DVCon-2020
San Jose, CA
Voted Best Paper
1st Place

World Class SystemVerilog & UVM Training

World Class Design & Verification Services

UVM Reactive Stimulus Techniques

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com

Heath Chambers
HMC Design Verification, Inc.
hmcdvi@msn.com

Stephen D'Onofrio
Paradigm Works
stephen.D'Onofrio@paradigm-works.com

ABSTRACT

***Abstract* - UVM reactive stimulus techniques allow sequences to receive feedback from a Design Under Test (DUT) to determine what stimulus should be sent next. Existing documentation and examples describe some of the requirements to create sequences and drivers with both request (REQ) type and response (RSP) type parameters, but the descriptions are somewhat incomplete regarding how to create the response (RSP) transaction that is sent back to the sequence.**

This paper describes all of the necessary steps to create efficient reactive stimulus sequences. The paper describes how those techniques can be used to test an example synchronous FIFO design.

Table of Contents

Table of Figures

Table of Examples

Table of Tables

I. Introduction

It is very common for a UVM test to execute a pre-defined set of sequences regardless of the status of the Design Under Test (DUT). An alternate approach is to execute stimulus that reacts to status from the DUT.

Reactive stimulus is stimulus that executes commands based on feedback from the DUT. The execution of stimulus is not just a fixed sequence of commands, but a set of commands that execute until a certain condition is detected. Frequently, stimulus reacts to status bits that are returned from the DUT. For example, the stimulus generation source might execute a series of write and read commands with more frequent write operations until a FIFO is full, then it might execute a series of read commands with intermittent write commands until the FIFO is empty.

UVM drivers, sequencers and sequences can be configured in a UVM test environment to be reactive in nature.

II. uvm_driver & uvm_sequencer Parameters

The **`uvm_driver`** and **`uvm_sequencer`** base classes are both parameterized classes with two parameters each. Each of these classes has a **`type REQ=uvm_sequence_item`** parameter and a **`type RSP=REQ`** parameter. The second parameter is typically only modified if reactive stimulus is used where the **`RSP`** (response transaction type) is different from the **`REQ`** (stimulus transaction type) as shown in Table 1.

Stimulus generation ***without examining response***	`RSP` type not specified
Stimulus generation ***examining response*** using ***same*** transaction type	`RSP` type not specified
Stimulus generation ***examining response*** using ***different*** transaction type	`RSP` type ***IS*** used

Table 1 - REQ / RSP type parameter usage

III. REQ/RSP Handles

The OVM/UVM User Guides use **`req`** and **`rsp`** handles in examples, but they are never explained in the User Guide [2]. This is just one of many places where User Guide examples are confusing and poorly explained.

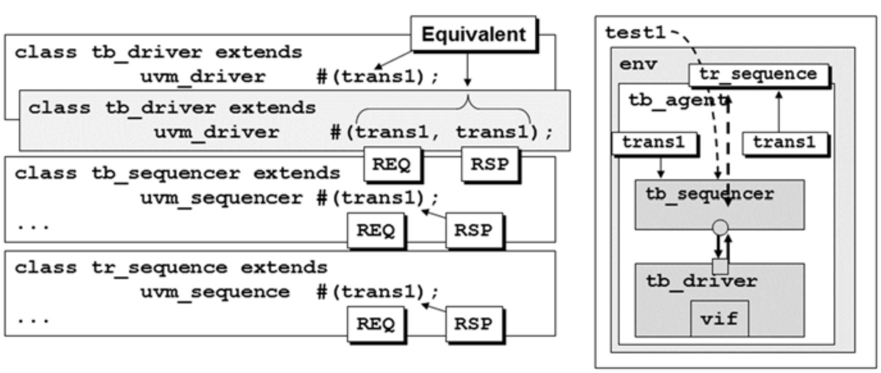

Figure 1 - Same trans1 REQ / RSP parameters used by drivers, sequencers and sequences

Users extend the **uvm_driver** base class with a user-defined driver class (for example: **tb_driver**) and pass the transaction type as the **REQ** parameter (for example **trans1**) as shown in Figure 1. From observation, it appears that most users then declare their own local **trans1 tr** handle and use that **tr** handle to drive stimulus to the virtual interface handle, also declared in the **tb_driver** class. This is actually completely unnecessary as the inherited base class has already declared an **req** handle of the **REQ** parameter type, and this type could be used in the **tb_driver** class in place of the user-declared **tr** handle. That being said, since the **req** handle documentation is both inadequate and frequently misunderstood, we recommend declaring your own local **trans1 tr** handle, which is common practice. The latter is much better understood by most verification engineers.

Similarly, the inherited base class has already declared an **rsp** handle of the **RSP** parameter type, so if a response transaction is returned to the reactive stimulus source, it is unnecessary to declare a local response handle name. Again, this is poorly understood so we recommend declaring your own response handle name.

IV. UVM_DRIVER BASE CLASS PARAMETERS & PORTS

The **uvm_driver** base class is extended from the **uvm_component** base class, and includes the following code:

```
class uvm_driver #(type REQ=uvm_sequence_item,
                   type RSP=REQ) extends uvm_component;

  uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;
  uvm_analysis_port      #(RSP)      rsp_port;

  REQ req;
  RSP rsp;
```

Figure 2 - uvm_driver class header & declarations

The **uvm_driver** base class header in Figure 2, shows that this class is parameterized with two type parameters.

Type parameter #1: **type REQ=uvm_sequence_item**

The first type parameter defines the **REQ** (request) type with a default value of **uvm_sequence_item** base class type. All user-defined transaction types are derivatives of the **uvm_sequence_item** type, so any transaction type can be assigned to the **REQ** type.

UVM compares the driver **REQ** type to the sequencer **REQ** type to ensure that there is a type match between the sequencer and the driver. If the types do not match, UVM issues a fatal error message. Depending on the simulator, the fatal message is reported in either the compilation or simulation steps.

Type parameter #2: **type RSP=REQ**

The second type parameter defines the **RSP** (response) type and the default value is also the **uvm_sequence_item** base class type. All user-defined transaction types are derivatives of the **uvm_sequence_item** type, so any transaction type can be assigned to the **RSP** type. As shown in Figure 2, the default **RSP** type is the same as the **REQ** type.

V. UVM_SEQUENCE BASE CLASS PARAMETERS & PORTS

The **uvm_sequence** base class is extended from the **uvm_sequence_base** base class, and includes the following code:

```
virtual class uvm_sequence #(type REQ = uvm_sequence_item,
                             type RSP = REQ) extends uvm_sequence_base;
  ...

  REQ req;
  RSP rsp;
```

Figure 3 - uvm_sequence class header & declarations

The **uvm_sequence** base class header in Figure 3, shows that this class is also parameterized with same two type parameters that are used in the **uvm_driver** base class, as described in Section IV.

The user-define **tr_sequence** class shown in Figure 4, is extended from the **uvm_sequence** base class shown in Figure 3. The sequence definition includes declarations and factory creation of the request (**tr**) transaction and might include a declaration for a response (**rsp**) transaction. Each command called by the **body()** task uses the request (**tr**) transaction and, if the sequence is reactive, the command will also get the response (**rsp**) transaction.

```
class tr_sequence extends uvm_sequence #(trans1);
  ...
  trans1 tr  = trans1::type_id::create("tr");
  trans1 rsp;                                // Used by reactive sequence
  ...
  task body;
    command1();
    ...
  endtask

  task command1;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    get_response(rsp); // Used by reactive sequence
    ...
  endtask
...
```

Figure 4 - User defined tr_sequence class header & common body() and command task

The user can use **req** of the **REQ** type if desired, or declare another request type to use, as shown in Figure 4. The UVM User Guide shows examples that use the **req** transaction type without describing where the **req** type comes from. As can be seen above, the **req** type is inherited from the **uvm_sequence** base class. That being said, most users declare their own request transaction type.

The user can also use **rsp** of the **RSP** type if desired or declare another response type to use as shown in Figure 5. The default **RSP** type matches the **REQ** type, but a user can choose to use a second response type. Most users tend to use the same default transaction type as the request transaction type.

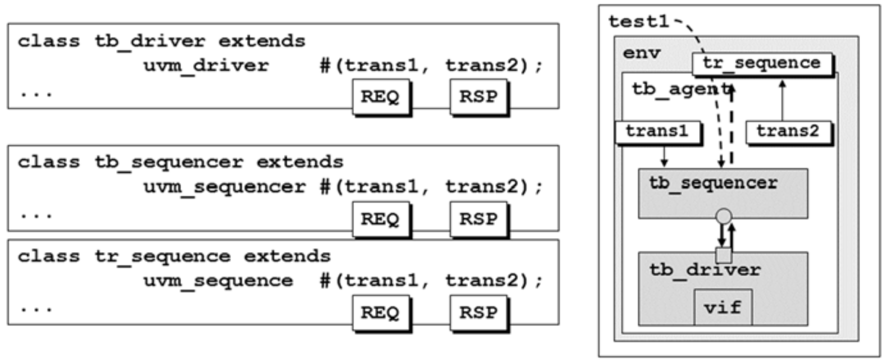

Figure 5 - Different trans1/trans2 REQ/RSP parameters used by drivers, sequencers & sequences

VI. Response Transaction Usage

When the test is required to examine some form of DUT status to calculate the next desired stimulus, then a response transaction is returned to the stimulus source for examination.

Using response transactions typically requires small but important modifications to the driver, sequencer and sequence coding styles that are not required in non-reactive stimulus generation. The following subsections give an overview of those high-level actions.

A. General

The driver, sequencer and sequence must all declare the same stimulus and response transaction handles. If the same transaction type is used for both the stimulus and response, then the driver, sequencer and sequence will just declare the same transaction handle types. If a different transaction type is used to pass a response to the sequence, then the driver, sequencer and sequence will list the second transaction type as additional class parameters as was shown in Figure 5.

B. Driver

For reactive stimulus generation, the driver needs both stimulus (**tr**) and response (**rsp**) transaction handles. The handles might be declared as the same transaction type as shown in Example 1, or they might be declared as different transaction types as shown in Example 2. The driver factory-creates the stimulus (**tr**) transaction but only declares a response (**rsp**) transaction handle

```
trans1 tr  = trans1::type_id::create("tr");
trans1 rsp;
```

Example 1 - Stimulus and response, both of the trans1 type

```
trans1 tr  = trans1::type_id::create("tr");
trans2 rsp;
```

Example 2 - Stimulus and response, declared as different trans1 and trans2 types

The driver's **run_phase()** needs a modified version of the **forever** loop with the following three modifications, (1) after getting the next item, the response needs to capture the transaction id using the **rsp.set_id_info(tr)**, (2) the **drive_item(tr, rsp)** includes both the **tr** and **rsp** handles, and (3) the

seq_item_port.item_done(rsp) method must return the response transaction handle. These modifications are highlighted in Example 3.

```
task run_phase(uvm_phase phase);
  trans1 tr;
  ...
  forever begin
    seq_item_port.get_next_item(tr);
    rsp.set_id_info(tr);
    drive_item(tr, rsp);
    seq_item_port.item_done(rsp);
  end
endtask
```

Example 3 - Driver run_phase() modifications for reactive stimulus generation

C. Sequencer

For reactive stimulus generation, the sequencer needs both stimulus (**tr**) and response (**rsp**) transaction handles declared in the class header as was shown in Figure 5. This is the only modification required for the reactive version of the sequencer.

D. Reactive Sequence

The reactive sequence needs access to the DUT outputs, including the important status signals, that were sampled by the **drive_item()** task in the driver's **forever** loop. The driver sent the response transaction handle (**rsp**) back through the sequencer to the reactive sequence using the **seq_item_port.item_done(rsp)** command shown in Example 3, and it is the reactive sequence's job to retrieve that handle using the **get_response(rsp)** after the **finish_item(tr)** command as highlighted in Example 4.

```
task command1;
  start_item(tr);
    …randomize transaction…
  finish_item(tr);
  get_response(rsp);
  ...
endtask
```

Example 4 - Command task example used by a reactive sequence

VII. Stimulus Coding Techniques

Stimulus can be coded at least three different ways, (1) send stimulus without examining any type of response status, (2) send stimulus and examine the response status using the same transaction type, and (3) send stimulus and examine the response status using the a different transaction type.

The first two styles do not require any special modifications to the sequencer code, while the third style simply requires the sequencer code to be parameterized to both request and response transaction types.

The coding considerations for the three styles are described in this section.

A. Stimulus without response transactions

UVM verification engineers are generally familiar with sequences and drivers that do not have response transactions.

The class headers for the **tr_sequence**, **tb_sequencer** and **tb_driver** shown in Example 5, Example 6 and Example 7 are all required to be parameterized to the same transaction type, **trans1** in these examples.

```
class tr_sequence extends uvm_sequence #(trans1);
...
  task body;
    command1();
    command2();
    ...
  endtask

  task command1;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    ...
  endtask

  task command2;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    ...
  endtask
...
```

Example 5 - Sequence with no response transaction

```
class tb_sequencer extends uvm_sequencer #(trans1);
...
```

Example 6 - Sequencer with no response transaction

```
class tb_driver extends uvm_driver #(trans1);

  task run_phase(uvm_phase phase);
    trans1 tr;
    ...
    forever begin
      seq_item_port.get_next_item(tr);
      drive_item(tr);
      seq_item_port.item_done();
    end
  endtask
...
```

Example 7 - Driver with no response transaction

As shown in the **tb_driver** of Example 7, stimulus without response transactions finishes communicating with the sequencer by issuing the command: **seq_item_port.item_done();**

Note that the **item_done()** command does not return a transaction handle. Non-response stimulus that returns a transaction handle is a mistake, which causes sequencer FIFO overflows as describe in Section XI.

B. Stimulus with matching request and response transaction types

There is reasonable existing documentation that shows how to code sequences and drivers with matching request and response transaction types.

When sequences, sequencers and drivers use matching request and response transaction types (**tran1** in these examples), the class headers for the **tr_sequence** and **tb_driver** shown in Example 8 and Example 9 are the same as the class headers used in the non-response versions of **tr_sequence** and **tb_driver** shown in Example 5 and Example 7. It should be noted that there is no modification required to use the **tb_sequencer** shown in Example 6 in a UVM testbench with matching request and response transaction types.

The user-defined sequence declares and creates the **trans1 tr** object and declares but does not create the **trans1 rsp** transaction object. The **body()** task of a sequence typically calls other command tasks, which in turn might call additional nested command tasks.

Each **command()** task called by the **body()** task is going to **start_item(tr)**, randomize the transaction, then issue the **finish_item(tr)** command. The **command()** task will then do a **get_response(rsp)** command to get the returned response-transaction as shown below in Example 8. The **rsp** handle can then be used by the sequence to test individual response fields, as will be shown in Section X.

```
class tr_sequence_rsp extends… #(trans1);
  ...
  trans1 tr  = trans1::type_id::create("tr");
  trans1 rsp;
  ...
  task body;
    command1();
    command2();
    ...
  endtask

  task command1;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    get_response(rsp);
    ...
  endtask

  task command2;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    get_response(rsp);
    ...
  endtask
...
```

Example 8 - Sequence with response transaction

```
class tb_driver extends uvm_driver #(trans1);
  ...
  task run_phase(uvm_phase phase);
    trans1 tr;
    trans1 rsp;
    ...
    forever begin
      seq_item_port.get_next_item(tr);
      rsp.set_id_info(tr);
      drive_item(tr, rsp);
      seq_item_port.item_done(rsp);
    end
  endtask
```

```
    ...
    task drive_item (trans1 tr, output trans1 rsp);
      trans1 resp;
      if(!$cast(resp,tr.clone())) `uvm_fatal("DRVI", "cast to resp failed")
      vif.cb1.rst_n <= tr.rst_n;
      vif.cb1.din   <= tr.din;
      vif.cb1.write <= tr.write;
      vif.cb1.read  <= tr.read;
      @vif.cb1;
       // copy all outputs at end of cycle to resp
      resp.full  = vif.cb1.full;
      resp.af    = vif.cb1.af;
      resp.empty = vif.cb1.empty;
      resp.ae    = vif.cb1.ae;
      resp.dout  = vif.cb1.dout;
      rsp = resp; // copy resp handle to drive_item rsp output handle
    endtask
  endclass
```

Example 9 - Driver with response transaction

The **tb_driver run_phase()** shown in the top part of Example 9, uses a **forever** loop that executes the following set of commands:

1. **seq_item_port.get_next_item(tr)**
2. The **rsp** transaction needs to set the id (**rsp.set_id_info(tr)**) of the driven transaction so that responses can be matched to the corresponding driven transactions.
3. Drive the item executing the user-defined **drive_item(tr, rsp)** task with both stimulus transaction and response transaction handles, **tr** and **rsp**.
4. Then finish communication with the sequencer using the **seq_item_port.item_done(rsp)** command.

Note that the **item_done(rsp)** command returns a response transaction (**rsp**) handle, which will be passed through the sequencer back to the sequence. The sequence may then use response transaction fields to determine the next set of commands to be executed by the sequence.

The **drive_item()** method shown at the bottom of Example 9 has input and output **trans1** handles called **tr** and **rsp**. A **trans1 resp** handle must then be declared beneath the **drive_item()** prototype, but the creation of this local **resp** object is accomplished by cloning of the **tr** input transaction handle using the command **if(!$cast(resp,tr.clone())) …**

The **resp** object, cloned from the **tr** input transaction, now has a copy of both the input and output transaction fields. The input transaction fields will be returned with the response transaction, but the output transaction fields are only copied from the virtual interface *after* the **drive_item()** method is first synchronized to the next **posedge clk** (**@vif.cb1**) and then the output fields are sampled just before the **@vif.cb1** using the clocking block defined inside of the interface. The reason for sampling the outputs at the end of the cycle are described in Cummings [1]. The sampled outputs have now been placed into the **resp** transaction and the final step is to copy the **resp** transaction handle to the output **rsp** handle.

The sampling of the transaction outputs at the end of the cycle and copying them to the output fields of the **resp** transaction is the step that is poorly documented in most sources that we have examined.

C. Stimulus with different request and response transaction types

When creating responsive stimulus, verification engineers typically use the same transaction type for both the request and response types. Verification engineers must include declarations for the desired response fields in this one and only transaction type and frequently these response fields are status bits that were already defined in the transaction. This makes it possible to use the simple responsive stimulus techniques described in the previous subsection.

If a verification engineer has a compelling reason to use a different response transaction type, small modifications to the previous procedure are required.

For the transaction sequence, the first difference is that the **`tr_sequence`** prototype is now parameterized to two different transaction types (**`trans1`**, **`trans2`**) as shown in the class header of Example 10.

The second difference is that both transaction types must be declared and factory-constructed inside of the **`tr_sequence`** class, also highlighted near the top of Example 10. The rest of the transaction code is the same as was shown in the **`tr_sequence`** of Example 8.

The second difference is that both transaction types must be declared inside of the **`tr_sequence`** class, but only the request (**`tr`**) transaction handle must be factory created since the **`get_response(rsp)`** command gets a constructed response from the sequencer's response queue. The rest of the transaction code is the same as was shown in the **`tr_sequence`** of Example 8.

```
class tr_sequence extends uvm_sequence #(trans1, trans2);
  ...
  trans1 tr  = trans1::type_id::create("tr");
  trans2 rsp;
  ...
  task body;
    command1();
    command2();
    ...
  endtask

  task command1;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    get_response(rsp);
    ...
  endtask

  task command2;
    start_item(tr);
      …randomize transaction…
    finish_item(tr);
    get_response(rsp);
    ...
  endtask
...
```

Example 10 - Sequence with different response transaction type

The driver must also be declared with two transaction types in the class header, where the second transaction type is the response type, as highlighted in Example 11. Even though declaring the **RSP** type to be **`trans2`**, and even though the driver would inherit the declaration **`trans2 rsp;`** (since **RSP** = **`trans2`**) we still recommend that engineers make the **`trans2 rsp;`** declaration shown near the top of Example 11. As long as the response handle name matches **`rsp`**, this declaration is not really necessary.

```
class tb_driver extends uvm_driver #(trans1, trans2);
  ...
  trans2 rsp;
  ...
  task run_phase(uvm_phase phase);
    trans1 tr;
    ...
    forever begin
      seq_item_port.get_next_item(tr);
      drive_item(tr, rsp);
      rsp.set_id_info(tr);
      seq_item_port.item_done(rsp);
    end
  endtask
  ...
  task drive_item (trans1 tr, output trans2 rsp);
    trans2 resp = trans2::type_id::create("resp");
    vif.cb1.rst_n <= tr.rst_n;
    vif.cb1.din   <= tr.din;
    vif.cb1.write <= tr.write;
    vif.cb1.read  <= tr.read;
    @vif.cb1;
     // sample all outputs at end of cycle into resp handle
    resp.full  = vif.cb1.full;
    resp.af    = vif.cb1.af;
    resp.empty = vif.cb1.empty;
    resp.ae    = vif.cb1.ae;
    resp.dout  = vif.cb1.dout;
    rsp = resp; // copy resp handle to drive_item rsp output handle
  endtask
endclass
```

Example 11 - Driver run_phase() with drive_item(tr,rsp) and rsp.set_id_info(tr)

The second difference in the driver is that the `trans2 resp` handle inside of the `drive_item()` task is typically not cloned from the `tr` input transaction, as it was in Example 9. This is because the second transaction type might not include all of the input fields that were defined in the `trans1` transaction type and therefore the second transaction type might not be `$cast`-compatible with the `tr` transaction type. Instead, in the `drive_item()` task, the sampled output fields required by the `resp` transaction type are sampled using the clocking block timing in the virtual interface and directly assigned to the output fields of the `resp` transaction.

VIII. Reactive Stimulus Test Plan for 1-Clock FIFO Example

To demonstrate the use of reactive stimulus, we will use as an example a well-understood design block, a 1-clock (synchronous), 16-deep FIFO design. The test plan for this type of FIFO might include the following desirable stimulus generation operations:

- **Write** specific word to FIFO.
- **Read** word from FIFO.
- Random **Block-writes** to FIFO with fixed word pattern.
- Random **Block-writes** to FIFO with random word patterns.
- Random **Block-reads** from FIFO.
- **Write until not Almost Empty** (AE) - intended to take the design from FIFO-empty to a safe fill-depth by continuously writing patterns - this requires reactive stimulus to monitor the AE flag on the DUT.
- **Write until Almost Full** - this requires reactive stimulus to monitor the AF flag on the DUT.

- **Write until Full** - this requires reactive stimulus to monitor the Full flag on the DUT.
- Attempt to **write past full**, which should not succeed. This can be accomplished by writing until full and then performing more random write commands without simultaneous read commands. This tests to make sure that the FIFO Full flag does not change and that the current FIFO contents are not over-written.
- **Read until Almost Empty** - this requires reactive stimulus to monitor the AE flag on the DUT.
- **Read until Empty** - this requires reactive stimulus to monitor the Empty flag on the DUT.
- Attempt to **read past empty**, which should not succeed. This can be accomplished by reading until empty and then performing more random read commands without simultaneous write commands. This tests to make sure that the FIFO Empty flag does not change.
- **Reset** the FIFO - this should cause the Empty flag to be set (if not already set) and the Full flag to be cleared (if it was set). Reset testing should be executed in at least three scenarios: (1) when the FIFO is already empty, (2) when the FIFO is neither full nor empty, and (3) when the FIFO is full.

Five of the above operations require the examination of DUT status and therefore use reactive stimulus. The five operations that use reactive stimulus are: (1) **Write until not Almost Empty**, (2) **Write until Almost Full**, (3) **Write until Full**, (4) **Read until Almost Empty**, (5) **Read until Empty**.

IX. FIFO Design tb_driver - Development of the drive_item() Method

The testing of the 1-clock, 16-deep FIFO design is described in this section. The **`drive_item()`** task used in the **`tb_driver`** to test this FIFO example was shown in Example 9. In this section, the development of the different parts of the **`drive_item()`** task will be explained in more detail.

The **`drive_item()`** task for the FIFO design samples the FIFO flag outputs at the end of the cycle and stores them into the response transaction. The step that is poorly described in descriptions of request/response stimulus generation is how the response transaction fields are sampled and assigned.

In the FIFO example, the FIFO inputs, reset (**`rst_n`**), input data (**`din`**), **`write`** and **`read`** signals, are driven to the DUT 20% of the cycle past the **`posedge clk`**. The recommended use of the 20% / 80% stimulus generation times is described in [1].

The driving of the inputs is accomplished in the **`drive_item()`** task snippet in the user-define **`tb_driver`**, as shown below:

```
task drive_item (input trans1 tr, ... );
  ...
  `uvm_info("drive_item", tr.input2string(), UVM_FULL)
  ...
  vif.cb1.rst_n <= tr.rst_n;
  vif.cb1.din   <= tr.din;
  vif.cb1.write <= tr.write;
  vif.cb1.read  <= tr.read;

  @vif.cb1;   // Synchronizes to the next active clock edge

  ...
endtask
```

Example 12 - Driver drive_item() method part 1 - drives DUT inputs using clocking block timing

For sequences that use response transactions, the response transaction might return both the driven inputs and the sampled outputs.

In part 2, as shown in the modified snippet of Example 13, a **`trans1 resp`** handle has been declared just below the **`drive_item`** task prototype and then the **`input trans1 tr`** handle was cloned and **`$cast`** to the **`resp`** handle. Now the **tr** and **resp** handles are identical, but we do not recommend using the **resp** version of the input signals to drive the DUT inputs. The reason to *not* use the **resp** inputs in the **drive_item()** task is because an existing **drive_item()** task would typically already be driving the **tr** version of the inputs and we want to minimize changes in case we decide to remove the response sampling in the future for whatever reason.

```
task drive_item (input trans1 tr, ... );
  trans1 resp;
  `uvm_info("drive_item", tr.input2string(), UVM_FULL)
  if(!$cast(resp,tr.clone()))
      `uvm_fatal("DRVI", "cast to resp failed")
  vif.cb1.rst_n <= tr.rst_n;
  vif.cb1.din   <= tr.din;
  vif.cb1.write <= tr.write;
  vif.cb1.read  <= tr.read;
  @vif.cb1;
  ...
endtask
```

Example 13 - drive_item() part 2 - uses cloned response transaction

The FIFO outputs, **full**, **af** (almost full), **empty**, **ae** (almost empty), and output data (**dout**) signals, should be sampled at the end of the cycle just before the next **posedge clk**.

In part 3, as shown in the modified snippet of Example 14, an output **trans1 rsp** handle declaration has been added to the header of the **drive_item()** task. At the end of the cycle, the **drive_item()** task re-synchronizes to the next **posedge clk** defined in the interface clocking block using the **@vif.cb1** syntax, and then uses clocking block timing to sample the outputs *just before* the **posedge clk** edge using the **vif.cb1.*signal_name*** syntax. Each sampled signal is then assigned to the corresponding output signal in the **resp** transaction. Finally the **resp** transaction handle is copied to the **drive_item output trans1 rsp** handle, to be returned to the calling transaction sequence.

```
task drive_item (input trans1 tr, output trans1 rsp);
  trans1 resp;
  `uvm_info("drive_item", tr.input2string(), UVM_FULLG)
  if(!$cast(resp,tr.clone()))
      `uvm_fatal("DRVI", "cast to resp failed")
  vif.cb1.rst_n <= tr.rst_n;
  vif.cb1.din   <= tr.din;
  vif.cb1.write <= tr.write;
  vif.cb1.read  <= tr.read;
  @vif.cb1;
  // copy all outputs at end of cycle to resp
  resp.full  = vif.cb1.full;
  resp.af    = vif.cb1.af;
  resp.empty = vif.cb1.empty;
  resp.ae    = vif.cb1.ae;
  resp.dout  = vif.cb1.dout;
  rsp = resp;
endtask
```

Example 14 - drive_item() part 3 - samples DUT transaction outputs using clocking bock timing

This last part of sampling DUT outputs at the end of the cycle and assigning them to the response transaction is what is frequently missing from existing examples. The response transaction now has the sampled FIFO output flags and output data at the end of the cycle, which can be examined and tested in the reactive sequence.

X. FIFO Design Command Tasks

Many of the FIFO **tr_sequence** command tasks randomize the transaction data fields and it is important that the randomization be tested to ensure that the constraints are met. Since this randomization is a common activity, we included a macro definition to print a consistent "RANDOMIZE FAIL" message as shown in Example 15.

A. *RANDOMIZE_FAIL message macro*

Each call to **tr.randomize()** in the **reset()**, **do_item()**, **write()** and **read()** tasks calls the common **RANDOMIZE_FAIL** macro that was placed just before the **tr_sequence** class, as shown in Example 15.

```
`ifndef RANDOMIZE_FAIL
  `define RANDOMIZE_FAIL \
          `uvm_fatal("TR_S", "tr_sequence randomization failed")
`endif
```

Example 15 - Common RANDOMIZE_FAIL macro

The **tr_sequence** class declares the stimulus and response transactions **tr** and **rsp** respectively, and then factory-creates the stimulus transaction **tr**, as shown in Example 16.

```
class tr_sequence extends uvm_sequence #(trans1);
  `uvm_object_utils(tr_sequence)

  trans1 tr  = trans1::type_id::create("tr");
  trans1 rsp;

  function new (string name = "tr_sequence");
    super.new(name);
  endfunction
```

Example 16 - Transaction sequence declares and creates transaction and response-transaction

B. *FIFO tr_sequence body() task*

The **body()** task (stimulus command source) of the **tr_sequence** is shown below in Example 17 and the sequence executes the following stimulus actions:

- Line 2 - The stimulus first resets the FIFO for two clock periods.
- Line 3 - Then completely fills the FIFO.
- Line 4 - Later, after the FIFO is detected to be full, the stimulus reads the FIFO until it is empty.
- Line 5 - The FIFO is written until it is past the Almost Empty mark.
- Line 6 - Then 6 random read/write commands are issued.
- Line 7 - The FIFO is then written until it is Almost Full.
- Line 8 - Then 10 random read/write commands are issued.
- Line 9 - The FIFO is written until full.
- Line 10 - An attempt is made to randomly do 4-8 additional write commands, which should not change anything in the FIFO.
- Line 11 - Read until the FIFO is Almost Empty.
- Line 12 - Write until the FIFO is full.
- Line 13 - Read until the FIFO is empty.
- Line 14 - An attempt is made to randomly do 5-9 additional read commands, which should not change anything in the FIFO.
- Line 15 - Write until the FIFO is Almost Full.
- Line 16 - Do 100 random read/write commands. And finish this sequence.

```
task body;
  repeat(2) reset(tr);
  write_until_full(tr);
  read_until_empty(tr);
  write_until_not_AE(tr);
  repeat(6) do_item(tr);
  write_until_AF(tr);
  repeat(10) do_item(tr);
  write_until_full(tr);
  repeat($urandom_range(4,8)) write(tr);
  read_until_AE(tr);
  write_until_full(tr);
  read_until_empty(tr);
  repeat($urandom_range(5,9)) read(tr);
  write_until_AF(tr);
  repeat(100) do_item(tr);
endtask
```

Example 17 - Sequence body task to test FIFO design

C. reset() and do_item() tasks

There are two general purpose testing tasks called **`reset()`** and **`do_item()`**. The **`reset()`** task does randomization with **`tr.rst_n`** asserted as shown in Example 18, while the **`do_item()`** task does randomization with **`tr.rst_n`** disabled, as shown in Example 19. The **`do_item()`** task will randomly generate **`write()`** and **`read()`** commands.

```
task reset (trans1 tr);
  `uvm_info("do_item", "executing", UVM_FULL)
  start_item(tr);
  if (!(tr.randomize() with {tr.rst_n=='0;})) `RANDOMIZE_FAIL
  finish_item(tr);
  get_response(rsp);
  `uvm_info("reset", tr.convert2string(), UVM_DEBUG)
endtask
```

Example 18 - reset() task

```
task do_item (trans1 tr);
  `uvm_info("do_item", "executing", UVM_FULL)
  start_item(tr);
  if (!(tr.randomize() with {tr.rst_n=='1;})) `RANDOMIZE_FAIL
  finish_item(tr);
  get_response(rsp);
  `uvm_info("do_item", tr.convert2string(), UVM_DEBUG)
endtask
```

Example 19 - do_item() task

D. FIFO write commands

The FIFO write commands are composed of the following simulation tasks:

`write()`, which does the **`start_item(tr)`** command, followed by transaction randomization with inline constraint that sets the **`tr.write`** bit, clears the **`tr.read`** bit and disables the **`tr.rst_n`** input. Then the **`write()`** command completes by calling the **`finish_item(tr)`** and **`get_response(tr)`** commands.

```
task write(trans1 tr);
  start_item(tr);
  if (!(tr.randomize() with {tr.write=='1; tr.read=='0;
                             tr.rst_n=='1;})) `RANDOMIZE_FAIL
  finish_item(tr);
  get_response(rsp);
  `uvm_info("FLAGS", sample_flags(rsp), UVM_HIGH)
endtask
```

Example 20 - FIFO write () command task

Three additional reactive write commands call this **write()** command:

write_until_full(trans1 tr) uses a **while (!rsp.full)** loop to continue writing until **rsp.full** is detected in the response, as shown in Example 21. This task also prints the message "**starting write_until_full**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. The **HIGH** verbosity message can be helpful during test and sequence development and the sample test-run shown in Figure 6 shows these messages enlarged in the simulation output transcript.

```
task write_until_full(trans1 tr);
  `uvm_info("body", "\n\nstarting write_until_full\n", UVM_HIGH)
  while (!rsp.full) write(tr);
endtask
```

Example 21 - FIFO write_until_full() command task

write_until_AF(trans1 tr) uses a **while (!rsp.af)** (while not Almost-Full) loop to continue writing until **rsp.af** is detected in the response, as shown in Example 22. This task also prints the message "**starting write_until_AF**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. Figure 6 shows these messages enlarged in the simulation output transcript.

```
task write_until_AF(trans1 tr);
  `uvm_info("body", "\n\nstarting write_until_AF\n", UVM_HIGH)
  while (!rsp.af) write(tr);
endtask
```

Example 22 - FIFO write_until_AF() command task

write_until_not_AE(trans1 tr) uses a **while (rsp.ae)** (while Almost-Empty) loop to continue writing while **rsp.ae** is still true in the response, as shown in Example 23. This task also prints the message "**starting write_until_not_AE**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. Figure 6 shows these messages enlarged in the simulation output transcript.

This command is used after resetting the FIFO to continue writing until the Almost Empty flag is cleared, which allows data values to partially fill the FIFO buffer right after releasing reset.

```
task write_until_not_AE(trans1 tr);
  `uvm_info("body", "\n\nstarting write_until_not_AE\n", UVM_HIGH)
  while (rsp.ae) write(tr);
endtask
```

Example 23 - FIFO write_until_not_AE() command task

E. FIFO read commands

The FIFO read commands are composed of the following simulation tasks:

read(), which does the **start_item(tr)** command, followed by a transaction randomization with inline constraint that clears the **tr.write** bit, sets the **tr.read** bit and disables the **tr.rst_n** input. Then the **read()** command completes by calling the **finish_item(tr)** and **get_response(tr)** commands.

```
task read(trans1 tr);
  start_item(tr);
  if (!(tr.randomize() with {tr.write=='0; tr.read=='1;
                              tr.rst_n=='1;})) `RANDOMIZE_FAIL
  finish_item(tr);
  get_response(rsp);
  `uvm_info("FLAGS", sample_flags(rsp), UVM_HIGH)
endtask
```

Example 24 - FIFO read() command task

Two additional reactive read commands call this **read()** command:

read_until_empty(trans1 tr) uses a **while (!rsp.empty)** loop to continue reading until **rsp.empty** is detected in the response, as shown in Example 25. This task also prints the message "**starting read_until_empty**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. Figure 6 shows these messages enlarged in the simulation output transcript.

```
task read_until_empty(trans1 tr);
  `uvm_info("body", "\n\nstarting read_until_empty\n", UVM_HIGH)
  while (!rsp.empty) read(tr);
endtask
```

Example 25 - FIFO read_until_empty() command task

read_until_AE(trans1 tr) uses a **while (!rsp.ae)**, (while not Almost-Empty), loop to continue reading until **rsp.ae** is detected in the response, as shown in Example 26. This task also prints the message "**starting read_until_AE**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. Figure 6 shows these messages enlarged in the simulation output transcript.

```
task read_until_AE(trans1 tr);
  `uvm_info("body", "\n\nstarting read_until_AE\n", UVM_HIGH)
  while (!rsp.ae) read(tr);
endtask
```

Example 26 - FIFO read_until_AE() command task

F. sample_flags() method

The **write()** and **read()** commands, which are also called by the other **write**-variation and **read**-variation commands, both call the **sample_flags()** method shown in Example 27 to display the **full** / **af** / **ae** / **empty** flags when run-time simulation verbosity is increased to **UVM_HIGH**

```
function string sample_flags(trans1 rsp);
  return($sformatf("full=%b / af=%b / ae=%b / empty=%b",
                   rsp.full, rsp.af, rsp.ae, rsp.empty));
endfunction
```

Example 27 - sample_flags() function

G. *FIFO simulation printout*

An abbreviated printout of the simulation results with **+UVM_VERBOSITY=HIGH** is shown in Figure 6. The write and read task messages have been enlarged to help review the simulation transcript. Also many of the individual write and read command display messages have been removed and replaced by "..." to help show the abbreviated results.

```
UVM_INFO @ 1: uvm_test_top.e.agnt.drv [INIT] Initialize (time @0)
UVM_INFO @ 5: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=xx  write=x  read=x  rst_n=0  full=0  af=0  empty=1  ae=1  dout=xx RESET  Actual:full=0  af=0  empty=1  ae=1  dout=xx RESET
UVM_INFO @ 15: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=ff  write=1  read=1  rst_n=0  full=0  af=0  empty=1  ae=1  dout=xx RESET  Actual:full=0  af=0  empty=1  ae=1  dout=ff RESET
UVM_INFO @ 25: uvm_test_top.e.agnt.sqr@@seq [body]
starting write_until_full
UVM_INFO @ 25: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=cc  write=0  read=0  rst_n=0  full=0  af=0  empty=1  ae=1  dout=xx RESET  Actual:full=0  af=0  empty=1  ae=1  dout=ff RESET
UVM_INFO @ 35: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 35: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=eb  write=1  read=1  rst_n=0  full=0  af=0  empty=1  ae=1  dout=xx RESET  Actual:full=0  af=0  empty=1  ae=1  dout=eb RESET
UVM_INFO @ 45: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 45: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=7d  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=1  dout=7d  Actual:full=0  af=0  empty=0  ae=1  dout=7d
UVM_INFO @ 55: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
...
UVM_INFO @ 175: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=48  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=7d  Actual:full=0  af=1  empty=0  ae=0  dout=7d
UVM_INFO @ 185: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0
UVM_INFO @ 185: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=f2  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=7d  Actual:full=0  af=1  empty=0  ae=0  dout=7d
UVM_INFO @ 195: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 195: uvm_test_top.e.agnt.sqr@@seq [body]
starting read_until_empty
UVM_INFO @ 195: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=5d  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=7d  Actual:full=1  af=1  empty=0  ae=0  dout=7d
UVM_INFO @ 205: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 205: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=57  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=7d  Actual:full=1  af=1  empty=0  ae=0  dout=7d
UVM_INFO @ 215: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0
...
UVM_INFO @ 345: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=91  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=1  dout=f2  Actual:full=0  af=0  empty=0  ae=1  dout=f2
UVM_INFO @ 355: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 355: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=f9  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=1  dout=5d  Actual:full=0  af=0  empty=0  ae=1  dout=5d
UVM_INFO @ 365: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 365: uvm_test_top.e.agnt.sqr@@seq [body]
starting write_until_not_AE
UVM_INFO @ 365: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=06  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=7d IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=7d IGNORED
UVM_INFO @ 375: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 375: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=85  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=7d IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=7d IGNORED
UVM_INFO @ 385: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 385: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=6f  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=1  dout=6f  Actual:full=0  af=0  empty=0  ae=1  dout=6f
UVM_INFO @ 395: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
...
UVM_INFO @ 465: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=17  write=0  read=0  rst_n=1  full=0  af=0  empty=0  ae=0  dout=65  Actual:full=0  af=0  empty=0  ae=0  dout=65
UVM_INFO @ 475: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=bd  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=0  dout=65  Actual:full=0  af=0  empty=0  ae=0  dout=65
UVM_INFO @ 485: uvm_test_top.e.agnt.sqr@@seq [body]
starting write_until_AF
UVM_INFO @ 485: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=5d  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=f3  Actual:full=0  af=0  empty=0  ae=0  dout=f3
UVM_INFO @ 495: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
UVM_INFO @ 495: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=40  write=1  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=06  Actual:full=0  af=0  empty=0  ae=0  dout=06
UVM_INFO @ 505: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
...
UVM_INFO @ 645: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=ab  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=40  Actual:full=0  af=0  empty=0  ae=0  dout=40
UVM_INFO @ 655: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=2f  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=ee  Actual:full=0  af=0  empty=0  ae=0  dout=ee
UVM_INFO @ 665: uvm_test_top.e.agnt.sqr@@seq [body]
starting write_until_full
UVM_INFO @ 665: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=32  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=0  dout=ee  Actual:full=0  af=0  empty=0  ae=0  dout=ee
UVM_INFO @ 675: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
UVM_INFO @ 675: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=e3  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=4e  Actual:full=0  af=0  empty=0  ae=0  dout=4e
UVM_INFO @ 685: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
...
UVM_INFO @ 755: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=c8  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=4e  Actual:full=1  af=1  empty=0  ae=0  dout=4e
UVM_INFO @ 765: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 765: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=30  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=4e  Actual:full=1  af=1  empty=0  ae=0  dout=4e
UVM_INFO @ 775: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 775: uvm_test_top.e.agnt.sqr@@seq [body]
starting read_until_AE
UVM_INFO @ 775: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=63  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=4e  Actual:full=1  af=1  empty=0  ae=0  dout=4e
UVM_INFO @ 785: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 785: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=5d  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=4e  Actual:full=1  af=1  empty=0  ae=0  dout=4e
UVM_INFO @ 795: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0
...
UVM_INFO @ 885: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=7c  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=f9  Actual:full=0  af=0  empty=0  ae=0  dout=f9
UVM_INFO @ 895: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
UVM_INFO @ 895: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=e7  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=0  dout=35  Actual:full=0  af=0  empty=0  ae=0  dout=35
UVM_INFO @ 905: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 905: uvm_test_top.e.agnt.sqr@@seq [body]
starting write_until_full
UVM_INFO @ 905: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=32  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=1  dout=64  Actual:full=0  af=0  empty=0  ae=1  dout=64
UVM_INFO @ 915: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 915: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=21  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=1  dout=8a  Actual:full=0  af=0  empty=0  ae=1  dout=8a
UVM_INFO @ 925: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
...
```

```
UVM_INFO @ 1025: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=ca  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=8a  Actual:full=0  af=1  empty=0  ae=0  dout=8a
UVM_INFO @ 1035: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0
UVM_INFO @ 1035: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=ef  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=8a  Actual:full=0  af=1  empty=0  ae=0  dout=8a
UVM_INFO @ 1045: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 1045: uvm_test_top.e.agnt.sqr@@seq [body]

starting read_until_empty

UVM_INFO @ 1045: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=b9  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=8a  Actual:full=1  af=1  empty=0  ae=0  dout=8a
UVM_INFO @ 1055: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=1 / af=1 / ae=0 / empty=0
UVM_INFO @ 1055: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=74  write=1  read=0  rst_n=1  full=1  af=1  empty=0  ae=0  dout=8a  Actual:full=1  af=1  empty=0  ae=0  dout=8a
UVM_INFO @ 1065: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0

. . .
UVM_INFO @ 1205: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=fa  write=0  read=1  rst_n=1  full=0  af=0  empty=0  ae=1  dout=b9  Actual:full=0  af=0  empty=0  ae=1  dout=b9
UVM_INFO @ 1215: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 1215: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=1f  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=8a IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=8a IGNORED
UVM_INFO @ 1225: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 1225: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=d6  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=8a IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=8a IGNORED
UVM_INFO @ 1235: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1

. . .
UVM_INFO @ 1255: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=d9  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=8a IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=8a IGNORED
UVM_INFO @ 1265: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 1265: uvm_test_top.e.agnt.sqr@@seq [body]

starting write_until_AF

UVM_INFO @ 1265: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=3b  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=8a IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=8a IGNORED
UVM_INFO @ 1275: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=1
UVM_INFO @ 1275: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=66  write=0  read=1  rst_n=1  full=0  af=0  empty=1  ae=1  dout=8a IGNORED  Actual:full=0  af=0  empty=1  ae=1  dout=8a IGNORED
UVM_INFO @ 1285: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0
UVM_INFO @ 1285: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=c9  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=1  dout=c9  Actual:full=0  af=0  empty=0  ae=1  dout=c9
UVM_INFO @ 1295: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=1 / empty=0

. . .
UVM_INFO @ 1385: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=0 / ae=0 / empty=0
UVM_INFO @ 1385: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=50  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=0  dout=c9  Actual:full=0  af=0  empty=0  ae=0  dout=c9
UVM_INFO @ 1395: uvm_test_top.e.agnt.sqr@@seq [FLAGS] full=0 / af=1 / ae=0 / empty=0
UVM_INFO @ 1395: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=09  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=c9  Actual:full=0  af=1  empty=0  ae=0  dout=c9
UVM_INFO @ 1405: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=04  write=1  read=0  rst_n=1  full=0  af=1  empty=0  ae=0  dout=c9  Actual:full=0  af=1  empty=0  ae=0  dout=c9

. . .
UVM_INFO @ 2385: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=06  write=1  read=0  rst_n=1  full=0  af=0  empty=0  ae=1  dout=63  Actual:full=0  af=0  empty=0  ae=1  dout=63
UVM_INFO @ 2395: uvm_test_top.e.sbd.cmp [PASS ] Expected:din=19  write=0  read=0  rst_n=1  full=0  af=0  empty=0  ae=1  dout=63  Actual:full=0  af=0  empty=0  ae=1  dout=63
UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 2395: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO @ 2395: uvm_test_top.e.sbd.cmp [PASSED]

*** TEST PASSED - Vectors: 240 Ran / 240 Passed ***

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  374
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[FLAGS]   121
[INIT]     1
[PASS ]   240
[PASSED]     1
[RNTST]     1
[TEST_DONE]     1
[body]     9
```

Figure 6 - FIFO simulation messages using responsive stimulus generation

XI. Common Response Transaction Coding Mistake

A common driver coding mistake is to cut-and-paste the **seq_item_port.get_next_item(tr)** to **seq_item_port.item_done(tr)** after the **drive_item(tr)** task-call, as shown in Example 28. For sequences that do not require a response, this will cause a **tr** handle to be pushed onto the sequencer response queue and quickly overflow the response queue. When the stimulus is driven as a non-reactive source, the **item_done()** method should not include the **tr** handle.

```
seq_item_port.get_next_item(tr);
drive_item(tr);
seq_item_port.item_done(tr);
```

Example 28 - Common driver response-transaction error

Per the UVM Class Reference [3], the sequencer has a response queue with a default depth of eight response transactions. Using **seq_item_port.item_done(tr)** pushes a **tr** handle into the response queue. If the sequence does not do a **get_response(tr)** command, then the response queue will fill up and start to issue **Response queue overflow, response was dropped** error messages, as shown in the partial simulation output messages of Figure 7. This was from a driver that improperly used **item_done(tr)** commands

```
UVM_INFO  @ 0:    [RESET] Initial reset
UVM_INFO  @ 5:    [PASS ] Expected:din=0000  ld=0  inc=0  rst_n=0  dout=0000  Actual:dout=0000
UVM_INFO  @ 15:   [PASS ] Expected:din=ffff  ld=1  inc=1  rst_n=0  dout=0000  Actual:dout=0000
UVM_INFO  @ 25:   [PASS ] Expected:din=73b9  ld=0  inc=1  rst_n=0  dout=0000  Actual:dout=0000
UVM_INFO  @ 35:   [PASS ] Expected:din=7c15  ld=0  inc=1  rst_n=1  dout=0001  Actual:dout=0001
UVM_INFO  @ 45:   [PASS ] Expected:din=f946  ld=0  inc=1  rst_n=1  dout=0002  Actual:dout=0002
UVM_INFO  @ 55:   [PASS ] Expected:din=011a  ld=0  inc=1  rst_n=1  dout=0003  Actual:dout=0003
UVM_INFO  @ 65:   [PASS ] Expected:din=f801  ld=0  inc=1  rst_n=1  dout=0004  Actual:dout=0004
UVM_INFO  @ 75:   [PASS ] Expected:din=7ccb  ld=0  inc=1  rst_n=1  dout=0005  Actual:dout=0005
UVM_INFO  @ 85:   [PASS ] Expected:din=5359  ld=0  inc=1  rst_n=1  dout=0006  Actual:dout=0006
UVM_ERROR @ 95:   [uvm_test_top.e.agnt.sqr.seq] Response queue overflow, response was dropped
UVM_INFO  @ 95:   [PASS ] Expected:din=d8c3  ld=0  inc=1  rst_n=1  dout=0007  Actual:dout=0007
UVM_ERROR @ 105:  [uvm_test_top.e.agnt.sqr.seq] Response queue overflow, response was dropped
UVM_INFO  @ 105:  [PASS ] Expected:din=3657  ld=0  inc=1  rst_n=1  dout=0008  Actual:dout=0008
UVM_ERROR @ 115:  [uvm_test_top.e.agnt.sqr.seq] Response queue overflow, response was dropped
UVM_INFO  @ 115:  [PASS ] Expected:din=990b  ld=0  inc=1  rst_n=1  dout=0009  Actual:dout=0009
UVM_ERROR @ 125:  [uvm_test_top.e.agnt.sqr.seq] Response queue overflow, response was dropped

...

UVM_ERROR @ 1015: [uvm_test_top.e.agnt.sqr.seq] Response queue overflow, response was dropped
UVM_INFO  @ 1015: [PASS ] Expected:din=4caf  ld=0  inc=1  rst_n=1  dout=ae39  Actual:dout=ae39
UVM_INFO  @ 1015: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO  @ 1015: [PASSED]

*** TEST PASSED - Vectors: 102 Ran / 102 Passed ***
```

Figure 7 - Response queue overflow messages

As an interesting side-note, even though the response queue overflow error messages were reported, since the stimulus did not need the response transactions, the simulation runs successfully as can be seen by the Test Passed message at the bottom of Figure 7. Despite the passing simulation result, the stray error messages should be removed by correcting the **item_done()** method call.

A. Verification IP consideration

Verification IP frequently includes response transactions that include **seq_item_port.item_done(rsp)** commands as was described above. Since this command is already embedded in the Verification IP (VIP) and since VIP is frequently encrypted, engineers typically cannot modify this command and if user-defined sequences do not call the **get_response(rsp)** command, the **queue overflow** error messages, described above, will be generated throughout the simulation.

To address these VIP issues, add the command **set_response_queue_error_report_disabled(1)** near the top of the sequence **body()** task or at the top of the commands called from the sequence **body()** task as shown in Example 29.

```
task do_item (trans1 tr);
  set_response_queue_error_report_disabled(1);
  start_item(tr);
  if (!(tr.randomize() ...
  ...
  finish_item(tr);
endtask
```

Example 29 - do_item() task with queue error reports disabled

The **set_response_queue_error_report_disabled(1)** command does require the argument **(1)** to be passed to the command. Omitting the parameter results in an error message reporting that this method does not have a default value.

Using this command, the sequencer response queue will still overflow since the VIP cannot be modified, but the annoying error messages will be suppressed.

XII. Summary & Conclusions

This paper has shown how to develop a UVM testbench for reactive stimulus generation, including the very important step of sampling status signals at the end of the cycle.

When the sequencer queues up a transaction from a test sequence, there are two primary modes of operation.

(1) The sequence is non-reactive and transactions are driven without expecting responses to be queued by the sequencer.

(2) The sequence *IS* reactive causing transactions to be driven and then the sequence issues a **get_response(rsp)** command to accept response transactions that are sent from the driver using the **seq_item_port.item_done(rsp)**, through the sequencer, back to the test sequence.

The request/response transaction types of the sequencer, driver and sequence must all match. Using the same transaction type for both request and response transaction types is a straightforward technique as described in Section VII-B. If different request and response transaction types are desired, then the techniques described in Section VII-C can be used.

The user-defined sequence (extended from the **uvm_sequence** base class), the user-defined sequencer (extended from the **uvm_sequencer** base class) and the user-defined driver (extended from the **uvm_driver** base class), are already parameterized to default request (**req**) and response (**rsp**) transaction types. The driver is not required to declare separate transaction and response handles, but we generally recommend doing so, to avoid confusion.

One of the keys to creating sequences that can examine response transactions is to have the driver both drive signals into the virtual interface, from the driven transaction, AND to have the driver sample the output signals from the virtual interface at the end of the cycle. It is this sampling at the end of the cycle that is not well documented in many industry examples.

This paper also included a test plan for a 1-clock, 16-deep FIFO design and showed how to create the reactive stimulus commands that were executed by a sequence to satisfy the test plan.

Finally, this paper showed that if a sequence is not reactive, which is to say it does not get a response transaction, and if the driver issues the command, **`seq_item_port.item_done(tr)`**, the sequencer response queue will overflow and issue many, possibly thousands, of response error messages of the form: **`Response queue overflow, response was dropped`**. These error messages typically do not impact the simulation results but are annoying and frequently avoidable by not using the **`seq_item_port.item_done()`** command to return a transaction. When Verification IP is used, making corrections to the **`item_done(tr)`** command might not be possible, but the queue-overflow messages can be suppressed by using the command:

```
set_response_queue_error_report_disabled(1);
```

References

[1] Clifford E. Cummings, "Applying Stimulus & Sampling Outputs - UVM Verification Testing Techniques," SNUG (Synopsys Users Group) 2016 (Austin, TX). Also available at: www.sunburst-design.com/papers/CummingsSNUG2016AUS_Verification_TimingTesting.pdf

[2] Universal Verification Methodology (UVM) 1.1 Users Guide - May 2011

[3] Universal Verification Methodology (UVM) 1.2 Class Reference - June 2014

XIII. Author & Contact Information

Cliff Cummings is founder of Sunburst Design, which merged with Paradigm Works in February of 2020, and is now Vice President of Training at Paradigm Works. Mr. Cummings has 38 years of ASIC, FPGA and system design experience and 28 years of Verilog, SystemVerilog, UVM verification, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 17 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Email address: cliffc@sunburst-design.com

Stephen D'Onofrio is a Principal Engineer (Verification Architect) at Paradigm Works. Mr. D'Onofrio has led and mentored several successful verification teams spanning the globe. His Functional Verification skills include test planning, testbench architecture, infrastructure development and test writing. Various clients include NASA JPL, IBM, Motorola, AMD, Teradyne, BAE and several smaller/mid-sized companies.

Mr. D'Onofrio has published and presented over a dozen papers and articles. The venues include Club-V, Embedded Computing, The Journal of New England Technology, SNUG, CDNLive, EDA Tech Forum, and DVCon. Mr. D'Onofrio has been recognized with a Best Paper award.

Mr. D'Onofrio participated on the Accellera UVM and Unified Coverage Committees. Mr. D'Onofrio is also on the DVCon Technical Program Committee.

Mr. D'Onofrio holds a BSEE from University of Massachusetts.

Email address: stephen.donofrio@paradigm-works.com

Heath Chambers is President of HMC Design Verification, Inc., a company that specializes in design and verification consulting and high-tech training. Mr. Chambers is a consultant with 22 years of Verilog Experience, 15 years of SystemVerilog experience, 20 years of consulting and verification lead experience for multiple projects and has been an instructor for Sunburst Design since the year 2000. Mr. Chambers was previously a contract Specman Basic Training instructor for Verisity. Mr. Chambers has ASIC and system verification, firmware, and self-test design experience and can answer the very technical questions asked by experienced verification engineers.

Mr. Chambers was a member of the IEEE 1364 Verilog and IEEE 1800 SystemVerilog Standards Groups from 2000 to 2012 and has helped to develop and improve Sunburst Design Verilog, SystemVerilog, UVM and synthesis training courses. Mr. Chambers has co-authored and presented several award-winning papers at Club-V, SNUG, and DVCon.

Mr. Chambers holds a BSCS from New Mexico Institute of Mining and Technology.

Email address: hmcdvi@msn.com

Sunburst Design World Class Training

Sunburst Design merged with Paradigm Works in February of 2020 and still provides World Class SystemVerilog, Synthesis and UVM Verification training. For more information about training, contact Cliff Cummings (cliffc@sunburst-design.com) or Michael Hoyt (michael.hoyt@paradigm-works.com)

Paradigm Works Expert Design Services, Verification Services & IP

Paradigm Works provides expert services in Semiconductor Architecture, Design, Synthesis, Functional Verification, and DFT. For more information about Paradigm Works services, contact Michael Hoyt at michael.hoyt@paradigm-works.com

Last Updated: October 2020