2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Pedal Faster! Or Make Your Verification Environment More Efficient. You Choose.

Rich Edelman, Mentor Graphics, Fremont, CA (rich_edelman@mentor.com)

Raghu Ardeishar, Mentor Graphics, McLean, VA (raghu_ardeishar@mentor.com)

Rohit Jain, Mentor Graphics, Fremont, CA (rohit_jain@mentor.com)

***Abstract*—Building a block or system level testbench is as hard as building the underlying RTL or System to be tested. This paper describes some performance problems that may creep into the testbench, and some performance problems that are fundamental in things like the UVM.**

Keywords—SystemVerilog; UVM; Testbench performance; Testbench optimizations

I. INTRODUCTION

Verification projects are getting larger with more parts, including SystemVerilog, Verilog, VHDL, SystemC, C++, C, Assertion languages, and third party verification IP. Aside from poorly written RTL, there are many places that can make a verification project slow. Most testbenches are slower than they should be.

The paper will introduce certain kinds of slow-downs and possible solutions, including poorly written RTL, UVM testbench architecture issues, C coding issues, inter-language data issues and SystemVerilog testbench issues. We'll also discuss ways to measure performance, including how to architect some DPI code to automatically collect elapsed time and memory usage.

We'll examine code snippets on the following:

- Poorly written RTL code, including event loops which are processed too many times, misuse of level checks when it should be event checks, signals that take a long time to settle and glitching signals.
- Poorly written UVM code, including misuse of configuration settings, misuse of sequence/sequence-item interactions, objection misuse, and poorly written driver code.
- Optimizations for fast and compact DPI C code argument passing will be discussed.
- SystemVerilog class based misuse will be discussed - when to call new, and when to reuse existing handles. We'll discuss object lifetime and garbage collection.
- Randomize() call misuse will be examined, including how to use post_randomize() to speed up randomize() calls.

Finally, a detailed exploration of why UVM_INFO is so slow will be discussed. We'll make recommendations about how to use logging (UVM_INFO), but still have good performance. The reader of this paper will gain some good ideas and actual examples of how to re-architect his testbench and DUT for easy speed-ups.

II. QUANTITY OF CODE

An obvious performance improvement is to reduce the quantity of code being executed. In a testbench, there may be behavioral Verilog, RTL and C or C++. The testbench is becoming more like software. The most you can do to remove lines of code that get executed the better. Simple code inspection may help you find lines of code that are unneeded. If you are using the UVM library, A deeper understanding of UVM internals will be needed to understand the impact that specific usages have on quantity of code executed.

A. `uvm_info and uvm_report_info

Most testbench designers generate logfiles to follow the behavior of their testbenches. The UVM has a "message logging system" which can be used to create pre-formatted messages of the form:

```
<MessageType> <FileName:LineNumber> @ <Time>: <UVM Instance Name> [ <MsgID> ] <Message>
```

For example:

```
# UVM_INFO vip_a/agentA.sv(169) @ 1334724:
      uvm_test_top.i2_agentA.sequencer@@ggp_seq_A2.gp_seq.p_seq.A_seq [sequence_A]
           MATCH @ Addr=766. Wrote fdb, Read fdb
```

The simplest, most recommended way to use this logging system is with the macro `uvm_info. Using a macro allows for automatic insertion of file and line number, as well as verbosity checking in order to not perform unnecessary message formatting in the case that the message is not actually printed. For example:

```
`uvm_info(get_type_name(),
    $sformatf("MATCH @ Addr=%0x. Wrote %0x, Read %0x", t.addr, wdata, rdata), UVM_HIGH)
```

The goal is to only perform the potentially expensive $sformat() when the message will be printed. In this example, if the VERBOSITY level is set to UVM_MEDIUM, then this message will not be printed (it is a UVM_HIGH message). In that case we don't need to bother calling $sformatf().

The `uvm_info macro implementation is simple enough. It has a conditional to check the verbosity level. If the current setting and the message verbosity allow, then the 'then' part is taken, and the message will be printed. The function uvm_report_enabled() checks the verbosity settings. Then uvm_report_info actually does the printing.

```
`define uvm_info(ID,MSG,VERBOSITY) \
  begin \
    if (uvm_report_enabled(VERBOSITY,UVM_INFO,ID)) \
      uvm_report_info (ID, MSG, VERBOSITY, `uvm_file, `uvm_line); \
  end
```

An example macro expansion of

```
`uvm_info("myid", "mymessage", UVM_MEDIUM)
```

is

```
if (uvm_report_enabled(UVM_MEDIUM,UVM_INFO,"myid"))
   uvm_report_info ("myid", "mymessage", UVM_MEDIUM, "myagent.sv", 4250);
```

Most of the time, the MSG is a complex function which calculates a message string. Using the `uvm_info macro allows that complex function to never be called if the message verbosity will cause it not to be printed.

Given a complicated "mymessage" formatting job, the two lines below are not equivalent in terms of speed, for the reasons outlined above. They are however equivalent in functionality.

```
`uvm_info("myid", "mymessage", UVM_MEDIUM)
uvm_report_info ("myid", "mymessage", UVM_MEDIUM, "myagent.sv", 4250);
```

Given this discussion, it is natural to assume that `uvm_info is fast, and should be used throughout your code. To a certain extent this is true, but even the act of checking the verbosity level with uvm_report_enabled() takes time, and would be better removed. `uvm_info is not fast.

Two simple removal techniques come to mind, both requiring a redefinition of `uvm_info.

1) Remove `uvm_info completely from your code. Redefine uvm_info as the empty string.

```
`ifdef uvm_info
`undef uvm_info
`define uvm_info(ID,MSG,VERBOSITY)
`endif
```

The problem with this solution is that your testbench now goes silent.

2) Change `uvm_info so it skips the verbosity check for everything above UVM_LOW. Add the red line below as a very fast way to filter verbosity levels. All UVM_MEDIUM, UVM_HIGH and UVM_DEBUG messages will only execute the if-statement, never calling into the UVM.

```
`ifdef uvm_info
`undef uvm_info
`define uvm_info(ID,MSG,VERBOSITY) \
   begin \
     if ( VERBOSITY <= UVM_LOW ) \
       if (uvm_report_enabled(VERBOSITY,UVM_INFO,ID)) \
         uvm_report_info (ID, MSG, VERBOSITY, `uvm_file, `uvm_line); \
   end
`endif
```

This solution allows UVM_LOW and UVM_NONE messages to be emitted. In a simple example using this solution allowed a large increase in performance. For long simulations or emulations, re-implementing the uvm_info macro will save time.

III. QUALITY OF CODE

Some code is better than other code. All code is not created equal in terms of quality. Coding quality can be a subjective measure, often specific for certain development groups or divisions. Having high quality code makes finding and fixing bugs easier, and makes general readability easier.

A. Arrays / Containers

When you are building a container to hold class handles or integers or other important structures in your scoreboard, driver, monitor or sequence, you need to choose between various "container" data structures that SystemVerilog offers – for example, an array, a dynamic array, an associative array or a queue.

Each of these containers could hold 10, 100 or 1000 elements easily. Each of these containers has a different implementation underneath which bring along pros and cons for each container. The decision about which container to use should be based on how it is being used.

Container Type	Pros	Cons	Memory	Elapsed time
Array	Fast	Size fixed before simulation	1	1
Dynamic Array	Fast Size can be changed during simulation	Resize may be expensive	1	1
Associative Array	Sparse memory	Can be slow	> 1	>> 1
Queue	Ordered list	Can be slow	> 1	>> 1

The data in the table above is relative. It was generated using 1 Billion array accesses on an array of size 1 Million (1 Million items were stored in the array, repeated 1000 times). For small arrays, and few accesses no problems will arise. Any of these containers will perform well. Once you have a large number of elements or access them many times, you must choose wisely.

Code snippets below illustrate that container usage is relatively the same for all the types.

ARRAY	DYNAMIC	ASSOCIATIVE	QUEUE

`int val;`
`int array[1000000];`
`initial begin`
 	
`...`
`array[i] = val;`
	`int val;`
	`int array[];`
	`initial begin`
	`array = new[1000000];`
	`...`
	`array[i] = val;`
		`int val;`
		`int array[int];`
		`initial begin`
		
		`...`
		`array[i] = val;`
			`int val;`
			`int array[$];`
			`initial begin`
			
			`...`
			`array.push_back(val);`

When you use an array, be sure you understand how it will be used. Interestingly, some of the UVM data structures (containers) will suffer slowdowns as they fill up. The best recommendation is to keep them relatively empty – use few configs, use few factory overrides, etc.

B. Smarter Sequences and Sequence Items

A sequence is a relatively heavyweight UVM object to create and bring into execution. Once a sequence is constructed it should stay "alive" for a reasonable amount of time. It should not simply create one sequence_item (transaction), and send it to the driver, but rather it should create a series of transactions.

Sequence/Sequence Item Style	Number of transactions constructed per body() call	Number of calls to new() to construct transactions	Elapsed Time
One sequence == one sequence item sent per body() call	1	N	Slowest
One sequence == many sequence items sent per body() call (a call to new() for each sequence item)	N	N	Slower
One sequence == many sequence items sent per body() call (a call to new once for a single, reused sequence item)	N	1	Fastest

The first entry in the table above is the one to stay away from. It is slower by the fact that sequences are heavy to get started, and creating a sequence to just send one transaction is inefficient. Instead, create a sequence that send many transactions, and if possible, only construct one transaction. Constructing one transaction can be a problem, depending on how transactions are used in the testbench. If a transaction may be modified or saved by other downstream components, constructing one transaction may not be appropriate. Further discussion of copy-on-write semantics are beyond the scope of this paper. Try to build a testbench that will operate properly when transaction construction is kept to a minimum. And try to avoid the copy-constructor, especially for large transactions. Copying is expensive, and should be avoided where possible.

For each call to body(), construct one transaction, randomize it and send it to the driver.

```
class seq_one_transaction extends uvm_sequence#(transaction);
  ...
  transaction t;

  task body();
    t = transaction::type_id::create("t");
    start_item(t);
    if (!t.randomize())
      `uvm_error(get_type_name(), "Randomize failed...")
    finish_item(t);
  endtask
endclass
```

For each call to body(), construct a transaction, randomize it and send it to the driver. Repeat N times.

```
class seq_many_transaction extends uvm_sequence#(transaction);
  ...
  transaction t;
  int max_count;

  task body();
    for (int i = 0; i < max_count; i++) begin
      // Construct one per count
      t = transaction::type_id::create("t");
      start_item(t);
      if (!t.randomize())
        `uvm_error(get_type_name(), "Randomize failed...")
```

```
      finish_item(t);
    end
  endtask
endclass
```

For each call to body(), construct one transaction, randomize and send it to the driver N times.

```
class seq_many_transaction_construct_once extends uvm_sequence#(transaction);
  ...
  transaction t;
  int max_count;

  task body();
    // Construct once.
    t = transaction::type_id::create("t");
    for (int i = 0; i < max_count; i++) begin
      start_item(t);
      if (!t.randomize())
        `uvm_error(get_type_name(), "Randomize failed...")
      finish_item(t);
    end
  endtask
```

Test harness. Create the three sequences and run the appropriate one per the test.

```
class test extends uvm_test;
  ...
  seq_one_transaction                 one_seq;
  seq_many_transaction                many_seq;
  seq_many_transaction_construct_once many_seq_construct_once;
  ...
  task run_phase(uvm_phase phase);

    many_seq_construct_once = seq_many_transaction_construct_once::type_id::
                              create("many_seq_construct_once");
    many_seq                = seq_many_transaction::type_id::create("many_seq");
    one_seq                 = seq_one_transaction::type_id::create("seq");

    if (run_one_transaction) begin
      for (int i = 0; i < max_count; i++)
        one_seq.start(e.sqr);
    end
    else if (run_many_transaction) begin
      many_seq.max_count =  max_count;
      many_seq.start(e.sqr);
    end
    else begin
      many_seq_construct_once.max_count = max_count;
      many_seq_construct_once.start(e.sqr);
    end
```

IV. BAD CODE

Poorly written code is a large contributor to performance issues, but is generally easiest to fix since it is usually user written code.

A. *Using memory – references and copying*

A verification testbench naturally uses data structures to help verify behavior. You must understand your data structures and what Verilog does with each one. A simple memory illustrates the points.

1) Copying memories when you don't intend to – be careful with local variables and assignments

The code below has two implementations. The fast one, and the not so fast one. In reality the user code only contained the not so fast one. It was code that evolved over time as the verification team solidified what kinds of models they were building and what kinds of information they had available. Additionally, they may have been getting used to a new language – SystemVerilog.

```
typedef int memory[int];
memory PAGES[int];

function automatic void assign_memory(int value, int address, int page_address);
    int m[int];
    memory p[int];

    if (fast) begin                          // Fast LEG
      PAGES[page_address][address] = value; // No accidental copying
    end
    else begin                               // Slow LEG
      if (PAGES.exists(page_address)) begin
        m = PAGES[page_address];             // Accidental copy
      end
      m[address] = value;                    // Update value into the copy
      PAGES[page_address] = m;               // Copy the copy back in
    end
  endfunction
```

This code is trivial code which demonstrates the "inadvertent" or "undesired" copying. In the 'fast' leg of the if-statement, no copying occurs. A "two-dimensional" array is updated with a value. In the 'slow' leg of the if statement we will usually copy the PAGES[page_address] into a new array named 'm'. Then we update the entry in 'm'. Then we copy the local 'm' array back into the PAGES[page_address]. Even for just 10,000 calls to this function the performance is seconds versus minutes.

2) Using call-by-reference and call-by-value – using the 'ref' keyword.

Many testbenches are factored into collections of function calls. This factoring practice is good software technique, adding to better understanding and easier maintenance. Unfortunately it can also introduce unsuspected slowdowns. For the code below, using 'ref' reduced a 3 minute run time to 2 seconds.

```
`define N 1000000
module top();
  int memory[`N];

  function           void check_copyin( input int i, input int m[`N] ); // Using 'input'
    // Code checking validity of memory 'm'
  endfunction

  function automatic void check_ref    ( input int i,   ref int m[`N] ); // Using 'ref'
    // Code checking validity of memory 'm'
  endfunction

  initial begin
    bit use_ref;
    ...
    if (use_ref) check_ref(   i, memory);
    else         check_copyin(i, memory);
    ...
endmodule
```

These kinds of problems often originate from testbench code that is used on small blocks or small testbenches. The amount of copying is small. But when the testbench is reused at higher levels with larger packets or larger memories, the amount of copying is large.

The key for large memory structures is to use the 'ref' keyword. The 'ref' keyword causes the compiler to just reference the variable, not to copy it onto the stack. Conceptually it behaves like an 'inout' memory would,

except an inout memory would be copied twice – once on the way in and once on the way out. A ref memory argument will not be copied.

B. `uvm_info + sprint() + m_string

Don't mix `uvm_info, sprint() and m_string. Some methodologies have guided users to writing code like the example below. If you do, you will consume all memory. This is a "feature" of the UVM. Instead, use convert2string() which is simple, does not incur the field automation overhead, and will not consume all memory.

Original m_string mistake (don't do this):

```
class transaction extends uvm_sequence_item;
   ...
  function void do_print(uvm_printer printer);
    printer.m_string = "transaction"; // Format your transaction as you please...
  endfunction
endclass

class env extends uvm_env;
  ...
  task run_phase(uvm_phase phase);
    transaction t = new();
    forever
      #10 `uvm_info("GEN", t.sprint(), UVM_MEDIUM)
  endtask
endclass
```

To fix this code, change the user code changes from

```
      #10 `uvm_info("GEN", t.sprint(), UVM_MEDIUM)
```

To

```
      #10 `uvm_info("GEN", t.convert2string(), UVM_MEDIUM)
```

Delete do_print() from the transaction class definition and implement convert2string():

```
  function string convert2string();
    return $sformatf("%0d, ...", ...);
  endfunction
```

C. uvm_config_db

The UVM configuration database is a very slow, hard to use collection of code. It is hard to debug to figure out what went wrong. You will have the best luck if you create one thing to put in the configuration database. For example, create a configuration class, and put it in the database. Then retrieve it later.

```
class config_class extends uvm_object;
  int delay_config = 42;
  int time_between_transactions = 12;
endclass
```

Lookup the configuration class once, and "hand-out" the configuration variables. (delay_config and time_between_transactions). Don't look up individual integers.

The UVM configuration database can be thought of as a large array of configuration items. There is one large configuration database. If you put many things into the database it will become very slow. When a lookup happens with a wildcard, each entry in the configuration database is visited, and the regular expression is evaluated to see if it matches – very inefficient. But you only see this problem if you have many configuration items in the database.

```
foreach configuration_setting {
     evaluate a regular expression, if match, perform configuration lookup. // Expensive
}
```

If you have many configuration entries, and you do lookups from high frequency loops – like a transaction monitor, then most of your simulation time will be spent in the configuration database system. Each monitored

transaction will cause the entire configuration database to be scanned. This will be very slow. Do not call configuration ::get() code from any high-frequency location like a monitor.

The get() implementation calls lookup_regex_names(), which is the code which looks through the entire configuration database – see the code snippets below.

```
static function bit get(uvm_component cntxt,
                        string inst_name,
                        string field_name,
                        inout T value);
    ...
    rq = rp.lookup_regex_names(inst_name, field_name, uvm_resource#(T)::get_type());
    ...
  endfunction

function uvm_resource_types::rsrc_q_t lookup_regex_names(string scope, string name,
                                                         uvm_resource_base type_handle = null);
    ...
    foreach (rtab[re]) begin
      rq = rtab[re];
      for(i = 0; i < rq.size(); i++) begin
        r = rq.get(i);
        if(uvm_re_match(uvm_glob_to_re(re),name) == 0)
          // does the type and scope match?
          if(((type_handle == null) || (r.get_type_handle() == type_handle)) &&
             r.match_scope(scope))
            result_q.push_back(r);
      end
    end
    return result_q;
  endfunction
```

D. iteration limit

RTL code can be written in such a way as to consume all cpu cycles or to be very hard to debug. The snippets below will do that. Avoid writing code like this. Although this code looks trivially easy to identify as bad code, it does occur in this and other forms in current designs.

1) In this example, the user wrote bad code. See papers by Cliff Cummings for good coding etiquette.

```
module top();
  reg a = 0, b = 0;

  always_comb b <= ~a;
  always_comb a <= ~b;
endmodule
```

2) In this example, the user forgot to assign the variable named ‘delay’ a value. It defaults to 0 and causes the problem.

```
module top();
  reg clk;
  int delay;
  always #(delay) clk = ~clk;
  always @(posedge clk)
    $display("%s:%0d", `__FILE__, `__LINE__);
endmodule
```

3) In this example, the user made a mistake building an always block without a “stop condition”, like “posedge clk”. This often happens in UVM driver implementations when a transaction is not fetched from the sequence, but instead a null handle (the sequence wasn’t ready with a transaction).

```
module top();
  reg rst;

  always
    if (rst != 0)
```

```
            $finish(2);
endmodule

class driver ...
     ...
     task run_phase(...);
       forever begin
         seq_item_port.get(t);
         ...
         if (t != null) begin
           seq_item_port.item_done();
         end
       end
     endtask
```

E. *Memory leaks due to inadvertent handle holding*

Transaction handles can be stored in arrays in any place in the testbench, especially scoreboards. When the handle is stored in an array it is said to have an additional reference. The code below is (mis-)designed to have a memory leak. The driver gets a transaction and saves it into a list. That list is never emptied, so the reference to the object (t) is never reclaimed by the garbage collector.

```
class driver extends uvm_driver#(tr);
  ...
  tr memory_leak[$];
  tr t;

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      memory_leak.push_front(t);
      …
      #10;
      seq_item_port.item_done();
    end
  endtask
endclass
```

Although no user would write this kind of code they do end up creating lists which accumulate class handles throughout simulation. For a short, small simulation the memory footprint will not grow to be a problem. As soon as this code is used in a larger testbench, the memory footprint will be a problem.

F. *Objections*

Some users use objections inside high frequency code, such as monitors or scoreboards. This causes the objection to get raised and dropped for each transaction. Objections are a very heavyweight part of the UVM. Use objections sparingly, for example, raise an objection when the test starts, and drop it when the test ends.

In the UVM objections can get themselves mixed up. Without any poorly written user code, a collection of UVM objections can lose count of the outstanding objections, due to event queue ordering in the simulator.

An objection is really just a fancy counter. It is quite easy to have the objection count become incorrect if there are many objections being triggered. This is a “feature” of UVM objections. Avoid using objections widely in your testbench. If possible use just one objection at the top of your test.

V. Optimizing Usage

A. *Optimized DPI Imports*

The SystemVerilog DPI is a powerful and simple way to call back and forth between C and SystemVerilog. By its nature, it is a pass-by-copy system. Any arguments that need to go from SystemVerilog to C are copied onto the stack in SystemVerilog, and used from the stack in C. For small data the pass-by-copy is not a problem, but when large packets of data need to move between SystemVerilog and C it can be quite slow.

Simulator companies have the ability to optimize certain calls across the DPI-C interface. For example DPI import functions with input or inout arguments of type array of byte, int, shortint or longint can be optimized to be pass-by-reference. The speed-up moving from unoptimized to optimized will be quite dramatic, since the copy of the large data is no longer needed.

Simply specifying the DPI call using the efficient data structures allows the simulator to not do a pass-by-copy. Defining the packet you are sending through a DPI-C interface as an array of integers will improve your performance.

B. Optimizing class handle allocation and reuse

Calling new() on a class handle can be expensive, depending on the design of the class. Handles can be reused under careful conditions. When new() is called, memory is allocated – a new object comes into existence. The handle returned by new() is a pointer to the newly allocated object. It is an address in memory.

In the case of a sequence, it may create 1000 transactions, and send them to the driver. If we know that the driver is going either consume all the transaction data immediately, or if the driver is going to make its own copy, then the sequence is safe to reuse the allocated class handle.

C. Randomizing efficiently

Many constraints are written without thinking about performance. The foreach iteration in constraints is known to be slow. Eliminating it can help performance. Additionally, often many variables in a constraint are actually not random variables, but have a simple relationship with other variables. Using a pre_randomize() or post_randomize() call to set variables can make constraints much faster.

The example below shows are simple change to eliminate a foreach construct.

```
class C;
  rand int ascending[`NUMBER_OF_INTS];
  constraint values { foreach (ascending[i]) ascending[i] inside {[0:1000]}; }
  constraint order  { foreach (ascending[i]) if (i!=0) ascending[i] > ascending[i-1]; }
endclass

class D;
  rand int ascending[`NUMBER_OF_INTS];
  constraint values { foreach (ascending[i]) ascending[i] inside {[0:1000]}; }

  function void post_randomize();
    ascending.sort();
  endfunction
endclass
```

This is a very simple example where eliminating the order constraint increases run time significantly. In this example, the ordering constaint just says that the values should be sorted in ascending order. We've replaced the constraint with a 'sort' call in the post_randomize() function. We simply generate N random values, and then sort them later.

D. Faster RTL

The two simple loops below are each designed to do the same thing. They print a message on the positive clock edge, when the interrupt signal is high. Loop A is a high frequency loop, repeating once per clock, checking the value of interrupt. Loop B is a lower frequency loop, waiting for interrupt to be high and the positive edge of the clock.

The A loop

```
forever begin
  @(posedge clk);
  if (interrupt == 1)
    $display("interrupt=%0d", interrupt_count++);
end
```

The B loop

```
forever begin
  wait(interrupt == 1);
  @(posedge clk);
  if (interrupt == 1)
    $display("interrupt=%0d", interrupt_count++);
end
```

Careful examination of your coding style may yield some performance improvements.

VI. MEASURING PERFORMANCE

Without the ability to measure performance, either memory usage or elapsed time, it is hard to make improvements. Using the simple DPI code below you can create timers and get "splits" or incremental usage for both memory and elapsed time. The getrusage() call is unreliable for memory size, use the linux mapped files for accurate data.

The appendix contains some example C code which can be used to measure elapsed time or memory footprint live during simulation. The most common usage of these routines is to show incremental footprint growth within loops, or to time loops. They are relatively short and easy to understand and can be changed to meet specific use model for performance measurement.

VII. SUMMARY

Performance gains exist in every testbench no matter how large or small. Finding them can be hard. One of the easiest to find in an UVM testbench is `uvm_info. The best way to have a fast testbench is to design it to be fast from the beginning. Use the tips and tricks here to help you. As you write your testbench, keep track of how fast it is, and don't let it get slower as development continues. As an example, a simple testbench in UVM 1.1d got 20% slower just by moving to UVM 1.2.

Performance is a never ending battle. You want to fastest testbench, because as soon as you move you RTL DUT to emulation, the testbench speed becomes the speed of the entire verification run. Keep your testbench fast.

VIII. REFERENCES

[1] Cliff Cummings – Many really good papers with really good recommendations. www.sunburst-design.com/papers

IX. APPENDIX – MEASURING TIME AND SPACE USED

A. *Timer and Memory Footprint library*

This library of code is useful to print the time and memory used. A simple use model is described below, with additions left as an exercise for the reader.

```
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "mti.h"       /* To get mti_Malloc() */
#include "svdpi.h"     /* Standard DPI header */
#include "dpiheader.h" /* For interface matching. */

extern void io_printf(char *format, ...);

typedef struct rusage * rusagep;
typedef void          * chandle;

/* timer_new() - Malloc a new rusage structure. */
static rusagep
timer_new()
{
    return (rusagep)mti_Malloc(sizeof(struct rusage));
}

/*
 * timer_restart()
 *
 * Given an existing rusage structure, "refill" the
 * entries from now.
 */
static rusagep
timer_restart(rusagep p)
{
    if (getrusage(RUSAGE_SELF, p) != 0) {
        /* Error */
        perror("timer_restart()");
    }
    return p;
}

/*
 * timer_start()
 *
 * Create a new rusage, fill it in, and return a pointer to it.
 */
chandle
timer_start()
{
    rusagep p;

    p = timer_new();
    timer_restart(p);
    return p;
}

/*
 * timer_split()
```

```
 *
 * Returns the number of useconds since either the start of the
 * timer (*pp) or the last split time.
 *
 * The parameter "pp" will have its contents reset to the current
 * "rusage" at the end of this function. It is an inout.
 */
int64_t
timer_split(chandle *pp)
{
    int64_t seconds, useconds;
    struct rusage now, *p;

    p = *((rusagep *)pp);
    timer_restart(&now);

    seconds  = now.ru_utime.tv_sec  - p->ru_utime.tv_sec;
    useconds = now.ru_utime.tv_usec - p->ru_utime.tv_usec;
    if ( useconds < 0 ) {
        /* We need to borrow 1 seconds worth of microseconds. */
        if (seconds > 0) {
            seconds -= 1;
            useconds += 1000000;
        } else {
            /* Error? Or just a clock rounding error? Call it ZERO. */
            seconds = 0;
            useconds = 0;
        }
    }
    useconds = seconds * 1000000 + useconds;
    timer_restart(p); /* Reset the timer for the next split */
    return useconds;
}

//
// mail.nl.linux.org/kernelnewbies/2006-04/msg00209.html
//
void
get_mem_used()
{
  int pid = getpid();
  int pagesize = getpagesize();

  int fd;
  char filename[25];
  char statm[1024];
  size_t length;
  unsigned int vm_pages, rss_pages;

  vm_pages = 0;
  rss_pages = 0;

  // On windows the open will fail.
  // Just use '0' as the answer.
  // We don't really care.
#if !defined(__WIN32__)
  sprintf(filename, "/proc/%d/statm", (int)pid);
  fd = open(filename, O_RDONLY);

  if (fd == -1) {
      fprintf(stderr,
          "error opening file %s: %s\n",
          filename, strerror(errno));
      exit(1);
  }

  length = read(fd, statm, sizeof(statm));
```

```
    if (length == -1) {
        fprintf(stderr,
            "error reading file %s: %s\n",
            filename, strerror(errno));
        exit(1);
    }
    close(fd);

    statm[length] = 0;
    sscanf(statm, "%u %u", &vm_pages, &rss_pages);
#endif // __WIN32__

    io_printf("VM=%llu RSS=%llu\n",
      (long long)vm_pages  *pagesize,
      (long long)rss_pages *pagesize);
}
```

B. Simple test program showing API usage.

```
import "DPI-C" function void get_mem_used();
import "DPI-C" function longint timer_split(inout chandle p);
import "DPI-C" function chandle timer_start();

module top();

    chandle splittime; // The split timer.
    chandle totaltime; // The total execution timer.

    longint useconds;  // Split time, in microseconds.
    real seconds;      // Total time, in seconds.
    int sum;           // The result from our calculation.

    initial begin
        totaltime = timer_start();  // Timer to measure total time.
        splittime = timer_start();  // Timer to measure splits.

        for(int i = 0; i < 10; i++) begin
            // Reset the split timer.
            useconds = timer_split(splittime);

            // Perform a time consuming computation.
            consumeTime(sum);

            // Calculate how long since the last split.
            useconds = timer_split(splittime);
            $display(" split - %0d microseconds", useconds);
            get_mem_used();
        end

        // Calculate total time since execution began.
        useconds = timer_split(totaltime);

        seconds = useconds / 1000000.0;
        if (seconds > 1)
            $display("Test complete - %0g seconds", seconds);
        else
            $display("Test complete - %0d useconds", useconds);
        foreach (handles[i])
          $display("i=%0d", i);
        $finish(2);
    end
endmodule
```