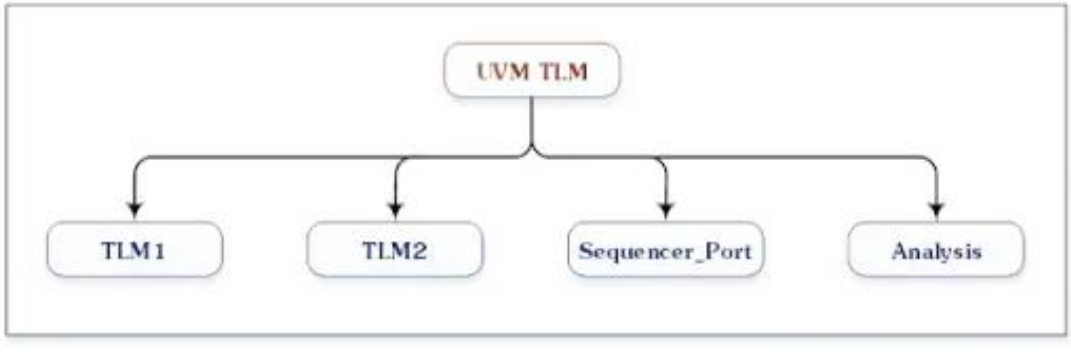
Lab 11: UVM

UVM Transaction level modeling (UVM TLM)

Transaction level modeling Is a modeling style for building more abstract models of components and systems. In UVM, data is represented as transactions that flow in and out of different components via special ports called TLM interfaces. TLM Provides a higher level of abstraction which is very much required due to the large number of signals associated with different protocols. UVM provides a set of transaction level communication interfaces to connect between components through the TLM library. The main advantages of TLM, are that it isolates a component from the changes in other components, promotes reusability and flexibility, and eases the debugging process.

UVM provides TLM library with transaction-level interfaces, ports, exports, imp ports, and analysis ports. All these TLM elements are required to send a transaction, receive a transaction, and transport from one component to another with a specific role. Each TLM Interface consists of methods for sending and receiving the transaction. The TLM1 ports provide blocking and non-blocking pass-by-value transaction-level interfaces. While the TLM2 sockets provide blocking and non-blocking transaction-level interfaces with well-defined completion semantics. For the Sequencer Port, there is a push or pull port, with well-defined completion semantics. Furthermore, for the analysis interface, it is used to perform non-blocking broadcasts of transactions to connected components.



1. UVM TLM Port

TLM Port is used to send the transactions. It has unidirectional and bidirectional ports. As a port can be connected to any compatible port, export, or imp port. A constructor method is used for the creation of the TLM Port.

```
function new (string name, uvm_component parent, int min_size=1, int max_size=1);
```

```
uvm_*_port #(T)          //unidirectional port class
uvm_*_port #(REQ,RSP) //bidirectional port class
```

Type parameters,

- T – The type of transaction to be communicated by the port, type T is not restricted to class handles and may be a value type such as int, enum, struct or similar
- REQ – The type of request transaction to be communicated by the port
- RSP – The type of response transaction to be communicated by the port

- **UVM TLM Put Port**

Any component can *send* a transaction to another component through the TLM port. The receiving component should define an implementation of that port. The implementation gives the receiver the chance to define what has to be done with the incoming transaction. For blocking the TLM port the put method will block execution in the sender until the receiver accepts the object. A uvm_blocking_put_port would stall the sender from resuming until the put task returns. Similarly, UVM TLM also has a non-blocking port uvm_nonblocking_put_port where the sender has to use try_put to see if the *put* was successful or can_put method to see if the receiver is ready to accept a transfer. Like before, the UVM TLM non-blocking_put_port should be connected to a non-blocking put implementation port.

- **UVM TLM Get Port**

Any UVM component can request to receive a transaction from another component through the TLM get port. The sending component should define an implementation of the get port. The implementation gives the sender the chance to define what needs to be sent to the requestor. This is just the opposite of a put port mentioned above. The TLM Get port can be either blocking or nonblocking, which will decide whether it's get method will block execution in the receiver until the sender sends the object. A uvm_blocking_get_port is blocking in nature where the receiver gets stalled until the sender finishes with the get task. Similarly, UVM TLM also has a non-blocking port of type uvm_nonblocking_get_port where the sender has to use try_get to see if the get was successful or can_get method to see if the sender is ready to start a transfer. Like before, the UVM TLM non-blocking get port should be connected to a non-blocking get implementation port.

2. UVM TLM Export

TLM Export is a port that forwards a transaction from a child component to its parent. It has unidirectional and bidirectional ports. As an export can be connected to any compatible child export or imp port. A constructor method is used for the creation of TLM Export.

```
function new (string name, uvm_component parent, int min_size=1, int max_size=1);
```

```
uvm_*_export#(T)         //unidirectional export class
uvm_*_export #(REQ,RSP) //bidirectional export class
```

Type parameters,

- T – The type of transaction to be communicated by the export
- REQ – The type of request transaction to be communicated by the export
- RSP – The type of response transaction to be communicated by the export

3. UVM TLM Imp port

TLM Imp Port is used to receive the transactions at the destination. It has unidirectional and bidirectional ports. A constructor method is used for the creation of TLM Import.

```
function new(string name, IMP imp)
```

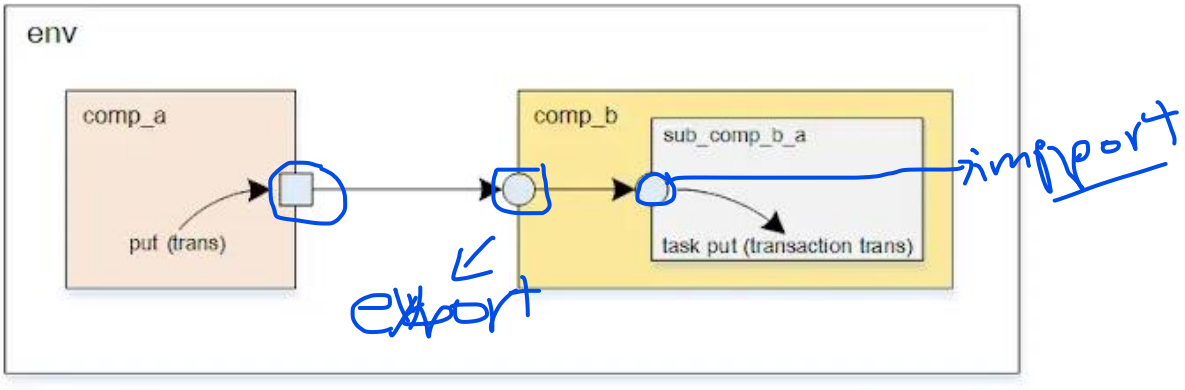
```
uvm_*_imp #(T,IMP)
uvm_*_imp #(REQ, RSP,
            IMP, REQ_IMP, RSP_IMP)
```

Type parameters,

- T – The type of transaction to be communicated by the imp
- IMP – The type of the component implementing the interface. That is the class to which this imp will delegate.
- REQ_IMP – The component type that implements the request side of the interface. Defaults to IMP. For master and slave imps only.
- RSP_IMP – The component type that implements the response side of the interface. Defaults to IMP. For master and slave imps only.

4. UVM TLM Example

In this example, a TLM port in component a is connected to a TLM imp port in subcomponent b_a through a TLM export.




```
1   class component_a extends uvm_component;
2     //Step-1. Declaring blocking port
3     uvm_blocking_put_port #(transaction) trans_out;
4
5     `uvm_component_utils(component_a)
6
7     function new(string name, uvm_component parent);
8       super.new(name, parent);
9       trans_out = new("trans_out", this); //Step-2. Creating the port
10    endfunction : new
11
12    virtual task run_phase(uvm_phase phase);
13      phase.raise_objection(this);
14
15      trans = transaction::type_id::create("trans", this);
16      void'(trans.randomize());  //Step-3. randomizing the transction
17      `uvm_info(get_type_name(),$sformatf(" tranaction randomized"),UVM_LOW)
18      `uvm_info(get_type_name(),$sformatf(" Printing trans,
19                                          \n %s",trans.sprint()),UVM_LOW)
20      `uvm_info(get_type_name(),$sformatf(" Before calling port put method"),UVM_LOW)
21      trans_out.put(trans);  //Step-4. Sending trans through port put method
22      `uvm_info(get_type_name(),$sformatf(" After  calling port put method"),UVM_LOW)
23
24      phase.drop_objection(this);
25    endtask : run_phase
26  endclass : component_a
```

```
1   class sub_component_b_a extends uvm_component;
2     transaction trans;
3     //Step-1. Declaring blocking imp port
4     uvm_blocking_put_imp#(transaction,sub_component_b_a) trans_in;
5     `uvm_component_utils(sub_component_b_a)
6     //----------------------------------------
7     // Constructor
8     //----------------------------------------
9     function new(string name, uvm_component parent);
10      super.new(name, parent);
11      trans_in = new("trans_in", this);  //Step-2. Creating imp port
12    endfunction : new
13    //----------------------------------------
14    // Imp port put method
15    //----------------------------------------
16    //Step-3. Implementing imp port
17    virtual task put(transaction trans);
18      `uvm_info(get_type_name(),$sformatf(" Recived trans On IMP Port"),UVM_LOW)
19      `uvm_info(get_type_name(),$sformatf(" Printing trans,
20                                          \n %s",trans.sprint()),UVM_LOW)
21    endtask
22  endclass : sub_component_b_a
23  ////////////////////////////////////////////////
24  ////////////////////////////////////////////////
```

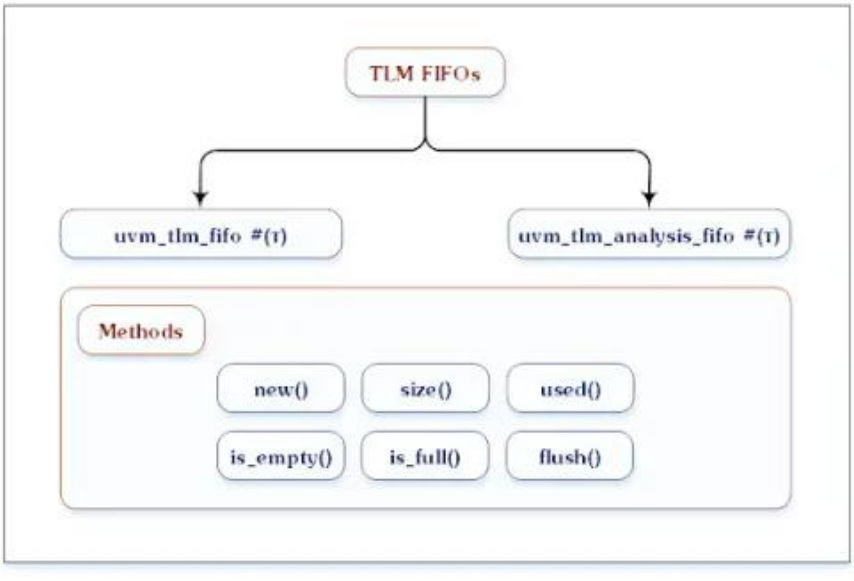
```
29  class component_b extends uvm_component;
30
31    sub_component_b_a sub_comp_b_a;
32    //Step-1. Declaring blocking export
33    uvm_blocking_put_export#(transaction) trans_in;
34    `uvm_component_utils(component_b)
35    //----------------------------------------
36    // Constructor
37    //----------------------------------------
38    function new(string name, uvm_component parent);
39      super.new(name, parent);
40      trans_in = new("trans_in", this);  //Step-2. Creating export
41    endfunction : new
42    //----------------------------------------
43    // build_phase - Create the components
44    //----------------------------------------
45    function void build_phase(uvm_phase phase);
46      super.build_phase(phase);
47      sub_comp_b_a = sub_component_b_a::type_id::create("sub_comp_b_a", this);
48    endfunction : build_phase
49    //----------------------------------------
50    // Connect_phase
51    //----------------------------------------
52    function void connect_phase(uvm_phase phase);
53      trans_in.connect(sub_comp_b_a.trans_in); //Step-3. Connecting export to the imp port
54    endfunction : connect_phase
55  endclass : component_b
```

```
//Inside the Env class
function void connect_phase(uvm_phase phase);
  comp_a.trans_out.connect(comp_b.trans_in); //Connecting port with export
endfunction : connect_phase
```

5. UVM TLM FIFO

The TLM FIFO provides storage for the transactions between two independently running processes. As we have seen put and get methods to operate with only one outstanding transaction at a time. What if case where the sender does not wait for the receiver's acknowledgment, it just wants to store it in memory and the receiver can consume it whenever required. So in this case, the sender and the receiver need not be in sync. In TLM FIFO, the sender pushes the transactions to FIFO and whenever the receiver requires a transaction it fetches it from the FIFO. Therefore, transactions are put into the FIFO via the put_export method, and fetched from the FIFO via the get_peek_export method. As it is called FIFO (First In First Out), transactions are fetched from the FIFO in the order they are put. A constructor method is used for the creation of TLM FIFO.

```
function new (string name, uvm_component parent, int size=1);
```

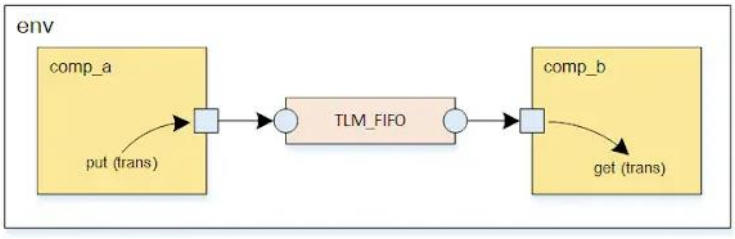


TLM FIFO Methods

- Size(): Returns the size of the FIFo, a return value of 0 indicates the FIFO capacity has no limit.
- Used(): Returns the number of entries put into the FIFO.
- Is_empty(): Returns 1 when there are no entries in the FIFO, 0 otherwise.
- Is_full(): Returns 1 when the number of entries in the FIFO is equal to its size, 0 otherwise.
- Flush(): Remove all entries from the FIFO.

6. UVM TLM FIFO Example

In this example, a UVM TLM FIFO is used to connect between a TLM put port in component_a and a TLM get port in component_b inside the env class.




```
1   class component_a extends uvm_component;
2       transaction trans;
3       //Step-1. Declaraing the put port
4       uvm_blocking_put_port#(transaction) trans_out;
5       `uvm_component_utils(component_a)
6       //---------------------------------------
7       // Constructor
8       //---------------------------------------
9       function new(string name, uvm_component parent);
10        super.new(name, parent);
11        trans_out = new("trans_out", this); //Step-2. Creating the port
12      endfunction : new
13      //---------------------------------------
14      // run_phase
15      //---------------------------------------
16      virtual task run_phase(uvm_phase phase);
17        phase.raise_objection(this);
18        trans = transaction::type_id::create("trans", this);
19        void'(trans.randomize());
20        `uvm_info(get_type_name(),$sformatf(" tranaction randomized"),UVM_LOW)
21        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
22                                            trans.sprint()),UVM_LOW)
23        //Step-3) Calling put method to push transaction to TLM FIFO
24        `uvm_info(get_type_name(),$sformatf(" Before calling port put method"),UVM_LOW)
25        trans_out.put(trans);
26        `uvm_info(get_type_name(),$sformatf(" After  calling port put method"),UVM_LOW)
27        phase.drop_objection(this);
28      endtask : run_phase
29    endclass : component_a
```

```
1   class component_b extends uvm_component;
2       transaction trans;
3       //Step-1. Declaraing the get port
4       uvm_blocking_get_port#(transaction) trans_in;
5       `uvm_component_utils(component_b)
6
7       function new(string name, uvm_component parent);
8         super.new(name, parent);
9         trans_in = new("trans_in", this);  //Step-2. Create the port
10      endfunction : new
11
12      virtual task run_phase(uvm_phase phase);
13        phase.raise_objection(this);
14
15        `uvm_info(get_type_name(),$sformatf(" Before calling port get method"),UVM_LOW)
16        trans_in.get(trans);  //Step-3. Get the transaction from TLM FIFO
17        `uvm_info(get_type_name(),$sformatf(" After  calling port get method"),UVM_LOW)
18        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
19                                            trans.sprint()),UVM_LOW)
20        phase.drop_objection(this);
21      endtask : run_phase
22    endclass : component_b
```

```
1   class environment extends uvm_env;
2       //----------------------------------------
3       // Components Instantiation
4       //----------------------------------------
5       component_a comp_a;
6       component_b comp_b;
7       //Step-1. Declaring the TLM FIFO
8       uvm_tlm_fifo #(transaction) fifo_ab;
9
10      `uvm_component_utils(environment)
11
12      //----------------------------------------
13      // Constructor
14      //----------------------------------------
15      function new(string name, uvm_component parent);
16        super.new(name, parent);
17      endfunction : new
18      //----------------------------------------
19      // build_phase - Create the components
20      //----------------------------------------
21      function void build_phase(uvm_phase phase);
22        super.build_phase(phase);
23        comp_a = component_a::type_id::create("comp_a", this);
24        comp_b = component_b::type_id::create("comp_b", this);
25
26        //Step-2. Creating the FIFO
27        fifo_ab = new("fifo_ab", this);
28      endfunction : build_phase
```

```
29      //----------------------------------------
30      // Connect_phase
31      //----------------------------------------
32      function void connect_phase(uvm_phase phase);
33
34        //Step-3. Connecting FIFO put_export with producer port
35        comp_a.trans_out.connect(fifo_ab.put_export);
36        //Step-4. Connecting FIFO get_export with consumer port
37        comp_b.trans_in.connect(fifo_ab.get_export);
38      endfunction : connect_phase
39    endclass : environment
```

7. UVM TLM Analysis

The main difference between a TLM port and an Analysis port is that an analysis port supports one-to-many connections, while TLM ports support only one-to-one connections.

1. Analysis Ports

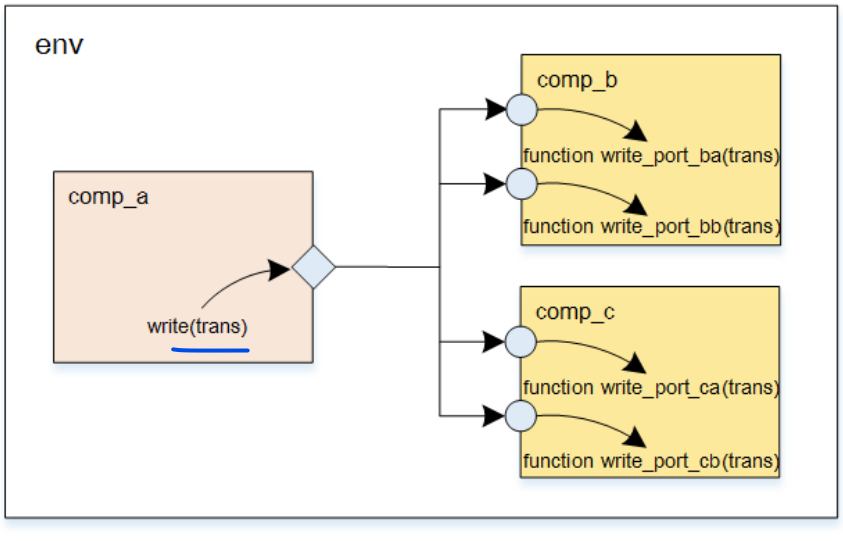
Analysis ports support one to many connections (uvm_analysis_port), that could be used by coverage collectors and scoreboards. It also contains a list of analysis_exports that are connected to it. Such that, when a component calls analysis_port.write(), the analysis_port cycles through the list and calls the write() method of each connected export.

2. Analysis FIFOs

They are extended from TLM FIFO. As it provides an "implementation" style export. And the implementation of "write" method that is responsible for placing the date into a FIFO.

8. UVM TLM Analysis Example

In this example, a TLM Analysis port in component_a is connected to multiple Analysis imp ports in component_b and component_c.



```
1   class component_a extends uvm_component;
2
3       transaction trans;
4       //Step-1. Declaring analysis port
5       uvm_analysis_port#(transaction) analysis_port;
6
7       `uvm_component_utils(component_a)
8
9       //----------------------------------------
10      // Constructor
11      //----------------------------------------
12      function new(string name, uvm_component parent);
13        super.new(name, parent);
14
15        //Step-2. Creating analysis port
16        analysis_port = new("analysis_port", this);
17      endfunction : new
```

```
18      //----------------------------------------
19      // run_phase
20      //----------------------------------------
21      virtual task run_phase(uvm_phase phase);
22        phase.raise_objection(this);
23
24        trans = transaction::type_id::create("trans", this);
25        void'(trans.randomize());
26        `uvm_info(get_type_name(),$sformatf(" tranaction randomized"),UVM_LOW)
27        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
28                                            trans.sprint()),UVM_LOW)
29
30        `uvm_info(get_type_name(),$sformatf(" Before calling port write method"),UVM_LOW)
31        //Ste-3. Calling write method
32        analysis_port.write(trans);
33        `uvm_info(get_type_name(),$sformatf(" After  calling port write method"),UVM_LOW)
34
35        phase.drop_objection(this);
36      endtask : run_phase
37    endclass : component_a
```

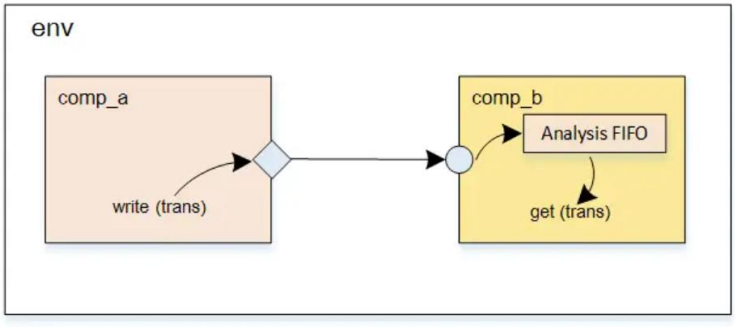
```
1  //Step-1. Define analysis imp ports
2  `uvm_analysis_imp_decl(_port_ba)
3  `uvm_analysis_imp_decl(_port_bb)
4  class component_b extends uvm_component;
5
6    transaction trans;
7    //Step-2. Declare the analysis imp ports
8    uvm_analysis_imp_port_ba #(transaction,component_b) analysis_imp_a;
9    uvm_analysis_imp_port_bb #(transaction,component_b) analysis_imp_b;
10
11   `uvm_component_utils(component_b)
12
13   //---------------------------------------
14   // Constructor
15   //---------------------------------------
16   function new(string name, uvm_component parent);
17     super.new(name, parent);
18     //Step-3. Create the analysis imp ports
19     analysis_imp_a = new("analysis_imp_a", this);
20     analysis_imp_b = new("analysis_imp_b", this);
21   endfunction : new
22   //Step-4. Implement the write method write_port_ba
23   //---------------------------------------
24   // Analysis port write method
25   //---------------------------------------
26   virtual function void write_port_ba(transaction trans);
27     `uvm_info(get_type_name(),$sformatf(" Inside write_port_ba method.
28                                 Received trans On Analysis Imp Port"),UVM_LOW)
29     `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
30                                         trans.sprint()),UVM_LOW)
31   endfunction
32
33   //Step-4. Implement the write method write_port_bb
34   //---------------------------------------
35   // Analysis port write method
36   //---------------------------------------
37   virtual function void write_port_bb(transaction trans);
38     `uvm_info(get_type_name(),$sformatf(" Inside write_port_bb method.
39                                 Received trans On Analysis Imp Port"),UVM_LOW)
40     `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
41                                         trans.sprint()),UVM_LOW)
42   endfunction
43
44 endclass : component_b
```

```
1   //Step-1. Define analysis imp ports
2     `uvm_analysis_imp_decl(_port_ca)
3     `uvm_analysis_imp_decl(_port_cb)
4     class component_c extends uvm_component;
5
6       transaction trans;
7       //Step-2. Declare the analysis imp ports
8       uvm_analysis_imp_port_ca #(transaction,component_c) analysis_imp_a;
9       uvm_analysis_imp_port_cb #(transaction,component_c) analysis_imp_b;
10
11      `uvm_component_utils(component_c)
12
13      //---------------------------------------
14      // Constructor
15      //---------------------------------------
16      function new(string name, uvm_component parent);
17        super.new(name, parent);
18        //Step-3. Create the analysis imp ports
19        analysis_imp_a = new("analysis_imp_a", this);
20        analysis_imp_b = new("analysis_imp_b", this);
21      endfunction : new
```

```
22      //Step-4. Implement the write method write_port_ca
23      //---------------------------------------
24      // Analysis port write method
25      //---------------------------------------
26      virtual function void write_port_ca(transaction trans);
27        `uvm_info(get_type_name(),$sformatf(" Inside write_port_ba method.
28                                  Received trans On Analysis Imp Port"),UVM_LOW)
29        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
30                                      trans.sprint()),UVM_LOW)
31      endfunction
32
33      //Step-4. Implement the write method write_port_cb
34      //---------------------------------------
35      // Analysis port write method
36      //---------------------------------------
37      virtual function void write_port_cb(transaction trans);
38        `uvm_info(get_type_name(),$sformatf(" Inside write_port_bb method.
39                                  Received trans On Analysis Imp Port"),UVM_LOW)
40        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
41                                      trans.sprint()),UVM_LOW)
42      endfunction
43    endclass : component_c
```

```
//Inside the Env class
  function void connect_phase(uvm_phase phase);
    //Connecting analysis port to imp port
    comp_a.analysis_port.connect(comp_b.analysis_imp_a);
    comp_a.analysis_port.connect(comp_b.analysis_imp_b);
    comp_a.analysis_port.connect(comp_c.analysis_imp_a);
    comp_a.analysis_port.connect(comp_c.analysis_imp_b);
  endfunction : connect_phase
```

9. UVM TLM Analysis Example




```
1   class component_a extends uvm_component;
2
3       transaction trans;
4       //Step-1. Declaring analysis port
5       uvm_analysis_port#(transaction) analysis_port;
6
7       `uvm_component_utils(component_a)
8
9       function new(string name, uvm_component parent);
10        super.new(name, parent);
11
12        //Step-2. Creating analysis port
13        analysis_port = new("analysis_port", this);
14      endfunction : new
```

```
15      virtual task run_phase(uvm_phase phase);
16        phase.raise_objection(this);
17
18        trans = transaction::type_id::create("trans", this);
19        void'(trans.randomize());
20
21        `uvm_info(get_type_name(),$sformatf(" tranaction randomized"),UVM_LOW)
22        `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",
23                                            trans.sprint()),UVM_LOW)
24
25        `uvm_info(get_type_name(),$sformatf(" Before calling port write method"),UVM_LOW)
26        //Ste-3. Calling write method
27        analysis_port.write(trans);
28        `uvm_info(get_type_name(),$sformatf(" After  calling port write method"),UVM_LOW)
29
30        phase.drop_objection(this);
31      endtask : run_phase
32    endclass : component_a
```

```
class component_b extends uvm_component;
    transaction trans;
    //Step-1. Declaring analysis FIFO
    uvm_tlm_analysis_fifo #(transaction) analy_fifo;
    `uvm_component_utils(component_b)

    function new(string name, uvm_component parent);
      super.new(name, parent);
      //Step-2. Creating analysis FIFO
      analy_fifo = new("analy_fifo", this);
    endfunction : new

    virtual task run_phase(uvm_phase phase);
      phase.raise_objection(this);

      #100;
      `uvm_info(get_type_name(),$sformatf(" Before calling analysis fifo get method"),UVM_LOW)
      //Step.3 - Getting trans from FIFO
      analy_fifo.get(trans);
      `uvm_info(get_type_name(),$sformatf(" After  calling analysis fifo get method"),UVM_LOW)
      `uvm_info(get_type_name(),$sformatf(" Printing trans, \n %s",trans.sprint()),UVM_LOW)

      phase.drop_objection(this);
    endtask : run_phase
  endclass : component_b
```

```
//Inside the Env class
  function void connect_phase(uvm_phase phase);
    //Connecting analysis port to analysis FIFO
    comp_a.analysis_port.connect(comp_b.analy_fifo.analysis_export);
  endfunction : connect_phase
```

10.Driver Sequencer Communication

The UVM driver class contains a TLM port called uvm_seq_item_pull_port which is connected to a uvm_seq_item_pull_export in the UVM sequencer in the connect phase of a UVM agent. The driver can use TLM methods to get the next item from the sequencer when required. These methods help the driver to get a series of sequence_items from the sequencer's FIFO that contains data for the driver to drive to the DUT. Also, there is a way for the driver to communicate back with the sequence that it has finished driving the given sequence item and can request for the next item. Declaration for the TLM ports can be found in the class definitions of uvm_driver and uvm_sequencer as follows.

- A uvm_sequencer has an inbuilt TLM pull implementation port called seq_item_export which is used to connect with the driver's pull port.
- The port in uvm_driver is connected to the export in uvm_sequencer in the connect phase of the UVM component in which both the driver and sequencer are instantiated. Typically, a driver and sequencer are instantiated in a uvm_agent.

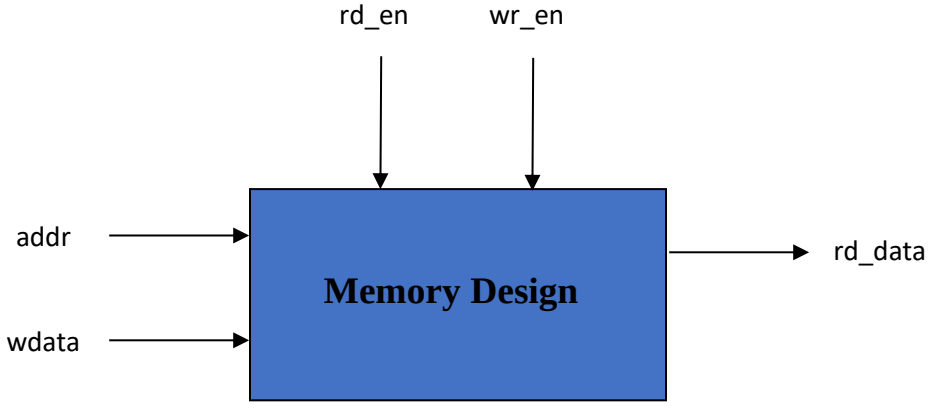
The connection between a driver and sequencer is a one-to-one connection. Multiple drivers are not connected to a sequencer nor are multiple sequencers connected to a single driver.

1. Lab Task

Part 1: Implement a communication between a UVM Monitor and a UVM scoreboard.

1) Implement a UVM monitor that sends a transaction to a UVM scoreboard that displays this transaction.
2) Connect between these UVM components using a UVM analysis port and analysis FIFO.

Part 2: Build a UVM testbench for the below RTL Verilog Mem DUT



```
module memory
  #( parameter ADDR_WIDTH = 2,
     parameter DATA_WIDTH = 8 ) (
    input clk,
    input reset,
    //control signals
    input [ADDR_WIDTH-1:0]  addr,
    input                   wr_en,
    input                   rd_en,
    //data signals
    input  [DATA_WIDTH-1:0] wdata,
    output [DATA_WIDTH-1:0] rdata
  );

  reg [DATA_WIDTH-1:0] rdata;

  //Memory
  reg [DATA_WIDTH-1:0] mem [2**ADDR_WIDTH];

  //Reset
  always @(posedge reset)
    for(int i=0;i<2**ADDR_WIDTH;i++) mem[i]=8'hFF;

  // Write data to Memory
  always @(posedge clk)
    if (wr_en)    mem[addr] <= wdata;

  // Read data from memory
  always @(posedge clk)
    if (rd_en) rdata <= mem[addr];


endmodule
```