# Basic UVM Training

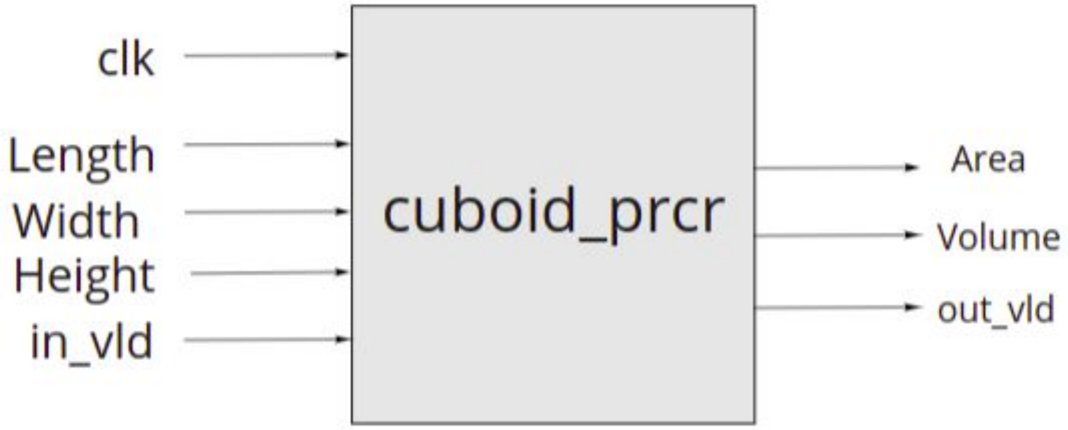
By
Maarij Raheem
Principal Engineer
Sahil Semiconductor

## Contents

## Example DUT (Cuboid Processor)

- We will create a Verification for this DUT in this presentation
- Interface

```
 input              clk,
 // input interface
 input      [16-1:0] length,
 input      [16-1:0] width,
 input      [16-1:0] height,
 input              in_valid,

// output interface
 output reg          out_valid,
 output reg [32-1:0] area,
 output reg [32-1:0] volm
```



## UVM Methodology

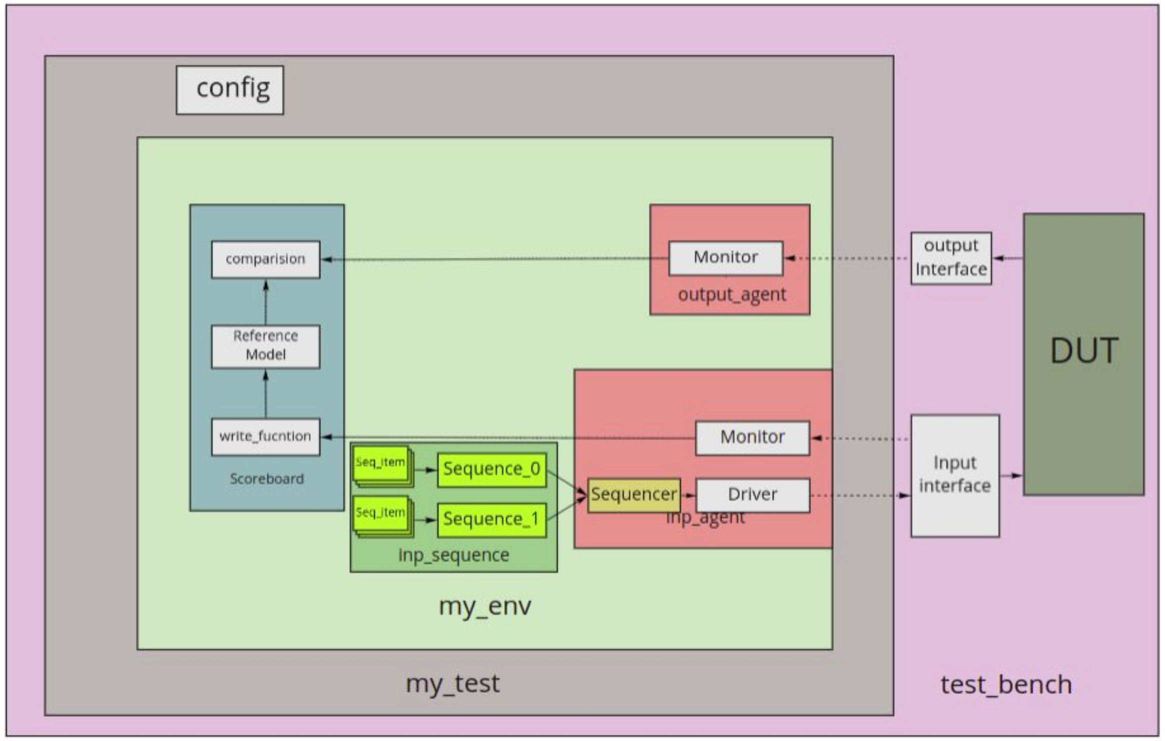
- Base Class Library
  - A library of generic building blocks
- Why UVM
  - UVM is a standard Methodology throughout the industry
  - Consistent approach
  - Has Many Advantages

## UVM Advantages

- Increases Readability
- Increases Reusability
- Provides Higher Level of abstraction i.e. converts pin level to transaction level
- Provides various built-in factory methods for all classes
- Provides a user-controllable messaging and debug utility
- Provides Command line processor to take runtime arguments +args
- Provides a standard resource sharing database (uvm_config_db)
- Provides built in test phases for synchronization purpose
- Tests can be instantiated at runtime. You need not to re-compile anything
- Easier to Document the Verification Plan

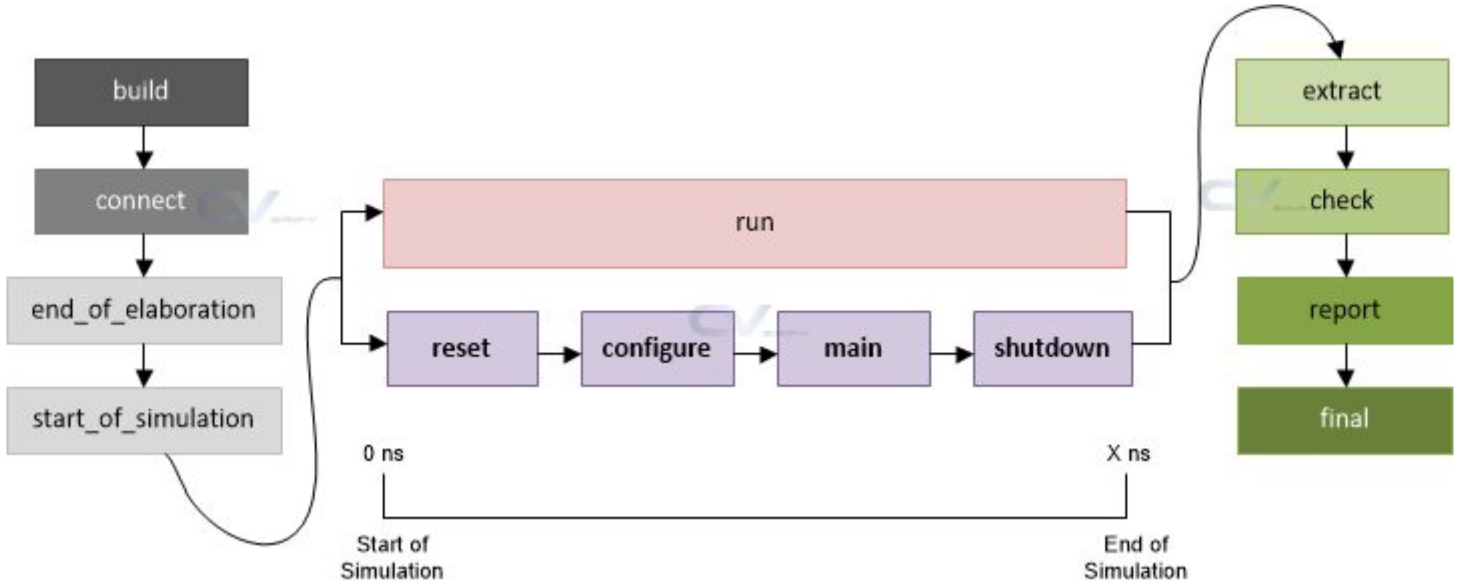
# UVM Testbench Architecture

- To setup a uvm testbench we need to work out the architecture of the environment
- Components
    - uvm_test
    - uvm_env
    - uvm_agent
    - uvm_driver
    - uvm_monitor
    - uvm_scoreboard
- Objects
    - uvm_sequence
    - Uvm_sequence_item
    - uvm_configs



## uvm_phases

- Uvm phases act like as a synchronization mechanism
- All phases except run phase execute in zero time



## UVM Testbench

- Instantiates the DUT
- Instantiates the interfaces and connects that to DUT
- Set the interfaces in uvm_config_db;
- Dynamically Instantiates the test class at run time (run_test();)
    - Thus enables us to run multiple tests without re-compiling the design or environment
    - sim_cmd +UVM_TESTNAME=my_test

## Configuration Database

- uvm_config_db is the built in database in which we can store values of any type under a given name for a given scope. That can be retrieved with in that scope i.e. testbench component.
- All the functions are static and must be called using the :: scope operator.
- Setting an interface in uvm_config_db

```
uvm_config_db #(virtual cuboid_inp_intf)::set(null, "uvm_test_top.env.ingr_agnt.drvr" , "cuboid_in_intf", cuboid_inp_if);
```

- Getting the same interface in uvm_test_top.env.ingr_agnt.drvr from uvm_resource_db using uvm_config_db

```
uvm_config_db#(virtual cuboid_inp_intf)::get(this, "", "cuboid_in_intf", vif);
```

## UVM Test

- Uvm_test is the top-level UVM component in the uvm_testbench.
- It instantiate the top level environment
- It also configures the environment
- We should implement all the basic logic in a base_test and extend all the test cases from that base test.
- You can also implement the interface initializations tasks.

# UVM Test Example Code

**Example Code:**

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

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

  cuboid_env          env          ;
  common_config    common_cfg   ;

// Build Phase Method
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    common_cfg = common_config::type_id::create("common_cfg", this);
    common_cfg.randomize();
    uvm_config_db #(common_config)::set(null, "*", "common_cfg", common_cfg);
    env = cuboid_env::type_id::create("env", this);
  endfunction

endclass
```

## Factory usage in UVM

- Factory is essential part of UVM
- Required for test registration and operation
- Recommended for all components (env, agent, sequencer, driver, monitor, scoreboard, etc.)
- Recommended for all objects (config, transaction, seq_item, etc.)
- Not appropriate for static interconnect (TLM port, TLM FIFO, interface, etc.)
- Responsible for build and functional behavior

## UVM Environment

- Its a hierarchical component that groups together other verification components.
- Typically it contains:
  - Agents
  - Scoreboards
  - Other Environments
- It should also make connections between implementation ports of scoreboard and analysis ports of monitors in connect_phase
- It can also start sequences using implicit calls in build_phase

# Environment Example Code

```
class cuboid_env extends uvm_env;
  // factory registration, constructor method and class handle declarations are not shown here

// build Phase
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // creating Agents
    ingr_agnt   = inp_agent::type_id::create("ingr_agnt", this);
    egrs_agnt   = out_agent::type_id::create("egrs_agnt", this);
    scrbrd      = scoreboard::type_id::create("scrbrd", this);
    // Getting common_config
    uvm_config_db #(common_config)::get(this, "*", "common_cfg", common_cfg);
// Implicit Call for inp_sequence
uvm_config_db#(uvm_object_wrapper)::set(this,"ingr_agnt.sqncr.main_phase", "default_sequence",
inp_sequence::type_id::get());
  endfunction //build_phase
```

**continued...**

## Environment Example Code continued..

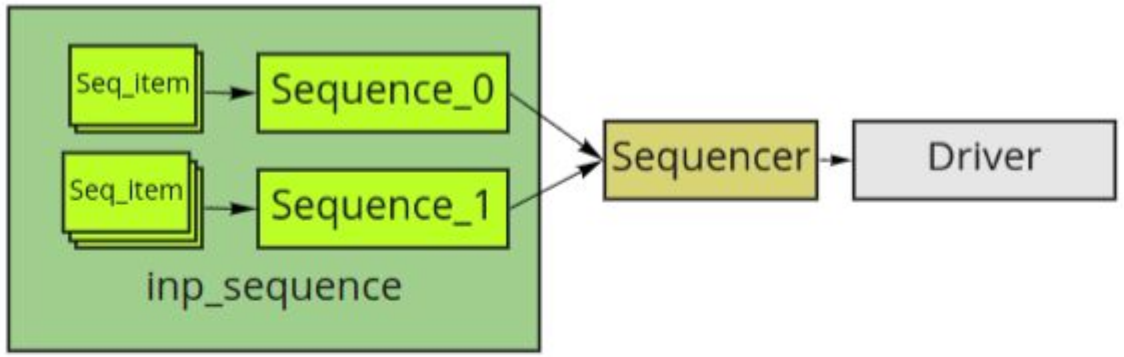
```
// Connection Phase
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //connecting analysis port to scoreboard
ingr_agnt.mntr.mon_analysis_port.connect(scrbrd.ingr_imp_export);
egrs_agnt.mntr.mon_analysis_port.connect(scrbrd.egrs_imp_export);
  endfunction  //connect_phase

// Main Phase
  virtual task main_phase(uvm_phase phase);
    phase.raise_objection(this);
    `uvm_info("cuboid_env", "Starting main_phase.. ", UVM_MEDIUM)
    super.main_phase(phase);
    wd_timer = common_cfg.watchdog_timer;

    fork
      #wd_timer;
      scb_evnt.wait_trigger();
    Join_any

    `uvm_info("cuboid_env", "main_phase done.. ", UVM_MEDIUM)
    phase.drop_objection(this);

  endtask // main_phase

endclass // cuboid_env
```

## UVM Objections raise/drop

- Objections allows status communication between participating components and objects
- The component or sequence will raise a phase objection at the beginning of an activity, that must be completed before the phase stops
- The objection will be dropped at the end of that activity.
- Once all of the raised objections are dropped, the phase terminates.
- UVM automatically kills all the forever loops inside the phases when terminates a phase

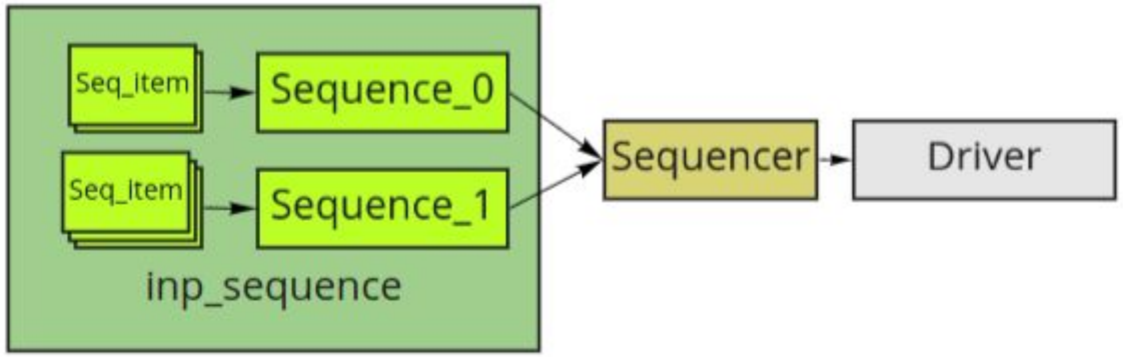
## UVM Sequence

- Generates stimulus
- Sequences/ Stimulus should be start based on test configuration
- Uvm_sequence can operate hierarchically with one sequence (called parent sequence) invoking another sequence (call a child sequence).
  - We use this as a standard practice
  - The parent sequence will be controlled generate stimulus
  - The child sequence(s) will generate required number of transactions depending upon the configuration
  - Multiple Child sequences are required when you need interleaving within transactions for multichannel behaviour.



# UVM Sequence continued..

- Each uvm_seuquence is bound to a particular phase of a sequencer when it is being started.
- Sequence can be start using:
    - Explicit call
        - Starting a sequence using start method.
        - **We use this call inside parent sequence to call child sequences.**
    - Implicit call
        - This is done by using uvm_config_db set function in environment
        - **We use implicit calls method as a standard practice for stimulus generation.**
        - It allows to have test reusability and run time reconfiguration
- It is an object



## Uvm Sequence Example Code (Parent sequence)

```
class inp_sequence extends uvm_sequence #(cuboid);
  // factory registration, constructor method are not shown here

  cuboid_sequence cboid_seq ; // child sequence

  virtual task body();
    cboid_seq = cuboid_sequence::type_id::create("cboid_seq");
    // Start Child Sequence
    cboid_seq.start(get_sequencer()); // Or you can give the complete hierarchical path of the sequencer
  endtask

endclass //inp_sequence
```

# Uvm Sequence Example Code (Child sequence)

```
class cuboid_sequence extends uvm_sequence #(cuboid);
  // factory registration, constructor method and class handle declarations are not shown here

  cuboid            cboid          ;
  cuboid_config  cboid_cfg      ;
  int                  num_cboids  ;

  virtual task body();
    cboid            = cuboid::type_id::create("cboid");                          // creating seq_item object
    cboid_config = cuboid_config::type_id::create("cboid_config"); // creating seq_item configuration object

    for (int i = 0; i < num_cboids; i++)  begin
      start_item(cboid);
      if (!cboid_cfg.randomize()) `uvm_fatal("cuboid_sequence", "cboid_cfg Randomization Failed"); // Randomize confg
      cboid.cboid_cfg = cboid_cfg;                    // assign randomized config to sequence item
      if(!cboid.randomize()) `uvm_fatal("cuboid_sequence", "cboid Randomization Failed"); // Randomize sequence item
      finish_item(cboid);                                // Transfer the control to Driver
    end // for (i = 0; i < num_cboids; i++)
  endtask // body

endclass // cuboid_sequence
```

## UVM sequence item

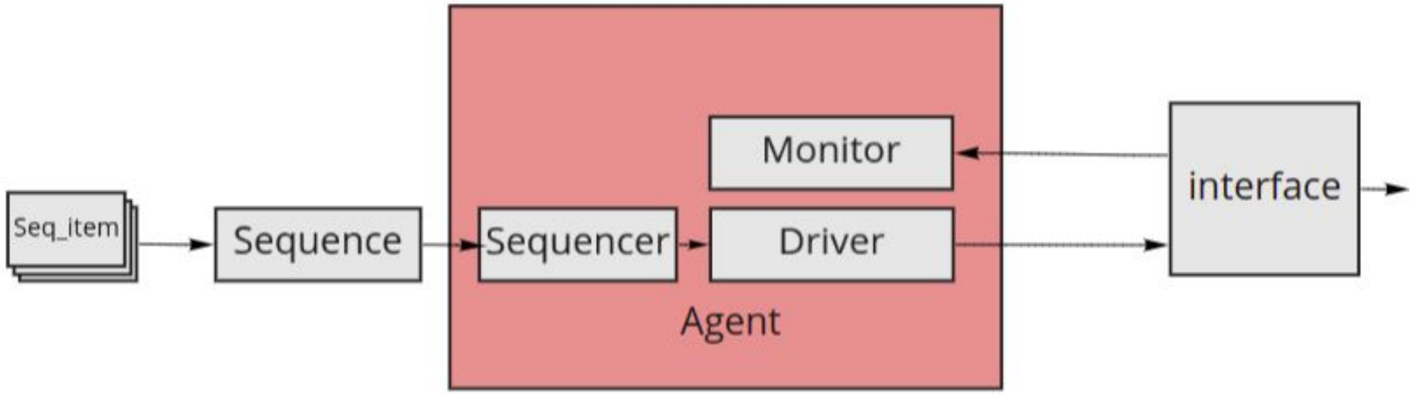
- Sequence Items are transactions
- It is being generated by uvm sequences
- It is an object
- Sequence items are continuously being created and destroyed throughout the simulation
- We need to use uvm_field_* macros inside uvm_sequence_item factory registration
    - We can control various functionalities using field macros like comparing, packing, printing etc.
- We can write constraints inside sequence item for input transactions
- We can write different Methods inside the sequence items as per our requirement like Display functions

# UVM Sequence Item Example Code

```
class cuboid extends uvm_sequence_item;
  // constructor method and class handle declarations are not shown here

  rand bit      [16-1:0] length    ;
  rand bit      [16-1:0] width     ;
  rand bit      [16-1:0] height    ;
  bit           [32-1:0] area      ;
  bit           [32-1:0] volm      ;
  cuboid_config          cboid_cfg ;

// Factory Registration
  `uvm_object_utils_begin(cuboid)
    `uvm_field_int(length, UVM_ALL_ON|UVM_NOCOMPARE)
    `uvm_field_int(width,  UVM_ALL_ON|UVM_NOCOMPARE)
    `uvm_field_int(height, UVM_ALL_ON|UVM_NOCOMPARE)
    `uvm_field_int(area,   UVM_ALL_ON)
    `uvm_field_int(volm,   UVM_ALL_ON)
  `uvm_object_utils_end

// constraints...
// Methods …

endclass // cuboid
```

## UVM Agent

- An agent deals with a specific DUT interface
- Typically agents contains Driver, Monitor and Sequencer.
- If an agent contains a driver it is called an active agent otherwise its a passive monitor



## UVM Agent Example Code

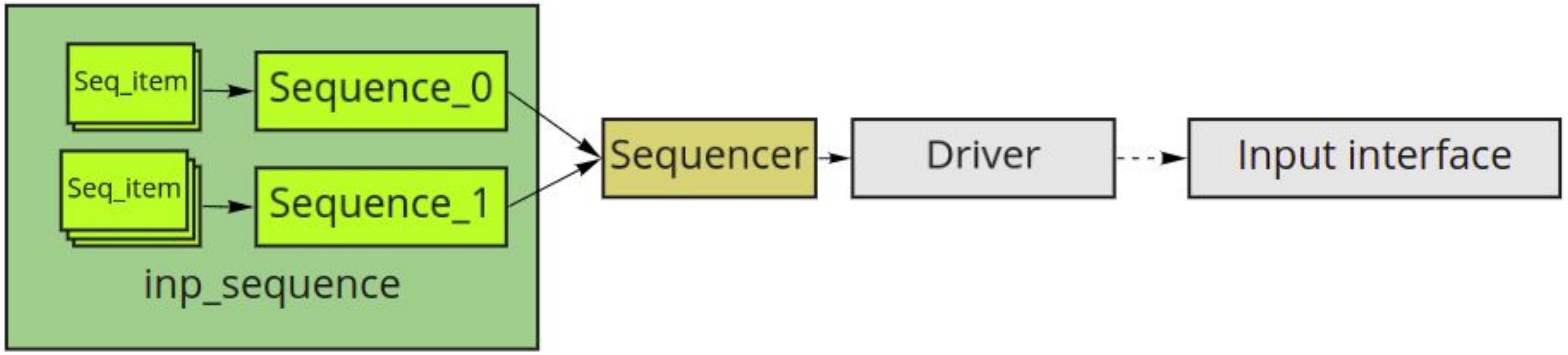
```
class inp_agent extends uvm_agent;
  // factory registration and constructor method are not shown here

  inp_monitor                     mntr ; // Monitor handle
  inp_driver                      drvr  ; // Driver  handle
  uvm_sequencer #(cuboid)   sqncr; // Sequencer Handle

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sqncr = uvm_sequencer#(cuboid)::type_id::create("sqncr", this);
    mntr = inp_monitor::type_id::create("mntr", this);
    drvr = inp_driver::type_id::create("drvr", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drvr.seq_item_port.connect(sqncr.seq_item_export); // connect driver built in seq item port with sqncr export
  endfunction
endclass
```

## UVM Sequencer

- Serves as a connection element between sequence and driver.
- Multiple uvm_sequence instances can be bound to same uvm_sequencer
- Arbiterates between different sequence items from multiple sequences
- By default it implements a round robin arbitration.
- Generally we use default uvm_sequencer i.e. we only create a handle and connect it in agent.
- It is a component.

## UVM Driver

- Receives sequence item from the uvm_sequencer and drives it on the DUT Interface.
- Converts Transaction-level stimulus into pin-level stimulus.
- It is a component.



## UVM Driver Example Code

```
class inp_driver extends uvm_driver #(cuboid);
// factory registration and constructor method are not shown here
  virtual cuboid_inp_intf  vif        ;
  cuboid                   cboid    ;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    cboid = cuboid::type_id::create("cboid", this);
    if (!uvm_config_db#(virtual cuboid_inp_intf)::get(this, "", "cuboid_in_intf", vif)) // Get the interface to driver the data
      `uvm_fatal("INP_DRIVER", "Could not get vif")
   endfunction // build_phase

  virtual task main_phase(uvm_phase phase);
    super.main_phase(phase);
    forever begin
      seq_item_port.get_next_item(cboid); // wait until a trasaction is recieved
      drive_item(cboid);                  // Main logic you need to write to drive the  data
      seq_item_port.item_done();          // Acknowledge  the sequence that it is done
    end
  endtask // main_phase
```

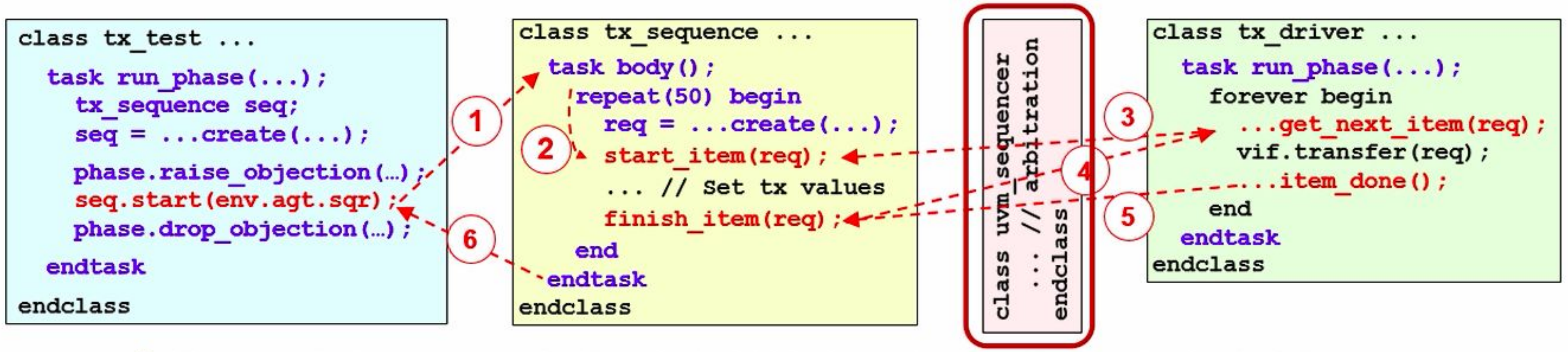
**s**

## UVM Driver Example Code continued

**continued..**

```
  virtual task drive_item(cuboid drv_cboid);
    vif.valid  <= 1 ;
    vif.length <= drv_cboid.length ;
    vif.width  <= drv_cboid.width  ;
    vif.height <= drv_cboid.height ;
    @(posedge vif.clk);
    vif.valid   <= 0 ;
    vif.length <= 0 ;
    vif.width  <= 0 ;
    vif.height <= 0 ;
  endtask // drive_item

endclass // inp_driver
```

# Handshaking Between Test/Sequence/Driver



```
class tx_test ...
  task run_phase(...);
    tx_sequence seq;
    seq = ...create(...);
    phase.raise_objection(…);
    seq.start(env.agt.sqr);
    phase.drop_objection(…);
  endtask
endclass
```

```
class tx_sequence ...
  task body();
    repeat(50) begin
      req = ...create(...);
      start_item(req);
      ... // Set tx values
      finish_item(req);
    end
  endtask
endclass
```

```
class uvm_sequencer
  ... // arbitration
endclass
```

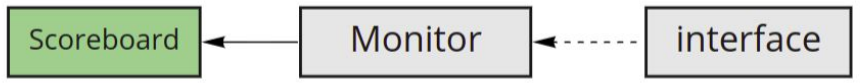
```
class tx_driver ...
  task run_phase(...);
    forever begin
      ...get_next_item(req);
      vif.transfer(req);
      ...item_done();
    end
  endtask
endclass
```

1. The test calls the sequence's start() method, which initializes its properties, then invokes body()
   - The test blocks (waits at that point) in the start() method until body() exits
   2. The sequence body() method calls start_item()
      - start_item() blocks until the driver asks for a transaction
      3. The driver calls seq_item_port.get_next_item() method to pull in a transaction
      - The driver then blocks until a transaction is received from the sequence
      4. The sequence fills in the item, calls finish_item() to send transaction to the driver, and blocks
      5. The driver calls item_done() to tell the sequence it is done with that object
   6. When the sequence body() exits, control is returned back to the test
7. The test continues with its next statement (such as allowing the run_phase to end)

* This slide is taken from the Mentor Learning paths videos. (Just to describe the flow)

## UVM Monitor

- Samples the DUT interface using virtual interfaces
- Monitor captures the information in form of transactions (sequence_items)
- Monitor normally sends the information to scoreboard using analysis port.



# Monitor Example Code

```
class inp_monitor extends uvm_monitor;
  // factory registration, constructor method and class handle declarations are not shown here

  uvm_analysis_port#(cuboid)  mon_analysis_port; // Analysis Port to broadcast the transaction
  virtual cuboid_inp_intf            vif                          ; // virtual interface to monitor
  cuboid                                    cboid                        ; // sequence item

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual cuboid_inp_intf)::get(this, "", "cuboid_in_intf", vif)) // gent the vif
      `uvm_fatal("INP_MONITOR", "Could not get vif")
    mon_analysis_port = new ("mon_analysis_port", this);                                          // create the ap
  endfunction // build_phase

  virtual task main_phase(uvm_phase phase);
    super.main_phase(phase);
    collect_data();     // Main logic you need to write to collect data (in forever loop inside this task)
  endtask // main_phase
```

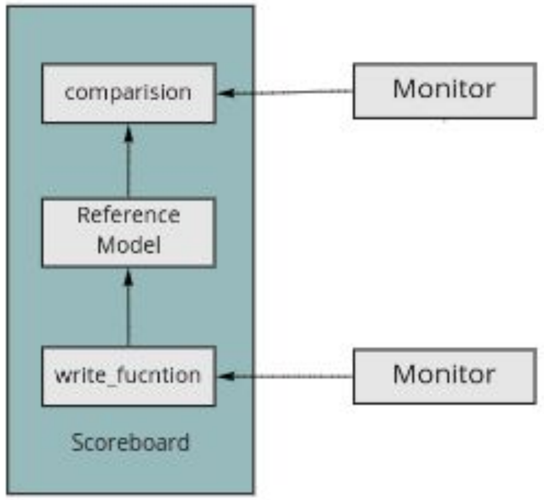
**continued..**

## Monitor Example Code continued..

```
  task collect_data ;
    forever begin
      if (vif.valid) begin
        cboid        = cuboid::type_id::create("INP Monitor Pkt");
        cboid.length = vif.length;
        cboid.width  = vif.width ;
        cboid.height = vif.height;
        mon_analysis_port.write(cboid); // write the sequence item on analysis port
        cboid.display_cuboid("INPUT_MONOTOR");
      end
      @(posedge vif.clk);
    end
  endtask // collect data

endclass
```

## UVM Scoreboard

- Its main function is to check the behavior of a certain DUT.
- It receives transactions from different input and output Monitors via implementation ports.
- For every implementation port we should have a write function.
- It may also contains reference model also known as predictor to produce expected output.



## UVM Scoreboard Example Code

```
`uvm_analysis_imp_decl(_ingr)
`uvm_analysis_imp_decl(_egrs)

class scoreboard extends uvm_scoreboard;
  uvm_analysis_imp_ingr #(cuboid, scoreboard) ingr_imp_export;
  uvm_analysis_imp_egrs #(cuboid, scoreboard) egrs_imp_export;
  cuboid exp_cboid, cboid, ingr_cboid_q[$];

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ingr_imp_export = new ("ingr_imp_export", this);
    egrs_imp_export = new ("egrs_imp_export", this);
  endfunction // build_phase

  virtual function void write_ingr (cuboid cboid);
    // Calculate expected Output that should be compared   using Reference Model function
    ….
    // Pushing the expected cboid in ingr_cboid_q
    ingr_cboid_q.push_back(cboid) ;
  Endfunction //write_ingr
```

# UVM Scoreboard Example Code Continued

**Continued..**

```
 virtual function void write_egrs (cuboid cboid);

   if (ingr_cboid_q.size() == 0)
     `uvm_error("SCB", $sformatf("Data not Present"))
   else
     exp_cboid = ingr_cboid_q.pop_front();

   if(cboid.compare(exp_cboid))
     match++;
   else
     mismatch++;
 endfunction  // write_egrs

virtual function void report_phase (uvm_phase phase);
   `uvm_info("SCB", $sformatf("cuboid Matched=%0d, Mismatched=%0d", match, mismatch), UVM_MEDIUM)
 endfunction // report_phase

endclass // scoreboard
```

## Configurations

- We can use different configurations in the uvm_architecture.
- Configuration are simple classes with **constraints and post_randomize function**
- Typically we use 3 type of configurations
    - Common Configurations
        - Test configuration
        - Remains constant for a single test
        - Can be used to generate random tests
    - Ral Configuration
        - Register file configurations
        - Remains constant for a single test
        - Can be used to generate random configurations
    - Sequence_item_configurations
        - Transaction configuration
        - Can be different per transaction.
        - Can be use generate random stimulus

## Command line processor

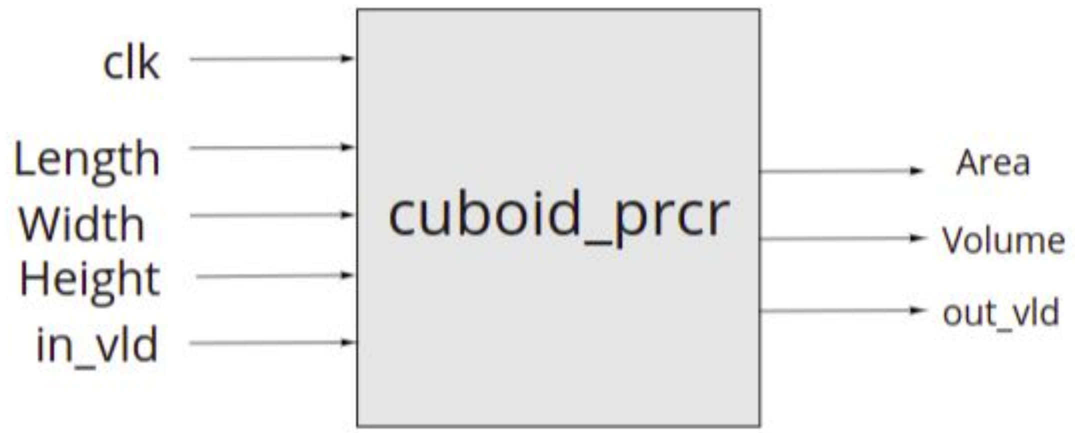
- An interface to the command line arguments
- Is used to get the runtime configurations
- Normally we call get_arg_value in post_randomize function of configs
  **function int get_arg_value (string  match,    ref   string  value)**

## Command Line Processor Usage

```
function void post_randomize();
  string arg_value;
  super.post_randomize();
  if (clp.get_arg_value("+inp_num_cboids=" , arg_value))
     inp_num_cboids = arg_value.atoi();
endfunction // post_randomize
```

## Exercise

- Try to Create a Verification Environment Block Diagram for the cuboid_prcr.



# Directory Structure

- <yourname>_training
    - UVM_assignment
        - design
            - cuboid_prcsr/
            - cuboid_prcsr_2/
            - cuboid_prcsr_3/
            - complex_number_prcsr/
        - Verif
            - Cuboid_prcsr
                - Uvm_tb
                    - agents/
                    - configs/
                    - environment/
                    - interface/
                    - package/
                    - sequences/
                    - sequence_item/
                    - base_test.sv
                - Vlib
                    - tb_top.sv
                - Script
                    - Makefile
                - tests/
            - cuboid_prcsr_2/
            - cuboid_prcsr_3/
            - complex_number_prcsr/
    - book_examples/
    - RTL_assignment/

Lets run our example ...

# Thank You...

Feel Free to ask Questions ??