SIEMENS

Home > Verification Methodology Cookbooks > UVM

SystemVerilog Performance Guidelines

These guidelines are aimed at enabling you to identify coding idioms that are likely to affect testbench performance.

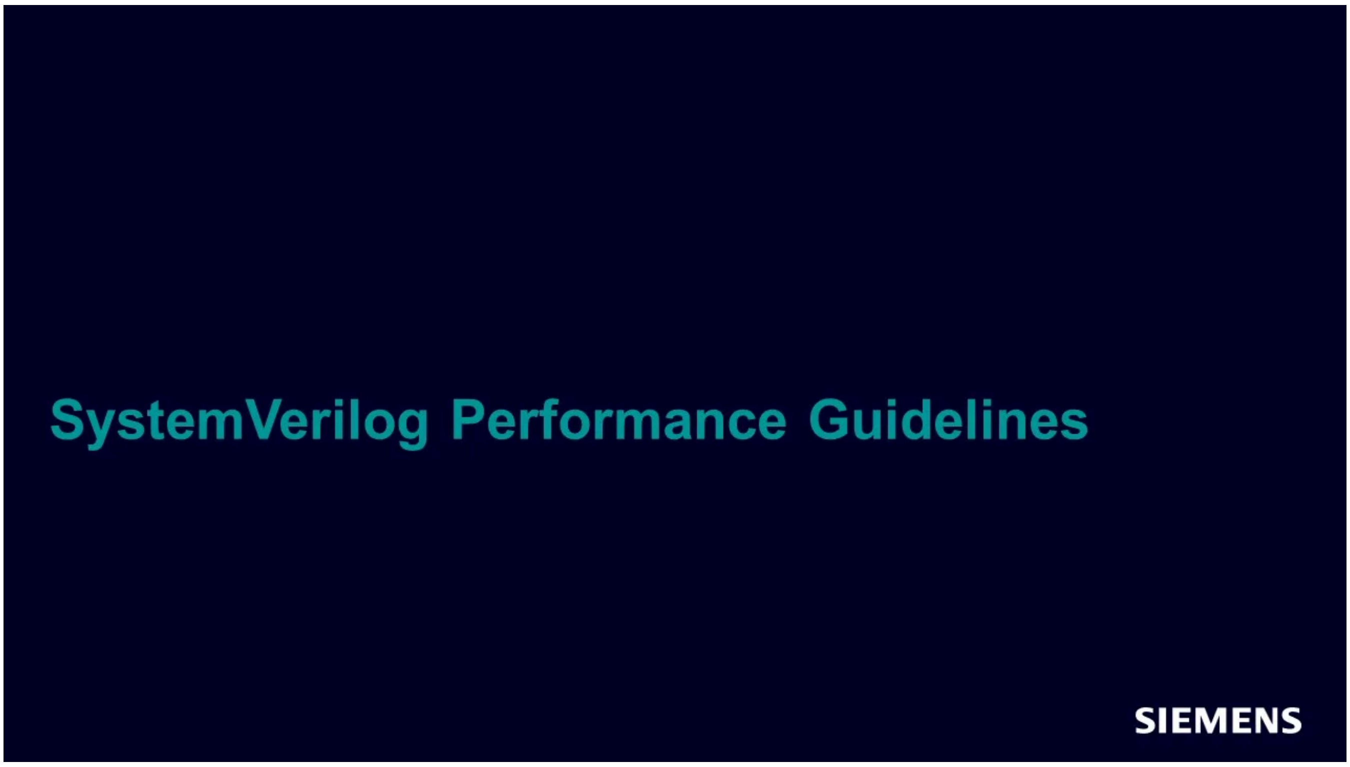
UVM - Universal Verification Methodology

Verification Methodology Team

Last Updated Mar 2014

UVM, SystemVerilog, Appendix, Guidelines, Performance, Beginner

Mark as viewed



se note that a number of these guidelines run counter to other recommended coding ctices 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 realizing 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 analyzed
- Random seeding - causing different levels of activity in the testbench
- Dominant behavior 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 analyzing 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 unnecessary, 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 optimizations 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 necessary.

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 Version

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

Higher Performance Version

```
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 optimizations. 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 minimized by the following guidelines:

Avoid Unnecessary Object Construction

Constructing an object can have an overhead associated with it. As a general rule, try to minimize 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;

functionvoid set_A(int value);
  A= value;
endfunction: set_A

function int get_A();
 return A;
endfunction: get_A

endclass: any_thing

// code that uses this class
// and its access methods

any_thing m;
int V;

initial begin
  m=new();
  V=1;
  repeat(10) begin
    m.set_A(V);
    V= V+ m.get_A();
 end
end
```

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 reusable 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 minimize the number of levels of methods involved.

Lower Performance Version

```
class mailbox_e#(type T=int);
 localmailbox#(T) mb;

 // standard mailbox API
```

```
  extern function new(int bound=0);
  extern function int num();
  extern task put(T item);
  extern function int try_put(T item);
  extern task get(ref T item);
  extern function int try_get(ref T item);
  extern function int try_peek(ref T item);

  // extended API
  extern function void reset();
endclass: mailbox_e

function mailbox_e::new(int bound=0);
  mb = new(bound);
endfunction

function int mailbox_e::num();
 return mb.num();
endfunction: num

task mailbox_e::put(T item);
  mb.put(item);
endtask: put

function int mailbox_e::try_put(T item);
 return mb.try_put(item);
endfunction: try_put

task mailbox_e::get(ref T item);
  mb.get(item);
endtask:get

function int mailbox_e::try_get(ref T item);
 return mb.try_get(item);
endfunction: try_get

function int mailbox_e::try_peek(ref T item);
 return mb.try_peek(item);
endfunction: try_peek

function void mailbox_e::reset();
  T obj;
 while (mb.try_get(obj));
endfunction: reset
```

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

```
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
```

Higher Performance Version

```
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 summarizes 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 minimizes performance impact
Associative arrays int a_ray[string];	Index is by a defined type, not an integerHas methods to aid managementSized 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
- Minimize 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_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

```
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
```

Higher Performance Version

```
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 performance 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

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

Higher Performance Version

```
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.

Previous Next

UVM Navigation

SystemVerilog Performance Guidelines

Recent Forum Discussions About SystemVerilog Performance Guidelines

Performance comparison between $system and DPI SystemV...

Using nonblocking assignments in combinational logic SystemV...

On Demand Web Seminar Recording: UVM Coding Guidelines: Tips ... Announc...

Overhead of sampling a covergroup every clock cycle SystemV...

SV assertions - Capturing an event in between two edges of a signal SystemV...

Ask a Question ›

SIEMENS

Siemens Digital Industries Software

Portfolio

Cloud

Design, Manufacturing and PLM Software

Electronic Design Automation

Insights Hub

Mendix

How to Buy

Buying with Siemens

Buy Online

Partners

Academics

Renewals

Siemens

About Us

Careers

Community

Events

Leadership

News and Press

Trust Center

Contact

VA - Contact Us

HLS - Contact Us

PLM - Contact Us

EDA - Contact Us

Worldwide Offices

Support Center

Provide Feedback

Report Piracy