =
                       packet_info_cfg::type_id::create("pkt_info");
// This created in the producer component and the consumer component
// has a handle to the object:

// In the producer component:
forever begin
  // The work resulting in a current_id update
  pkt_info.current_id = current_id;
  current_id++;
end

// In the consumer component:
forever begin
  @(pkt_info.current_id);
  // Start working with the new id
end
```

The principle at work in the higher performance version is that the current_id information is inside an object. Both the consumer and the producer components share the handle to the same object, therefore when the producer object makes a change to the current_id field, it is visible to the consumer component via the handle. This avoids the use of repeated set() and get() calls in the uvm_config_db and also the use of the expensive wait_modified() method.

Do not use the UVM field macros in transactions

The UVM field macros may seem like a convenient way to ensure that the various do_copy(), do_compare() methods get implemented, but this comes at a heavy cost in terms of performance. This becomes very evident if your testbench starts to use sequence_items heavily.

Lower Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

// Field macros:
`uvm_object_utils_begin(apb_seq_item)
  `uvm_field_int(addr, UVM_DEFAULT)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_enum(we, apb_opcode_e, UVM_DEFAULT)
`uvm_object_utils_end

function new(string name = "apb_seq_item");
  super.new(name);
endfunction

endclass: apb_seq_item
```

Higher Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

`uvm_object_utils(apb_seq_item)

function new(string name = "apb_seq_item");
  super.new(name);
endfunction

// Sequence Item convenience method prototypes:
extern function void do_copy(uvm_object rhs);
extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
extern function string convert2string();
extern function void do_print(uvm_printer printer);
extern function void do_record(uvm_recorder recorder);
extern function void do_pack();
extern function void do_unpack();
endclass: apb_seq_item
```

Although the lower performance code example looks more compact, compiling with an -epretty flag will reveal that they expand out into many lines of code. The higher performance example shows the templates for the various uvm_object convenience methods which should be implemented manually, this will always improve performance and enhance debug should you need it.

The definitive guide on the trade-offs involved in using or not using these and the various other UVM macros can be found here.

Minimise factory overrides for stimulus objects

The UVM factory can be used to override or change the type of object that gets created when a object handle's ::type_id::create() method is called. During stimulus generation this could be applied to change the behavior of a sequence or a sequence_item without rewriting the testbench code. However, this override capability comes at a cost in terms of an extended lookup in the factory each time the object is created. To reduce the impact of creating an object overridden in the factory, create the object once and then clone it each time it is used to avoid using the factory.

Lower Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item item;

  repeat(200) begin
    item = apb_seq_item::type_id::create("item");
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

Higher Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item original_item = apb_seq_item::type_id::create("item");
  apb_seq_item item;

  repeat(200) begin
    $cast(item, original_item.clone());
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

The higher performance example only makes one factory create call, and uses clone() to create further copies of it, so saving the extended factory look-up each time that is expended each time round the generation loop in the lower performance example.

Avoid embedding covergroups in transactions

Embedding a covergroup in a transaction adds to its memory footprint, it also does not make sense since the transaction is disposible. The correct place to collect coverage is in a component. Transactional coverage can be collected by sampling a covergroup in a component based on transaction content.

Lower Performance Version

```
// APB Sequence item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we

covergroup register_space_access_cg;

ADDR_RANGE: coverpoint addr[7:0];
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

function void sample();
  register_space_access_cg.sample();
endfunction: sample

// Rest of the sequence item ...

endclass: apb_seq_item

// Sequence producing the sequence item:
class bus_access_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item apb_item = apb_seq_item::type_id::create("apb_item");

  repeat(200) begin
    start_item(apb_item);
    assert(apb_item.randomize());
    apb_item.sample();
    finish_item(apb_item);
  end
endtask: body
```

Higher Performance Version

```
// apb_seq_item implemented without a covergroup
// Therefore bus_access_seq does not sample covergroup
// Sampling coverage in the driver:
class apb_coverage_driver extends apb_driver;

covergroup register_space_access_cg() with
                        function sample(bit[7:0] addr, apb_opcode_e we);
```

```
ADDR_RANGE: coverpoint addr;
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

task run_phase(uvm_phase phase);
  apb_seq_item apb_item;

  forever begin
    seq_item_port.get(apb_item);
    register_space_access_cg.sample(apb_item.addr[7:0], apb_item.we);
    // Do the signal level APB cycle
    seq_item_port.item_done();
  end

endtask: run_phase

// ....
endclass: apb_coverage_driver
```

The lower performance example shows the use of a covergroup within a transaction to collect input stimulus functional coverage information. This adds a memory overhead to the transaction that is avoided by the higher performance example which collects coverage in a static component based on the content of the transaction.

Use the UVM reporting macros

The raw UVM reporting methods do not check the verbosity of the message until all of the expensive string formating operations in the message assembly have completed. The `uvm_info(), `uvm_warning(), `uvm_error(), and `uvm_fatal() macros check the message verbosity first and then only do the string formatting if the message is to be printed.

Lower Performance Version

```
function void report_phase(uvm_phase phase);
if(errors != 0) begin
  uvm_report_error("report_phase", $sformatf(
                 %0d errors found in %0d transfers", errors, n_tfrs));
end
else if(warnings != 0) begin
  uvm_report_warning("report_phase", $sformatf(
           "%0d warnings issued for %0d transfers", warnings, n_tfrs));
end
else begin
  uvm_report_info("report_phase", $sformatf(
                              "%0d transfers with no errors", n_tfrs));
end
endfunction: report_phase
```

Higher Performance Version

```
function void report_phase(uvm_phase phase);
if(errors != 0) begin
  `uvm_error("report_phase", $sformatf(
                  "%0d errors found in %0d transfers", errors, n_tfrs))
end
else if(warnings != 0) begin
  `uvm_warning("report_phase", $sformatf(
             "%0d warnings issued for %0d transfers", warnings, n_tfrs))
end
else begin
  `uvm_info("report_phase", $sformatf(
                     "%0d transfers with no errors", n_tfrs), UVM_MEDIUM)
end
endfunction: report_phase
```

In the example shown, the same reports would be generated in each case, but if the verbosity settings are set to suppress the message, the higher performance version would check the verbosity before generating the strings. In a testbench where there are many potential messages and the reporting verbosity has been set to low, this can have a big impact on performance, especially if the reporting occurs frequently.

Do not use the uvm_printer class

The uvm_printer is a convenience class, originally designed to go with the use of field macros in order to print out component hierarchy or transaction content in one of several formats. The class comes with a performance overhead and its use can be avoided by using the convert2string() method for objects. The convert2string() method returns a string that can be displayed or printed using the UVM messaging macros.

Lower Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  bus_req.print();
end
```

Higher Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  `uvm_info("BUS_SEQ", bus_req.convert2string(), UVM_DEBUG)
end
```

Note also that the print() method calls $display() without checking verbosity settings.

Avoid the use of get_xxx_by_name() in UVM register code

Using the get_field_by_name(), or the get_register_by_name() functions involves a regular expression search of all of the register field name or register name strings in the register model to return a handle to a field or a register. As the register model grows, this search will become more and more expensive.

Use the hierarchical path within the register model to access register content, it is far more efficient as well as being a good way to make register based stimulus reusable.

Lower Performance Version

```
task set_txen_field(bit[1:0] value);
  uvm_reg_field txen;

  txen = rm.control.get_field_by_name("TXEN");
  txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

Higher Performance Version

```
task set_txen_field(bit[1:0] value);
  rm.control.txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

The higher performance version of the set_txen_field avoids the expensive regular expression lookup of the field's name string.

Minimize the use of get_registers() or get_fields() in UVM register code

These calls, and others like them return return queues of object handles, this is for convenience since a queue is an unsized array. Calling these methods requires the queue to be populated which can be an overhead if the register model is a reasonable size. Repeated calls of these methods is pointless, they should only need to be called once or twice within a scope.

Lower Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;

repeat(200) begin
  regs = rm.encoder.get_registers();
  no_regs = regs.size();
  repeat(no_regs) begin
    tassert(this.randomize() with {idx =< no_regs;});
    assert(regs[idx].randomize());
    regs[idx].update();
  end
end
```

Higher Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;

regs = rm.encoder.get_registers();
repeat(200) begin
  regs.shuffle();
  foreach(regs[i]) begin
    assert(regs[i].randomize());
    regs[i].update();
  end
end
```

The higher performance version of the code only does one get_registers() call and avoids the overhead associated with the repeated call in the lower performance version.

Use UVM objections, but wisely

The purpose of raising a UVM objection is to prevent a phase from completing until a thread is ready for it to complete. Raising and dropping objections causes the component hierarchy to be traversed, with the objection being raised or dropped in all the components all the way to the top of the hierarchy. Therefore, raising and lowering an objection is expensive, becoming more expensive as the depth of the testbench hierarchy increases.

Objections should only be used by controlling threads, and the proper place to put objections is either in the run-time method of the top level test class, or in the body method of a virtual sequence. Using them in any other place is likely to be unecessary and also cause a degradation in performance.

Lower Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
task body;
  uvm_objection objection = new("objection");
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    objection.raise_objection(this);
    finish_item(item);
    objection.drop_objection(this);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  do_adpcm.start(ADPCM);
endtask
```

Higher Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
task body;
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    finish_item(item);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  uvm_objection objection = new("objection");
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  objection.raise_objection(ADPCM);
  do_adpcm.start(ADPCM);
  objection.drop_objection(ADPCM);
endtask
```

In the higher performance version of the code, the objection is raised at the start of the sequence and dropped at the end, bracketing in time all the sequence_items sent to the driver, this is far more efficient than raising an objection per sequence_item.

Minimize the use of UVM call-backs

The implementation of call-backs in the UVM is expensive both in terms of the memory used and the code associated with registering and executing them. The complications arise mainly from the fact that the order in which the call-backs are registered is preserved. For performance, avoid the use of UVM call-backs by using alternative approaches to achieve the same functionality.

For example, register accesses can be recorded and viewed using transaction viewing either by extending the uvm_reg class or by using a call-back class. The class extended from uvm_reg overloads the pre_read() and pre_write() methods to begin a transaction when a register read() or write() method is called, and overloads the post_read() and the post_write() methods to end the transaction when the register transfer has completed. This will result in a transaction being recorded for each register access, provided the extended class is used as the base class for the register model. The alternative is to use a uvm_reg_cbs class which contains call-backs for the uvm_reg pre_read(), pre_write(), post_read() and post_write() methods. As with the extended class, the pre_xxx() methods start recording a transaction and the post_xxx() methods end recording transactions. A call back class object is then registered for each register using the package function enable_reg_recording().

Lower Performance Version Using Call Backs

```
//
// Call-Back class for recording register transactions:
//
class record_reg_cb extends uvm_reg_cbs;

  virtual      task pre_write(uvm_reg_item rw);

  endtask

  virtual task post_write(uvm_reg_item rw);

  endtask

  virtual task pre_read(uvm_reg_item rw);

  endtask

  function void do_record(uvm_recorder recorder);

  endfunction

endclass : record_reg_cb

//
// Package function for enabling recording:
//
function void enable_reg_recording(uvm_reg_block reg_model,
                           reg_recording_mode_t record_mode = BY_FIELD);
  uvm_reg regs[$];
  record_reg_cb reg_cb;

  //Set the recording mode
  uvm_config_db #(reg_recording_mode_t)::set(
                           null,"*","reg_recording_mode", record_mode);

  //Get the queue of registers
  reg_model.get_registers(regs);

  //Assign a callback object to each one
  foreach (regs[ii]) begin
    reg_cb = new({regs[ii].get_name(), "_cb"});
    uvm_reg_cb::add(regs[ii], reg_cb);
  end

  reg_cb = null;

  uvm_reg_cb::display();

endfunction : enable_reg_recording
```

Higher Performance Version Using Class Extension

```
//
// Extension of uvm_reg enables transaction recording
//
class record_reg extends uvm_reg;

  virtual      task pre_write(uvm_reg_item rw);

  endtask
  virtual task post_write(uvm_reg_item rw);

  endtask

  virtual task pre_read(uvm_reg_item rw);

  endtask

  function void do_record(uvm_recorder recorder);

  endfunction

endclass : record_reg
```

The main argument for using call-backs in this case is that it does not require that the register model be used with the extended class, which means that it can be 'retro-fitted' to a register model that uses the standard UVM uvm_reg class. However, this comes at the cost of a significant overhead - there is an additional call-back object for each register in the register model and the calling of the transaction recording methods involves indirection through the UVM infrastructure to call the methods within the call-back object. Given that a register model is likely to be generated, and that there could be thousands of registers in larger designs then using the extended record_reg class will deliver higher performance with minimum inconvenience to the user.

For the latest product information, call us or visit: www.mentor.com

Corporate Headquarters	**Silicon Valley**	**Europe**	**Pacific Rim**	**Japan**
Mentor Graphics Corporation	**Mentor Graphics Corporation**	**Mentor Graphics**	**Mentor Graphics (Taiwan)**	**Mentor Graphics Japan Co., Ltd.**
8005 SW Boeckman Road	46871 Bayside Parkway	**Deutschland GmbH**	Room 1001, 10F	Gotenyama Garden
Wilsonville, OR 97070-7777	Fremont, California 94538 USA	Arnulfstrasse 201	International Trade Building	7-35, Kita-Shinagawa 4-chome
Phone: 503.685.7000	Phone: 510.354.7400	80634 Munich	No. 333, Section 1, Keelung Road	Shinagawa-Ku, Tokyo 140-0001
Fax: 503.685.1204	Fax: 510.354.1501	Germany	Taipei, Taiwan, ROC	Japan
Sales and Product Information	**North American Support Center**	Phone: +49.89.57096.0	Phone: 886.2.87252000	Phone: +81.3.5488.3030
Phone: 800.547.3000	Phone: 800.547.4303	Fax: +49.89.57096.400	Fax: 886.2.27576027	Fax: +81.3.5488.3021