Watch Out! Generating Coordinated Random Traffic in UVM

Nigasan Ragunathan
Microsoft Corporation
niraguna@microsoft.com

Christine Thomson
Microsoft Corporation
chthomso@microsoft.com

***Abstract*- Traditional designs with unique data paths use constrained random stimulus to generate legal traffic, only requiring virtual sequence level coordination across interfaces. For more complex designs, this approach is not always sufficient. For a design with interdependent data paths and multiple input interfaces, stimulus generation must be coordinated and constrained to account for transactions sent on any input interface. This paper will introduce a methodology for generating coordinated random traffic with dependencies to overcome this barrier.**

I. Introduction

As Intellectual Property (IP) within a System on Chip (SoCs) become more complex, verification for these IPs become more complex as well. It can become increasingly difficult to generate legal constrained-random stimulus to comprehensively verify such IPs with complex traffic conditions without resorting to an increased number of tests written as more directed-random scenarios. This paper looks at one such scenario of complex constrained-random stimulus generation which is not able to be contained through standard UVM random sequences and sequence items and proposes a coordinated methodology to address this challenge. This method reduces the number of directed-random tests needed by automating the complexity systematically.

II. Background

In a simple Testbench, stimulus is generally contained to driving one or more primary data path interfaces, a register or control interface, and miscellaneous sideband signals. Generating stimulus for this standard Testbench is straightforward: generate initialization transactions on the register control interface, followed by constrained-random or directed transactions on the primary data path(s). Data path transactions are typically a series of sequence items which often do not have any dependencies on the prior transaction to be considered legal input. This simple testbench is shown in Figure 1 below.

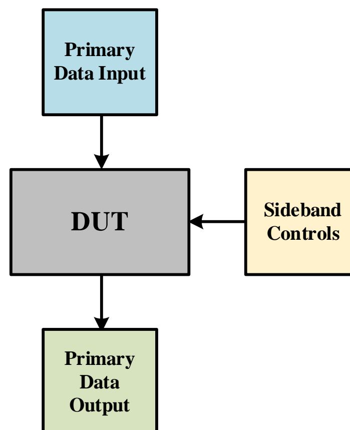

Figure 1: Simple Testbench

Unlike these traditional Testbenches, this paper will focus on a more complex Design Under Test (DUT) which has multiple input interfaces, each of which are dependent on all the past traffic received by the DUT. In this scenario, there is a need for complex constrained-random stimulus generation which is not able to be contained through standard UVM random sequences and sequence items and proposes a coordinated methodology to address this challenge.

Stepping back, when an IP has multiple input interfaces requiring complex stimulus patterns which are mutually dependent on each other, it is logical to assume that constrained random testing would take too much effort and maintenance to effectively develop and test in the traditional sense. Regardless, attempting the traditional method of constrained-random stimulus would result in producing illegal traffic to the DUT since subsequent traffic has to be dependent on prior traffic to be considered legal input. This cannot be handled through normal sequence items and sequences – therefore directed stimulus would be more valuable in the short run. On the other hand, limiting a Testbench to directed stimulus will result in fewer unique conditions getting produced. This causes a reduction in the coverage space of traffic, leading to a much higher chance that a bug will be missed. The other alternative is to use a model to generate legal traffic, but this is essentially a different form of directed traffic and holds the same disadvantages. An example of this complex testbench is shown in Figure 2 below.

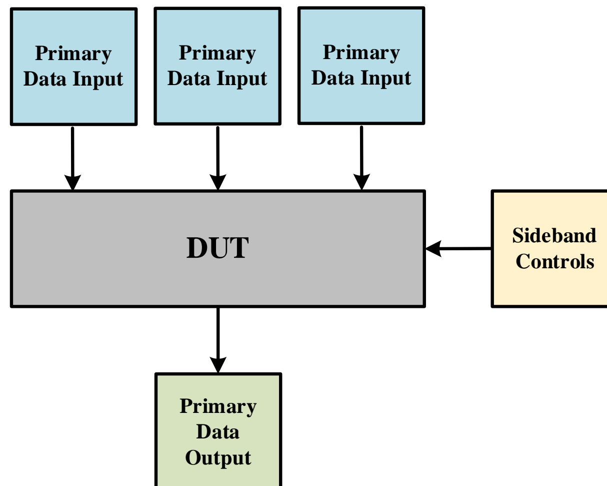

Figure 2: Complex Testbench

III. Proposed Method

The proposed solution to this problem of needing mutually dependent random traffic is to create a UVM object which acts as a "traffic manager". This traffic manager class must be a single object and is shared across sequences to manage the automatic legal traffic generation without relying on the test writer to manage what traffic is being sent in a manual or directed way. The traffic manager is responsible for aiding each interface's sequence in sequence item randomization. This is done by logging the contents of randomized sequence items on each interface in a database before it is sent to the DUT. The information stored in the database is then used to randomize sequence items when the sequence executes to produce coordinated and automated random traffic. Figure 3 below illustrates the flow for a sequence using the traffic manager.

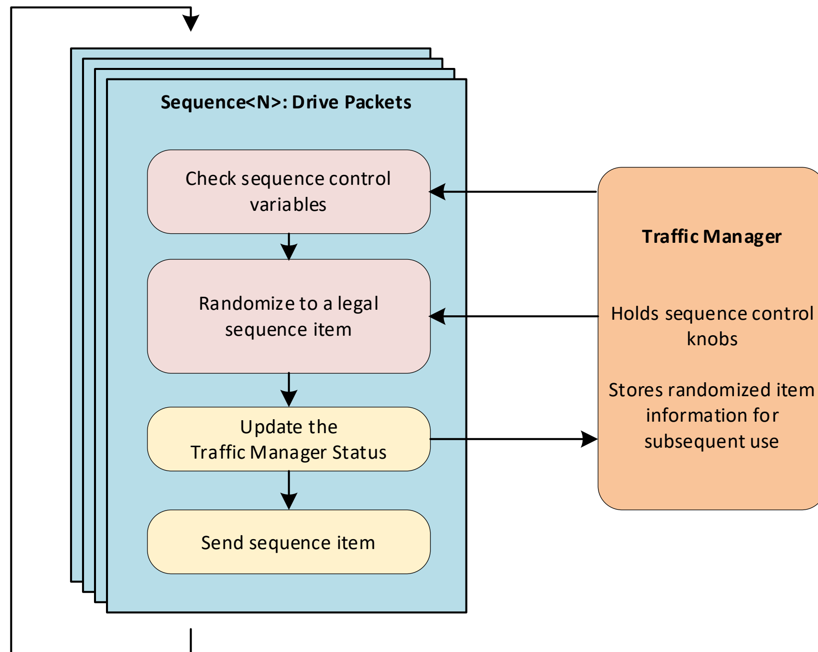

Figure 3: Example High-level Flow of Sequences

The traffic manager exists as a standalone object to isolate the traffic generation from all the other components in the Testbench. It is necessary to separate the traffic manager from what the Testbench observes through the monitors due to timing constraints – the monitor only observes transactions after it has already occurred, whereas the sequence needs to know what stimulus is already generated and "in flight" before producing more traffic to queue up and be sent. All of this is done to ensure that no illegal traffic is generated before a monitor has time to see it, as this can take several cycles depending on the protocol for that interface. Since the traffic manager is a standalone UVM object used in conjunction with a sequence, it is also easily integrated into higher level testbenches for vertical re-use in subsystem and SoC level Testbenches.

III. EXAMPLE DESIGN

To demonstrate the concept of the traffic manager, consider the case for a DUT which takes as input data values and operation codes (opcodes) to perform arithmetic operations. For this example, data is supplied on unique input interfaces and opcodes are supplied on a third input interface. Legal opcodes which can be provided are add, subtract, multiply and divide. It is required that the operation code must not be sent until both data operands are provided as the operation executes at the time the opcode is received. To add additional complexity for the sequences, let's also say a requirement exists in which multiply operations are unique. Multiply operations do not require two new data values to be loaded and can only be sent following an add operation. If a multiply operation is received, it will execute on the two previously loaded data values. Figure 4 below shows a high-level block diagram for this example design.

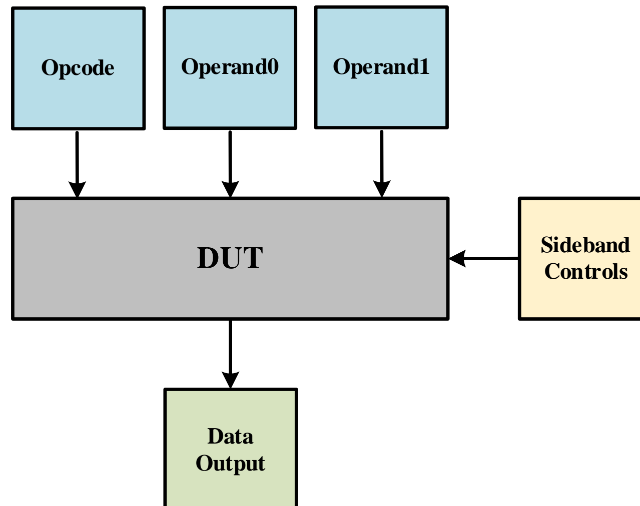

Figure 4: Example Testbench

Note that, since the data is supplied on unique interfaces, it is possible that data can arrive at any time with respect to the other interface; i.e. the data for operand0 may arrive before the data for operand1 and vice versa.

To apply the traffic manager to this example design, the traffic manager would store the opcodes supplied within a previous_opcodes queue. This is then leveraged within the sequence body to look at the previous_opcodes sent and automatically determine the next legal operation to generate when randomizing the sequence item. The traffic manager class in Figure 5 demonstrates one such implementation given the example design's requirements.

```
 1 class traffic_manager #(int OPCODE_WIDTH = `OPCODE_WIDTH_DEFAULT,
 2                         int NUM_OPERANDS = `OPERAND_WIDTH_DEFAULT)
 3                                             extends uvm_object;
 4
 5   // Associative arrays containing
 6   // expected internal state of DUT
 7   bit [OPCODE_WIDTH-1:0] previous_opcode[$];
 8
 9   // Operand Load Indicator
10   bit [NUM_OPERANDS-1:0] data_loaded;
11
12   // Control access to the Traffic Manager
13   semaphore access;
14
15   // Add shared variables here
16
17   function new(string name="traffic_manager");
18     super.new(name);
19
20     sa_access = new(1);
21
22   endfunction : new
23
24 endclass: traffic_manager
```

Figure 5: Traffic Manager Class

This example code shows the queue of previous_opcodes which stores the previous operations randomized in the sequence and sent to the DUT. Additional fields shown include: data_loaded which indicates when the operands have been sent by the data interface sequences, and an access semaphore which is used to control accessing the traffic manager object. The access semaphore is described further in the text below. Based on design and stimulus requirements, additional variables can be defined for this class and shared by the sequences to support further visibility, sequence coordination, or stimulus generation.

Recall that one of the requirements is that multiply operations are unique and can only be sent following an add operation without loading new data operands. To apply this requirement, the opcode sequences would access the traffic manager's previous_opcode queue and observe if the previous operation was an add. If an add was just sent, then and only then is it legal to send a multiply operation. Additionally, since each opcode must wait on the data operands to be loaded, the sequences would access the traffic manager's data_loaded signal to detect and wait for each operand sequence to report the data has been loaded before the new opcode is sent to the driver. Since multiply operations are unique and do not require new operands to be sent, the sequence can bypass waiting on the data_loaded indicator if the next operation is a multiply. Figure 6 shows the randomization of the sequence item within the operand sequence leveraging the contents of the traffic manager to produce legal stimulus as described above.

```
1     // Randomize req_item with data from Traffic Manager
2     if (!(std::randomize(req_item with {
3       if (tm.previous_opcode[tm.previous_opcode.size()-1] != ADD) {
4         req_item.opcode != MULTI;
5       }
6     }))) begin
7       `uvm_fatal(REPORT_TAG, "Unable to randomize req_item")
8     end
9
10    // Wait for operands to load
11    if (req_item.opcode != MULTI) begin
12      wait (&tm.data_loaded == 1);
13    end
```

Figure 6: Opcode Randomization with Traffic Manager

Since multiple sequences are accessing the same instance of the traffic manager object, care must be taken to avoid using stale information. This can occur if two or more sequences were attempting to update the class members of the traffic manager simultaneously. To prevent sequence update collisions, an access semaphore with one key is instanced in the traffic manager. Each sequence that shares the traffic manager is then responsible for obtaining the key before accessing the traffic manager's contents and returning the key once this is complete. This controlled access guarantees that only one sequence has access to the traffic manager at any given time and preserves the fidelity of the database.

The traffic manager concept was used to verify a real-world design at Microsoft. The method outlined above proved beneficial for normal operation testing to comprehensively verify the design with randomized sequence items without the need for directed testing. For non-normal operation testing (error testing) error conditions also needed to be accounted for in stimulus generation. For this real-world design, when a hardware error was encountered, an interrupt would be asserted. Upon detecting the interrupt, status registers needed to be read to determine what the cause of the error was and if the error was as expected. For our example design, this could be analogous to a multiply operation which was received after an operation that was not an add. In verifying error conditions such as this, it was found that the input sequences would continuously execute without providing sufficient time for the status register reads to take place. This resulted in subsequent traffic corrupting the expected contents of the status registers. To overcome this, new fields were added to the traffic manager to support automated pause functionality with

the sequence. When an interrupt was detected, a pause_traffic field would be set within the traffic manager and used by the sequence to stall the sequence until the status register were read and the pause_traffic field cleared. This avoided the need to add logic to each individual error testcase and contained updates to a single location by taking advantage of the traffic manager. Figure 7 shows the traffic manager with the addition of the pause_traffic field.

```
 1 class traffic_manager #(int OPCODE_WIDTH = `OPCODE_WIDTH_DEFAULT,
 2                         int NUM_OPERANDS = `OPERAND_WIDTH_DEFAULT)
 3                                             extends uvm_object;
 4
 5   // Associative arrays containing
 6   // expected internal state of DUT
 7   bit [OPCODE_WIDTH-1:0] previous_opcode[$];
 8
 9   // Operand Load Indicator
10   bit [NUM_OPERANDS-1:0] data_loaded;
11
12   // Control access to the Traffic Manager
13   semaphore access;
14
15   // Control traffic generation
16   bit [NUM_OPERANDS-1:0] pause_traffic;
17
18   // Add shared variables here
19
20   function new(string name="traffic_manager");
21     super.new(name);
22
23     sa_access = new(1);
24
25   endfunction : new
26
27 endclass: traffic_manager
```

Figure 7: Updated Traffic Manager Class

Figure 8 below shows an example sequence which uses the traffic manager class for sequence item randomization, demonstrates the access permissions flow of the semaphore, and incorporates use of the pause_traffic feature.

```
 1 class seq extends uvm_sequence #(seq_item);
 2
 3   // Sequence Items
 4   seq_item req_item;
 5   seq_item resp_item;
 6
 7   // Traffic Manager
 8   traffic_manager tm;
 9
10   virtual function set_traffic_manager(traffic_manager tm);
11     this.tm = tm;
12   endfunction: set_traffic_manager
13
14   virtual task body();
15
16     // Wait for Traffic Manager to allow this
17     // sequence to drive traffic
18     if (tm.pause_traffic) begin
19       wait (tm.pause_traffic == 0);
20     end
21
22     req_item = seq_item::type_id::create("req_item");
23
24     // Get access to the Traffic Manager
25     tm.access.get(1);
26
27     // Randomize req_item with data from Traffic Manager
28     if (!(std::randomize(req_item with {
29       if (tm.previous_opcode[tm.previous_opcode.size()-1] != ADD) {
30         req_item.opcode != MULTI;
31       }
32     }))) begin
33       `uvm_fatal(REPORT_TAG, "Unable to randomize req_item")
34     end
35
36     // Wait for operands to load
37     if (req_item.opcode != MULTI) begin
38       wait (&tm.data_loaded == 1);
39     end
40
41     // Update the contents of the Traffic Manager
42     // with the randomized req_item
43     update_tm(.item(req_item));
44
45     // Indicate that the loaded data has been consumed
46     tm.data_loaded = 0;
47
48     // Release access to the Traffic Manager
49     tm.access.put(1);
50
51     start_item(req_item);
52     finish_item(req_item);
53     get_response(resp_item);
54
55   endtask: body
56
57 endclass: seq
```

Figure 8: Sequence Using Traffic Manager and Access Controls

The optimal use case for the traffic manager is in constrained randomized testing, however it can be used in conjunction with directed testing as well. For instance, it is common to have raw data or transaction level input files for stimulus generation provided from architecture models. These files are then parsed by the testcase, sent to the driver, and leverage the UVM checkers to perform checking on a specific set of input data. In this setup, instead of having the sequence generate transactions, the testcase can provide the pre-defined data to the sequence which is packed into the sequence item instead of randomizing the sequence item. In the implementation of this at Microsoft, the traffic manager methodology proved beneficial by

leveraging the sequence constraints for legal stimulus which would trigger constraint failures if there were any discrepancies between the model data and the testbench assumptions. For instance, if the model generated stimulus files were incorrect with respect to the design requirements, the sequence constraints would catch these issues immediately through constraint failures. Alternatively, if the testbench was providing constraints that were too narrow, this was also quickly identified and fed back into improving the randomized testing by increasing the randomization scope.

IV. SCALABILITY

The traffic manager and sequences are constructed to be portable and scalable for reuse in verifying interfaces exposed in a subsystem or SoC testbench. Given a subsystem with exposed input interfaces, the traffic manager and sequences can be instances within the subsystem testbench and used directly. This standard method of sequence reuse is complicated for subsystem or SoC testbenches which have multiple instances of a single DUT that need to be verified with comprehensive combinations of random stimulus. To further describe this, consider a case where the example DUT is instanced two times within a subsystem. The DUTs are connected as slaves on a subsystem fabric with one master input interface. Transaction are supplied on the master input interfaces and target either DUT0 or DUT1 as the destination. Figure 9 show a high-level block diagram for this example subsystem.

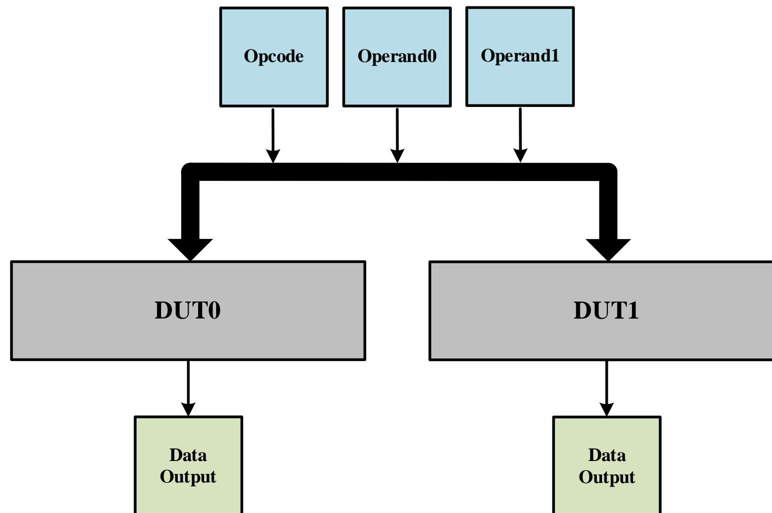

Figure 9: Subsystem Testbench

This example is comparable to a real-world use case at Microsoft in which it was necessary to leverage the sequences and traffic manager in both the subsystem and SoC testbenches. In this use case, the number of DUT instances was greater than 20 and the design's stimulus requirements where complex enough that it was impractical and inefficient to focus on directed or end-to-end testcases in these environments.

The standard UVM approach to reuse the sequences would be to instance the sequence and traffic manager for the input interface to the subsystem. However, this is not a valid solution when there are multiple DUT instances which have independent state information that must be stored in the traffic manager. Since the traffic manager only stores the state for a single DUT instance, it becomes necessary for the sequence to keep track of the state of each DUT to determine legal stimulus. This proved to be an interesting challenge. One potential solution to this problem would be to have multiple instances of the sequence, one sequence for each DUT with its own traffic manager, then use a single sequencer to be an arbiter between the multiple sequences to drive the single input. With the real-world use case at Microsoft having a large number of DUTs, this would have been slow and inefficient, therefore another approach was taken instead.

To work through this challenge, the sequence was updated to support dynamic assignment of the traffic manager to the sequence. This allowed the test writer to set the destination traffic manager in the sequence, before the sequence was started. The sequence would then operate using the state information of the provided traffic manager to ensure the stimulus was legal based on the destination DUT. The traffic manager would be set through a function added in the sequence which used a pass by reference argument to assign the sequence's local traffic manager. This could be changed at any point during the test to target another DUT instance. Instead of creating the traffic manager within the sequence, the test would instance an array of traffic managers for each instance of the DUT. Figure 10 below shows an example usage to create and assign the traffic manager to the sequence. For this test, a randomized set of traffic is sent through the input interface of the fabric for each DUT instance.

```
1 function void build_phase(uvm_phase phase);
2   // ...
3
4   // Create Traffic Managers
5   foreach (traffic_manager[ii]) begin
6     traffic_manager[ii] = traffic_manager::type_id::create(
7                             $sformatf("traffic_manager[%0d]", ii), this);
8   end
9
10  // ...
11 endfunction : build_phase
12
13 function void run_phase(uvm_phase phase);
14  // ...
15
16  // Send transactions to each DUT
17  foreach (traffic_manager[ii]) begin
18    seq.set_traffic_manager(.tm(traffic_manager[ii]));
19    seq.start(seqr);
20  end
21
22  // ...
23 endfunction : run_phase
```

Figure 10: Assigning Traffic Managers to Sequences

V. CONCLUSIONS

This approach to coordinated stimulus generation takes constrained random stimulus a step further by building intelligent randomization based on past and current generated sequence items. This has proven beneficial in a real-world design which had stimulus requirements that were dependent on traffic generated across interfaces. Additionally, the need for more directed tests was eliminated by stressing the DUT and hitting more corner case scenarios through typical random tests.

This methodology was successfully applied for verifying a complex IP design at Microsoft and vertically re-used for verification at the SoC level. In using this methodology for IP verification, coverage holes were quickly identified and resolved by modifying underlying constraints. In reusing this single complex sequence for all design input interface, modifications to source code resulted in a multiplier effect for coverage closure. The need for directed test development was also reduced given this single intelligent sequence which would randomize to a more comprehensive range for a complex series of events. Additionally, in isolating the complexity of the traffic generation into a reusable sequence coupled with the traffic manager, SoC verification bring-up was simplified only requiring time to port and re-use the sequence for the common interfaces at the chip level.