Yikes! Why is My SystemVerilog *Still* So Slooooow?

Cliff Cummings, Sunburst Design, Inc.
John Rose, Cadence Design Systems, Inc.
Adam Sherer, Cadence Design System, Inc

I. Introduction

In the 2012 edition of this topic we observed that thousands of engineers were cranking out SystemVerilog code but often found the simulation of that code ran slower than expected. We offered a number of testbench-focused recommendations for optimizing that code. Since then, millions more lines of code have been generated, simulators have become faster, but the question engineers raise is often the same with one new word: "Why is my SystemVerilog still so slow?"

Generally speaking, all SystemVerilog simulators are faster in 2019 than they were in 2012. UVM testbench code, randomization, assertions, and design execution have all become faster as the underlying engines become faster. While that is necessary without any doubt, it isn't sufficient to address the compute requirements of modern verification environments. That leaves engineers with a few knobs they can turn: add compute, get more performance from their simulator vendors, construct more effective verification environments, and improve the code they feed into the first three. Ouch. Cloud and datacenter expansion are answers for compute access but that requires financial investment. Squeezing the simulator vendor is a go-to and can yield modest gains. More effective verification environments can have larger benefits ranging from new technologies like formal analysis and to refactoring algorithms. It's this last point – refactoring then can raise a "yikes" from engineers because it sounds like a lot of work for the same output. But if we dig into this topic we can find approaches that can speed execution multiple times by training engineers with some new best-practices.

II. UVM is Software

Like it or not, this is the simple truth. As such, engineers working with UVM need to become familiar with known-good software practices. As stated in the introduction, are continuously optimized to improve performance. To do so, vendors typically look at coding patterns then tune the simulator to recognize these patterns for faster execution. For commonly used code, like the UVM reference library, this manifests in faster simulation for most projects. For project-specific coding styles or application-specific coding styles, the simulation improvements may completely miss some projects. With that said, there are inefficient coding styles that are more difficult for the simulators to recognize and need to be served with software-knowledgeable coding practices.

Frequent function/task calls add overhead and may mask algorithmic order issues. Each function/task has to set-up the call, push data references and/or complete data copies to the call stack and process any specified return. When they are called in a loop, the performance cost is paid each time through the loop. For simple calls, the compiler may in-line the function/task to avoid the stack frame manipulation, but complex calls are typically not in-lined because they bloat the simulation code which can lead to cache misses creating a different performance issue. Moreover, the body of the function/task may itself contain loops and/or additional calls. These can increase the order of the algorithm especially as the application scales. Figure 1 is code that calculates the number of elements in an MDA (multi-dimensional array) of queues by iterating over the array and counting elements. The algorithm is slow because it counts every element every time. It would be more efficient to use the queue's size operation, but the most efficient approach is shown in Figure 2 where a separate count is maintained. While the runtime difference between the separate counter and the built-in function may be small for a small MDA, it may be hard to predict how the code will be used so the best practice would be to code for efficiency.

```
class container;
  trans_obj mda_q [][][$];
  int rows = 0, cols=0;
  function void set_queues(int row, int col);
    rows = row; cols = col;
    mda_q = new[row];
    foreach(mda_q[row]) mda_q[row] = new[col];
  endfunction
  function int get_num();
    foreach(mda_q[i][j][k]) get_num++;
    //Better would be
    // foreach(mda_q[i][j]) get_num += mda_q[i][j].size();
  endfunction
  function void add (trans_obj obj, int row, int col);
    //checks for legality not shown
    mda_q[row][col].push_back(obj);
  endfunction
  function trans_obj get_first(int row, int col);
    if(mda_q[row][col].size())
      get_first = mda_q[row][col].pop_front();
  endfunction
end
```

Figure 1: Inefficient get_num function (code in bold)

```
class container;
  trans_obj mda_q [][][$];
  int elements = 0;
  int rows = 0, cols=0;
  function void set_queues(int row, int col);
    rows = row; cols = col;
    mda_q = new[row];
    foreach(mda_q[row]) mda_q[row] = new[col];
    elements=0;
  endfunction
  function int get_num();
    return elements;
  endfunction
  function void add (trans_obj obj, int row, int col);
    //checks for legality not shown
    mda_q[row][col].push_back(obj);
    ++elements;
  endfunction
  function trans_obj get_first(int row, int col);
    if(mda_q[row][col].size()) begin
      get_first = mda_q[row][col].pop_front();
      --elements;
    end
  endfunction
end
```

Figure 2: Efficient get_num function utilizing separate counter variable "elements"

There are other general coding approaches, some of which were identified in the 2012 paper [1]. For example, minimize string manipulation wherever possible. Also removing loop invariants – code that inside a loop that is not affected by the loop iterations – will speed simulation especially if the loop count is high. This may not be apparent to the loop author if the loop limit is variable as the limit may be much higher in scaled usage. A complement to this point is moving code that is only used in conditional, such as `if` or `case`, inside the conditional especially if the conditional is unlikely to be true or the calculation is only needed in one branch. If the code is in the testbench, then the compiler will likely make this adjustment for you. But if the code is in the DUT, or has any delay condition, the compiler may not be able to make the optimization because another process may update the rhs (right-hand side) of the expression.

```
int tmp_expr = 2*m+6;
@(posedge vif.clk);
if(setval == 1) begin
    x = data + tmp_expr;
end
```

```
@(posedge vif.clk);
if(setval == 1) begin
    x = data + 2*m + 6;
end
```

Figure 3: Conditional code example with efficient coding on the right

III. SystemVerilog Semantics Support Syntax Skills

SystemVerilog syntax knowledge is enough to get you coding, but semantics will help you do it efficiently. The semantics cover information beyond the syntax including defaults, simulator function specified by the standard, simple coding masking time-expensive execution, and more. Performance issues associated with semantics are rarely manifest in simulator profiles making it difficult them difficult to discover unless you are can recognize coding issues associated with syntax semantics. This doesn't discount the value of profilers, but it does imply that good coding and profiling work together to achieve performance.

Let's take a look at the logic type. Introduced in SystemVerilog, it can have either wire or variable storage and that storage type is determined from context by the simulator if it is not explicitly declared. This matters to simulation because wires can be collapsed to be the same object for higher simulation speed whereas variables can't. Since the semantic for logic type is to default to variable storage in all cases except for the inputs or inouts of a design unit you may have a properly executing simulation that mysteriously runs more slowly than you'd expect. Some simulators provide some profiling and optimization support for this issue, but it is a best practice to code as explicitly as possible to avoid side-effects like this lower performance. Figure 4 shows a coding example where `A2.y`, `A2.X1.y`, and `A2.X1.T1.y` are distinct and where the bolded declarations allow them to be collapsed to a single object. Note that `wire` is implicit on the input port so the declaration isn't needed to assure that the instances of **`a`** collapse together. However, it is not implicit on the output and does need to be declared in order for the simulator to collapse **`y`** together. It may be easier for designers to simply add `wire` to each port declaration to enables faster simulation speed.

```
// Example with distinct objects
module A2(input logic a, input logic b, output logic y);
  A2_1 X1 (a, b, y);
endmodule

module A2_1(input logic a, input logic b, output logic y);
  A2_2 T1 (a, b, y);
endmodule

module A2_2(input logic a, input logic b, output logic y);
  logic y_r;
  assign y = y_r;
  always_comb y_r = a&b;
endmodule

//Example with collapsed object via explicit declarations
module A2(input wire logic a, input wire logic b, output wire logic y);
  A2_1 X1 (a, b, y);
endmodule

module A2_1(input wire logic a, input wire logic b, output wire logic y);
  A2_2 T1 (a, b, y);
endmodule

module A2_2(input wire logic a, input wire logic b, output wire logic y);
  var logic y_r;
  assign y = y_r;
  always_comb y_r = a&b;
endmodule
```

Figure 4: Example of logic type semantics

Another important semantic is that the simulator will typically operate faster on a full vector than individual bits. This coding style is known as bit-blasting and the syntax for it is straight forward; looping over the bits of a vector for example. In Figure 5, the inefficient example uses a generate statement which creates a static hierarchy. While the simulator may be able to optimize a simple example within a local process like this one, complex examples optimization across a hierarchy are often a source of hidden performance issues.

```
// Inefficient vector operation
module somemod (input clk, input [31:0] a, input [31:0] b, output [31:0] c);
  reg [31:0] a_t;
  reg [31:0] c_t;

  for(genvar g=0; g<32; ++g) begin
    always@(posedge clk)
      a_t[g] <= a[g];

    always@(posedge clk)
      c_t[g] <= (a_t[g] ^ a[g]) | b[g];

    assign c[g] = c_t[g];
  end

 endmodule

// Efficient coding
module somemod (input clk, input [31:0] a, input [31:0] b, output [31:0] c);
  reg [31:0] a_t;
  reg [31:0] c_t;

  always@(posedge clk)
      a_t<= a;

    always@(posedge clk)
      c_t<= (a_t^ a) | b;

    assign c= c_t;

endmodule
```

Figure 5: Example of bit-blasting semantics

A third example of performance due to semantics is pass by reference versus pass by value. Without the general concept of a pointer and/or without the impact from algorithmic scaling, SystemVerilog coders may ignore the cost of passing by value. Engineers should see the `ref` construct in their code wherever a function only needs read access to any large data objects, such a `struct`s with hundreds of fields or QDAs (queues, dynamic arrays, associative arrays) with hundreds of elements, but does not write back to it. Just keep in mind that all parameters in an argument list which follow the ref construct will pass by reference unless you explicitly use input, output or inout.

The semantics of dynamic data structures (QDAs) are also sources of common performance issues which are generally true of SystemVerilog and most languages that have these types. An easy one to recognize is the use of static arrays instead of dynamic arrays wherever possible. Even if there is a small amount of variability to the length of the array, it is better to specify the static array a bit larger rather than take on the overhead of the dynamic array (memory footprint and garbage collection time). Another common performance mistake is to use dynamic arrays where a queue is better and vice versa. Since dynamic arrays are best for look-up and random insertion/deletion operations and queues are best for front or back operations with automatic resizing, the simulators have different internal representations to optimize each group of operations. Comparing access between queues and associative arrays, an arbitrary indexing for an object in a queue is O(1) but it is O(logn) for associative arrays.

IV. Memory and Garbage Collection – Neither are Free

We focused on the general semantics of dynamic types in the previous section, but the memory and garbage collection aspects of those types warrant a separate section. Inefficient memory can lead to significant cache misses, heap management overhead, and garbage collection overhead, all of which can be difficult to discover through profiling.

Copying dynamic objects, especially deep copies, should be minimized. One approach mentioned earlier is to use the ref construct in function calls, but another critical approach is to leave deep copies to the consumer of your objects wherever possible. Another approach is to use a single object where every possible or object pools for objects that are often created and destroyed. This is certainly a situation where the coder needs to understand how their code will be used as it is simple to create objects with simple new and free methods. While the example in Figure 6 is focused on the single object example for brevity in this paper, one can envision a class with separate methods to create and destroy objects. The create method would check a dynamic object containing previously used objects and only new one if the pool is empty. Similarly, the destroy method should push the object into the pool rather than simply dereferencing it for garbage collection. And, yes, semantics here imply that a queue is the proper dynamic object for the pool.

```
// md object created and destroyed each time through the for loop
class mydata;
  rand int a1;
  rand int a2;
  ...
endclass
class generator;
  task doit;
    mydata md;
    for(int i=1; i<1000; ++i) begin
      md = new;
      md.randomize();
      send_data(md);
    end
  endtask
  ...
Endclass

// md object created and destroyed once for the for loop
class generator;
  task doit;
    mydata md;
    md = new;
    for(int i=1; i<1000; ++i) begin
      md.randomize();
      send_data(md);
    end
  endtask
  ...
endclass
```

Figure 6: Example of reducing garbage collection overhead with object management

Implicit heap management is another way to avoid hidden performance issues. Classes are heap objects and carry a fair amount of overhead. Wherever possible structs should be used instead – either inside the class or instead of the class. For example, if the main purpose of the class is to be a container of heterogenous data types, then a `struct` is a better choice. A scoreboard is a good example. it is more efficient to code a `struct` as data element within the class that manipulates the scoreboard instead of accessing it in a separate class because that separate class will require heap management and potentially engage garbage collection but the simple `struct` will not.

Putting interface-heavy functionality into the interface rather than in classes is also more simulation efficient with the added benefit of being more reusable, because the functionality is associated with the interface itself, and synthesizable because the interface is but the class is not. In Figure 7, the class operating on the interface contains basic operations associated with the interface but inefficiently references them through a virtual interface and leaves any future user of the interface to duplicate this basic code. Figure 8 shows a more efficient and more reusable practice to embed the code in the interface itself.

```
interface myintf(input logic clk);
  logic txstart;
  logic[31:0] addr;
  logic [7:0] data;
  ...
  task wait_tx_end;
    @(posedge clk iff txstart);
    @(posedge clk iff !txstart);
  endtask
  ...
endinterface
class mywaiter;
  virtual myintf vif;
  task dosomething;
    if(vif.txstart) @(posedge vif.clk iff !vif.txstart);
    vif.data = 'haa;
    @(posedge clk iff txstart);
    @(posedge clk iff !txstart);
  endtask
  ...
endclass
```

Figure 7: Inefficiently coding operations into classes associated with virtual interfaces

```
interface myintf(input logic clk);
  logic txstart;
  logic[31:0] addr;
  logic [7:0] data;
  ...
  task wait_tx_end;
    if(vif.txstart)
      @(posedge vif.clk iff !vif.txstart);
    else begin
      @(posedge clk iff txstart);
      @(posedge clk iff !txstart);
    end
  endtask
  ...
endinterface

class mywaiter;
  virtual myintf vif;
  task dosomething;
    vif.wait_tx_end;
    vif.data = 'haa;
    vif.wait_tx_end;
  endtask
  ...
endclass
```

Figure 8: Efficiently coding operations into standard interfaces

V. It's Best to Leave Sleeping Processes to Lie

SystemVerilog simulators are event driven – the more events they run at a given time point, the slower they go. The implication, to paraphrase an old adage, is to let sleeping processes lie and not wake them unnecessarily. Sometimes the wake-up may occur in unexpected circumstance.

Perhaps the most common process in SystemVerilog is the always block with a single sensitive signal, such as the clock. This static process is highly optimized in all simulators, but side-effects from dynamic tasks or functions such as DPI (or any external) functions, virtual class tasks/functions, and virtual interface tasks/functions may disable the optimization. Some of these may be handled by a given simulator, but these side-effects can be arbitrarily complex, so the optimization can't be maintained in all cases. In Figure 9, the DPI call was first coded outside the conditional. Because of this, the simulator will need to run the always process on every `posedge` of `clk` because it can't see any side-effect coming from the DPI function. By moving the DPI call inside the conditional, the simulator can optimize the process wake up to `posedge clk` and `txactive` reducing the number of times the process executes.

```
import "DPI-C" function void somedpi(logic active, int count);
//Verilog code
interface myintf(input logic clk);
  logic txactive;
  int counter = 0;

  always@(posedge clk) begin
    //move the dpi code into the condition if it is conditional
    ~~somedpi(txactive,counter);~~
    if(txactive) begin
      somedpi(txactive,counter);
      counter<=counter+1;
    end
  end
endinterface

//dpi code
void somedpi(int active, int count)
{
  if(active)
  {
    if((count%100) == 0) dowork();
  }
}
```

Figure 9: Reducing static process wake-up by blocking side-effects

In contrast to static processes, simulators have more difficulty optimizing the wake-up for dynamic processes. Again, this is due to the arbitrarily complex nature of the dynamic processes. SystemVerilog provides the means to guide the simulator with the `iff` operator which instructs the simulator to wake the dynamic process if and only if a condition is met. In Figure 10, the `iff vif.txstart` code is needed prior to the conditional to assure that the process doesn't run every `clk`, but only runs on both the `clk` and `txstart`.

```
class mymonitor;
  ...
  task collect;
    forever begin
      @(posedge vif.clk iff vif.txstart);
      if(vif.txstart) begin
        collect_data();
      end
    end
  endtask
  ...
endclass
```

Figure 10: Using `iff` to reduce dynamic process wake-up

And it is possible to combine both static and dynamic process wake-up issues masking wake-up performance issues. For example, code can be written such that execution begins by triggering a static process, but if that leads to code triggered based on execution flow, the subsequent processes can be dynamic. Figure 11 contains an example of a statemachine coded with while statements. This style will lead to the intended hardware implementation, but the simulator will wake-up the statement machine on every `posedge clk` regardless of the state variable. While the simulator may be able to optimize this simple coding example, the coding can be arbitrarily complex, so it can't be optimized in all cases. However, recoding as shown in Figure 12 will lead to both proper hardware and efficient simulation.

```
interface myintf(input logic clk, input logic rst);
  logic req, ack;
  logic [1:0] sm;
  int counter = 0;

  always@(posedge clk) begin
    if(rst) begin
      sm<=0;
    end
    else begin
      while(!req) @(posedge clk);
      sm <= 1;
      while(!ack) @(posedge clk);
      sm <= 2;
      while(req) @(posedge clk);
      sm <= 3;
      while(ack) @(posedge clk);
      sm <= 0;
    end
  end
endinterface
```

Figure 11: Mixed static and dynamic processes with inefficient wake-up

```
interface myintf(input logic clk, input logic rst);
  logic req, ack;
  logic [1:0] sm;
  int counter = 0;

  always@(posedge clk) begin
    if(rst) begin
      sm<=0;
    end
    else begin
      case(sm)
        0: if(req) sm<=1;
        1: if(ack) sm<=2;
        2: if(!req) sm<=3;
        3: if(!ack) sm<=0;
      endcase
    end
  end
endinterface
```

Figure 12: Mixed static and dynamic processes recoded for efficient simulation

VI. UVM Best Practices

Not surprising, all of the principles discussed so far apply to UVM. Wherever possible, deep-copy, pass by value, process wake-up associated with UVM should be minimized.

A common approach in UVM is to code message writing to interesting parts of the verification environment. A simple optimization is to guard the messaging using the uvm_report_enabled() function. In Figure 13, the messaging is triggered if the verbosity level is set at or above `UVM_DEBUG`. And if the messaging needs to be written frequently, the UVM tree printer or even the line printer should be used to get meaningful information while keeping the messaging overhead to a minimum.

```
class mymonitor;
  int mem[logic[47:0] ];
  ...
  task get_data();
    string memlayout;
    get_data_from_if();
    if(uvm_report_enabled(UVM_DEBUG, UVM_INFO, "MEMDATA")) begin
      memlayout = "  {\n";
      foreach(mem[i])
        memlayout = $sformatf("%s    mem[%x] : %x", memlayout, i, mem[i]);
      memlayout = {memlayout, "  }\n"};
    end
    `uvm_info("MEMDATA", memlayout, UVM_DEBUG)
   endtask
endclass
```

Figure 13: Conditional messaging in UVM

Another source on unnecessary execution is associated with TLM analysis ports. Since these are often used as a callback mechanism, monitors, collectors, or other components package data and then write the data to all of the consumers which are attached to the analysis port. As such, the component writer will often do all the packaging, but there is no requirement that an analysis port is hooked-up so the packaging may not be used. In fact, the hook-up can be environment dependent. Figure 14 shows the packaging in the component writer, but the use of a conditional to avoid executing that packaging for unconnected ports.

```
class mycollecter extends uvm_component;
  ...
  tlm_analysis_port#(mydata) data_port = new("data_port", this);
  task run_phase(uvm_phase phase);
    collect();
  endtask
  task collect();
    fork
      while(1) begin
        mydata md = mydata::create("mydata",this);
        get_txn_from_interface(md);
        data_port.write(md);
      end
    join_none
  endtask
endclass

  task run_phase(uvm_phase phase);
    if(data_port.size()) begin
      collect();
    end
  endtask
```

Figure 14: Conditionally connecting UVM analysis ports

There are some additional best practices that can help reduce the risk of unexpected overhead. If objections need to be frequently raised and lowered for objects deep in the hierarchy, the overhead can become prohibitive, so the use of global objection handling may be warranted. Configurations can also lead to unexpected overhead if the complexity of the types cases the expansion of the fields to hundreds or thousands of elements. In this case a configuration container should be used. A final point of guidance is to never use wildcards for field names unless absolutely necessary because of the unanticipated algorithmic order issues. A simple name lookup is of log(n) order, but the wildcards invoke the regular expression execution causing additional overhead.

VII. VERIFICATION BEST PRACTICES

This section probably warrants a paper unto itself. As such, we will focus on a few critical recommendations associated with randomization, assertions, and coverage execution. The authors agree that this just scratches the surface on this topic, but these simple recommendations can be broadly applied so they fit within the context of this paper.

SystemVerilog provides multiple ways to randomize variables so it's important to understand the advantages of each to maximize performance. `$urandom` adds thread stability to the older Verilog standard $random and remains the fastest way to do randomization of single, independent variables which are randomized often. More complex constraint-based randomization is often used in UVM environments and is subject to a lot of optimization work in each simulator. However, there are practices that can help each engine solve faster. Using solve order to break up or simplify constraint solving and using `pre_randomize/post_randomize` for sequential solving can speed-up simulation but you do need to be careful to avoid creating invalid solutions. Echoing recommendations made earlier in this paper regarding arrays, coders should be careful to avoid coupling such variables where possible. Figures 15 and 16 bring some of these concepts together. In Figure 15, the loop sets up a constraint on each array element based on its neighbor resulting in a list of 100 integers that have 100 32-bit variables that have to be solved simultaneously. Modifying the code to use `post_randomize` as in Figure 16 can improve runtime performance up to 1000x.

```
class txn;
  rand int addr;
  rand logic[15:0] payload[$];
  rand bit [2:0] del;
  constraint size_ct { payload.size() inside { [16:256]}; }
  constraint val_ct { payload[i] => i && (payload[i] <= 'hffff-i); }
  constraint sort_ct {
    foreach (payload[i]) {
      if(i) payload[i] >= payload[i-1];
    }
  }
```

Figure 15: Randomization of array element

```
class txn;
  rand int addr;
  rand logic[15:0] payload[$];
  rand bit [2:0] del;
  constraint size_ct { payload.size() inside { [16:256]}; }
  function void post_randomize();
    payload.sort();
  endfunction
endclass
```

Figure 16: Conditionally connecting UVM analysis ports

Assertions are also commonly used in UVM environments and subject to a lot of optimization in simulators. As with randomization, a set of best practices can help boost performance. In general, we want to balance the well-known value of assertions with efficient simulation. Minimizing the number of attempts by configuring the enabling condition to trigger only on the first cycle of the enabling condition, using single-cycle assertions wherever possible, an using single-clock assertions – even if that means splitting the assertion into two separate assertions – all result in improved performance. While local variables may be needed to manipulate data inside sequences and properties, they add overhead during simulation. Where possible, local variables should be avoided as shown in Figure 17.

```
// With local variable
property P;
  reg lv;
  @(posedge clk)
  (a, lv = b) ##1 c |=> d == lv;
endproperty
```

```
// Without local variable
property P;
  @(posedge clk)
  a ##1 c |=> d == $past(b, 2);
endproperty
```

Figure 17: Coding assertion with and without local variables

Coverage is a third common element of verification environments. When setting up the coverage bins, choose vectored or auto bins to maximize performance as scalar and fixed-size bins create additional simulation overhead and are more difficult to debug. Per the mantra of this paper, fewer coverage events will deliver faster simulation. This starts by sampling coverage using a specific event trigger rather than a generic event such as a system clock. Coverage sampling events can be further reduced by having `covergroups` share common expressions. A third method to reduce sampling events is to merge sample process that use the same event as shown in Figure 18.

```
// Sampling on separate events
covergroup cg1;
  ...
Endgroup
covergroup cg1;
  ...
endgroup
covergroup cg1;
  ...
endgroup

always@(event)
  if(collect_cov) cg1.sample();
always@(event)
  if(collect_cov) cg2.sample();
always@(event)
  if(collect_cov) cg3.sample();
```

```
// Sampling merged to a single event
always@(event iff collect_cov) begin
  cg1.sample();
  cg2.sample();
  cg3.sample();
end
```

Figure 18: Coding assertion with and without local variables

VIII. ACKNOWLEDGMENT

UVM/SystemVerilog environments have grown in size and complexity since we looked at SystemVerilog performance in 2012. With that growth, the cost of slow execution has also grown. This paper provided code examples and guidelines to make every simulation faster and reduce those “yikes” moments verification engineers have when they run their regressions.

REFERENCES

[1] “Yikes! Why is my SystemVerilog so Slooooow?” Frank Kampf, Justin Sprague, Adam Sherer, DVCon U.S. Proceedings 2012.
[2] *IEEE Std 1800-2012, IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language.* by IEEE, 3 Park Avenue, New York, NY 10016-5997, USA.
[3] *Assertion Writing Guide, Product Version 14.2, January 2015, Chapter 8 - "Maximizing Assertion Performance",* by Cadence Design Systems, Inc., 2655 Seely Avenue, San Jose, CA 65134, USA.
[4] “App Note Spotlight: Streamline Your SystemVerilog Code, Part I,” by John Rose. Blog post by Tyler Sherer 19 March 2018. Cadence Functional Verification Blogs. Retrieve from http://community.cadence.com/cadence_blogs_8/b/fv/posts/app-note-spotlight-streamline-your-systemverilog-code-part-i.