Cleaning Out Your Pipes - Pipeline Debug in UVM Testbenches

Rich Edelman
Mentor, A Siemens Company
46871 Bayside Parkway, Fremont, CA 94538

Neil Bulman
Arm Limited
110 Fulbourn Road,
Cambridge, UK CB1 9NJ

***Abstract*- Debugging SystemVerilog[1] UVM[2] Testbenches can be challenging. This paper will discuss some tips and tricks for debugging specific problems with a UVM Testbench for a pipelined design. These techniques can be used with any debugger, but are sometimes difficult to use without a roadmap. We'll describe and discuss a roadmap for doing high level UVM Testbench debug.**

I. INTRODUCTION

Debug can be the mundane day-to-day work that is consuming more and more time. Using advanced debug techniques can help speed up bug finding and improve quality.

In a modern UVM Testbench, there is usually a collection of code from many different places – different design and verification teams, various verification IP, automatically generated stimulus generators, automatically generated coverage collectors, and many constraints. Most verification engineers are likely only familiar with a small part of this code. Good debug means good visibility.

In this paper, better debug is achieved through better visibility and better analysis – even for simple things like waveform debug. We'll show how to instrument sections of code to provide better visibility - for example using transaction recording and transaction debug.

The following categories of problems and solutions will be discussed in this paper.

II. EXAMPLE

The motivation and example used for this paper was a sub-section of an Arm® Cortex-M33, which was announced October 2016. It shares characteristics with the Cortex-M4 plus TrustZone security and 3-stage instruction pipeline. The design code consist of 62 files containing a total of about 100,000 lines of code. The testbench code consists of 95 files containing about 12,000 lines of code. A typical test runs in about a minute and the tests use random seeds. The basic block diagram for the test environment is in Figure 1.

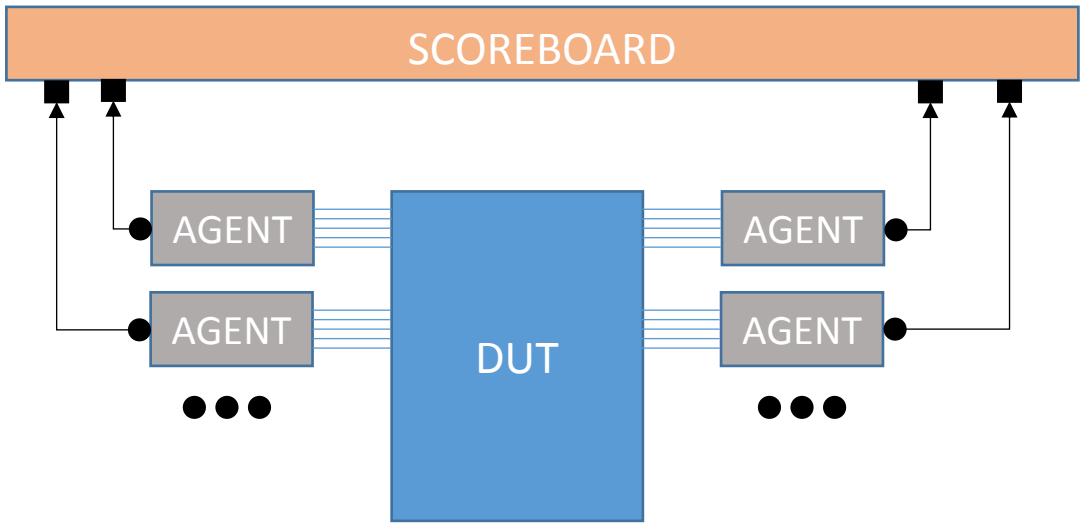


Figure 1 - Basic Testbench Architecture

III. TESTBENCH DEBUG

Data structures

A testbench typically will use many types of SystemVerilog data structures, including dynamic arrays, associative arrays and queues. It will contain lists of lists and arrays of arrays and arrays of lists. The job of the scoreboard is to keep track of things (make lists) and to model behavior (both with code and with data structures).

A pipeline scoreboard might have queues of expected items that are in the pipeline. The model needs to be kept consistent with the hardware. Pipeline items can be put in the wrong pipeline queue or duplicated, or not put in at all. Each of these issues – wrong place, duplicated and missing must be debugged. Who put it in the wrong place and why? Who duplicated it and why? And worst of all – who did NOT put it in the right place. Debugging things that happened and are incorrect is hard. Debugging things that didn't happen is even harder. We don't have an automatic answer for any of it, but by combining the techniques below debug can be made easier.

Coding Styles

Testbench scoreboards written in SystemVerilog may have a combination of UVM calls, macros, threads, if-then-else and case statements. The code is usually written at a "higher abstraction" level: pushing and popping a queue and comparing the items from two different queues. Or interpreting instructions streams to make an accurate replica of registers and memory. The best debug is writing clear and small code. Large nested if-then-else and multiple page function calls are ripe for big problems. Write simply.

IV. USE $DISPLAY

"I just use $display". That's a common way to do debug, and often the fastest way to figure out what is going on. But on a real design with lengthy compile, optimization, elaboration and simulation times, it is hard to repeatedly add $display in each new place it is needed.

But by all means use $display. But then don't forget to take them all out again. A simple way to take them in and out is with some kind of macro (like VERBOSE below).

```
`ifdef VERBOSE
  $display("Debug: %0d", loop_count);
`endif
```

Use your debug tool when possible to see values. $display is a last resort.

V. USE FANCY $DISPLAY WITH %P

Using $display can be cumbersome and time-consuming when printing a complex object, list a class or an array. Use the %p format type. SystemVerilog will auto-format the object, including arrays and class objects.

```
int gid;

class base;
  int id;
  int x;
  function new();
    id = gid++;
  endfunction
endclass

class extended extends base;
  int x;
  int y;
  int z;

  base b;

  int my_queue[$];
  int my_associative_array[int];
```

```
  function void print();
    b = new();
    $display("This = %p", this);
    $display("B    = %p", b);

    $display("Queue = %p", my_queue);
    $display("Associative Array = %p", my_associative_array);
  endfunction
```

The print() routine will produce the output below, printing a two classes, a queue and an associative array.

```
This = '{id:10, x:0, x:0, y:0, z:0, b:@base@10,
        my_queue:'{57, 74, 40, 79, 87, 100, 61},
        my_associative_array:'{0:61, 1:100, 2:87, 3:79, 4:40, 5:74, 6:57 }}
B    = '{id:10, x:0}
Queue = '{57, 74, 40, 79, 87, 100, 61}
Associative Array = '{0:61, 1:100, 2:87, 3:79, 4:40, 5:74, 6:57 }
```

VI. Even Fancier $display - Stylized Watch Window

Problem: A collection of variables should have consistent values. After some simulation they become inconsistent (they diverge). Keep an eye on those variables. Testbenches and designs are getting more complicated. When a test fails it may not be immediately obvious whether the design or testbench is at fault, being able to compare the testbench expected view of state can help quickly determine which is at fault and direct the next stage of debug.

A watch window is called for in your debug tool. Or you can have a "watch()" function defined which prints the "interesting" variables when called. Call this watch() function when things change or on every clock edge, or at regular times.

```
function void watch();
  $display(" * Watch ******************************************************");
  $display(" * Id: %0d", id);
  $display(" *     x=%0d, my_queue[0]=%0d, my_queue[$]=%0d, my_queue.size()=%0d",
                  x,     my_queue[0],     my_queue[$],     my_queue.size());
  $display(" *     y=%0d, \
                      my_associative_array[1]=%0d, my_associative_array.size()=%0d",
                  y,  my_associative_array[1],     my_associative_array.size());
  $display(" ************************************************************");
endfunction
```

Output Result:

```
* Watch ******************************************************
* Id: 10                                                     *
*     x=0, my_queue[0]=57, my_queue[$]=61, my_queue.size()=7  *
*     y=0, my_associative_array[1]=100, my_associative_array.size()=7 *
* ************************************************************
```

This is a simple list of variables with header and time. Fancier "blocks" of information could be printed, like a collection of registers or memory locations. Arguments could be passed in, either with the variables to be printed or object handles to use to reference the desired variables.

VII. Breakpoint

Problem: Testbench code is doing something wrong. Set a breakpoint to figure out how it got there. What the control conditions were. Use the debugger software to set a breakpoint. If you don't have a breakpoint capability, use a $stop.

Using SystemVerilog code, we could insert code such as

```
BP1: begin
       $stop;
     end
```

A Regular Breakpoint

Many debug tools have breakpoints. They are easy to specify using a syntax like

```
bp <FILE> <LINENUMBER>
```

If no debug tool with breakpoints are available, you can use $stop on the line you want to stop on.

VIII. CONDITIONAL BREAKPOINT

Problem: A breakpoint can be set, but we only want to stop at the breakpoint under certain conditions. For example, in a class, a breakpoint on a line might only apply to a certain instance of that class – we only want to stop in one particular instance of the class. In addition, other conditions can be tested when a breakpoint is hit, for example, if we are not in "reset mode", or if the number of items processed has reached 10000, or if 10 reads and 10 writes have happened.

Using SystemVerilog code, we could insert code such as

```
BP2: begin
       if ( !reset_mode && nreads > 10 && nwrites > 10) $stop;
     end
```

IX. BREAKPOINTS WITH A COUNT

Problem: Sometimes a problem occurs in a testbench at a certain line, but only after the breakpoint has been hit hundreds or thousands of times. Using a counter allows the breakpoint to be skipped until the counter is reached.

Using SystemVerilog code, we could insert code such as

```
BP3: begin
       if ( counter++ > LIMIT) $stop;
     end
```

X. BREAK IN A SPECIFIC INSTANCE

Problem: Only one particular instance of a class object is having a problem. Break on a line in just that one instance.

There are many ways to specify an instance, perhaps the easiest is with the "name". Stop when the name matches.

```
BP4: begin
       if(get_full_name() == "uvm_test_top.i2_ ... .ggp_seq_A2.gp_seq.p_seq.A_seq")
         $stop;
     end
```

XI. BREAK ON CHANGE - DETECTING A VARIABLE CHANGE

Problem: A class member variable is being changed, but the source of the change cannot be determined by code inspection. For example, a status register, an event trigger or a configuration field could be getting updated incorrectly. The source of the change must be found. In a microprocessor it may be possible for a general purpose register to be updated for many reasons: instruction execution, entering an interrupt hander, returning from an interrupt handler, external debug access, testbench initialization stimulus. If a number of these happen close together it can be difficult to track the order which the testbench modelled these. Using break-on-change in conjunction with a call stack can quickly show why the updates happened and the order the testbench handled them.

Use the debug tool to set a break-on-change. Run simulation until the break. The "change" is happening at the break spot. Now, debug can determine how and why this change is happening. If your debug tool doesn't have break-on-change, then using SystemVerilog code it is quite easy to stop after detecting a change in class member variable.

```
`define STOP_ON_CHANGE(obj,name) \
    fork \
      forever begin \
        @(obj.name);  \
        $display("Stopping. Object Member Variable Changed");  \
        $stop;  \
      end \
    join_none
```

In the testbench, execute the follow code, which creates a "stop-on-change" monitor.

```
`STOP_ON_CHANGE(this, write_expected_tr)
```

This macro adds a powerful debug aid. It waits for a class member to change, from some specific class. When it changes, then a message is printed and simulation stops.

XII. BREAK ON CHANGE – DYNAMIC ARRAY OR QUEUE CHANGE

Stopping on a variable change is important, but sometimes those variables are arrays, queues, dynamic arrays or associative arrays. This can add some complexity. When one of these dynamic elements changes size, it is important for debug. When one of the elements in one of these dynamic elements changes, it is important for debug. In both cases, an update has been made, and could be important for debug. The change is important, but more important is "who did it".

Many kinds of fancy checks can be implemented using SystemVerilog. In the case below, the check is to wait for the size of the queue to change. After five monitoring events, stop monitoring.

```
task wait_all_q();
  forever begin
    int size;
    size = all_q.size();
    wait (all_q.size() != size);
    $display("STOPONQUEUE: Size changed. size=%0d, q=%p", all_q.size(), all_q);
    if (count++ > 5) break; // Stop monitoring after 5 alerts
  end
  ...
endtask
...
task run_phase(...);
  fork
    wait_all_q();
  join
  ...
endtask
```

OUTPUT FILE:

```
# STOPONQUEUE: Size changed. size=1, q='{@sequence_item_A@3}
# STOPONQUEUE: Size changed. size=2, q='{@sequence_item_A@7,
   @sequence_item_A@3}
# STOPONQUEUE: Size changed. size=3, q='{@sequence_item_A@11,
   @sequence_item_A@7, @sequence_item_A@3}
# STOPONQUEUE: Size changed. size=4, q='{@sequence_item_A@15,
   @sequence_item_A@11, @sequence_item_A@7, @sequence_item_A@3}
```

XIII. STACKS AND THREADS AND SEQUENCES

Problem: Setting breakpoints and single stepping is too low level. We want to see sequences get created and watch any patterns form with long or short lives. Additionally, we want to debug threads they may spawn.

Closing coverage can be a time consuming part of a project, which can have an impact on either project delivery with additional time needed to close coverage, or project quality, with fixed deadlines and coverage holes.

Being able to visualize the coordination of sequences can help in being able debug why the expected tests are not producing the expected results in a more time efficient manner. Many debug tools have the ability to record transactions. Transactions can be placed in any part of the testbench or design. [3]

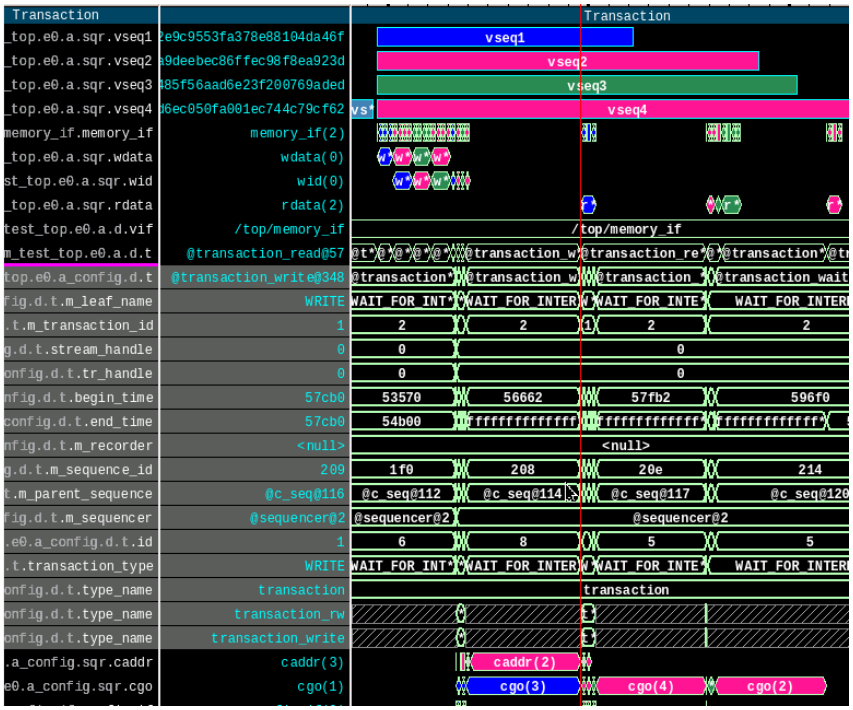


Figure 2 - Classes and Transactions In A Wave Window

XIV. CLASSES IN THE WAVE WINDOW

Problem: Seeing a class object with values at a current time is valuable, but seeing ALL the values of a class object across time is more valuable. Especially in post simulation. Use the debug tool to examine classes in the wave window over time.

Classes in the wave window are useful in two ways. Seeing a variable 't' change over time shows objects "passing through".

```
my_transaction t;

forever begin
  seq_item_port.get_next_item(t);
  …
end
```

The "contents" of 't' – the class member variables is also important to see. For example in a transaction going through the driver, seeing the packet priority or source and destination addresses might be a useful debug capability.

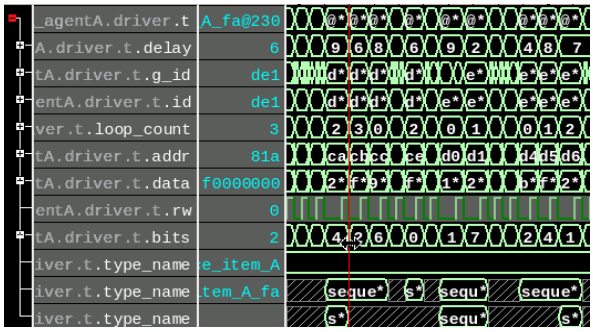


Figure 3 - Debugging Class Member Variables in a Wave Window

```
class sequence_item_A extends uvm_sequence_item;
    `uvm_object_utils(sequence_item_A)

    rand bit       rw  ;
    rand bit [31:0]addr;
    rand bit [31:0]data;

    struct packed {
      ...
    } bits;
```

XV. COLOR HIGHLIGHTS

As debug shows more and more values, it's handy to be able to color values that are found. For example, '58' is an important value in the display below.

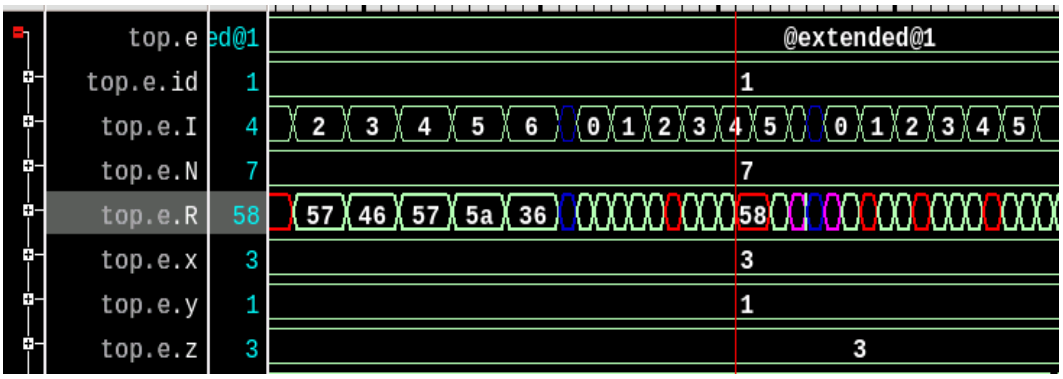


Figure 4 - Coloring Values

Every place that '58' occurs has been colored RED.

XVI. MONITORING RAM USAGE – MEMORY HOGS

Problem: A testbench that runs at a block level may suddenly consume all RAM when reused up in a larger block or system. Why? Who has a handle to my object? Or "Why is memory filling up?"

A UVM Testbench writer may create objects and store their handles for later processing. This happens in most scoreboards, stimulus generators and drivers. It is a common pattern. As the objects are created, they consume memory. When the handles are stored (for example in a queue), that space cannot be returned to the operating system – it is "kept" by the simulation. As more and more handles are allocated more and more space is consumed. The memory consumed by testbenches fluctuates over time growing and shrinking as tests complete. Occasionally, through coding errors and typos the allocated object handles are never removed from the storage place (from the queue). This causes the queue to grow quite large, and for the overall simulation memory requirement to increase. The space for the constructed objects is not being "free'd".

An interesting side note is that a testbench which has these "memory hogging" characteristics may run just fine at a block or sub-system level, since there are few objects allocated during the tests. When the testbench is reused in the system or at the chip level, suddenly memory requirements are out of control, and all RAM is consumed and the tests fail or crash. Without support for this feature in the debugger, a slightly different technique can help.

Usually, this accumulation of "non-free'd" objects also has a side-effect – very large dynamic data structures. Very long queues or large dynamic or associative arrays. By tracking array sizes, any large array can be flagged.

Any addition to a queue (or other memory element) could be annotated to check the size. If the size is above some threshold, then there may be a problem. For example if your scoreboard has a queue and is checking transactions pairwise in-order, there is no reason for the queue to grow beyond single digits. Certainly a queue size above 10,000 is likely a bug.

```
`define Q_PUSH_FRONT(q, item) \
   if (q.size() > Q_SIZE_LIMIT) begin \
```

```
        ... \
    end \
    q.push_front(item);
```

Using the new “push” functionality means change q.push_front(item) to `Q_PUSH_FRONT(q, item).

```
begin
   `Q_PUSH_FRONT(egress_q, transaction_h)
   ...
end
```

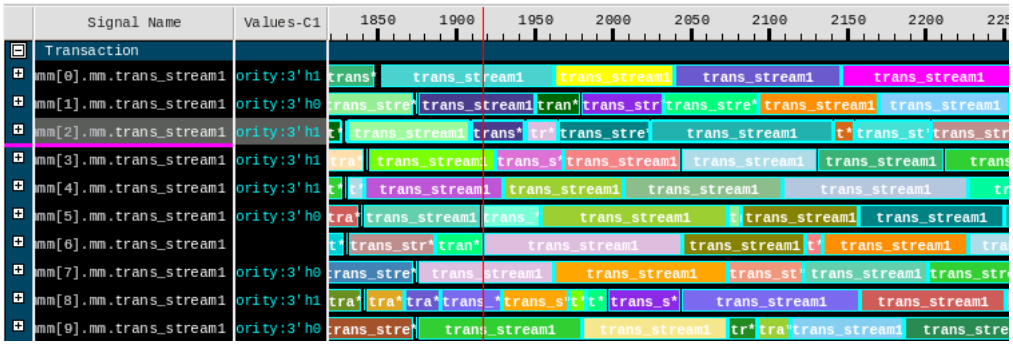
If your debug tool has the ability to track “un-free’d” handles, use it. Otherwise, you can track dynamic storage sizes using the technique above.

XVII. TRANSACTION DEBUG

Problem: Waves and Classes are too low level. Visualizing transactions can help the eye see key patterns. Using transactions allows the eye to see large patterns.

Transactions are like candy. The more you have, the more you want. Transactions come natively with the UVM (with limited functionality), and can be added to any testbench, whether it is a UVM testbench or otherwise. Transaction recording is really nothing more than a fancy $display() statement. Anyplace you can put a $display() statement, you can annotate with transaction recording.

Using transaction recording APIs is beyond the scope of this paper. Searching the literature will yield many useful references.



XVIII. MISCELLANEOUS

Checkpoint / Restart

Simulation are sometimes quite long running – hours, days or even weeks. Using a checkpoint enables a restore to begin from a checkpoint without all the previous simulation being repeated. Many users checkpoint their long running simulations every 3 hours. As simulation continues, bugs may be found. When a bug is found, then debug can occur by starting (restoring) from the previous checkpoint. In this way, a bug is only 3 hours of simulation away. Any convenient frequency for checkpointing can be used.

SEARCH

Debug is better visibility. Search can improve visibility by finding things quickly. Using search can help find the things that may be problematic – perhaps some verification engineer suspects the “exception_entry” code is problematic. Using a powerful search can find the suspect code quickly. Powerful search is critical to better debug.

UVM Config Debug

The UVM has a variety of data structures, one of which – the Configuration Database – is particularly hard to debug. There are UVM switches available for dumping the config database and interactions with it, but often the problems cannot be discovered with the built-in switches. The debug tool should provide some way to understand the type matching type specifications and the inner loop checking that can cause a mismatch. Use simulator command line

arguments for help debugging (+UVM_RESOURCE_DB_TRACE +UVM_CONFIG_DB_TRACE) or use your debugger. [4]

Macro Debug

Macros are notoriously hard to debug. Normally a macro is created for some collection of well-tested code with well-understood behavior. When a macro needs to be debugged, the debugger will need to support it. If not, the last resort is to expand the macro, and then use it expanded in-line. This is a tedious exercise, but necessary if the macro is a complicated collection of code. For example, debugging the UVM field-automation macros would be a case where a debugger or in-line expansion would be necessary.

XIX. CONCLUSION

Debug techniques for a variety of situations have been discussed. Adopting some or all of them will improve debug productivity. They can be used alone, or combined for even more productivity. Knowing what tools are in the toolbox is the first step to more product debug.

Breakpoints, Conditional breakpoints, Instance specific breakpoints and break-on-change are all powerful ways to debug simulation problems with live simulation.

Fancy $display and waveforms with classes and transactions can be used in live simulation and in post-simulation debug.

Whatever bugs you may be trying to uncover, the best way to do it is to know what tools you have available.

XX. REFERENCES

[1] SystemVerilog Language Reference Manual, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[2] UVM Language Reference Manual, http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[3] *"Thinking In Transactions",* DVCON India 2017, Rich Edelman, Mustufa Kanchwala.

[4] *"Go Figure – UVM Configure" The Good, The Bad, The Debug",* DVCON Europe 2016, Rich Edelman, Dirk Hansen.