Mastering Reactive Slaves in UVM

Jeff Montesano
Verilab Inc.

Co-authors: Mark Litterick, Taruna Reddy
September 18th, 2015
Austin, TX
Agenda

Introduction
Active Masters and Reactive Slaves
Operation
Storage
Control Agent
Error Injection
Conclusion
Introduction
Introduction

• Most protocols have masters and slaves
  – Master initiates transactions
  – Slave responds to transactions

• Sequence-based stimulus in UVM
  – Straightforward for masters

• Benefits of sequence-based stimulus
  – Achieve coverage closure using randomization

• How are slaves often done presently?
  – Bus-functional-models (BFMs) with non-random or pseudo-random responses
  – Inferior to sequence-based approach in achieving coverage closure
Introduction

How can we make slave verification components respond with sequence-based stimulus?

- If we could do that:
  - Coverage could be reached using constrained-random sequence items
  - Hitting more cases would just be a matter of running more random seeds
  - The power of UVM and randomization would be fully leveraged
Proactive Masters and Reactive Slaves
Proactive Masters and Reactive Slaves

• A verification component (VC) can play role of:
  – Master (design under test (DUT) is slave)
  – Slave (DUT is master)

• VC’s that drive signals: active role
• VC’s that drive no signals: passive role
• In most protocols, slave plays active role
  – Examples: AXI, I2C, SPI
  – Exception: display interface

Called “proactive master” and “reactive slave” because both play an active role
Proactive Masters and Reactive Slaves

- **Proactive masters:**
  - Test controls when sequences are executed on the VC
  - Stimulus blocks test flow

- **Reactive slaves:**
  - Timing of DUT requests is unpredictable
  - VC must respond autonomously without blocking test flow
Proactive Masters and Reactive Slaves

• Proactive master architecture:
  - Proactive master environment
  - Test case initiates transfer
  - 1. sequencer
    - item
    - TLM FIFO
  - 2. master agent
    - driver
    - monitor
      - coverage
      - checks
    - VIF
  - interface
  - sva
  - DUT (slave)
Proactive Masters and Reactive Slaves

• Reactive slave architecture:

  slave environment
  
  slave agent
  
  sequencer
  item
  TLM FIFO

  driver

  monitor
  • checks

  VIF

  interface

  VIF

  DUT
  (master)

  DUT initiates transfer
Operation of Reactive Slaves
Operation of Reactive Slaves

• Implementation of reactive slaves is very similar to that of proactive masters

<table>
<thead>
<tr>
<th></th>
<th>Proactive Masters</th>
<th>Reactive Slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence execution</strong></td>
<td>Test case calls on-demand</td>
<td>Forever loop called at time 0</td>
</tr>
<tr>
<td><strong>Monitor-to-Sequencer communication</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• The rest is identical
Operation of Reactive Slaves

- **Monitor**
  - Publishes requests from the DUT

- **Sequencer**
  - Runs the “forever” sequence
  - Subscribes to published requests from the monitor

- **“Forever” sequence**
  - Generates response items

- **Driver**
  - Converts response items into bus signals
Operation of Reactive Slaves

slave environment

slave agent

sequencer
- item
- TLM FIFO

driver
- VIF

monitor
- coverage
- checks

DUT

SVI

RESP

REQ

TRANSACTION
Operation: Monitor

class my_slave_monitor extends uvm_monitor;
uvm_analysis_port #(my_transaction) m_req_port;
uvm_analysis_port #(my_transaction) m_bus_port;
...
task monitor_bus();
...
forever begin
    // decode bus signals in accordance with protocol
    ...
    m_req_port.write(transaction);
    ...
    m_bus_port.write(transaction);
end
dendtask
...
endclass

Two analysis ports

Publish observed request (to sequencer)

Publish full transaction (to any subscribers)
Operation of Reactive Slaves

slave environment

slave agent

sequencer

item

TLM FIFO

driver

VIF

monitor

• coverage
• checks

DUT

sva

interface

RESP

REQ

TRANSACTION

VIF

VIF

VIF

VIF
class my_slave_sequencer extends uvm_sequencer #(my_seq_item);
...
  uvm_analysis_export #(my_transaction) m_request_export;
  uvm_tlm_analysis_fifo #(my_transaction) m_request_fifo;
...
  function new(string name, uvm_component parent);
    ...
    m_request_fifo = new("m_request_fifo", this);
    m_request_export = new("m_request_export", this);
  endfunction

  function void connect_phase(...);
    ...
    m_request_export.connect(m_request_fifo.analysis_export);
  endfunction
endclass

TLM analysis export receives requests published by monitor
Operation: Sequence

class my_slave_response_seq extends my_slave_base_seq;

... virtual task seq_body();
    forever begin
        p_sequencer.m_request_fifo.get(m_req);

        case (m_req.m_direction)
            MY_DIRECTION_WRITE : begin
                `uvm_do_with(m_item,{
                    m_item.m_resp_kind == MY_RESPONSE_ACK;
                })
            end
            ...
        endcase
    end

endtask
endclass

wait for a transaction request (get is blocking)
generate response based on observed request
Operation of Reactive Slaves

slave environment

slave agent

sequencer

item

TLM FIFO

driver

monitor

• coverage
• checks

VIF

VIF

DUT

sva

interface

TRANSACTION

REQ

RESP

REQ

RESP
Operation: Driver

```verilog
class my_slave_driver extends uvm_driver #(my_slave_seq_item);

... task run_phase(...);
...
forever begin
  seq_item_port.get_next_item(m_item);
  drive_item(m_item);
  seq_item_port.item_done();
end
endtask


task drive_item(my_slave_seq_item item);
...
endtask
endclass
```

Standard **driver-sequencer** interaction

**drive response** signals to DUT
Operation of Reactive Slaves

- Reactive slave sequence-driver interaction

<table>
<thead>
<tr>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
</tr>
<tr>
<td>get()</td>
</tr>
<tr>
<td>get_next_item()</td>
</tr>
<tr>
<td>item_done()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait for response item</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wait for request from monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate/do response item</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Drive response signals</th>
</tr>
</thead>
</table>
Operation of Reactive Slaves

slave environment

slave agent

sequencer

item
TLM FIFO

driver

VIF

monitor

VIF

- coverage
- checks

DUT

TRANSACTION

RESP

REQ
Operation: Agent

```cpp
function void my_slave_agent::connect_phase(...);
    super.connect_phase(phase);
    ...
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
    m_monitor.m_req_port.connect(m_sequencer.m_request_export);
    ...
endfunction
```

Connect **driver** and **monitor** to the **sequencer**
Operation of Reactive Slaves

Test environment

slave environment

slave agent

- sequencer
  - item
  - TLM FIFO
- driver
  - VIF
- monitor
  - coverage
  - checks
  - VIF

INTERFACE

sva

DUT

TRANSACTION

RESP

REQ
Operation: Test Environment

class **my_test_env** extends uvm_env;
...
my_uvc uvc_env;
...
function void build_phase(...);
  super.build_phase(phase);
...
uvm_config_db #(uvm_object_wrapper)::set(
    this,
    "uvc_env.slave_agent.m_sequencer.main_phase",
    "default_sequence",
    my_slave_response_seq::type_id::get());
...
endfunction
declass

Set the default sequence for the reactive slave agent
Additional Features

Storage, Control Agent, and Error Injection
Reactive Slave with Storage

• Storage serves to emulate real slave behavior

• API should provide tests the ability to:
  – Read from slave storage (e.g. for doing checks)
  – Write to slave storage (e.g. for doing error injection)
  – Initialize storage content

• API should provide reactive slave “forever” sequence the ability to
  – Read from slave storage

• Storage should be updated by monitor
Reactive Slave with Storage

**tests** can access storage (via sequencer handle) to **prefill, check & modify**

**sequences read** storage values for autonomous read **response items**

**monitor writes** to storage when DUT write transaction **observed**

But how does a **test** know **when** to check and/or modify **storage** values relative to **DUT traffic**?

- **monitor** writes to storage when DUT write transaction **observed**
- **sequences read** storage values for autonomous read **response items**
- **tests** can access storage (via sequencer handle) to **prefill, check & modify**
Reactive Slave with Storage

- Slave responds autonomously
- Test flow does not block on slave transactions
- How can test synchronize to slave activity?
- For example, make test wait for
  - A DUT transaction
  - A specific transaction kind and/or field value from DUT
  - A specific transaction kind to a specific address from DUT
**Driver** doesn’t drive signals, just interprets sequence items

**Test** can execute a sequence on control agent to block test flow
class my_control_driver extends uvm_driver#(my_control_seq_item);
    uvm_analysis_export #(my_slave_transaction) aexport;

    task drive_item(control_seq_item item);
        case (item.m_trans)
            WRITE: @write_detected;
            READ: @read_detected;
        endcase
    endtask

    function void write(my_slave_transaction transaction);
        ...
        case (transaction.operation)
            WRITE: ->write_detected;
            READ: ->read_detected;
        endcase
    endfunction
endclass

Test flow stops until DUT read/write transaction observed
Trigger events when monitor observes read/write
Reactive Slave with Control Agent

class my_test extends uvm_test;
    wait_seq control_seq;
    ...
    `uvm_do_on_with(
        control_seq,
        test_env.uvc_env.control_agent.sequencer, {
            transaction == WRITE;
        })
    ... 
    sequencer.storage.write(...)
endclass

- Block test flow until a write happens
- Modify the value that was written (e.g. for error injection purposes)
- What if we want error injection to happen autonomously, without the test blocking on slave activity?

• Test can:
  – Read a value that was written to perform checks
  – Launch other related stimulus after a read or write
  – Overwrite value that was written to inject errors
Reactive Slave with Error Injection

• Error injection with proactive masters
  – Sequence item gets one additional boolean field per error type to be injected
  – Testcase writer constrains sequence to set error injection field
  – Driver interprets sequence item and performs the error injection

• Error injection with reactive slaves is more complicated
  – Slave is acting autonomously
  – How can a test tell the slave to inject an error?
Reactive Slave with Error Injection

• Proposed approach
  – Sequence item gets one additional boolean field per error type to be injected (same as proactive masters)
  – Test case increments *counters* that store error requests (use of counters is for flexibility; a single field could work too)
  – Sequences set error injection field when counters are > 0
  – Driver injects errors based on sequence item contents (same as with proactive master)
Reactive Slave with Error Injection

```verilog
class my_slave_cfg extends uvm_object;
  int m_type1_error_count;
...
endclass
```

One counter per error type

```verilog
class my_control_incr_err_seq extends uvm_sequence;
...
  bit increment_type1_error;
...
  task body();
    if (increment_type1_error)
      p_sequencer.cfg.m_type1_error_count++;
...
endtask
endclass
```

Test increments error counter via sequence
class my_slave_response_err_seq extends uvm_sequence;

virtual task seq_body();
  forever begin
    p_sequencer.m_request_fifo.get(m_req);
    `uvm_create(m_item)
    if (!m_item.randomize() with {
      ...
    }) `uvm_fatal(“”, “randomization failed”)

    if (p_sequencer.cfg.m_type1_error_count > 0) begin
      m_item.m_inject_type1_error = 1;
      p_sequencer.cfg.m_type1_error_count--;
      `uvm_send(m_item);
    end
  end
endclass
class my_err_test extends uvm_test;

  ...

  my_uvc uvc_env;

  ...

  function void build_phase(...);
    super.build_phase(phase);
    ...

    uvm_config_db #(uvm_object_wrapper)::set(
        this,
        "uvc_env.slave_agent.m_sequencer.main_phase",
        "default_sequence",
        my_slave_response_err_seq::type_id::get());
    ...

  endfunction

endclass

Make the error sequence be the default sequence
Conclusion
Conclusion

• We have shown how reactive slaves can
  – Take a similar approach as is done with proactive masters
  – Make use of storage to emulate true slave behavior
  – Allow testcases to be in sync with slave activity (e.g. to check, modify, and inject errors in storage)
  – Allow testcases to inject errors into autonomous responses

• This approach is better than using a BFM approach because
  – It leverages the full power of UVM and constrained-random simulators
  – It will achieve coverage closure more efficiently
  – It will most likely uncover more bugs in less time

• All code examples available in paper for your own use
Thank You
Proactive Masters and Reactive Slaves

• Active/passive role not the same as active/passive mode
• A VC can operate in active or passive mode regardless of its role
slave environment

slave agent

control agent

sequencer

item

driver

storage

sequencer

item

TLM FIFO

monitor

VIF

• coverage
• checks

DUT

interface

sva

transaction

RESP

REQ

RESP

REQ