Mastering Reactive Slaves in UVM

Mark Litterick
Verilab GmbH.

Co-author: Jeff Montesano
June 23, 2016
Munich, Germany
Agenda

Introduction to masters and slaves
Overview of proactive master architecture & operation
Details of reactive slave architecture & operation
Additional features: storage, synchronization, error injection
Conclusion
Introduction

- Modern verification uses **constrained-random sequence-based stimulus**
  - ...including, but not limited to, SystemVerilog and the UVM
  - powerful, flexible, scalable & adaptable (e.g. modify stimulus one-the-fly)
  - restrict test scenarios by applying more constraints (e.g. directed test)
  - explore application space by loosening constraints (e.g. random test)
  - achieve functional coverage closure through randomization

- Most interface protocols comprise of **masters and slaves**
  - masters initiate transactions
  - slaves respond to masters

- Sequence-based **stimulus** in UVM
  - straightforward for masters
  - more complicated for slaves!

**How can we make slave UVCs generate constrained-random response based on current request using sequences?**
Active and Passive Operation

- **Active**
  - *generate* or affect stimulus
- **Passive**
  - *observe* and check behavior
- **Master**
  - *initiates* protocol traffic
- **Slave**
  - *responds* to initiator
- **Verification components**
  - all combinations required
  - active/passive master/slave

![Diagram of UVC A, B, C, D with labels and connections]

- **generate request stimulus** (DUT is slave)
- **observe external DUT behavior** (no response)
- **generate response stimulus** (DUT is master)
- observe internal DUT behavior (no stimulus)
Proactive Masters and Reactive Slaves

- **Proactive Masters**:
  - Test controls when sequences are executed on the UVC and request timing to DUT
  - Stimulus blocks test flow waiting for DUT response

- **Reactive Slaves**:
  - Timing of DUT requests is unpredictable (e.g. due to embedded microcode execution)
  - UVC must react to request and respond autonomously without blocking test flow
Proactive Master Architecture

- constrained-random sequence-based stimulus
- **DUT** interaction via signal **interface**
- **standard** UVM component structure (agent, sequencer, driver, monitor)
- independent **passive monitor** observes request and response
Proactive Master Operation

**Test** or higher-level: start / do *sequence* on sequencer

Generate **sequence item** & pass to **driver** via TLM

Drive *request signals* according to **protocol** (& wait for response)

Monitor publishes **full transactions** via TLM **analysis port** (REQ & RESP)

Decode response & return to sequence
Reactive Slave Architecture

- forever sequence executed on sequencer
- similar architecture to proactive master
- other common architectures are discussed in the paper...

- TLM analysis FIFO provides blocking get() for analysis path
- additional TLM analysis connection between monitor & sequencer
- use analysis port & FIFO (instead of blocking put port) => easier passive mode

SNUG 2016
Reactive Slave Operation

test or higher-level: start / do / default forever sequence that waits for REQ

generates sequence item & passed to driver via TLM

drive response signals according to protocol (based on request)

monitor publishes full transactions via TLM analysis port (REQ & RESP)

decode request whenever DUT initiates it & publish to sequencer
Monitor Code

class my_slave_monitor extends uvm_monitor;
...
  uvm_analysis_port #(my_transaction) transaction_aport;
  uvm_analysis_port #(my_transaction) request_aport;
  my_transaction m_transaction;
...
  task monitor_bus();
    forever begin
      // decode bus signals in accordance with protocol
      ...
      // when request is complete...
      request_aport.write(m_transaction);
      ...
      // continue to decode response...
      transaction_aport.write(m_transaction);
    end
  endtask
endclass
Sequencer Code

class my_slave_sequencer extends uvm_sequencer #(my_slave_seq_item);

  ...
  uvm_analysis_export #(my_transaction) request_export;
  uvm_tlm_analysis_fifo #(my_transaction) request_fifo;

  ...
  function new(string name, uvm_component parent);
    ...
    request_export = new("request_export", this);
    request_fifo = new("request_fifo", this);
  endfunction

  function void connect_phase(...);
    ...
    request_export.connect(request_fifo.analysis_export);
  endfunction

endclass

additional analysis export and TLM analysis FIFO

construct TLM components

connect analysis export direct to FIFO (no need to implement write method)
Sequence Code

class my_slave_response_seq extends uvm_sequence #(my_slave_seq_item);

my_slave_seq_item m_item;
my_transaction m_request;
...

task body();
   forever begin
      p_sequencer.request_fifo.get(m_request);
      case (m_request.m_direction)
         ...
            READ : begin
               `uvm_do_with(m_item, {
                  m_item.m_resp_kind == READ_RESPONSE;
                  m_item.m_delay <= get_max_delay();
                  m_item.m_data == get_data(m_request.m_addr);
                  ...
               })
            end
         ...
   end
...

call forever loop inside sequence (sequence runs throughout phase: ...do not raise and drop objections!)

wait for a transaction request (fifo.get is blocking)

generate response item based on observed request

note: code examples assume:
• protocol defines base transaction
• seq_item extends base transaction (to add constraints and control knobs)
class my_slave_driver extends uvm_driver #(my_slave_seq_item);
    my_slave_seq_item m_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 (based on sequence item fields)

identical code structure to proactive master
Agent Code

class my_slave_agent extends uvm_agent;
... uvm_analysis_port #(my_transaction) slave_aport;
...
function void connect_phase(...);
...
monitor.transaction_aport.connect(slave_aport);
if (is_active == UVM_ACTIVE) begin
driver.seq_item_port.connect(sequencer.seq_item_export);
monitor.request_aport.connect(sequencer.request_export);
end
...
endfunction
endclass

standard TLM port connections (monitor to agent) (sequencer to driver)

additional TLM port connection (monitor to sequencer)
Test Environment Code

- Select default sequence in environment (can override it from test):
  
  ```
  class my_test_env extends uvm_env;
  ...
  function void build_phase(...);
  ...
  uvm_config_db #(uvm_object_wrapper)::set(this, "uvc_env.slave_agent.sequencer.main_phase", "default_sequence", my_slave_response_seq::type_id::get());
  ```

- OR start explicit sequence from test (leave default as `null`):
  
  ```
  class my_test extends uvm_test;
  ...
  task run_phase(...);
  ...
  fork
  slave_seq.start(test_env.uvc_env.slave_agent.sequencer);
  join_none
  ```

set the `default_sequence` for the `main_phase` of slave sequencer

start a specific sequence on the required slave sequencer

note: `default_sequence` for sequencer component is deprecated in UVM-1.2 => use the `main_phase` instead
Additional Features

Storage, Control Agent, & Error Injection
Reactive Slave with Memory/Storage

- What do we mean by memory/storage?
  - so far slave has generated responses based on **current request**, not **previous traffic**
  - often we need capability to store values and use these in subsequent responses
  - storage serves to emulate real slave behavior in application (local memory)

- Add a local memory model to the slave agent
  - used by **slave** to generate and control **responses**
  - used by the **test** to control interesting **scenarios**

- API should allow:
  - **read** from slave storage (e.g. for doing checks)
  - **write** to slave storage (e.g. for doing error injection, or modifying values)
  - **initialize** storage content (e.g. from a file, or random content)
Slave With Storage Architecture

important to note that slave storage is not part of DUT memory model (…part of target slave)

handle from sequencer for stimulus

small or sparse memory array with API

handle from monitor for passive update & debug

handle from monitor for passive update & debug
Slave With Storage Operation

How can we **synchronize test** operations with autonomous **requests** from DUT?

```c
m_item.m_data == p_sequencer.storage.read(m_request.m_addr);
```

**tests:**
- preload memory
- check content
- modify content

```c
storage.init(); // on reset
storage.write(m_addr, m_data);
```

sequences: **use** storage to generate response data

monitor: **update** storage based on observed traffic

```c
m_item.m_data == p_sequencer.slave_sequencer.storage.write(...)
```

// in sequences via sequencer

```c
if(env.slave.storage.read(addr)!='h1234)...
```

// or via test component hierarchy
Reactive Slave With Control Agent

• What do we mean by control agent?
  – so far we focused on slaves generating responses on-demand for DUT master
  – typically enough for block-level verification environments...
  – ...but for practical high-level scenario generation we need more control

• Add a proactive control agent to the environment
  – provides stimulus synchronization for high-level scenario
  – enables interaction of stimulus with autonomous responses

• For example, make top-level test stimulus wait for:
  – any transaction on the slave interface (initiated by the DUT whenever...)
  – DUT write to a specific address: check the value is correct, modify the value in storage
  – DUT read to a specific address: increment the stored value to validate calibration algorithm
  – control error injection...
Control Agent Architecture & Operation

**control agent is not controlling slave response** directly, but high-level **scenario timing** based on observed DUT traffic.

test sequence waits for **slave traffic** of particular kind...

call **item_done()** to unblock sequence (optional return data)

driver:
- **drive_item()**: loop waiting for event of specified kind
- **aport write()**: decode transactions, generate events

slave generates **responses** and publishes **transactions**
Control Driver Code

```verilog
// task drive_item
task drive_item(my_control_seq_item item);
    case (item.m_operation)
        WAIT_WRITE_TO_ADDR:
            @(write_event iff m_transaction.m_addr == item.m_addr);
                ... // wait for requested event and specific condition
    ... // extend this concept to control error injection in autonomous responses

// function write
function void write(my_slave_transaction transaction);
    m_transaction.copy(transaction);
    case (transaction.m_direction)
        WRITE: -> write_event;
            ... // trigger events based on observed slave traffic
    ... // example test stimulus to synchronize with slave traffic and affect responses

// class my_test_seq
class my_test_seq
    ... // wait for DUT to write to 'h60
    'uvm_do_on_with(wait_wr_ad_seq,...,{a=='h60})
    // check value in storage
    if(...storage.read('h60)!=='hFF) 'uvm_error(...)
    // modify storage for subsequent read by DUT
    ...storage.write('h60,'h00);
    ... // example test stimulus to synchronize with slave traffic and affect responses
```
Reactive Slave with Error Injection

- Error injection with proactive masters
  - request sequence item has additional fields for error injection (e.g. crc_error_en)
  - test writer constrains sequence to set error injection field (e.g. do with crc_error_en==1)
  - driver interprets sequence item and performs the error injection (e.g. corrupt the CRC)

- Error injection with reactive slaves is more complicated
  - slave is acting autonomously, provided responses on-demand
  - how can a test scenario control when the slave to injects an error?

- Recommended solution
  - response sequence item has additional fields, and driver uses them (as for master)
  - slave is provided with error injection counters that are used by sequences
  - counters are incremented by test when required, decremented by sequence when used
  - control agent can be used to synchronize scenario (e.g. error on next access to addr N)
increment counter from test scenario
add error counters to config object
if >0 set error knob in item and decrement counter
inject error in next autonomous response

if (p_sequencer.cfg.error_cnt > 0) begin
  m_item.m_inject_error = 1;
  p_sequencer.cfg.error_cnt--;
end

if (m_item.m_inject_error)
  drive_bad_response(...);
else
  drive_good_response(...);
Conclusion

• Demonstrated how to **implement reactive slaves** in UVM
  – utilizing full-power of constrained-random sequence-based stimulus
  – allows slaves to react to current and previous requests
  – make use of storage to emulate application slave behavior
  – allow testcases to synchronize with slave activity
  – allow testcases to inject errors into autonomous responses

• **Proven UVM-compliant architecture**
  – used for many protocols, applications, projects & clients
  – uses similar UVM architecture to proactive masters
  – minimal changes to sequence and agent, no changes for item or driver
  – more powerful than BFM-based solution, more flexible than ad-hoc approaches

• All **code examples** available in paper
How to include your company logo?

1) Open the Slide Master
2) Double click on the "Company Logo" image
3) Click on "Change Picture"
4) Select your logo in a .png format (transparent background)
5) Delete this note when completed

Thank You