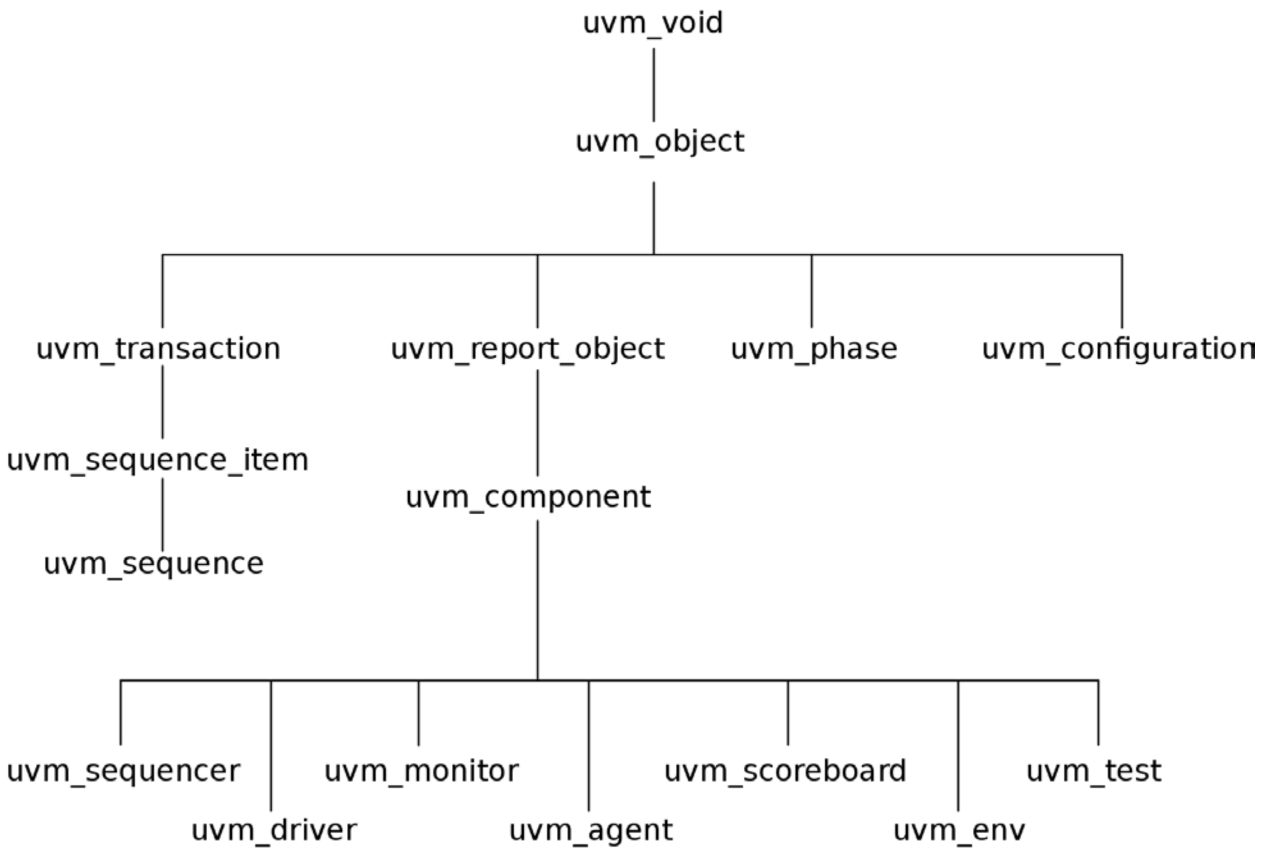
EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 5

OBJECTIVE: To learn about sequence items, sequences and sequencers, and their implementations in UVM.

UVM_CLASSES:
In UVM, all of the blocks mentioned in Lab1 as part of a layered testbench are represented as objects that are derived from already-existing classes.

UVM Class tree:



uvm_sequence_item:

These allow us to move from the signal level to the transaction level. A transaction is a data item that is eventually or directly processed by the DUT. For example, data packets (i.e. address + data), instructions and pixel values can all form transactions. Transactions are extended from either the uvm_transaction class or from the uvm_sequence_item class. uvm_transaction is a typedef of uvm_sequence_item.

uvm_sequence:

A sequence is a series of transactions, i.e. in a specific order. They create transactions suited to our needs and may generate as many as needed. For example, a sequence could be a set of valid addresses and corresponding data being sent to the DUT, a set of valid combinations of instructions, or an image sent to the DUT in the form of a set of pixel values. A sequence could also be used to apply other stimulus scenarios to the DUT, such as a reset condition or an initialization event.

uvm_component:

UVM verification component classes are derived from the uvm_component class, which provides useful characteristics such as hierarchy, phasing, configuration, reporting, factory registration and transaction recording.

The following are some important UVM component classes**:**

uvm_sequencer
The sequencer delivers a sequence to the driver.

uvm_driver
The driver converts the transaction-level stimulus into pin wiggles on the DUT.

uvm_monitor
The monitor converts pin values back into transactions.
The monitor is passive, in the sense that it does not drive any signals.

uvm_scoreboard
A scoreboard predicts and tracks the DUT output values to determine if the DUT is functioning properly.

uvm_env
The environment instantiates and connects all of the testbench components.

uvm_test
The test instantiates the environment.
It can also re-configure the environment based on the specific test requirements.
It controls the running of sequences and it can replace constraints as needed.

Let us look into **uvm_sequence_item** in more detail:

uvm_sequence_item is derived from the **uvm_object** class. The uvm_object has a number of virtual methods which are used to implement common data object functions (**copy, clone, compare, print, pack, unpack,** etc.) and these should be implemented to make the sequence_item more general-purpose. uvm_object also has various macros defined for it, namely **Utility Macros** and **Field Macros**.

Utility Macros:

When defining a class, its type has to be registered with the UVM factory to allow name-based overriding. The utility macros provide implementations of the create method (needed for cloning) and the get_type_name method (needed for debugging), etc.

Field Macros:

The `uvm_field_* macros are invoked inside of the `uvm_object_utils_begin and `uvm_object_utils_end, for the implementations of the methods: copy, compare, pack, unpack, record, print, etc. They are also known as field automation macros.

Commonly used field automation macros are as follows:

```
`uvm_field_int
`uvm_field_string
`uvm_field_enum
`uvm_field_real
`uvm_field_event
```

objects with field macros:

```
`uvm_object_utils_begin(TYPE)
      `uvm_field_*(FIELD,FLAG)
`uvm_object_utils_end
```

FLAGS:

Flag	**Description**
UVM_ALL_ON	Set all operations
UVM_DEFAULT	Use default flag settings
UVM_NOCOPY	Don’t copy this field
UVM_NOCOMPARE	Don’t compare this field
UVM_NOPRINT	Don’t print this field
UVM_NOPACK	Don’t pack or unpack this field
UVM_PHYSICAL	Treat as a physical field
UVM_ABSTRACT	Treat as an abstract field
UVM_READONLY	Treat as read only

Example:

```
`uvm_field_int(addr,UVM_ALL_ON | UVM_NOPRINT)
```

- This includes addr in all methods except for print.

User defined implementation of uvm_object utility methods:

The user can provide his/her own implementation for functions such as print, compare, copy, pack and unpack. To do this, define these methods in the transaction using do_<method_name> and call them using <method_name>.

Now, we can start building the sequence item for an adder RTL.

Step 1: In adder_sequence.sv, do the following in the adder_transaction class:

- Declare two random variables a, b each of 4-bit size.

```
rand bit [3:0] a;
rand bit [3:0] b;
```

- Write constraints for these two variables.

```
constraint c_a { a >= 0; a < 16; }
constraint c_b { b >= 0; b < 16; }
```

- Register them with the UVM factory using the utility and field macros.

```
`uvm_object_utils_begin(adder_transaction)
       `uvm_field_int(a,UVM_ALL_ON)
       `uvm_field_int(b,UVM_ALL_ON)
`uvm_object_utils_end
```

- Write a user implementation for the print method. Add the following code:

```
function void do_print(uvm_printer printer);
      super.do_print(printer);
      $display("Printing from do_print:\n");
endfunction
```

Now, let us look into **sequence** and **sequencer** in more detail:

UVM sequence:

A sequence is an ordered collection of transactions. Sequences can be reused, extended, randomized and combined sequentially and hierarchically, in various ways. Sequences are extended from the **uvm_sequence** class. The most important properties of a sequence are:

body method

A body method specifies what the sequence does.

m_sequencer handle

The m_sequencer handle contains the reference to the sequencer on which the sequence is running. It is initialized when the sequence is started.

A sequence will get executed upon calling the start of the sequence from the test.

```
sequence_name.start(sequencer_name);
```

where, sequencer_name specifies on which sequencer the sequence has to run.

UVM sequencer:

A sequencer is responsible for the coordination between a sequence and the driver. The sequencer sends the transaction to the driver and obtains the response from the driver. (A response transaction from the driver is optional.) When multiple sequences are running in parallel, then sequencer is responsible for arbitrating between the parallel sequences. A sequencer is extended from **uvm_sequencer**.

We will be using the default sequencer provided by UVM. Hence, we don’t have to add any code in the testbench.

Sequence and Driver Communication:

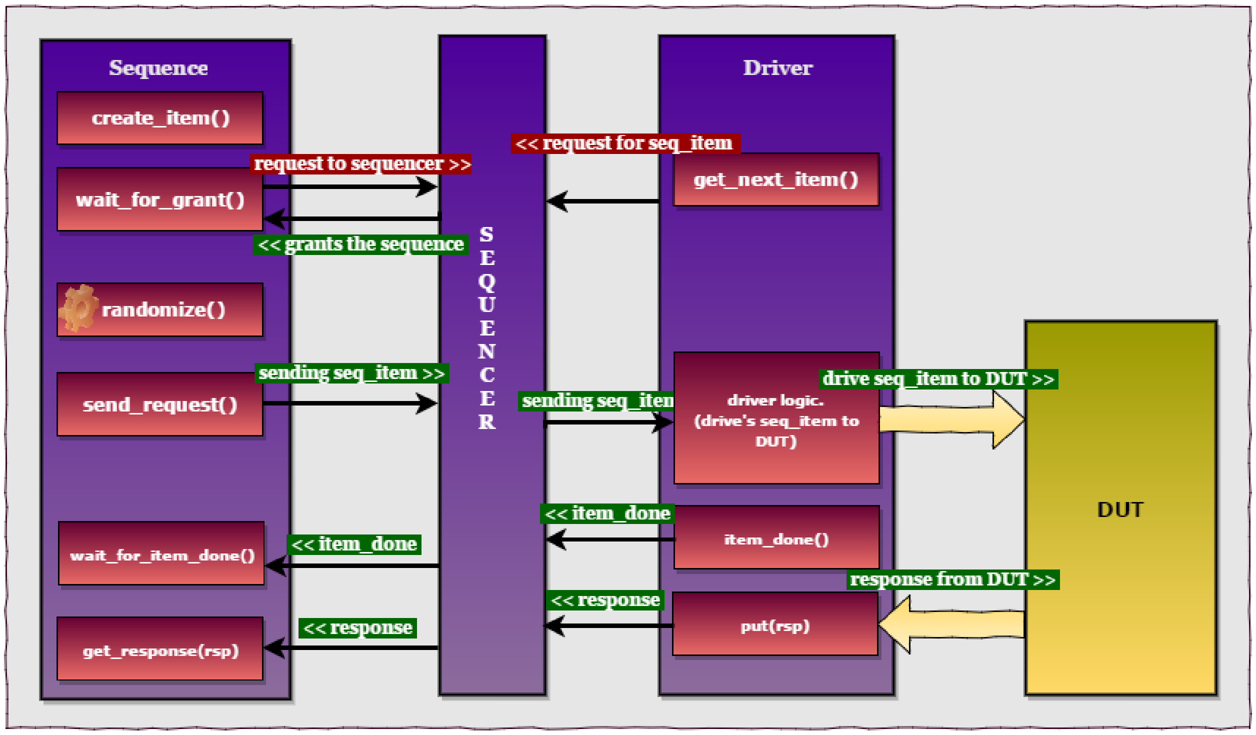


Fig. 1. Sequence and Driver Communication
(ref: http://www.verificationguide.com/p/uvm-sequence.html)

When the body() method is called in a sequence, the following actions occur:
1) A transaction is created using the create() method.
2) After a transaction is created, the wait_for_grant() method is called. This method is a blocking method.
3) In the run task of the driver, when seq_item_port.get_next_item() is called, then the sequencer unblocks the wait_for_grant() method. If more than one sequence is getting executed by sequencer, then using arbitration rules, it unblocks the wait_for_grant() method.
4) After the wait_for_grant() unblocks, the transaction can be randomized. Then, using the send_request() method, the transaction is sent to the driver.
5) After calling the send_request() method, the wait_for_item_done() method is called. This is a blocking method and the execution is blocked.
6) The transaction, which is sent from the sequence, is available in the driver as the seq_item_port.get_next_item(req) method argument. Then, the driver drives this transaction onto a bus or sends it to a lower level.
7) Once the driver operations have been completed, then by calling seq_item_port.put(rsp), the wait_for_item_done() method of the sequence gets unblocked. Using get_response(res), the response transaction from the driver is taken by the sequence and processed.

Step 2: In adder_sequence.sv, do the following in the adder_sequence class:

- In the task "body", create a sequence of 8 new transactions.
```
repeat(8) begin
      req = adder_transaction::type_id::create("req");
```
- Call the wait_for_grant() task.
```
wait_for_grant();
```
- Randomize the transactions.
```
if (!req.randomize()) begin
      `uvm_error("MY_SEQUENCE", "Randomize failed.");
end
```
- Call the print method of the transaction.
```
req.print();
```
- Send the request to the driver.
```
send_request(req);
```
- Wait for the transaction to be driven by the driver.
```
wait_for_item_done();
```
- Make sure that you end the repeat block.

Step 3: In the adder_driver.sv, do the following:

- Wait for a transaction from the sequencer.
```
forever begin
      seq_item_port.get_next_item(req);
```
- Drive the transaction onto DUT.
```
dut_vif.a = req.a;
dut_vif.b = req.b;
```
- Indicate to the sequencer that the transaction has been driven.
```
seq_item_port.item_done();
#5;
```
- Make sure that you end the forever block.

Step 4: In the adder_agent.sv, do the following:

- Declare the driver and sequencer.
```
adder_driver driver;
uvm_sequencer#(adder_transaction) sequencer;
```
- In the build phase, build the driver and the sequencer using a create function call.
```
driver = adder_driver ::type_id::create("driver", this);
sequencer =
uvm_sequencer#(adder_transaction)::type_id::create("sequenc
er", this);
```
- In the connect phase, connect the sequencer and the driver.
```
driver.seq_item_port.connect(sequencer.seq_item_export);
```

Step 5: In adder_test.sv, do the following:

Start the sequence by adding the following code in run_phase()

```
seq = adder_sequence::type_id::create("seq");
seq.start(env.agent.sequencer);
```

Step 6: Run the following commands on the command line to run the simulation. Take a snapshot of the results.

```
      vcs -Mupdate +v2k -sverilog -timescale=1ns/10ps +incdir+$UVM_HOME/src
$UVM_HOME/src/uvm.sv $UVM_HOME/src/dpi/uvm_dpi.cc -CFLAGS -DVCS top.sv
adder_interface.sv  adder.v -l compile.log +vcs+dumpvars+verilog.vpd -
debug_all

      ./simv +UVM_TESTNAME=adder_test +ntb_random_seed=5258623
```

(Note: Enter your student ID for the random seed value.)

Questions:

1. What is the difference between copy and clone member functions of the uvm_sequence_item class?

2. Uncomment the following code in **adder_sequence.sv** and run the simulation.

```
//adder_transaction req_1;
//adder_transaction req_2;

//wait_for_grant();
//req_1 = adder_transaction::type_id::create("req_1");
//if (!req_1.randomize()) begin
//  `uvm_error("MY_SEQUENCE", "Randomize failed.");
//end
//`uvm_info("req_1_print","Printing REQ_1",UVM_LOW);
//req_1.print();
//send_request(req_1);
//wait_for_item_done();

//wait_for_grant();

//$cast(req_2,req_1.clone());

//`uvm_info("req_2_print","Printing REQ_2",UVM_LOW);
//req_2.print();
//send_request(req_2);
//wait_for_item_done();
```

   What is the name of the sequence item being printed by the `req_2.print()` statement? Why is there an abnormality?

3. Give the values of a, b, sum and carry for req_1 and req_2 for the simulation with the following changes:

   Case 1: Uncomment the following line:

   ```
   //req_2.a = 8;
   ```

   Case 2: Comment the following lines:

```
$cast(req_2,req_1.clone());
req_2.a = 8;
```
Uncomment the following lines:
```
//req_2 = adder_transaction::type_id::create("req_2");
//req_2.copy(req_1);
```
Case 3: Uncomment the following line:
```
//req_2.a = 8;
```

4. What is a virtual sequence? Why is it used?

5. The field automation macro **`` `uvm_field_int ``** implements the data operations for any packed integer property. Name the field automation macro for a queue of integers.

What is to be turned in?

A report (pdf file) consisting of the screenshot of the simulation result described in Step 6 and the answers to the above questions.

References used in the preparation of this lab:

1. http://www.verificationguide.com
2. https://www.edaplayground.com
3. http://testbench.in
4. http://www.asic-world.com
5. https://verificationacademy.com
6. http://systemverilog.us/driving_into_wires.pdf
7. http://www.eetimes.com/document.asp?doc_id=1276112
8. http://www.embedded.com/print/4004083
9. https://colorlesscube.com/uvm-guide-for-beginners
10. https://www.doulos.com
11. http://www.chipverify.com/uvm