UVM

NOTES

Jairaj Mirashi
Design Verification Engineer

What are some of the benefits of UVM methodology?

Testbench prototyping becomes faster because of all the base classes for drivers, monitors, sequencers

1. There's a well-defined reporting system which supports various levels of verbosities like LOW, DEBUG, etc
2. Supports register model creation and maintenance which simplifies the way DUT registers are accessed
3. Well-structured plug and play style components like agents that can be plugged into any environment to support a particular protocol
4. Factory helps to override certain components without having to modify existing connections in the testbench
5. Configuration databases that allow components to share objects and data between each other
6. Make use of TLM features that enable different components that receive transactions to perform different operations on it.
7. Promotes re-usability, flexibility, uniformity and robustness to every testbench built on UVM

What are some of the drawbacks of UVM methodology?

With increasing adoption of UVM methodology in the verification industry, it should be clear that the advantages of UVM overweight any drawbacks.

1. For anyone new to the methodology, the learning curve to understand all details and the library is very steep.
2. The methodology is still developing and has a lot of overhead that can sometimes cause simulation to appear slow or probably can have some bugs

UVM FACTORY

1. We have to register the .class to the factory
2. It is a .class where it creates the components.and objects.
3. from test we can control, configure the factory to create whether parent.class .or child.classfor .this we have to register a .class .with the factory
4. There are three steps to register the .class .with factory
   1. Create Wrapper around component registry .class, typedefed to type_id
   2. static .function to get the type_id
   3. function to get the .type name

Why do we need to register a class with a factory?

You don't necessarily need to. It is registered with a factory simply to make the testbench more re-usable and have the capability to override it with some other derivative component later on.

What is the difference between creating an object using new() and create() methods?

1. create() is the factory method which is used in uvm to create components .and transaction .class objects & it is a .static method defined in registry .class
2. create() internally call new() constructor
3. new() is the system verilog constructor to create components .and transaction .class objects

/STEPS TO REGISTER CLASS WITH FACTORY

```
class my_component extends uvm_component;
 //wrapper class around the component/object registry class
 typedef uvm_component_registry#(my_component) type_id;
 /*//incase of object
 typedef uvm_object_registry#(my_object) type_id;*/

 //used to get the type_id wrapper
 static function type_id get_type();
  return type_id::get();
 endfunction : get_type

 //used to get the type_name as string
 function string get_type_name();
  return "my_component";
 endfunction : get_type_name

endclass : my_component
```

1. so writing .this much code to register the .class everytime is .bit lengthy
2. so .for registering the component/object .with factory we will .use macros

```
//MACRO FOR COMPONENT CLASSES          //MACRO FOR PARAMETRIZED COMPONENT CLASSES
-----------------------------          ----------------------------------------
 `uvm_component_utils()                   `uvm_component_param_utils()

//MACRO FOR OBJECT CLASSES             //MACRO FOR PARAMETRIZED OBJECT CLASSES
--------------------------             ---------------------------------------
 `uvm_object_utils()                      `uvm_object_param_utils()
```

//FACTORY OVERRIDEN

==========

//GLOBAL OVERRIDE

1.Make sure both classes should be polymorphically compatible

eg. original and substitude should have parent child relation

```
.-------------------------------------------------------------------------------------------------.
| set_type_override_by_type(original_type::get_type(),substitude_type::get_type(),bit replace=1); |
'-------------------------------------------------------------------------------------------------'
```

//EXAMPLE

```
.-------------------------------------------------------------------------.
| set_type_override_by_type(my_driver::get_type(),child_driver::get_type()); |
'-------------------------------------------------------------------------'
```

1. It will override type_id of parent .with type_id of child.
2. type_id of driver will be override by type_id of child_driver in entire testbench.
   so now driver will have type_id of child_driver
3. whenever create() method will called by driver it will create object .for child_driver.

//INSTANCE OVERRIDE

```
.-------------------------------------------------------------------------------------------------------.
| set_inst_override_by_type("path",original_type::get_type(),substitude_type::get_type(),bit replace=1); |
'-------------------------------------------------------------------------------------------------------'
```

/*Why replace=1 here so that if again want to override this instance with other then by checking this value
simulator will understand that previous overriding happened so it will first convert it to original type then override again with new type

What is factory in UVM ? Why is it needed and what is meant by factory override?

The purpose of factory in UVM is to change the behaviour of the testbench without any change in code .
A factory is a look up table used in UVM for creating objects of component or transaction types. The benefit of object creation using factory is that a testbench build process can decide at run-time which type of object has to be created. Based on this, a class type could be substituted with another derived class type without any real code change.

To ensure this feature and capability, all classes are recommended to be registered with factory.

In the example below , instances agt1 and agt2 is changed from type packet_agent_c to eth_packet_agent_c using the factory override .

Syntax

```
packet_agent_c agt1, agt2;
agt1 = packet_agent_c::type_id::create("agt1", this);
agt2 = packet_agent_c::type_id::create("agt2", this);
```

Override :

```
pkt_agent_c::type_id::set_type_override(eth_packet_agent_c::get_type());
```

How to register a class to the factory ?

By using macros :

`uvm_component_utils() for components
`uvm_object_utils() for objects
`uvm_component_utils() for parametrized components
`uvm_object_utils() for parametrized objects

these macros will get expended in three parts

1. wrapper --> uvm_component_registry/uvm_object_registry which typedefed to type_id
2. get_type() -->.static method which returns type_id
3. get_type_name() --> method which .return component/object name as a string

create(string ="name",uvm_component parent);

Why create accept two argument during building a components but only one during building objects ?

Because componets follow hierarchy so they have name and parent but object did not follow any hierarchy so they dont have parent that is why only one constructor which is of string type name only passed

STIMULUS MODELING

1.Transaction .class extends from uvm_sequence_item
2.Inside transaction.class we declare properties which is used as.input to the DUT.and valid constraints .and methods required.for the.class variables(properties) eg. copy,clone,compare,print etc.

what is purpose of field macros ?
register the.property.for copy,compare,clone & print method

There are two ways to implement those methods

1. Using field macros we can automate the methods by enabling the methods .for all the properties
2. By override the.virtual methods eg. do_copy, do_compare, do_print etc.

CLONE() METHOD

==========

1. While using copy mtheod we will make sure object for destination handle already created
   but for clone there is no need to create a destination object

2. clone does two things ---> create+copy
   1. first it will create object for destination
   2. then it call the copy method

   after the copy the object which is created it will return that object reference through the parent handle.

3. The return type of clone is uvm_object

```
function uvm_object clone();

 write_xtn t=write_xtn::type_id::create("t1");
 t.copy(t1);
 uvm_object p=t;
 return p;

endfunction
```

```
.-----------------------------------.
| write_xtn t1,t2;       |// t1 is of write_xtn type so clone will create a object of
|                          |// write_xtn type and copy all the properties of t1 to that object
| initial                  |          // and it pass handle of that object through parent handle.
| begin     |
|/*        t2=t1.clone();      */|
|  $cast(t2,t1.clone()); |
| end               |
'-----------------------------------'
```

//t2=t1.clone(); ===> //t2.copy(t1) --> t2.do_copy(t1)

it is returning through parent handle so we can't directly assign parent handle to child ,we use $cast here.

UVM PHASES

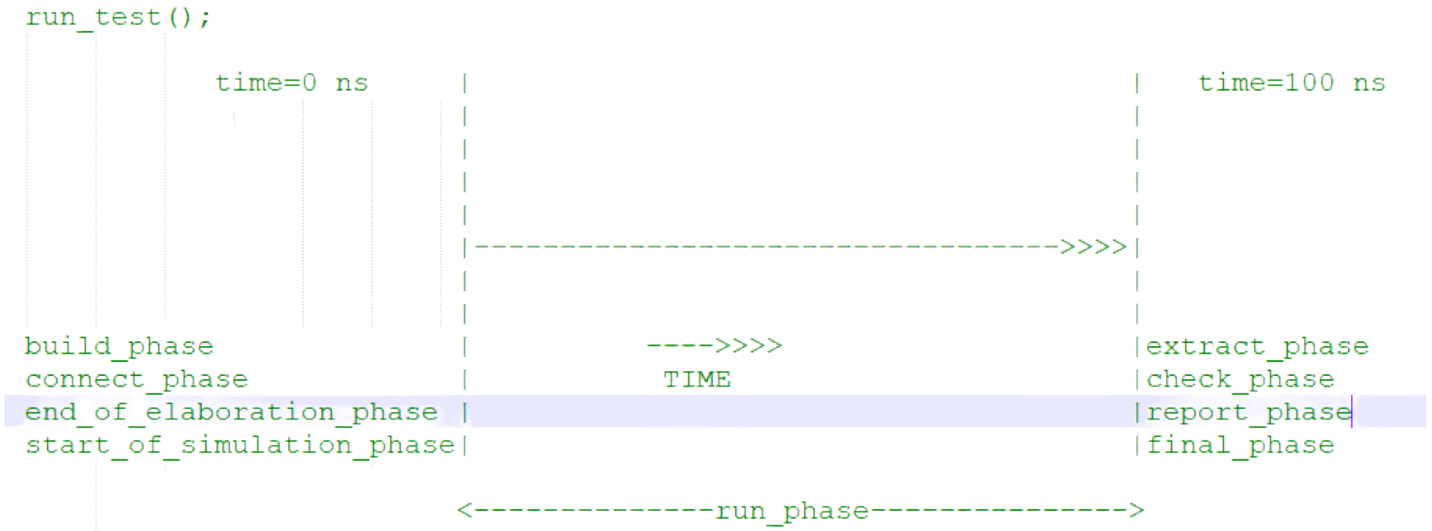
ORDER OF EXECUTION OF PHASES

---

1. build_phase }
2. connect_phase } // PRE-RUN PHASES
3. end_of_elaboration_phase }
4. start_of_simulation_phase }
5. run_phase ----------
6. extract_phase }
7. check_phase } // POST-RUN PHASES
8. report_phase }
9. final_phase }

All above phases automatically called by run_test() method

EXECUTION OF PHASES



//BUILD_PHASE

1.It is used to construct sub-components

2.Component hierarchy is therefore build top-down

build_phase() executed top down in uvm_component hierarchy?

In UVM, all the testbench components like a test, Env, Agent, Driver, Sequencer are based on the uvm_component class and there is always a hierarchy for the testbench components.

The build_phase() method is part of the uvm_component class and is used to construct all the child components from the parent component. So, to build the testbench hierarchy you always need to have a parent object first, which can then construct its children, and that can further construct its children using build_phase. Hence, build_phase() is always executed top-down.

//CONNECT_PHASE

1.It is used to connect the components .or any connection like connecting static and virtual interface

2.It follow bottom-up approach

3.First connect_phase of driver,monitor,sequencer executed then agents--->env

//END_OF_ELABORATION_PHASE

1.It is used to make any .final adjustments to the structure,configuration .or connectivity of the testbench .before simulation starts

2.It follow bottom-up approach

//START_OF_SIMULATION_PHASE

1.It is used to display banners: testbench topology,.or configuration information

2.It follow bottom-up approach

//RUN_PHASE

---

1. It is the only phase which is task remaining all phases is function.
2. It consumes simulation .time remaining phases completes in zero simulation .time
3. run tasks are executed in parallel, run_phases of all components executed in parallel.
4. 12 Sub phases are added in parallel with run_phase
5. It is recommended either .use run_phase .or sub-run_phases don't .use both coz multiple driver issue can happen like multiple driver driving same bus

DRIVER

```
,---------------------------------,
| fork                            |
|  run_phase();                   |
|  begin                          |
|   reset_phase();                |
|   configure_phase();            |
|   main_phase();                 |
|   shutdown_phase();             |
|  end                            |
| join                            |
'---------------------------------'
```

6. driver.and monitor should .use run_phase
7. don't.use phase domain .or jumping

OBJECTIONS

1. Components and Objects can raise and drop the objections
2. It remains in same phase till all the objections are dropped
3. Used in the run_phase

//OBJECTIONS
---

1. IN COMPONENTS
   -> Phase.raise_objection(this);
   -> Phase.drop_objection(this);

   2. IN OBJECTS
   -> Starting_phase.raise_objection(this);
   -> Starting_phase.drop_objection(this);

```
//EXAMPLES
------------

class agent extends uvm_agent;

 virtual task run_phase(uvm_phase phase);
  phase.raise_objection(this);   //this keyword will tell which component raised the objection
    #100;         //phase will tell phase eg. reset,configure,main,shutdown
  phase.drop_objection(this);    //this keyword will tell which component droped the objection
 endtask

endclass


class driver extends uvm_driver;

 virtual task run_phase(uvm_phase phase);
  phase.raise_objection(this);
    #10;
  phase.drop_objection(this);
 endtask

endclass
```

Component which is using run_phase should tell the simulator that I'm using the run_phase by raising a objections

once it complete all the.task.or functionalities in run_phase then it should drop the objection

Simulator will track the objections , It will done via variable count, .this count variable will increment every .time when a component raise_objection.and will decrement every.time when component drop the objection..

So when all the raise_objection will droped the count will become zero at that.time simulator will understand that all the components have completed the run_phase,.and then it will terminate the run_phase .and move to post run_phases

```
uvm_objection::raise_objection(uvm_object null,string description="",int count=1);

uvm_objection::raise_objection(uvm_object null,string description="",int count=1);
```

1. uvm_object ---> only.this argument we have to passed
2. description ---> default value =""
3. count   ---> default value = 1

REPORTING MECHANISM

IT is used to display message on terminal according to verbosity level

Why Reporting Mechanism over $display ?????

==========

1. It can filter display message manually
2. We can enable/disable certain message from MakeFile without going back to tb.and change it in code
3. We can give severity here. eg. fatal,error,warning,info etc.
4. We can call reporting mechanism by.function or UVM Macros(recommended).
5. During developing tb we want certain messages and after completing TB, We don't want these messages so we can simply filter out using verbosity levels.

TRANSACTION LEVEL MODELING(TLM)

WHY TLM ?????

==========

1. TLM Stands for Transaction Level MODELING
2. TLM Promoted Re-usability as they have same .interface
3. Interoperability for mixed language verification environment
4. Maximizes reuse and minimize the time and effort

WHAT IS TLM ??

==========

1. Messsage passing system
2. TLM Interface
   1. Ports- set of methods Ex. Get(), Put(), Peek() etc
   2. Exports/Imp- Implementation for the METHODS

PORT: 1.In a initiator you will have a port with port we call the method.
eg. get_port or put_port

-> Denoted by square
-> If component is initiator and sending the data then it should have put_port
-> If component is initiator and recieving the data then it should have get_port
-> Port always parametrize by single parameter which is transaction type (write_xtn)

IMPLEMENTATION PORT: 2.get(),put() methods are implemented by the implementation port inside the target

eg. get_imp or put_imp
-> put()/get() method should be implemented by imp_ports otherwise it will give error.
-> Denoted by circle(O)
-> imp_port always parametrize by two parameter which is transaction type (write_xtn) & .class name in which put/get method is implemented

EXPORT: 3. It will not implement any method

TLM FIFO: 4. Both are initiator here and Generator sending the data so it should have put_port
-> Driver recieving the data so it should have get_port
-> TLM_FIFO have put_export & get_export
-> Methods like get(), put(), peek() etc are implemented inside TLM_FIFO

The get() operation will return a transaction (if available) from the TLM FIFO and
also removes the item from the FIFO. If no items are available in the FIFO, it will block
and wait until the FIFO has at least one entry.

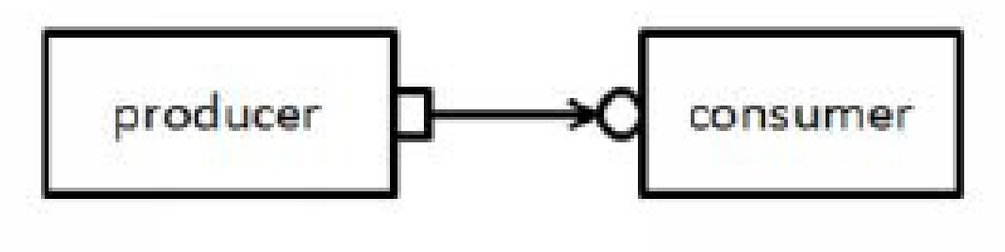
The peek() operation will return a transaction (if available) from the TLM FIFO without actually
removing the item from the FIFO. It is also a blocking call that waits if FIFO has no available entry.

ANALYSIS PORT: 4. Denoted by diamond/rhombus symbol
--> One to Many connection
--> It acts like a Broadcaster
--> Analysis port may be connected to zero, one or many analysis_export
--> Analysis port has single write() method
--> If analysis_export not implemented write() method in target then it won't
give any error

ANALYSIS FIFO: 5. Analysis fifo have analysis_imp port and write() method is implemented here

What are TLM ports and exports?

In Transaction Level Modelling, different components or modules communicate using transaction objects. A TLM port defines a set of methods (API) used for a particular connection while the actual implementation of these methods are called TLM exports. A connection between the TLM port and the export establishes a mechanism of communication between two components.



Here is a simple example of how a producer can communicate to a consumer using a simple TLM port. The producer can create a transaction and “put” to the TLM port, while the implementation of “put” method which is also called TLM export would be in the consumer that reads the transaction created by producer, thus establishing a channel of communication.

What are TLM FIFOs?

A TLM FIFO is used for Transactional communication if both the producing component and the consuming component need to operate independently. In this case (as shown below), the producing component generates transactions and "puts" into FIFO, while the consuming component gets one transaction at a time from the FIFO and processes it.

What is the difference between a get() and peek() operation on a TLM fifo?

The **get**() operation will return a transaction (if available) from the TLM FIFO and also removes the item from the FIFO. If no items are available in the FIFO, it will block and wait until the FIFO has at least one entry.

The **peek**() operation will return a transaction (if available) from the TLM FIFO without actually removing the item from the FIFO. It is also a blocking call which waits if FIFO has no available entry.

What is the difference between analysis ports and TLM ports? And what is the difference between analysis FIFOs and TLM FIFOs? Where are the analysis ports/FIFOs used?

The TLM ports/FIFOs are used for transaction level communication between two components that have a communication channel established using put/get methods.

Analysis ports/FIFOs are another transactional communication channel which are meant for a component to distribute (or broadcast) transaction to more than one component.

TLM ports/FIFOs are used for connection between driver and sequencer while analysis ports/FIFOs are used by monitor to broadcast transactions which can be received by scoreboard or coverage collecting components.

SETTING CONFIGURATION

Configuration is used to configure our testbench, to configure agents ,environment etc..

We.use set().and get() method is used to configure these methods are defined in uvm_config_db class.and of static.type.

set method will create a database in which the variable which is last argument stored in that.
set() performs write operation in associative array.
get() performs read operation in associative array.

CONFIGURATION DATABASE is implemented using associative array where
starting three arguments works as INDEX(address) which is of string.type.for associative array in which configured variable is stored

What is uvm_config_db and what is it used for?

The UVM configuration mechanism supports sharing of configurations and parameters across different testbench components. This is enabled using a configuration database called uvm_config_db. Any testbench component can populate the configuration database with variables, parameters, object handles etc. Other testbench components can get access to these variables, parameters, object handles from the configuration database without really knowing where it exists in the hierarchy.

For Example, a top level testbench module can store a virtual interface pointer to the uvm_config_db . Then any uvm_driver or a uvm_monitor components can query the uvm_config_db to get handle to this virtual interface and use it for actually accessing the signals through the interface.

How do we use the get() and set() methods of uvm_config_db?

The get() and set() are the primary methods used to populate or retrieve information from the uvm_config_db. Any verification component can use the set() method to populate the config_db with some configuration information and can also control which other components will have visibility to same information. It could be set to have global visibility or visible only to one or more specific testbench components. The get() function checks for a shared configuration from the database matching the parameters.

The syntax for the get() and set() methods are as follows:

uvm_config_db#(<type>)::set(uvm_component context, string inst_name, string field_name,<type> value)

uvm_config_db#(<type>)::get(uvm_component context, string inst_name, string field_name, ref value)

SEQUENCE

What is a subsequence?

A subsequence is asequence that is started from another sequence. From the body() of a sequence, if start() of another sequence is called, it is generally called a subsequence.

What is the difference between get_next_item() and try_next_item() methods in UVM driver class?

The get_next_item() is a blocking call (part of the driver-sequencer API) which blocks until a sequence item is available for driver to process, and returns a pointer to the sequence item.

The try_next_item() is a nonblocking version which returns a null pointer if no sequence item is available for driver to process.

What is m_sequencer handle?

When a sequence is started, it is always associated with a sequencer on which it is started.
The m_sequencer handle contains the reference to the sequencer on which sequence is running. Using this handle, the sequence can access any information and other resource handles in the UVM component hierarchy.

What is a p_sequencer handle and how is it different in m_sequencer?

A UVM sequence is an object with limited life time unlike a sequencer or a driver or a monitor which are UVM components and are present throughout simulation time. So if you need to access any members or handles from the testbench hierarchy (component hierarchy), the sequence would need a handle to the sequencer on which it is running.
m_sequencer is a handle of type uvm_sequencer_base which is available by-default in a uvm_sequence. However, to access the real sequencer on which sequence is running, we need to typecast the m_sequencer to the real sequencers, which is generally called p_sequencer (though you could really use any name and not just p_sequencer).

Here is a simple example where a sequence wants to access a handle to a clock monitor component which is available as a handle in the sequencer.

```
class test_sequence_c extends uvm_sequence;
test_sequencer_c p_sequencer;
clock_monitor_c my_clock_monitor;

task pre_body();
if(!$cast(p_sequencer, m_sequencer)) begin
`uvm_fatal("Sequencer Type Mismatch:", " Wrong Sequencer");
end
my_clock_monitor = p_sequencer.clk_monitor;
endtask
endclass
class test_Sequencer_c extends uvm_sequencer;
clock_monitor_c clk_monitor;
endclass
```

DRIVER AND SEQUENCE HANDSHAKING

Explain the protocol handshake between a sequencer and driver?

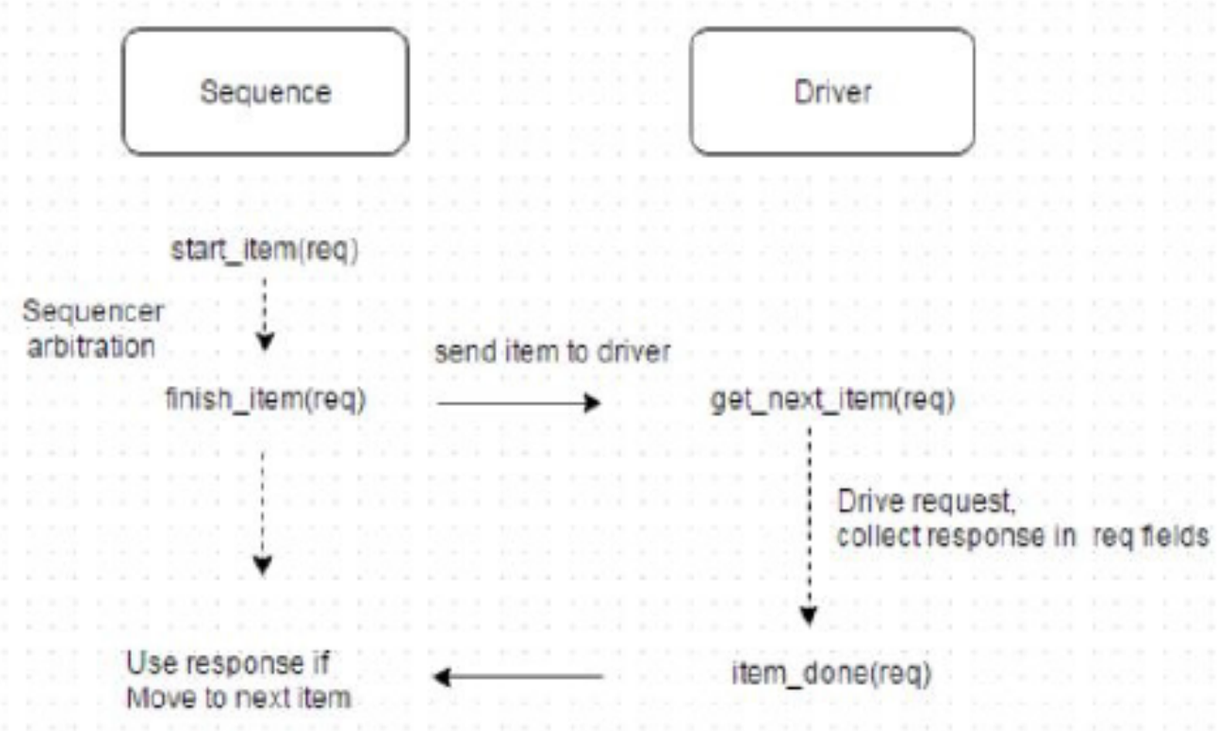
On the sequence side, there are two methods as follows:

1. start_item(<item>): This requests the sequencer to have access to the driver for the sequence item and returns when the sequencer grants access.
2. finish_item(<item>): This method results in the driver receiving the sequence item and is a blocking method which returns only after driver calls the item_done() method.

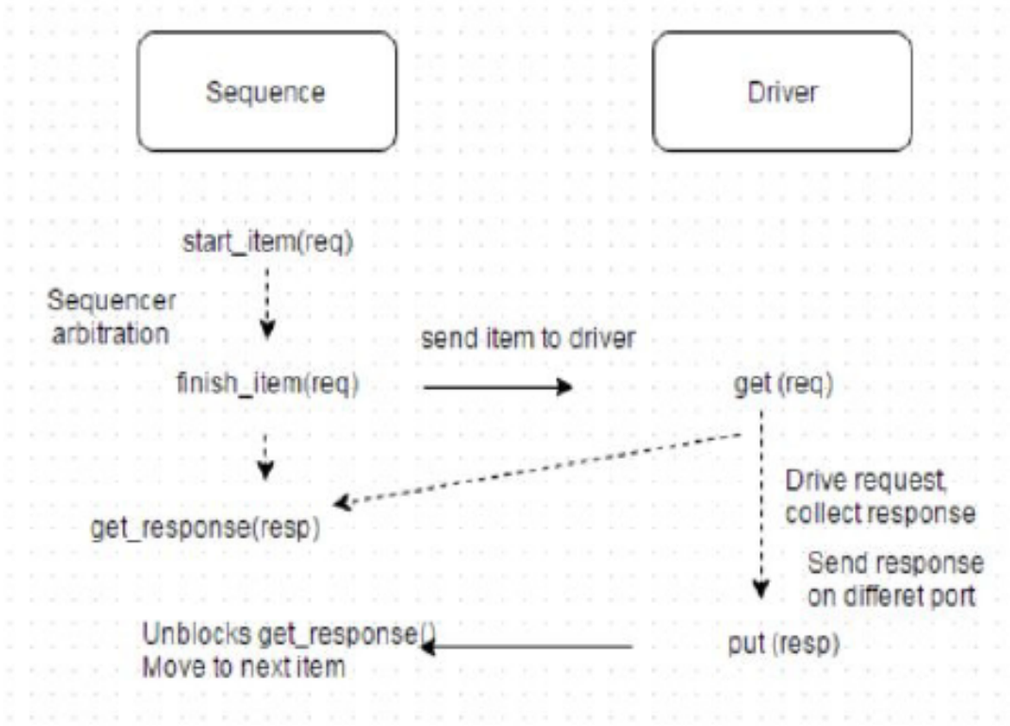
On the driver side,

1. get_next_item(req) : This is a blocking method in driver that blocks until a sequence item is received on the port connected to sequencer. This method returns the sequence item which can be translated to pin level protocol by the driver.
2. item_done(req): The driver uses this nonblocking call to signal to the sequencer that it can unblock the sequences finish_item() method, either when the driver accepts the sequences request or it has executed it.

Following diagram illustrates this protocol handshake between sequencer and driver which is the most commonly used handshake to transfer requests and responses between sequence and driver.



Few other alternatives methods are: get() method in driver which is equivalent to calling get_next_item() along with item_done(). Sometimes,there would also be need for a separate response port if the response from driver to sequence contains more information than what could be encapsulated in the request class. In this case, sequencer will use a get_response() blocking method that gets unblocked when the driver sends a separate response on this port using put() method. This is illustrated in below diagram.



task drive();

//drive the DUT i/p
//collect DUT o/p

EXAMPLE

rsp.data_out=vif.cb.data_out;

endtask

class driver extends uvm_driver#(write_xtn);

task run_phase(uvm_phase phase);
forever
begin
seq_item_port.get_next_item(req);
drive(req);
seq_item_port.item_done();
// seq_item_port.put_response(rsp); // this response send back to the sequence using
// rsp_port.write(rsp); //item_done,put_response,rsp_port.write

end
endtask

endclass
/*this response stored in response fifo whose depth is 8.
and sequence will get the response by calling the method get_response();

if we are using put_response(rsp) then in sequence it is must to get response using get_response(rsp) method otherwise
sequence will not initiate next transaction*/

class wr_seq extends uvm_sequence#(write_xtn);
//uvm_sequencer_base m_sequencer;

`uvm_object_utils(wr_seq)

function new (string "name");

endfunction

task body();

//repeat(10)
//begin
req=write_xtn::type_id::create("req");
//repeat(10)
begin
start_item(req); //wait for driver request (get_next_item)
req.randomize();
finish_item(req); //give req(data) to driver and wait for acknowledgment through item_done
end
endtask

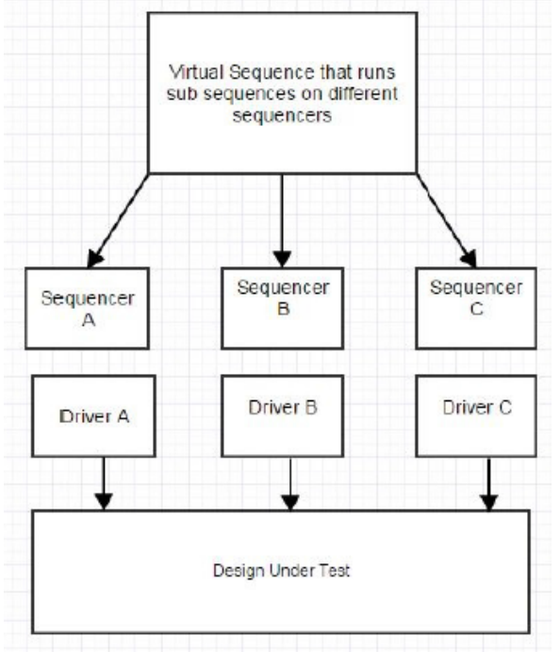
get_response(rsp);

//if the next transaction object depend on data_out of DUT ,
if(rsp.data_out == .....(some logic)) /*here we can decide how data should we generated bassed on DUT response*/
begin
start_item(req); /*wait for driver request (get_next_item)*/
req.randomize()with {.....}; /*inline constraint*/
finish_item(req); /*give req(data) to driver and wait for acknowledgment through item_done*/
end
endclass : wr_seq

What is a virtual sequence and where do we use a virtual sequence? What are its benefits?

A virtual sequence is a sequence which controls stimulus generation across multiplensequencers. Since sequences, sequencers and drivers are focused on single interfaces, almost all testbenches require a virtual sequence to co-ordinate the stimulus and the interactions across different interfaces. Virtual sequences are also useful at a Subsystem or System level testbenches to have unit level sequences exercised in a co-ordinated fashion.

Following diagram shows this conceptually where a Virtual sequence has handles to three sequencers which connect to drivers for three separate interface to DUT. The virtual sequence can then generate sub sequences on each of the interfaces and run them on the corresponding sub-sequencer.



REGISTER ABSTRACTION LAYER

UVM RAL (Register Abstraction Layer) is a feature supported in UVM that helps in verifying the registers in a design as well as in configuration of DUT using an abstract register model.

The UVM register model provides a way of tracking the register content of a DUT and a convenience layer for accessing register and memory locations within the DUT. The register model abstraction reflects the structure of the design specification for registers which is a common reference for hardware and software engineers working on the design.

Some other features of RAL include support for both front door and back door initialization of registers and built in functional coverage support.

CALLBACKs

The uvm_callback class is a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class.
Typically, the component developer defines an application-specific callback class that extends from this class and defines one or more virtual methods, called as callback interface. The methods are used to implement overriding of the component class behavior.

One common usage can be to inject an error into a generated packet before the driver sends it to DUT.
Following pseudo code shows how this can be implemented.
1) Define the packet class with an error bit
2) Define the driver class that receives this packet from a sequence and sends it to
DUT
3) Define a driver callback class derived from the base uvm_callback class and
add a virtual method which can be used to inject an error or flip a bit in the packet.
4) Register the callback class using `uvm_register_cb() macro
5) In the run() method of driver that receives and send the packet to the DUT,
based on a probability knob, execute the callback to cause a packet corruption

```
class Packet_c;
byte[4] src_addr, dst_addr;
byte[] data;
byte[4] crc;
endclass
//User defined callback class extended from base class
class PktDriver_Cb extends uvm_callback;
function new (string name = “PktDriver_Cb”);
super.new(name);
endfunction
virtual task corrupt_packet (Packet_c pkt);
//Implement how to corrupt packet
//example - flip one bit of byte 0 in CRC
pkt.crc[0][0] = ~pkt.crc[0][0]
endtask
endclass : PktDriver_Cb
//Main Driver Class
class PktDriver extends uvm_component;
`uvm_component_utils(PktDriver)
//Register callback class with driver
`uvm_register_cb(PktDriver,PktDriver_Cb)
function new (string name, uvm_component parent=null);
super.new(name,parent);
endfunction
virtual task run();
forever begin
seq_item_port.get_next_item(pkt);
`uvm_do_callbacks(PktDriver,PktDriver_Cb, corrupt_packet(pkt))
//other code to derive to DUT etc
end
endtask
endclass
```

Enjoy The Content?

If this post useful for your business,
please like, share, comment and save.

Thank you