Universal Verification Methodology Lecture2

Pankaj Badhe

DUT-Testbench Connections

Using the uvm_config_db to pass virtual interface handles from hdl_top to an agent

Connecting a UVM testbench to a DUT using the uvm_config_db

To pass a virtual interface handle from the static module based part of the testbench (hdl_top), to a UVM component, use the uvm_config_db as follows:

Step 1: In the HDL part of the testbench assign the static interface to a virtual interface entry in the uvm_config_db:

```
sfr_if SFR_IF();
 // Declaration of SFR interface – This will be connected to modules
initial begin
  uvm_config_db #(virtual sfr_if)::set(null, "uvm_test_top", "SFR", SFR_IF);
end
```

- The uvm_config_db is parameterised with the type ***virtual sfr_if***
- The first argument of the set() method is context, intended to be assigned a UVM component object handle;

in this case since we are in the HDL part of the testbench, a null object handle is assigned.

- The second argument of the set() method is a string used to identify the UVM component instance name(s) within the UVM testbench component hierarchy that may access the data object. This is "uvm_test_top" here to restrict access to the top level UVM test object.
- The third argument of the set() method is a string, intended as the lookup name, i.e. a string that can be used to uniquely identify the virtual interface from within the uvm_config_db.
- The final argument of the set() is the static interface assigned to the virtual interface handle entry that is created within the uvm_config_db.

```
class sfr_config extends uvm_object;
...
virtual sfr_if sfr_vif;
...
endclass

class test extends uvm_test;
...

function void build_phase(uvm_phase phase);
  sfr_config cfg = sfr_config::type_id::create("cfg");

  // Assigning the SFR virtual interface handle via the uvm_config_db
#(...)::get() method
  if(!uvm_config_db #(virtual sfr_if)::get(this, "", "SFR",
cfg.sfr_vif)) begin
    `uvm_error("VIF_GET", "Unable to find the SFR virtual interface in
the uvm_config_db", UVM_LOW)
  end
```

- The uvm_config_db::get() method is a function that returns a bit value to indicate whether the object retrieval was been successful or not; this is tested to ensure that the testbench does not proceed if the lookup fails.
- The uvm_config_db::get() method is parameterised with the virtual interface type so that the right type of object is retrieved from the database.
- The first argument of the get() method, context, is passed the handle for the UVM component in which the call is being made – this
- The second argument, instance_name, is passed an empty string, “”, this means that only the path string for the component is applied based on the component path string derived from the component handle passed in context (i.e. “uvm_test_top”).
- The third argument is the uvm_config_db lookup string, this should match the lookup string assigned in the HDL module, in this case “SFR”.
- The fourth argument is the virtual interface handle. In this case the handle is inside a uvm_config_object that will be passed to an agent.

```
interface sfr_if(input clk, input reset);

  logic[7:0] address;
  logic[7:0] write_data;
  logic[7:0] read_data;
  logic we;
  logic re;

endinterface: sfr_if
```

driver

```
task sfr_driver::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    seq_item_port.get_next_item(item);
    SFR.execute(item);
    seq_item_port.item_done();
  end
```

Driver bfm

```
task execute(sfr_seq_item item);
   if(reset == 1) begin
      wait(reset == 0);
   end
   else begin
      @(posedge clk);
      address = item.address;
      we <= item.we;
      write_data <= item.write_data;
      re <= item.re;
      @(posedge clk);
      if(re == 1) begin
         item.read_data = read_data;
         re <= 0;
      end
      we <= 0;
   end
endtask: execute
```

"'The Monitor'"

```
task sfr_monitor::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    item = sfr_seq_item::type_id::create("item");
    SFR.monitor(item);
    ap.write(item);
  end

endtask: run_phase
```

'''The Monitor BFM'''

```
task monitor(sfr_seq_item item);
  @(posedge clk);
  if((we == 1) || (re == 1)) begin
    item.we = we;
    item.re = re;
    item.address = address;
    item.write_data = write_data;
    item.read_data = read_data;
  end
endtask: monitor
```

```
module hdl_top;

import uvm_pkg::*;

sfr_if SFR(.clk(clk), .reset(reset));

sfr_dut dut (.clk(clk),
             .reset(reset),
             .address(SFR.address),
             .write_data(SFR.write_data),
             .we(SFR.we),
             .re(SFR.re),
             .read_data(SFR.read_data));

initial begin
  uvm_config_db #(virtual sfr_if)::set(null, "uvm_test_top", "SFR",
SFR);
end

endmodule
```

```
module hvl_top;

import uvm_pkg::*;
import sfr_test_pkg::*;

initial begin
  run_test();
end

endmodule: hvl_top
```

Using Packages

```
// Note that this code is contained in a file called spi_env_pkg.sv
//
// In Questa it would be compiled using:
// vlog +incdir+$UVM_HOME/src+<path_to_spi_env> <path_to_spi_env>/spi_env_pkg.sv
//


//
// Package Description:
//
```

```
package spi_env_pkg;

// Standard UVM import & include:
import uvm_pkg::*;
`include "uvm_macros.svh"

// Any further package imports:
import apb_agent_pkg::*;
import spi_agent_pkg::*;
import spi_register_pkg::*;

// Includes:
`include "spi_env_config.svh"
`include "spi_virtual_sequencer.svh"
`include "spi_env.svh"

endpackage: spi_env_pkg
```

The Virtual Interface

- In SystemVerilog a class cannot make a hierarchical reference to a signal without being declared within a module or interface scope where those kinds of references are allowed.
- For the purposes of developing reusable testbenches using packages, this is very restrictive.
- SystemVerilog classes defined in a package can reference signals within an interface via an interface handle stored in a virtual interface variable.
- This allows a class to either assign or sample values of signals inside an interface, or to call tasks or functions within an interface

```
class protocol_driver extends uvm_driver #(protocol_seq_item);

virtual protocol_interface vif;
// Virtual interface declaration

....

task run_phase(uvm_phase phase);

  @(posedge vif.clock);
  // Accessing the clock signal via the virtual interface variable

  vif.send_packet(....);

  // Calling a BFM method via the virtual interface variable
```

To pass a virtual interface handle from the static module based part of the testbench (hdl_top), to a UVM component, use the uvm_config_db as follows:

Step 1: In the HDL part of the testbench assign the static interface to a virtual interface entry in the uvm_config_db:

```
sfr_if SFR_IF();
 // Declaration of SFR interface - This will be connected to modules
initial begin
  uvm_config_db #(virtual sfr_if)::set(null, "uvm_test_top", "SFR", SFR_IF);
end
```

Step 2: In a UVM test component assign the virtual interface handle from the uvm_config_db entry to the virtual interface handle inside a configuration object:

```
// Virtual interface handle declared inside an agent configuration
object
class sfr_config extends uvm_object;
...
virtual sfr_if sfr_vif;
...
endclass

class test extends uvm_test;
...

function void build_phase(uvm_phase phase);
  sfr_config cfg = sfr_config::type_id::create("cfg");

  // Assigning the SFR virtual interface handle via the uvm_config_db
#(...)::get() method
  if(!uvm_config_db #(virtual sfr_if)::get(this, "", "SFR",
cfg.sfr_vif)) begin
    `uvm_error("VIF_GET", "Unable to find the SFR virtual interface in
the uvm_config_db", UVM_LOW)
  end
```

Handling Parameterization

```
package tb_params_pkg;

// Import the VIP package to get access to all the VIP class types
import sfr_agent_pkg::*;

// Class declaration with localparams - this creates a defined
namespace
// and is useful for handling multiple VIPs of the same type
class SFR;

  localparam sfr_addr_width = 16;
  localparam sfr_data_width = 32;

endclass
```

```
typedef sfr_config_object #(SFR::sfr_addr_width, SFR::sfr_data_width)
SFR_cfg_t;
typedef sfr_agent #(SFR::sfr_addr_width, SFR::sfr_data_width)
SFR_agent_t;

// Declaration of virtual interface handles
typedef virtual sfr_monitor_bfm #(SFR::sfr_addr_width,
SFR::sfr_data_width) SFR_monitor_bfm_t;
typedef virtual sfr_master_bfm #(SFR::sfr_addr_width,
SFR::sfr_data_width) SFR_master_bfm_t;

endpackage
```

Testbench Configuration

- Config object
  - Bfm cfg (active / passive)
  - Seq_item cfg
  - Sequence cfg (normal/error)
  - Driver cfg (normal/error_inj)

Analysis Components

- **Monitoring DUT Activity**
- The analysis part starts its decision process by observing the DUT responses. This is done by one or more monitors (proxy and BFM) that observe the signal-level activity on the DUT interfaces (via virtual interfaces).
- A UVM monitor converts signal-level activity into TLM transaction objects and broadcasts these to interested analysis components - i.e. subscribers - via analysis port connections.
- These subscribers capture the transactions and conduct their analysis.

- **Scoreboards**
- Scoreboards are analysis components that collect the transactions sent by a monitor and perform specific (often comparative) analysis computations to determine whether or not the design is functioning as expected.
- A good scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation.
- A predictor model, sometimes referred to as "Golden Reference Model", receives the same stimulus transactions as the DUT and computes known good response transactions.
- The scoreboard evaluates the predicted with respect to the actual responses.
- The most common evaluation technique uses a comparator of some sort, capable of comparing transaction streams in in-order or a particular out-of-order fashion.

- **Coverage Collectors**
- Coverage information is paramount to answer the questions "Are we done (testing) yet?" or "Have we done enough (testing) yet?"
- Coverage collectors are analysis port subscribers that sample observed transactions and activity into SystemVerilog functional coverage groups.
- The coverage data collected from each test is stored into a shared coverage database used to determine overall verification progress.

- **Metric Analyzers**
- Metric Analyzers watch and record non-functional behavior such as timing/performance and power utilization.
- Depending on the number of observed transaction streams, they are implemented as a uvm_subscriber or with analysis exports.
- Metric analyzers perform ongoing calculations during the run phase, and/or during the post-run phases.

- **Analysis Reporting**
- All data for analysis is collected during simulation.
- Most analysis takes place dynamically during the course of the simulation run (i.e. on-the-fly), though some analysis may be performed after the UVM run_phase ends when timed simulation completes.
- UVM provides three phases for this purpose: extract_phase, check_phase, and report_phase.
- These post-run phases provide components with the option to extract relevant data collected on-the-fly during the run, perform a post-run check, and finally produce comprehensive reports on all analysis results.

Analysis Port

- UVM provides three objects to meet the requirements of the observer pattern as well as the requirement of noninterference:
  - **analysis ports, analysis exports, and analysis fifos.**
- Analysis ports, analysis exports, and analysis fifos follow the standard UVM transaction-level communication semantics.
- An analysis port **requires** an implementation of write() to be connected to it. An analysis export **provides** the implementation of the write() function.

Analysis Exports
Analysis Port
AP
exp3
exp2
exp1
AP.connect(exp1);
AP.connect(exp2);
AP.connect(exp3);
Exports subscribe by connection

Analysis Exports
Analysis Port
exp3
exp2
exp1
AP
AP.write(txn)
Analysis Port broadcasts to all subscribers

- All other UVM TLM ports and exports, such as blocking put ports and blocking put exports, perform point-to-point communication. Because of the one-to-many requirement for analysis ports and exports, an analysis port allows
- multiple analysis exports to be connected to it. It also allows for no connections to be present. The port maintains a list of connected exports.
- The Analysis port provides a single void function named write() to perform the broadcast behavior.
- When code calls the write() function on an analysis port, the port then uses its internal list to broadcast by calling write() on all
- connected exports. This causes the write() function to be executed on all components containing the connected exports.
- If no exports are connected to the analysis port, then no action is performed when the write() function is called and no error occurs.

Analysis Exports

An "imp" export is placed on a component that actually implements the write() function directly.

A hierarchical export is used for hierarchical connections, where the component that implements the write() function is a hierarchical child of the component with the export. A hierarchical export forwards the call to write() to the child component.

Analysis Fifos

- UVM provides a pre-built component called an analysis fifo that has an "imp"-style analysis export and an implementation of the write() function that places the data written into a fifo buffer.
- The buffer can grow indefinitely, in order to prevent blocking and adhere to the non-interference requirement.
- The analysis fifo extends the tlm_fifo class, and so it also has all of the exports and associated operations of a tlm fifo such as blocking get, etc.

Implementing the write() function

Analysis Connections
Single Transaction Stream: uvm_subscriber

- Many analysis components just need to sample the data sent by the Monitor, and perform some calculation such as coverage measurement.
- For these components, which only deal with a single stream of transactions, UVM provides a base component class called uvm_subscriber.
- This class extends uvm_component and includes a single "imp"-style export named analysis_export, with a pure virtual declaration of write().
- To use this class, you must extend it and override the write() function to perform the analysis calculation.

```
class coverage_sb extends uvm_subscriber #(Packet);
 `uvm_component_utils(coverage_sb)

 Packet pkt;
 int pkt_cnt;

 covergroup cov1;
  s: coverpoint pkt.src_id {
      bins src[8] = {[0:7]};
     }
  d: coverpoint pkt.dest_id {
      bins dest[8] = {[0:7]};
     }
  cross s, d;
 endgroup : cov1
```

```
 function new( string name , uvm_component parent);
  super.new( name , parent );
  cov1 = new();
 endfunction

 function void write(Packet t);
  real current_coverage;
   pkt = t;
   pkt_cnt++;
   cov1.sample();  // cause sampling of covergroup
   current_coverage = $get_coverage();
   `uvm_info("COVERAGE",$sformatf("%0d Packets sampled,  Coverage =
%f%% ",
                    pkt_cnt,current_coverage), UVM_MEDIUM)
 endfunction

endclass
```

Multiple Transaction Streams Without Synchronization

- One complication when using uvm_analysis_imps with multiple streams is that each uvm_analysis_imp expects to be tied to a write() function.
- Since there can only be one function named write() in the component, a workaround is needed.
- UVM provides a workaround by defining the `uvm_analysis_imp_decl macro.
- This macro allows you to declare a specialized "imp"-style analysis export that calls a function named write_SUFFIX where you specify _SUFFIX.
- The `uvm_analysis_imp_decl macro is creating an extension of the

- delay_analyser.sv

Multiple Transaction Streams With Synchronization

- Implementing the write() functions directly is easier when the required behavior of the analysis component allows for each stream's processing to be done independently.
- Many situations, however, require synchronization between the transaction streams, so that data is not lost.
- For example, an in-order Comparator has two streams, one from the Predictor and one from the Monitor.
- It must wait until it has a matching pair of transactions, one from each stream, to compare.

- In this case, rather than performing all the synchronization work manually in two write() functions, you can instantiate two analysis FIFOs and place two hierarchical exports on the Comparator.
- Since you are not implementing the behavior of the Comparator in the write() functions, a different approach is required.
- In this case, you can implement the Comparator behavior in the run() task of the component and use the blocking operations on the analysis FIFOs to perform the necessary synchronization.
- Inorder_comparator.sv

- predictors

End of Test

Objections

End UVM Test

UVM Sequences

- Sequence execution is started by calling the uvm_sequence start() method, this associates the sequence with a sequencer and then calls the sequences body method.
- Inside the sequence body method, other sequences can be executed or sequence_items can be generated and sent to the driver.
- Generally, stimulus generation is started with one main controlling sequence in the test class run() method, and this sequence then spawns other sequences which generate the stimulus within the testbench.
- Once a chain of sequences is started, the flow of execution can be hierarchical, or parallel, or executed in a randomized order.
- Sequences can also be randomized to change control or data variables.

- In order to handle the sequence_items arriving from the sequence, the driver has access to methods which are implemented in the sequencer, these give the driver several alternate means to indicate that a sequence_item has been consumed or to send responses back to the sequence.
- The handling of sequence_items inside a sequence often relies to some extent on the how the driver processes sequence items

Sequence Stimulus Generation Variations

1. **Controlling Stimulus Generation On More Than One Driver**

- The sequence architecture of a driver, sequencer, sequence_item and sequence is orientated around a single interface.
- In most cases, a testbench will need to handle transactions on multiple interfaces and this is achieved through the use of the virtual sequence.
- A virtual sequence is a sequence that can start sub-sequences on multiple sequencers.
- The way to implement a virtual sequence is for it to have handles for the target sequencers on which the sub-sequences run.
- The approach is to use a virtual sequencer which is a

2. Controlling Multiple Sequences Connected To A Single Driver

- Multiple sequences can interact concurrently with a driver.
- The sequencer has an arbitration mechanism to ensure that only one sequence_item has access to the driver at any point in time.
- The choice of which sequence item is sent is dependent on an user selectable sequencer arbitration algorithm.
- There are five built in sequencer arbitration mechanisms and a hook for a user defined algorithm.
- A sequence can be started with a priority, this enables some of the arbitration mechanisms to give the correct order of precedence to multiple sequences running on a sequencer.

- If responses are being returned from the driver to one of several sequences, the sequence id field in the sequence_item is used by the sequencer to route the response back to the right sequencer.
- The response handling code in the driver should use the set_id_info call to ensure that any response items have the same sequence id as their originating request.
- In some cases, such as processing an interrupt, a sequence needs to gain exclusive access to the driver, in this case a sequence can use the grab or lock methods.

3. Layering

- In many cases, sequences are used to generate streams of data objects which can be abstracted as layers, serial communication channels being one example and accessing a bus through a register indirection is another.
- The layering mechanism allows sequences to be layered on top of each other, so that high level layer sequences are translated into lower level sequences transparently.
- This form of sequence generation allows complex stimulus to be built very rapidly.

4. Waiting For A Hardware Event

- Whilst the driver takes care of normal hardware synchronisation, a sequence execution flow may need to be synchronised to a hardware event such as a transition on a sideband signal, or an end of reset condition.
- Rather than extend the driver and add another field to the sequence_item, it is possible to implement a wait_for_hardware_event method in a configuration object that contains a pointer to a virtual interface.

5. Interrupt Driven Stimulus

- One variation of the waiting for a hardware event theme is to use interrupts to trigger the execution of sequences.
- This might result in the exclusive execution of an interrupt service routine, or it might emulate the control state machine for a hardware device that generates interrupts when it is ready or has completed a task.

Sequences

- sequence_items
  - **Data Property Members**
    - Control - i.e. What type of transfer, what size
    - Payload - i.e. The main data content of the transfer
    - Configuration - i.e. Setting up a new mode of operation, error behavior etc
    - Analysis - i.e. Convenience fields which aid analysis - time stamps, rolling checksums etc

Transaction Methods

Method called by user	Virtual method	Purpose
copy	do_copy	Performs a deep copy of an object
clone	do_copy	Creates a new object and then does a deep copy of an object
compare	do_compare	Compares one object to another of the same type
convert2string	-	Returns a string representation of the object
print	do_print	Prints a the result of convert2string to the screen
sprint	do_print	Returns the result of convert2string
record	do_record	Handles transaction recording
pack	do_pack	Compresses object contents into a bit format
unpack	do_unpack	Converts a bit format into the data object format
to_struct	-	Optional: returns a struct representation of the object
from_struct	-	Optional: populates the data members of the object from the struct input

Sequence API

A uvm_sequence is derived from an uvm_sequence_item and it is parameterized with the type of sequence_item that it will send to a driver via a sequencer.

The two most important properties of a sequence are the body method and the m_sequencer handle.

- **The body Method**

An uvm_sequence contains a task method called body. It is the content of the body method that determines what the sequence does.

- **The m_sequencer Handle**

When a sequence is started it is associated with a sequencer.

The m_sequencer handle contains the reference to the sequencer on which the sequence is running.

The m_sequencer handle can be used to access configuration information and other resources in the UVM component hierarchy.

Running a sequence – 3 steps

Step 1 - Sequence Creation

Since the sequence is derived from an uvm_object, it is created using the factory:

```
my_sequence m_seq; // Declaration

m_seq = my_sequence::type_id::create("m_seq");
```

Using the factory creation method allows the sequence to be overridden with a sequence of derived type as a means of varying the stimulus generation.

Step 2 - Sequence Configuration

The sequence may need to be configured - examples of configuration include:

- Setting up start values - e.g. a start address or data value
- Setting up generation loop variables - e.g. number of iterations, which index number to start from
- Setting up pointers to testbench resources - e.g. a register map

```
m_seq.no_iterations = 10; // Direct assignment of values

// Using randomization
if(!m_seq.randomize() with {no_iterations inside {[5:20]};}) begin
  `uvm_error("marker", "Randomization failure for m_seq")
end

// Assigning a test bench resource
m_seq.reg_map = env.reg_map;
```

- **Step 3 - Starting The Sequence**
- A sequence is started using a call to its start() method, passing as an argument a pointer to the sequencer through which it will be sending sequence_items to a driver.
- The start() method assigns the sequencer pointer to a sequencer handle called m_sequencer within the sequence and then calls the body task within the sequence.
- When the sequence body task completes, the start method returns.
- Since it requires the body task to finish and this requires interaction with a driver, start() is a blocking

```
virtual task start
      uvm_sequencer_base sequencer, // Pointer to sequencer
      uvm_sequence_base parent_sequence = null, // Relevant if called within
      a sequence
      integer this_priority = 100, // Priority on the bit sequencer
      bit call_pre_post = 1); // pre_body and post_body methods called

// For instance - called from an uvm_component - usually the test:
apb_write_seq.start(env.m_apb_agent.m_sequencer);

// Or called from within a sequence:
apb_compare_seq.start(m_sequencer, this);
```

Sending a sequence_item to a driver

- **Step 1 - Creation**
- The sequence_item is derived from uvm_object and should be created via the factory:
- Using the factory creation method allows the sequence_item to be overridden with a sequence_item of a derived type if required.
- **Step 2 - Ready - start_item()**
- The start_item() call is made, passing the sequence_item handle as an argument. This call blocks until the sequencer grants the sequence and the sequence_item access to the driver.
- **Step 3 - Set**
- The sequence_item is prepared for use, usually through randomization, but it may also be initialized by setting properties directly.

- **Step 4 - Go - finish_item()**
- The finish_item() call is made, which blocks until the driver has completed its side of the transfer protocol for the item.
- No simulation time should be consumed between start_item() and finish_item().
- **Step 5 - Response - get_response()**
- This step is optional, and is only used if the driver sends a response to indicate to indicate that it has completed transaction associated with the sequence_item.
- The get_response() call blocks until a response item is available from the sequencers response FIFO.

```
// Inside some sequence connected to a sequencer

my_sequence_item item;

task body;
  // Step 1 - Creation
  item = my_sequence_item::type_id::create("item");

  // Step 2 - Ready - start_item()
  start_item(item);
  // Step 3 - Set
  if(!item.randomize() with {address inside {[0:32'h4FFF_FFFF]};}) begin
    `uvm_error("body", "randomization failure for item")
  end
  // Step 4 - Go - finish_item()
  finish_item(item);
endtask: body
```

Sequencer

uvm_sequencer #(REQ, RSP)
uvm_seq_item_pull_export
REQ seq_items
RSP seq_items
uvm_tlm_analysis_export
RSP seq_items
uvm_driver #(REQ, RSP)
uvm_seq_item_pull_port
Instance handle: seq_item_port
uvm_tlm_analysis_port
Instance handle: rsp_port
Sequencer-Driver Connections

```
// Agent containing a driver and a sequencer - uninteresting bits left
out
class adpcm_agent extends uvm_agent;
```

```
adpcm_driver m_driver;
adpcm_agent_config m_cfg;
// uvm_sequencer parameterized with the adpcm_seq_item for request &
response
uvm_sequencer #(adpcm_seq_item) m_sequencer;

// Sequencer-Driver connection:
function void connect_phase(uvm_phase phase);
  if(m_cfg.active == UVM_ACTIVE) begin // The agent is actively driving
 stimulus
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export); // TLM
 connection
    m_driver.vif = cfg.vif; // Virtual interface assignment
  end
endfunction: connect_phase
```

Bidirectional connection

```
// Same agent as in the previous bidirectional example:
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
uvm_sequencer #(adpcm_seq_item) m_sequencer;
adpcm_agent_config m_cfg;

// Connect method:
function void connect_phase(uvm_phase phase );
  if(m_cfg.active == UVM_ACTIVE) begin
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export); //
Always need this
    m_driver.rsp_port.connect(m_sequencer.rsp_export); // Response
analysis port connection
    m_driver.vif = cfg.vif;
  end
  //...
endfunction: connect_phase

endclass: adpcm_agent
```

Driver-Sequence API

- The uvm_driver is an extension of the uvm_component class that adds an uvm_seq_item_pull_port which is used to communicate with a sequence via a sequencer.
- The uvm_driver is a parameterized class and it is parameterized with the type of the request sequence_item and the type of the response sequence_item.
- In turn, these parameters are used to parameterize the uvm_seq_item_pull_port.
- The fact that the response sequence_item can be parameterized independently means that a driver can return a different response item type from the request type.

- The purpose of the driver sequencer API is for the driver to receive a series of sequence_items from sequences containing the data required to initiate transactions, and for the driver to communicate back to the sequence that it has finished with the sequence_item and that it is ready for the next item.
- **get_next_item() followed by item_done()**

  This use model allows the driver to get a sequence item from a sequence, process it and then pass a hand-shake back to the sequence using item_done(). No arguments should be passed in the item_done() call.
- The corresponding sequence implementation would be a start_item() followed by a finish_item()
- The drivers call to item_done() unblocks the finish_item() call in the sequence and then the sequence can access the fields in the sequence_item, including those which the driver may have updated as part of the response side of the pin level transaction.

Sequence
Sequencer
Driver
BFM Interface/ DUT
start_item()
Arbitration
get_next_item()
finish_item()
Pin level transaction activity
item_done()
Execution flow for get_next_item() followed by item_done()

```
//
// Driver run method
//
task run_phase( uvm_phase phase );
  bus_seq_item req_item;

  forever begin
    seq_item_port.get_next_item(req_item); // Blocking call returning
the next transaction

    //BFM handles all pin wiggling and population of req_item with
response data
    m_bfm.drive(req_item);

    seq_item_port.item_done(); // Signal to the sequence that the
driver has finished with the item
  end
endtask: run
```

```
//
// Sequence body method:
//
task body();
  bus_seq_item req_item;
  bus_seq_item req_item_c;

  req_item = bus_seq_item::type_id::create("req_item");

  repeat(10) begin
    $cast(req_item_c, req_item.clone); // Good practice to clone the
req_item item
    start_item(req_item_c);
    req_item_c.randomize();
    finish_item(req_item_c); // Driver has returned REQ with the
response fields updated
    `uvm_info("body", req_item_c.convert2string())
  end
endtask: body
```

Generating Stimulus with UVM Sequences

Different Generation Flow Options

Serial Execution:

```
A.start(sqr);
B.start(sqr);
C.start(sqr);
```

Parallel Execution:

```
A.start(sqr);
fork
   B.start(sqr);
   C.start(sqr);
join
D.start(sqr);
```

Hierarchical Execution:

```
A.start(sqr);
// But this calls
// B & C
// And B calls D & E
```

- In any sequence stimulus generation process there are three primary layers in the flow:
- 1. **The master control thread** - This may be a run task in an UVM test component or a high level sequence such as a virtual sequence or a default sequence. The purpose of this thread is to start the next level of sequences.
- 2. **The individual sequences** - These may be stand-alone sequences that simply send sequence_items to a driver or they may in turn create and execute sub-sequences.
- 3. **The sequence_item** - This contains the information that enables a driver to perform a pin level transaction. The sequence item contains rand fields which are randomized with constraints during generation within a sequence.

```
// Parent sequence body method running child
sub_sequences within a fork join
task body();
//
// Code creating and randomizing the child sequences
//
fork
        seq_A.start(m_sequencer);
        seq_B.start(m_sequencer);
        seq_C.start(m_sequencer);
join
```

examples

- **Test sequence (example)**

- Random order sequence

Overriding Sequences and Sequence Items

- Override_sequence

Virtual Sequences

- A virtual sequence is a sequence which controls stimulus generation using several sequencers.
- Since sequences, sequencers and drivers (proxy and BFM) are focused on point interfaces, almost all testbenches require a virtual sequence to coordinate the stimulus across different interfaces and the interactions between them.
- A virtual sequence differs from a normal sequence in that its primary purpose is not to send sequence items.
- Instead, it generates and executes sequences on different target agents.
- To do this it contains handles for the target sequencers and these are used when the sequences are started.

```
// Creating a useful virtual sequence type:
typedef uvm_sequence #(uvm_sequence_item) uvm_virtual_sequence;

// Virtual sequence example:
class my_vseq extends uvm_virtual_sequence;
  ...
  // Handles for the target sequencers:
  a_sequencer_t a_sequencer;
  b_sequencer_t b_sequencer;

  task body();
    ...
    // Start interface specific sequences on the appropriate target
sequencers:
    aseq.start( a_sequencer , this );
    bseq.start( b_sequencer , this );
  endtask
endclass
```

In order for the virtual sequence to work, the sequencer handles have to be assigned. Typically, a virtual sequence is created in a test class in the run phase and the assignments to the sequencer handles within the virtual sequence object are made by the test. Once the sequencer handles are assigned, the virtual sequence is started using a null for the sequencer handle.

```
my_seq vseq = my_seq::type_id::create("vseq");

vseq.a_sequencer = env.subenv1.bus_agent.sequencer;
vseq.b_sequencer = env.subenv2.subsubenv1.bus_agent3.sequencer;

vseq.start( null );
```

Hierarchical Sequences

- When dealing with sequences, it helps to think in layers when considering the different functions that a testbench will be asked to perform.
- At the lowest layer associated with each agent are API sequences.
- The middle layer which makes use of the API sequences to get work done are worker sequences.
- Finally at the top of the testbench controlling everything is a virtual sequence which coordinates the running of worker sequences on the different target sequencers.

Virtual Sequences
Worker Sequences
API Sequences
test1_seq
moduleA
_init_seq
moduleB
_init_seq
spi_read_seq
spi_write_seq
spi_rd_mod_
wr_seq
wait_irq_seq
Testbench
Virtual
Sequencer
Agent Sequencer
Agent Sequencer

- **API Sequences**
- API sequences are the lowest layer in the sequence hierarchy.
- They also do the least amount of work in that they are doing a very targeted action when they are run.
- API sequences should perform small discreet actions like performing a read on a bus, a write on a bus, a read-modify-write on a bus or waiting for an interrupt or some other signal.
- The API sequences would be included in the SystemVerilog package which defines the Agent on which the sequences are intended to run.

- hier_sequences

- **Worker Sequences**
- Worker sequences make use of the low level API sequences to build up middle level sequences.
- These mid-level sequences could do things such as dut configuration, loading a memory, etc.
- Usually a worker sequence would only be sending sequence items to a single sequencer.

- Worker_sequence

Virtual Sequences

- Virtual sequences are used to call and coordinate all the worker sequences. In most cases, designs will need to be initialized before random data can be sent in.
- The virtual sequence can call the worker initialization sequences and then call other worker sequences or even API sequences if it needs to do a low level action.
- The virtual sequence will either contain handles to the target sequencers (recommended) or be running on a virtual sequencer which allows access to all of the sequencers that are needed to run the worker and API sequences.

- virtual_sequence

Code snippets…

```
class ubus_example_env extends uvm_env;
    // Provide implementations of virtual methods such as get_type_name()
      `uvm_component_utils(ubus_example_env)
  // UBus reusable environment
      ubus_env ubus0;
  // Scoreboard to check the memory operation of the slave
      ubus_example_scoreboard scoreboard0;
  // new()
      function new(string name, uvm_component parent);
        super.new(name, parent);
      endfunction : new
  // build_phase()
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase); // Configure before creating the
                                  // subcomponents.
        uvm_config_db#(int)::set(this,"ubus0",
                                      "num_masters", 1);
     uvm_config_db#(int)::set(this,".ubus0",
                                  "num_slaves", 1);
     ubus0 = ubus_env::type_id::create("ubus0", this);
        scoreboard0 =
    ubus_example_scoreboard::type_id::create("scoreboard0",
            this);;
      endfunction : build_phase
  virtual function connect_phase();
        // Connect slave0 monitor to scoreboard.
        ubus0.slaves[0].monitor.item_collected_port.connect(
        scoreboard0.item_collected_export);
      endfunction : connect
  virtual function void end_of_elaboration_phase(uvm_phase phase);
        // Set up slave address map for ubus0 (basic default).
        ubus0.set_slave_address_map("slaves[0]", 0, 16'hffff);
      endfunction : end_of_elaboration_phase
endclass : ubus_example_env
```

Other configuration examples include:

— Set the `masters[0]` agent to be active:

```
uvm_config_db#(uvm_active_passive_enum)::set(this,"ubus0.masters[0]",
    "is_active", UVM_ACTIVE);
```

— Do not collect coverage for `masters[0]` agent:

```
uvm_config_db#(int)::set(this, "ubus0.masters[0].monitor",
    "coverage_enable", 0);
```

— Set all slaves (using a wildcard) to be passive:

```
uvm_config_db#(uvm_active_passive_enum)::set(this,"ubus0.slaves*",
    "is_active", UVM_PASSIVE);
```

Base test

```
class ubus_example_base_test extends uvm_test;
      `uvm_component_utils(ubus_example_base_test)
      ubus_example_env ubus_example_env0;
      // The test's constructor
      function new (string name = "ubus_example_base_test",
        uvm_component parent = null);
        super.new(name, parent);
      endfunction
  // Update this component's properties and create the ubus_example_tb
    component.
      virtual function build_phase(); // Create the top-level environment.
        super.build_phase(phase);
        ubus_example_env0 =
    ubus_example_tb::type_id::create("ubus_example_env0", this);
      endfunction
endclass
```

Override test

```
class test_read_modify_write extends ubus_example_base_test;
  `uvm_component_utils(test_read_modify_write)
  // The test's constructor
      function new (string name = "test_read_modify_write",
         uvm_component parent = null);
        super.new(name, parent);
      endfunction
  // Register configurations to control which
      // sequence is executed by the sequencers.
      virtual function void build_phase(uvm_phase phase);
        // Substitute the default sequence.
                  uvm_config_db#(uvm_object_wrapper)::
            set(this, "ubus0.masters[0].sequencer.main_phase",
                "default_sequence", read_modify_write_seq::type_id::get());
      uvm_config_db#(uvm_object_wrapper)::
            set(this, "ubus0.slaves[0].sequencer.main_phase",
                "default_sequence",slave_memory_seq::type_id::get());
        super.build_phase(phase);
      endfunction
endclass
```

Virtual seqr

```
class simple_virtual_sequencer extends uvm_sequencer;
      eth_sequencer eth_seqr;
      cpu_sequencer cpu_seqr;
  // Constructor
      function new(input string name="simple_virtual_sequencer",
        input uvm_component parent=null);
        super.new(name, parent);
      endfunction
  // UVM automation macros for sequencers
      `uvm_component_utils(simple_virtual_sequencer)
endclass: simple_virtual_sequencer
```

```
class simple_virt_seq extends uvm_sequence;
      ... // Constructor and UVM automation macros
      // A sequence from the cpu sequencer library
      cpu_config_seq conf_seq;
      // A sequence from the ethernet subsequencer library
      eth_large_payload_seq frame_seq;
      // A virtual sequence from this sequencer's library
      random_traffic_virt_seq rand_virt_seq;
```

```
    virtual task body();
          // Invoke a sequence in the cpu subsequencer.
          `uvm_do_on(conf_seq, p_sequencer.cpu_seqr)
          // Invoke a sequence in the ethernet subsequencer.
          `uvm_do_on(frame_seq, p_sequencer.eth_seqr)
          // Invoke another virtual sequence in this sequencer.
          `uvm_do(rand_virt_seq)
        endtask : body
  endclass : simple_virt_seq
```

Connecting a Virtual Sequencer to Subsequencers

```
class simple_tb extends uvm_env;
      cpu_env_c cpu0; // Reuse a cpu verification component.
      eth_env_c eth0; // Reuse an ethernet verification component.
      simple_virtual_sequencer v_sequencer;
      ... // Constructor and UVM automation macros
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Configuration: Set the default sequence for the virtual sequencer.
        uvm_config_db#(uvm_object_wrapper)::set(this,
                                                "v_sequencer.run_phase",
                                                "default_sequence",
                                                simple_virt_seq.type_id::get());
        // Build envs with subsequencers.
        cpu0 = cpu_env_c::type_id::create("cpu0", this);
        eth0 = eth_env_c::type_id::create("eth0", this);

        // Build the virtual sequencer.
        v_sequencer =
    simple_virtual_sequencer::type_id::create("v_sequencer",
            this);
      endfunction : build_phase

  // Connect virtual sequencer to subsequencers.
      function void connect_phase();
        v_sequencer.cpu_seqr = cpu0.master[0].sequencer;
        uvm_config_db#(uvm_sequencer)::set(this,"v_sequencer",
                                           "eth_seqr",eth0.tx_rx_agent.sequencer);
      endfunction : connect_phase

endclass: simple_tb
```

Scbd

```
class cpu_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(cpu_scoreboard)

  uvm_analysis_export #(bus_xact) iss_export;
  uvm_analysis_export #(bus_xact) cpu_export;
  typedef uvm_in_order_class_comparator #(bus_xact) comp_t;
  comp_t m_comp;
  ...
  function void build_phase(uvm_phase phase);
    iss_export = new("iss_export", this);
    cpu_export = new("cpu_export", this);
    m_comp     = comp_t::type_id::create("comp", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    iss_export.connect( m_comp.before_export );
    cpu_export.connect( m_comp.after_export );
  endfunction
  ...
endclass
```

Reports every match/mismatch

Built-in FIFOs

```
uvm_in_order_comparator #(type T,
                          type comp,      // calls compare()
                          type convert,   // implements convert2string()
                          type pair_type)
```

Base class
(don't use directly)

```
uvm_in_order_class_comparator #(type T)
```

T is a class

```
uvm_in_order_built_in_comparator #(type T)
```

T is a built-in type

Each comparator writes its results to an analysis port:

```
uvm_analysis_port #( pair_type ) pair_ap;
```

Generates matched pairs

```
uvm_algorithmic_comparator #(type BEFORE,
                             type AFTER,
                             type TRANSFORMER)
```

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

  rand logic [3:0] opcode;
  rand logic [3:0] src;
  rand logic [3:0] src2;
  rand logic [3:0] dst;

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

  function string do_pack(uvm_packer packer);
    ...

  function void do_copy(uvm_object rhs);
    ...

  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    ...
endclass
```

Unconstrained

```
class instruction_seq extends uvm_sequence #(instruction);
 `uvm_object_utils(instruction_seq)

  int unsigned icount;

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

  task body;
    if( !std::randomize(icount) with { icount < 100; } )
      `uvm_warning("", "randomize failed")
    repeat (icount)
      `uvm_do(req)
  endtask

endclass
```

Sequence item type

Corresponding sequencer type

```
typedef uvm_sequencer #(instruction) instruction_generator;
```

```
class instruction_driver extends uvm_driver #(instruction);
  virtual mem_if m_bus_if;

  bit iwordstream [];
  shortint iword;
  ...

  task run_phase(uvm_phase phase);
    forever @(posedge m_bus_if.rom.clk)
      if (m_bus_if.rom.re) begin
        iwordstream.delete();  //clear previous bitstream

        seq_item_port.get(req);
        assert(req.pack(iwordstream));

        iword = {<<{iwordstream}};
        m_bus_if.rom.datar  <= iword;
      end
      else
        m_bus_if.rom.datar <= 'bz;
  endtask
  ...
```

Dynamic array holds instruction of any length

Reads ROM inputs and drives bus

Pack next instruction to bitstream

Pack bitstream to shortint and write to bus

Seqr driver interface

Driver may modify transaction

Blocking

TLM-1.0 style

```
seq_item_port.get_next_item(req);
...
... // e.g. req.data = bus.data;
...
seq_item_port.item_done();
```

```
seq_item_port.get(req);
```

Allow sequencer to move on

Separate response object

```
seq_item_port.get_next_item(req);
...
rsp.set_id_info(req);
seq_item_port.item_done(rsp);
```

```
seq_item_port.get(req);
...
rsp.set_id_info(req);
seq_item_port.put(rsp);
```

Constraining instruction

```
class same_regs_instruction extends instruction;
  `uvm_object_utils(same_regs_instruction)

  function new(string name = "");
    super.new(name);
  endfunction
  constraint assert_same_regs {
    src  == dst;
    src2 == dst;
  }
endclass
```

```
class test1 extends uvm_test;
 `uvm_component_utils(test1)

  verif_env env1;
  ...
  function void start_of_simulation_phase(uvm_phase phase);
    instruction::type_id::set_type_override(
                                  same_regs_instruction::get_type());
  endfunction

endclass
```

UVM factory override

```
class register_instruction extends instruction;
 `uvm_object_utils(register_instruction)

  constraint more_regs_and_stores {
    opcode dist { [nop_op:load_op]  := 1,
                   store_op         := 16,
                  [move_op:or_op]   := 4,
                  [jump_op:jneg_op] := 1 };
  }
endclass
```

:= is weight-per-item, :/ is weight-per-range

Each derived instruction class has a corresponding test class to override type passed to uvm_random_stimulus

```
class test2 extends uvm_test;
  verif_env env1;

  function void start_of_simulation_phase(uvm_phase phase);
    instruction::type_id::set_type_override(
                          register_instruction::get_type());
  endfunction
  ...
```

Using array..inside

```
class C;

  rand int unsigned i;

  constraint fixed_values {
    i inside { [0:7], [128:143], 512, 65536 };
  }

  rand byte a[];  // Dynamic array

  // constraint asize { a.size < 16; }

  function void pre_randomize;
    int n = $urandom_range(0, 15);
    a = new[n];
  endfunction

  constraint array {
    foreach( a[j] ) a[j] inside { 1, 2, 4, [16:64] };
  }

endclass
```

Implicit new[n]

Randomize size and contents of a[ ] in two steps

Using function call

```
class C;
  rand byte unsigned data;
  rand bit           parity;

  constraint parity_bit {
    parity == calc_parity(data);
  }

  function automatic bit calc_parity(byte unsigned arg);
    calc_parity = ^arg;
  endfunction

endclass
```

Like assignment in post_randomize, but a constraint may be overridden

```
C c;
c = new;
repeat(10) #10 if( !c.randomize() with { c.data < 128; } )
                `uvm_error("", "Constraint failed")

c.parity_bit.constraint_mode(0);
```

Check randomize for success

Inline constraint

Turn off constraint

Seq body method

- Defines overall flow of sequence
- Called automatically when sequence is executed
- Macros step through sequence phases

```
task body;

  `uvm_do_with(req, {opcode == movi_op;
                     {src,src2}==8'FF;
                     dst == base_reg;} )
  `uvm_do_with(req, {opcode == movhi_op;
                     {src,src2}==8'FF;
                     dst == 6;} )

  ...
endtask
```

Execute all sequence phases with additional constraints

Using virtual sequence

```
class my_virtual_sequence extends uvm_sequence;

  `uvm_object_utils(my_virtual_sequence)
  `uvm_declare_p_sequencer(my_virtual_seqr)

  sequence_0 seq0;
  sequence_1 seq1;

  ...

  task body;
    `uvm_do_on( seq0, p_sequencer.seqr0 )

    seq1 = sequence_1::type_id::create("seq1");
    seq1.randomize();
    seq1.start( p_sequencer.seqr1 );
  endtask

endclass
```

Sequence references

Reference to virtual sequencer

Sequencer to use

or set up a reference in the virtual sequence itself

Using virtual seqr

```
class my_virtual_seqr extends uvm_sequencer;
  `uvm_component_utils(my_virtual_seqr)
  ...
  sequencer_0 seqr0;
  sequencer_1 seqr1;
  ...
endclass
```

Sequencers to be controlled (optional)

```
class my_env extends uvm_env;
  ...
  function void build_phase(uvm_phase phase);
    v_sequencer = my_virtual_seqr::type_id::create(...);
    agent0      = agent_0::type_id::create(...);
    agent1      = agent_1::type_id::create(...);
  endfunction

  function void connect_phase(uvm_phase phase);
    v_sequencer.seqr0 = agent0.seqr0;
    v_sequencer.seqr1 = agent1.seqr1;
  endfunction
endclass
```

End