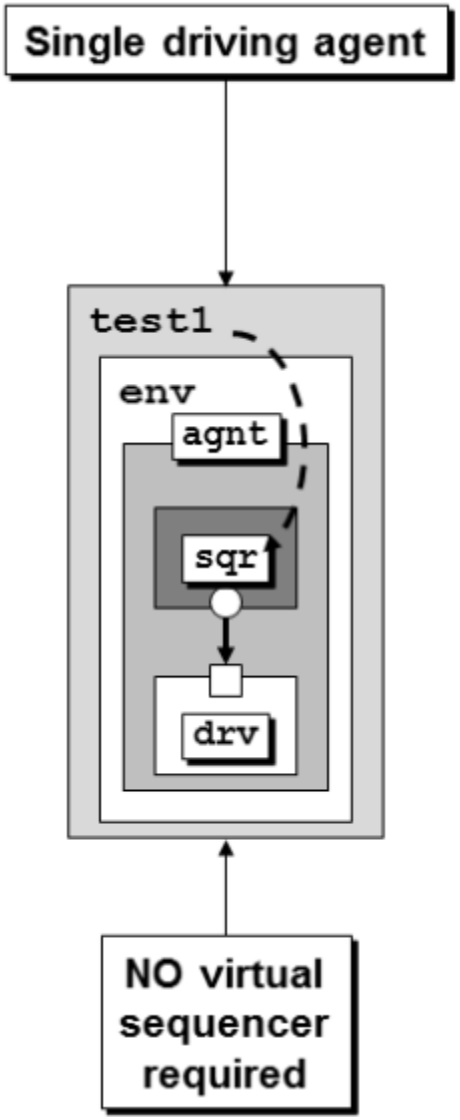
True DPRAM – UVM Testbench

For whom can know how to program

Tuan Nguyen-viet

Simple Protocol/Interface

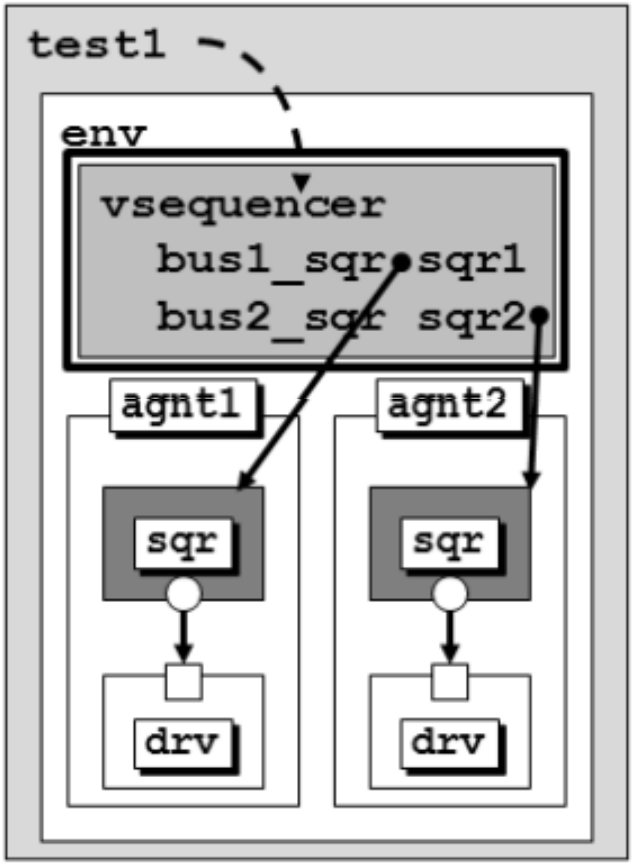
- Simple protocol/interface
  - Just one **sequencer** sending the stimulus to the **driver**.
  - **Top-level test** will use this **sequencer** to process the **sequences**.
  - May not need a **virtual sequence /a virtual sequencer**.



REF: Using UVM Virtual Sequencers & Virtual Sequences, by Clifford E. Cummings and Janick Bergeron

Two or More Interfaces/Protocols

- **DUT** is having 2 different **interface ports**,
  - There would be 2 **Agents** serving each **interface port** inside the **UVM Testbench**.
- **Virtual Sequence** will co-ordinate and synchronize the **Transactions** (**sequence items**)
  - for the 2 **Agents** to generate the simulation.
- **Virtual Sequence** acts like a Controller of the simulation data being generated for the **DUT**.
- From **top-level test**,
  - need a way to control two **sequencers**.
- ➔ using a **virtual sequencer** and **virtual sequences**.



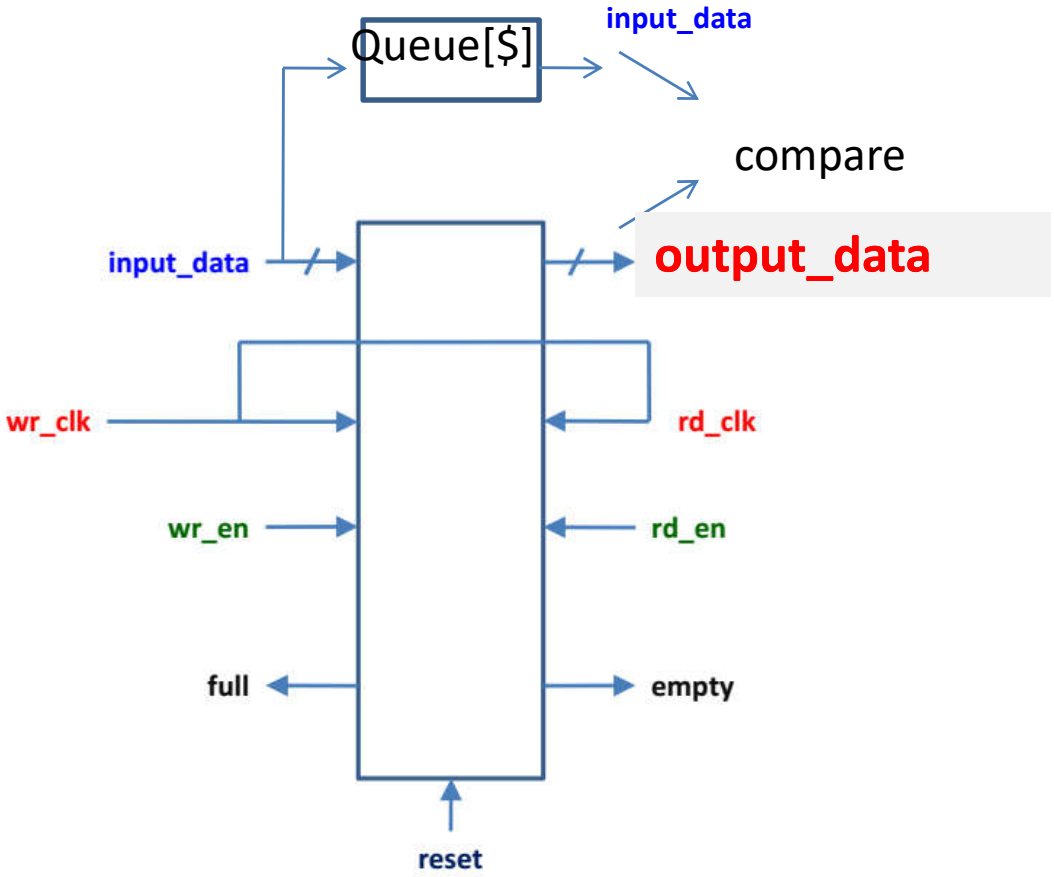
Two or More Interfaces/Protocols (2)

- In **SoC**: different modules that interact with different protocols.
  - need different **drivers** to drive corresponding **interfaces**.
  - keep separate **agents** to handle the different protocols.
  - execute **sequences** on corresponding **sequencers**.

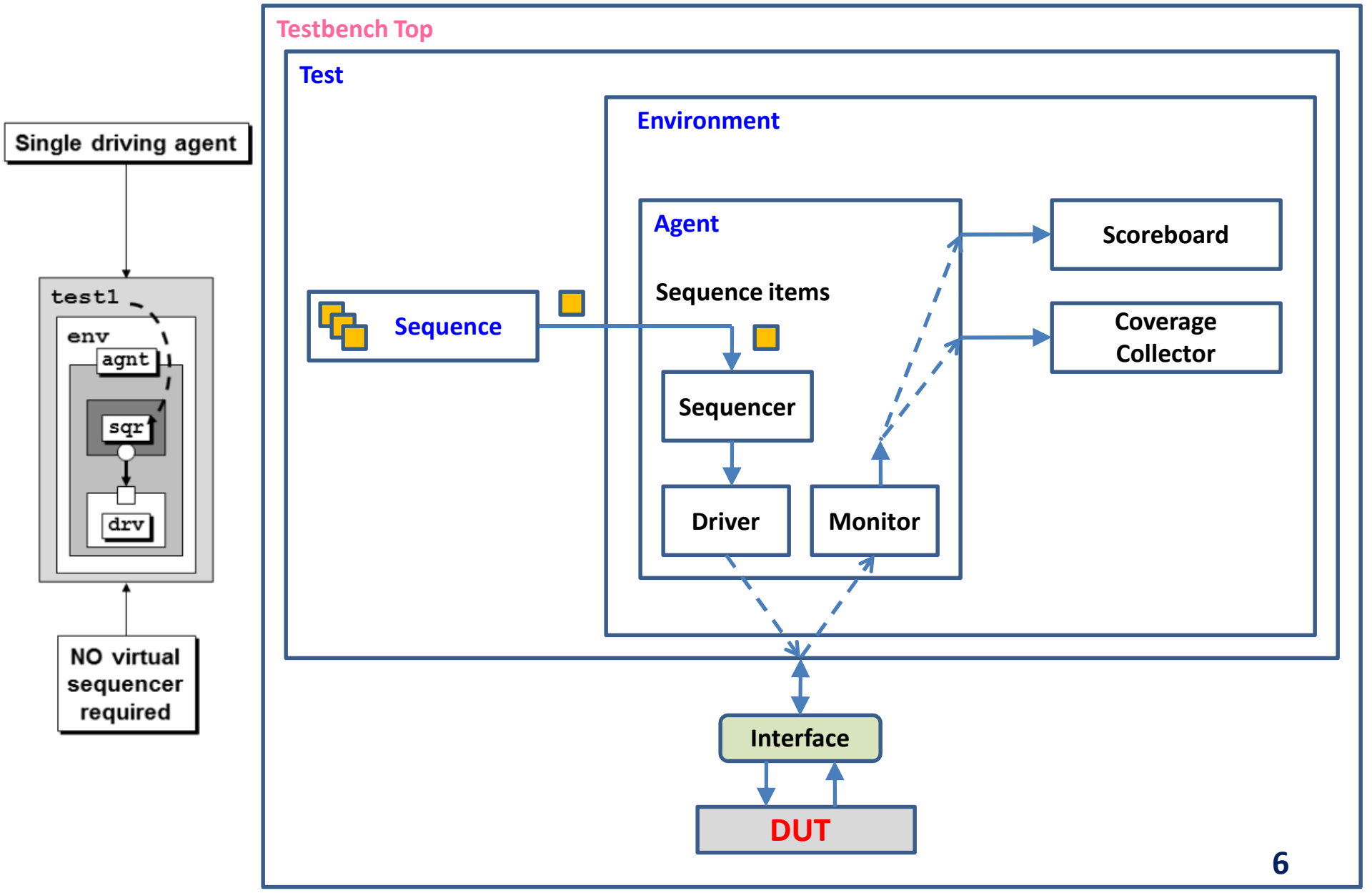
- multiple cores in **SoC**.
  - multiple cores present in **SoC** that can
    - *handle* different operations *on input* provided
    - and *respond to* the device/chip differently.
  - ➔ different **sequence** execution becomes important on different **sequencers**.

- It is recommended to use a **virtual sequencer/sequence**
  1. if we have multiple **agents**
  2. and **stimulus coordination** is required.

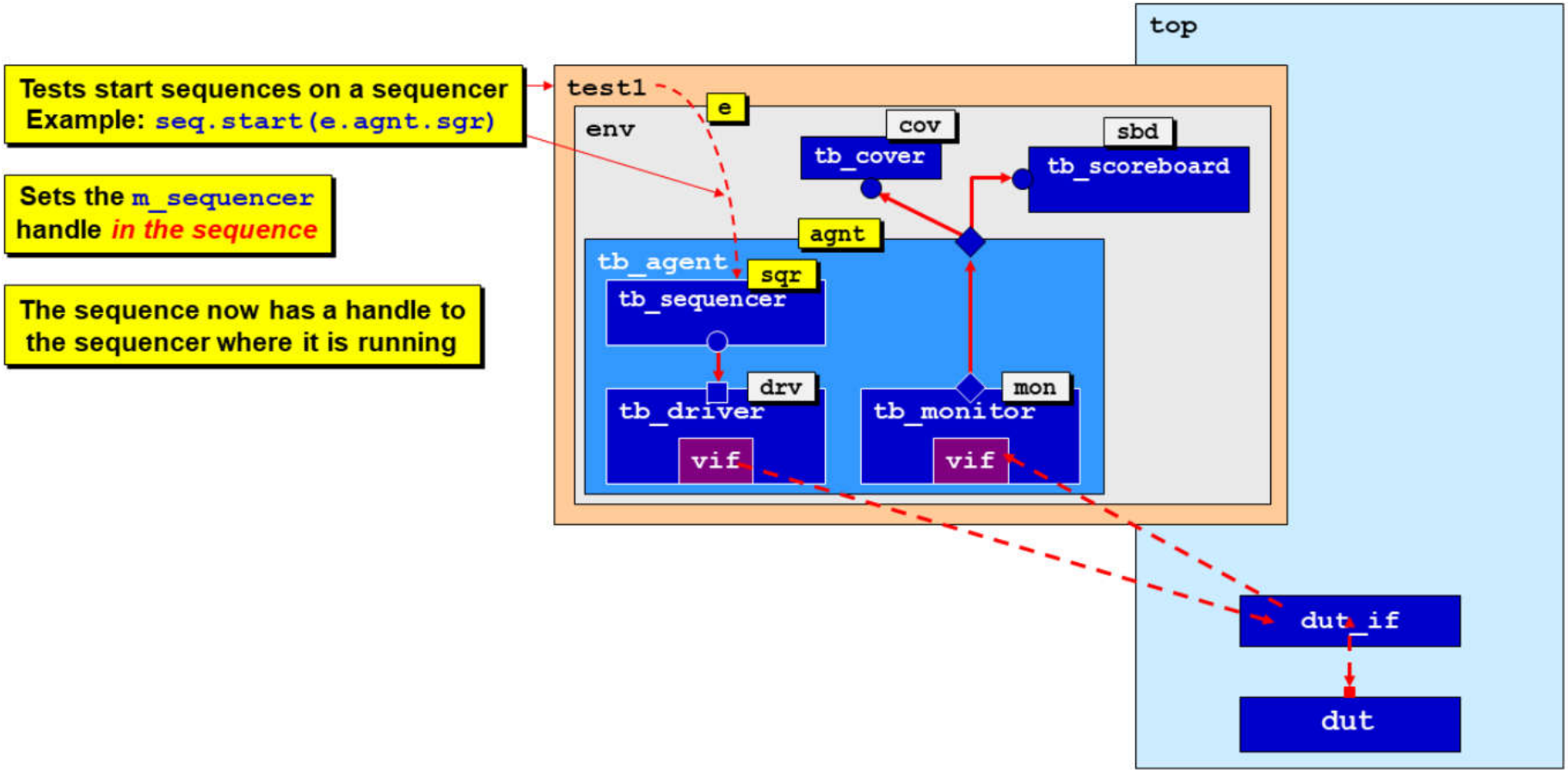
SFIFO


Queue[$]
input_data
compare
input_data
output_data
wr_clk
rd_clk
wr_en
rd_en
full
empty
reset

UVM - Simple Architecture w/ Single Agent

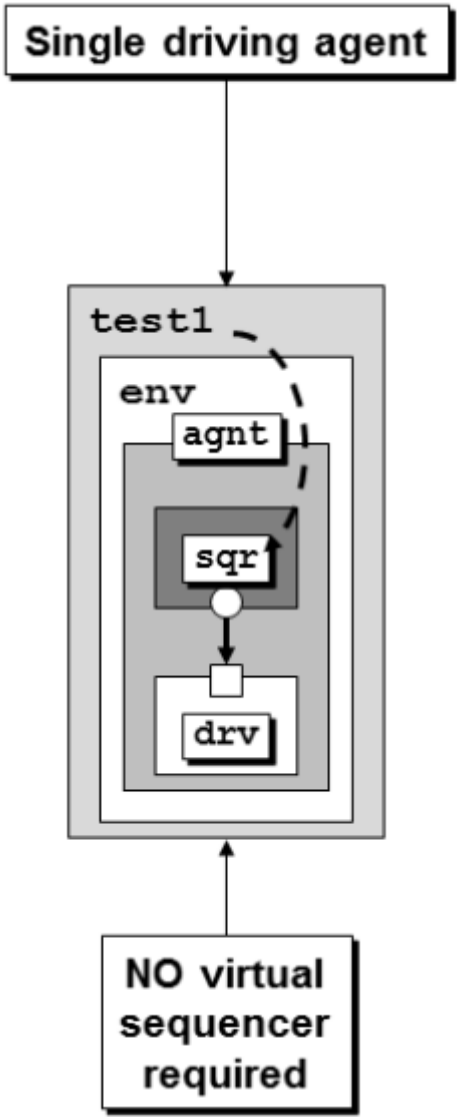


UVM - Simple Architecture w/ Single Agent (2)



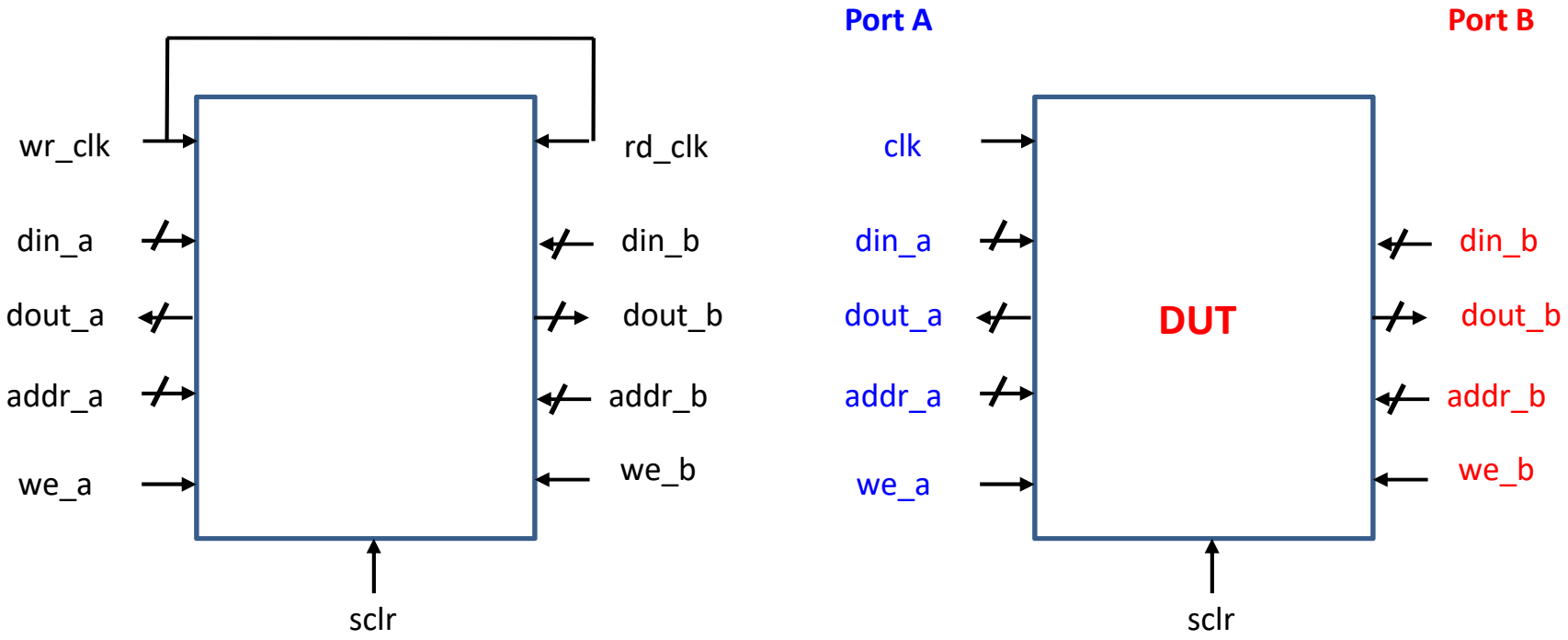
REF: Understanding the UVM m_sequencer, p_sequencer, handles, and the `uvm_declare_p_sequencer Macro, by Clifford E. Cummings

SFIFO UVM Testbench



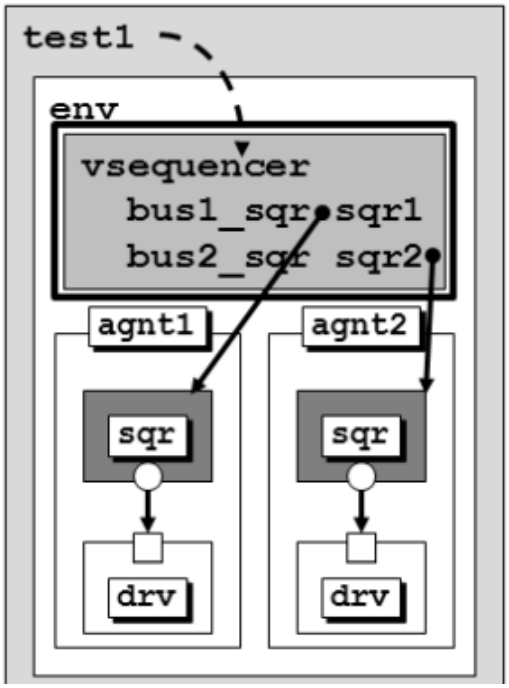

```
class sfifo_test extends uvm_test;
  sfifo_sequence f_seq;
  sfifo_environment f_env;
  `uvm_component_utils(sfifo_test)

  function new(string name = "sfifo_test", uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    f_seq = sfifo_sequence::type_id::create("f_seq", this);
    f_env = sfifo_environment::type_id::create("f_env", this);
  endfunction

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    f_seq.start(f_env.f_agt.f_seqr);
    phase.drop_objection(this);
    phase.phase_done.set_drain_time(this, 100);
  endtask

endclass
```

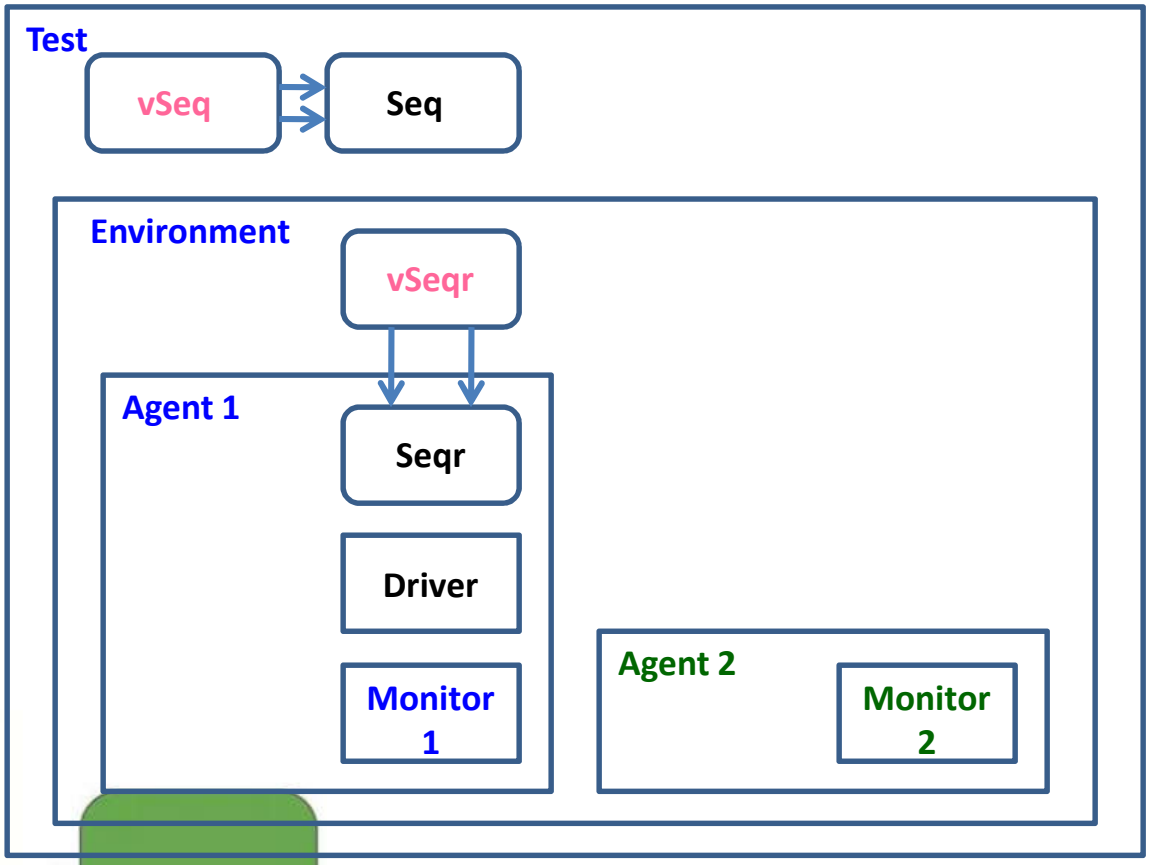
DUT – True DPRAM

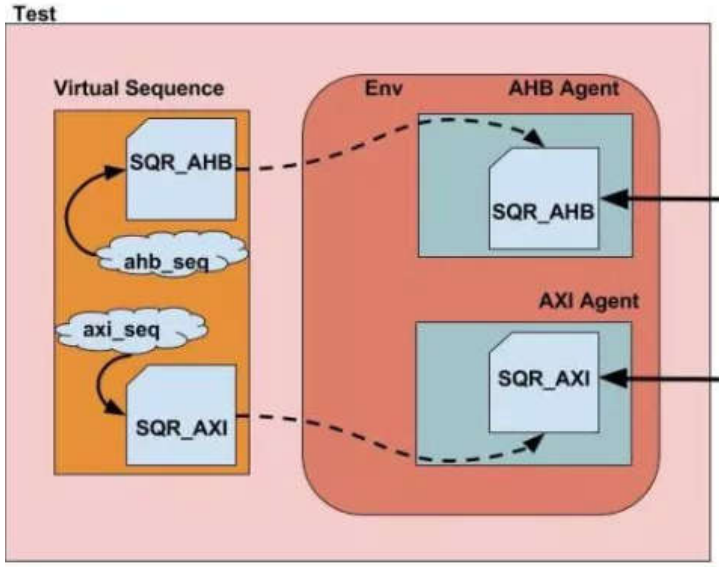


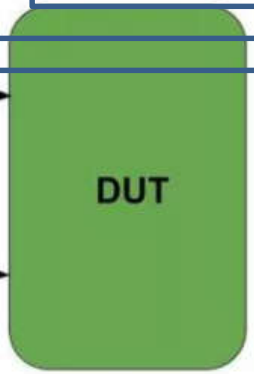
Sample: We will do a UVM Testbench for TDPRAM shown above.

UVM - Simple Architecture w/ Two Agents

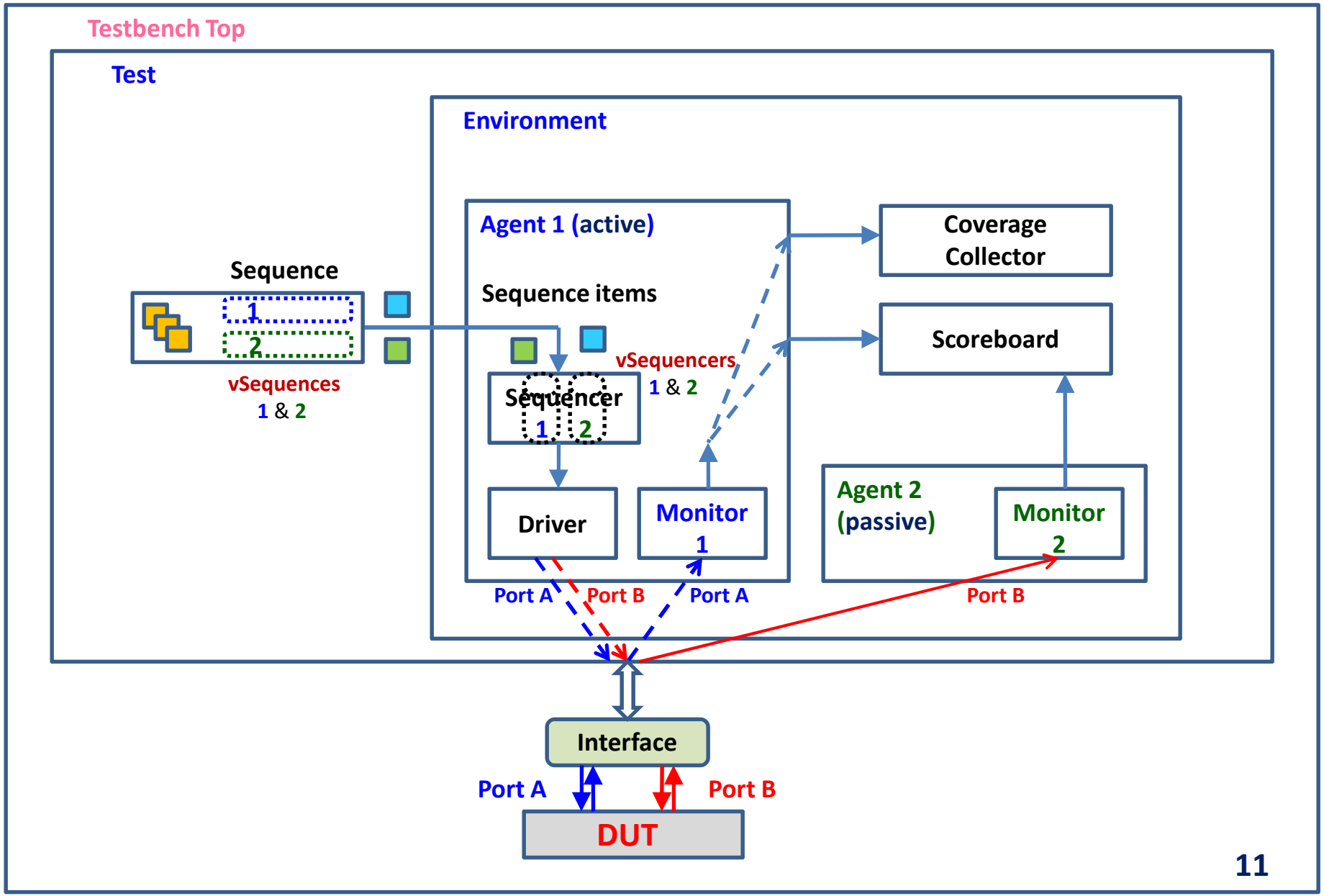






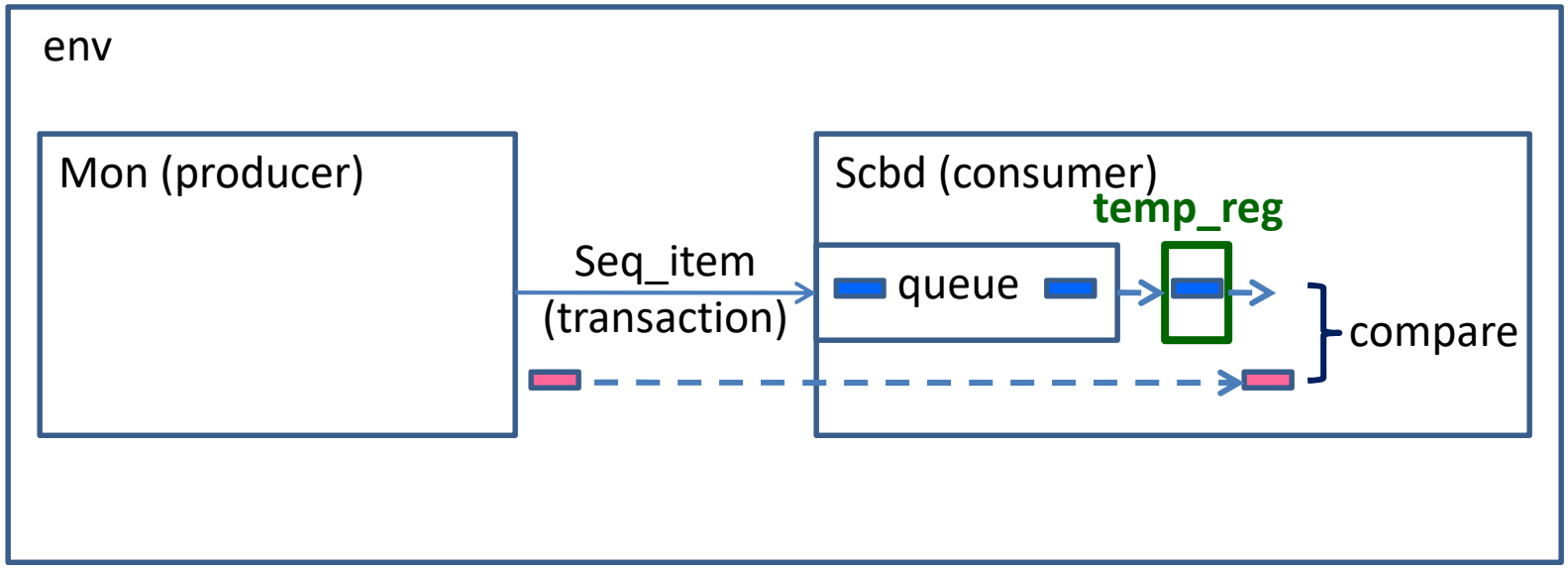


UVM - Simple Architecture w/ Two Agents (A1 and B2)

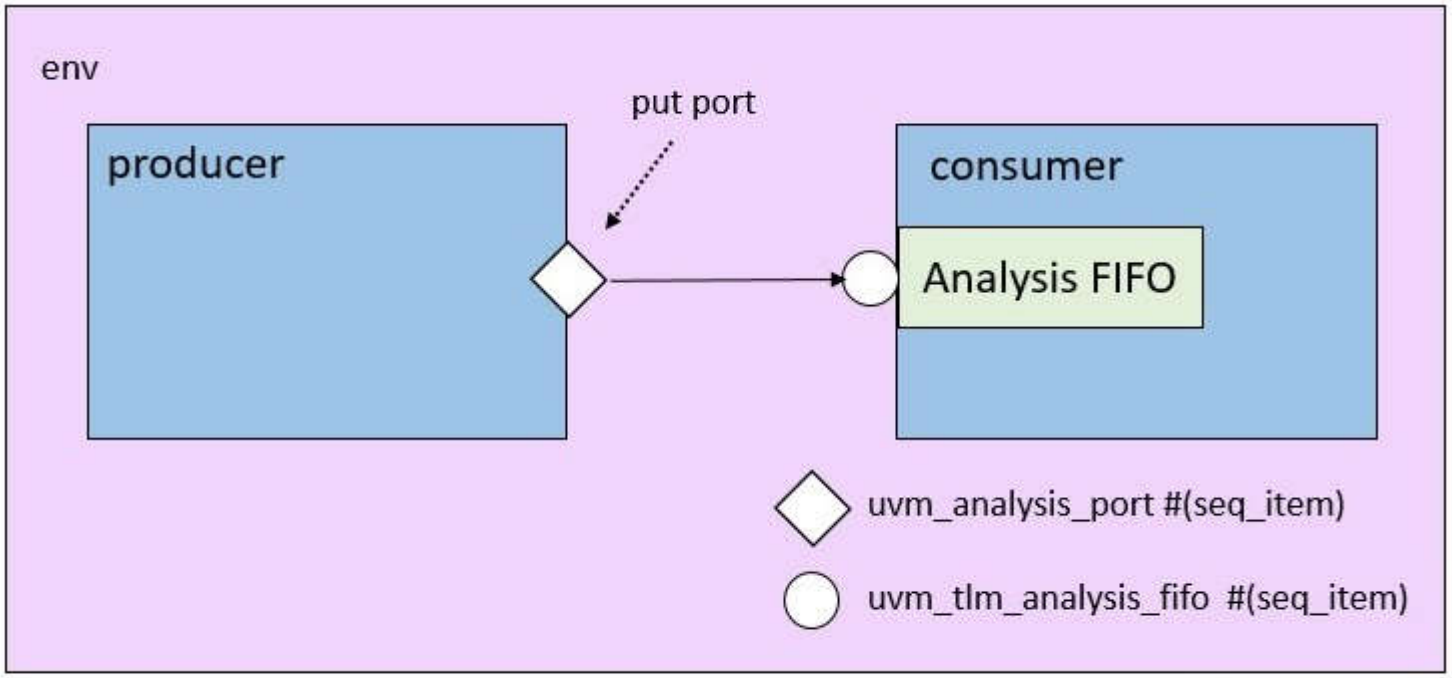


FIFO at Scoreboard

SV-Queue FIFO



TLM Analysis FIFO



uvm_tlm_analysis_fifo

Producer (e.g. Monitor)

```
class producer extends uvm_component;
  seq_item req;
  uvm_analysis_port #(seq_item) a_port;

  `uvm_component_utils(producer)

  function new(string name = "producer", uvm_component parent = null);
    super.new(name, parent);
    a_port = new("a_port", this);
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);

    repeat(10) begin
      req = seq_item::type_id::create("req");
      assert(req.randomize());
      a_port.write(req);
      `uvm_info(get_name(), $sformatf("Send value = %0h", req.value), UVM_NONE);
      #5;
    end
  endtask
endclass
```

REF: https://vlsiverify.com/uvm/tlm/tlm-analysis-fifo/

Consumer (e.g. Scoreboard)

```
class consumer extends uvm_component;
  seq_item req;
  uvm_tlm_analysis_fifo  #(seq_item) tlm_a_fifo;

  `uvm_component_utils(consumer)

  function new(string name = "consumer", uvm_component parent = null);
    super.new(name, parent);
    tlm_a_fifo = new("tlm_a_fifo", this);
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);

    repeat(10) begin
      #10;
      tlm_a_fifo.get(req);
      `uvm_info(get_type_name(), $sformatf("Received value = %0h", req.value), UVM_NONE);
    end
  endtask
endclass
```

Virtual Sequence and Virtual Sequencer

Virtual Sequence and Virtual Sequencer

- A **virtual sequence** starts multiple **sequences** on different **sequencers**.
- **Virtual sequencer** controls other **sequencers**
  - and it is not attached to any **driver**.
- A **virtual sequence** is usually executed on the **virtual sequencer**.

Why are the virtual_sequence and virtual_sequencer named virtual?

- System Verilog has *virtual methods*, *virtual interfaces*, and *virtual classes*.
  - "*virtual*" keyword is common in all of them.
- But, **virtual_sequence** and **virtual_sequencer** do not require any *virtual* keyword.
  - UVM does not have uvm_virtual_sequence and uvm_virtual_sequencer as base classes.

- A **virtual sequence** is derived from **uvm_sequence**.

- A **virtual_sequencer** is derived from **uvm_sequencer** as a base class.
- **Virtual sequencer** controls other **sequencers**.
- It is not attached to any **driver** and can not process any **sequence_items** too.
- Hence, it is named **virtual**.

Backup Slide 1

DIFFERENCE BETWEEN LOGIC AND BIT IN SYSTEMVERILOG

Difference b/w logic and bit in SystemVerilog

In SystemVerilog, "logic" and "bit" are two different data types with distinct characteristics:

1. **logic**: The logic data type is a multi-valued logic type that can represent values beyond the traditional binary 0 and 1.
   - It can represent the following values: 0, 1, X (unknown), and Z (high-impedance).
   - The logic data type is **the most commonly used** data type in SystemVerilog
     - and is the *recommended data type* for general use.
2. **bit**: The bit data type is a binary data type that can only represent the values 0 and 1.
   - It is a more compact representation compared to logic,
     - as it does not have the additional values of X and Z.
   - The bit data type is typically used for Boolean or single-bit operations
     - where the additional values provided by logic are not required.

REF: https://www.quora.com/What-is-the-difference-between-logic-and-bit-in-SystemVerilog#:~:text=bit%20is%20a%202%2Dstate,1%2C%20x%2C%20and%20z.

Difference b/w logic and bit in SystemVerilog (2)

The main differences between logic and bit are:

1. **Supported Values**: logic can represent 0, 1, X, and Z, while bit can only represent 0 and 1.

2. **Memory Usage**: bit is more *memory-efficient* than logic
   - because bit only requires a single bit of storage,
   - whereas logic requires more memory to represent the additional values.

3. **Functionality**: logic is more versatile
   - and can be used in a wider range of applications,
     - while bit is better suited for simple Boolean operations.

- In general, you should use **logic** as the **default data type** in SystemVerilog
  - unless you have a specific reason to use bit,
    - such as when you *need to optimize memory usage* or perform simple Boolean operations.

REF: https://www.quora.com/What-is-the-difference-between-logic-and-bit-in-SystemVerilog#:~:text=bit%20is%20a%202%2Dstate,1%2C%20x%2C%20and%20z.

Backup Slide 2

MODPORTS IN SV

Modports in SV

- **Modports** are a part of an **interface** that helps define the direction of signals.
- But we can also define the direction of signals within the **interface**,
  - so what is the need for **modports**?
- We can see how **modports** help us organize signals in *complex designs* as well as *testbenches*.

REF: https://www.theoctetinstitute.com/content/sv/modports/

What are modports?

- **Modports** can be considered as a sub-entity of **interface** which defines a particular direction to the signals.
- In an **interface**, there can be many **modports**
  - and all **modports** can define different directions to the signal.

Syntax:

modport <modport_name**> (**<direction> **<signal_name [range]>,** <direction> **<signal_name [range]>, ...);**

modport <name> (input <port_list>, output <port_list>);

modport TB **(**output **a,b, en,** input **out, ack);**
modport RTL **(**input **clk, reset, a,b, en,** output **out, ack);**

REF: https://www.theoctetinstitute.com/content/sv/modports/

Interface

```
interface mult_if (input logic clk, reset);
  logic [7:0] a, b;
  logic [15:0] out;
  logic en;
  logic ack;

  modport TB (output a,b, en, input out, ack);
  modport RTL (input clk, reset, a,b, en, output out, ack);
```

DUT (.sv)

```
module multiplier(mult_if inf);

  always@(posedge inf.clk or posedge inf.reset) begin
    if(inf.reset) begin
      inf.out <= 0;
      inf.ack <= 0;
    end
    else if(inf.en) begin
      inf.out <= inf.a * inf.b;
      inf.ack <= 1;
    end
    else inf.ack <= 0;
  end
endmodule
```

REF: https://vlsiverify.com/system-verilog/systemverilog-modport/

Need for modports

An **interface** is an entity that helps connect testbenches to

- designs
- or different modules of a design.

For example, let's consider the simple scenario of connecting a testbench to a design.

- Any inputs in the design need to be driven by the testbench.
- So, the signals that are *inputs for the design* are *outputs for the testbench*.
- However, inside an **interface**,
  - the same signal cannot take on two different directions.
  - This is where **modports** become **helpful**.

- We can define two **modports** inside the **interface**.
  - One will define the *directions* with respect to *the design*,
  - and the other will define the *directions* with respect to *the testbench*.

Need for modports (2)

- We can see that complex testbenches often have ***drivers*** and ***monitors***.
  - For a driver to drive signals, the direction would be *output*,
  - but a monitor, as its name suggests, needs to monitor the signals and thus all the signals will be *inputs* in this case.

- This is another scenario where **modports** are helpful.

- This is the main reason why we use **logic** as a data type for signals inside.
  - If we used **reg** or **wire**,
    - then it would be difficult to make it behave as both input and output
      - without worrying about the ***driver*** load issue.

How to define port directions ?

```
1  interface myBus (input clk);
2    logic [7:0]  data;
3    logic        enable;
4
5    // From TestBench perspective, 'data' is input and 'write' is output
6    modport TB  (input data, clk, output enable);
7
8    // From DUT perspective, 'data' is output and 'enable' is input
9    modport DUT (output data, input enable, clk);
10 endinterface
```

REF: https://www.chipverify.com/systemverilog/systemverilog-interface

m_sequencer and p_sequencer – Difference

- One of *the most confusing UVM stuff* is about **m_sequencer** and **p_sequencer** and the difference between the two.
  - In reality, its just a game of **polymorphism**.
- Referring to some forum answer, m_sequencer is a ***generic sequencer pointer*** of type uvm_sequencer_base.
  - It will always exist for a **uvm_sequence**
    - and is initialized when the **sequence** is started.
- The **p_sequencer** is a ***type specific sequencer pointer***, created by registering the **sequence** to a **sequencer** using the `uvm_declare_p_sequencer macros.
  - Being type specific,
    - you will be able to access anything added to the **sequencer** (i.e. pointers to other **sequencers**, etc.).
  - **p_sequencer** will not exist if the `uvm_declare_p_sequencer macros isn't used.

REF: https://asic4u.wordpress.com/2015/12/31/m_sequencer-p_sequencer-difference/#:~:text=The%20p_sequencer%20is%20a%20type,other%20sequencers%2C%20etc.).

Thank You