REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS ECE 387 – LECTURE 7

PROF. DAVID ZARETSKY

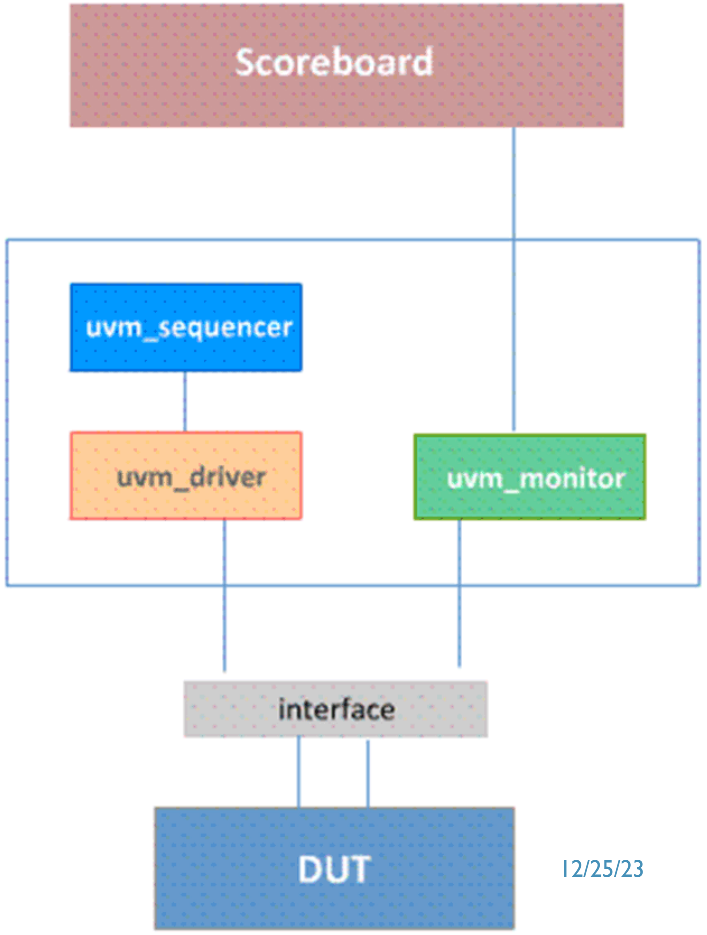
DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

- Universal Verification Methodology (UVM)
- Grayscale UVM demo

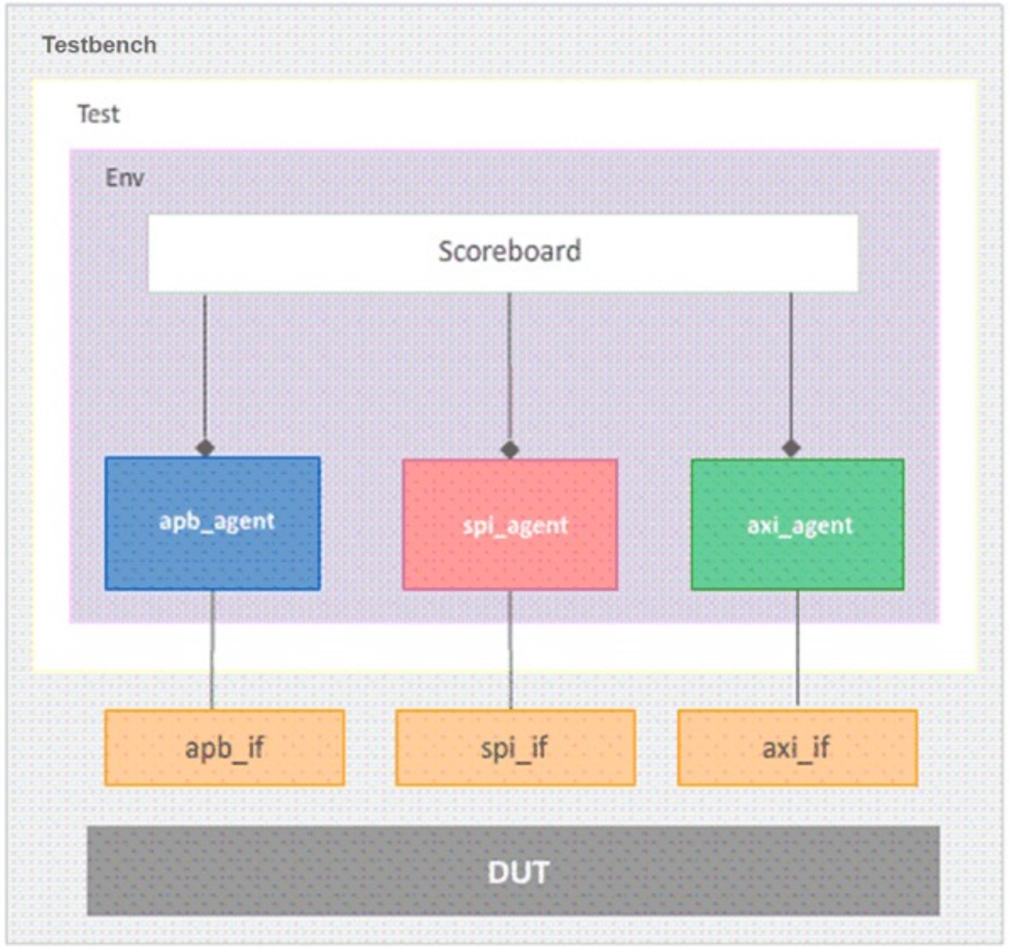
UVM INTRODUCTION

- Universal Verification Methodology (UVM) is a standard that enables faster development and reuse of verification environments and IP.
- It is a set of class libraries defined using the syntax and semantics of SystemVerilog (IEEE 1800) and is now an IEEE standard.
- UVM is a framework of SystemVerilog classes from which fully functional testbenches can be built in a more structured approach.
- The main idea behind UVM is to help companies develop modular, reusable, and scalable testbench structures by providing an API framework that can be deployed across multiple projects.



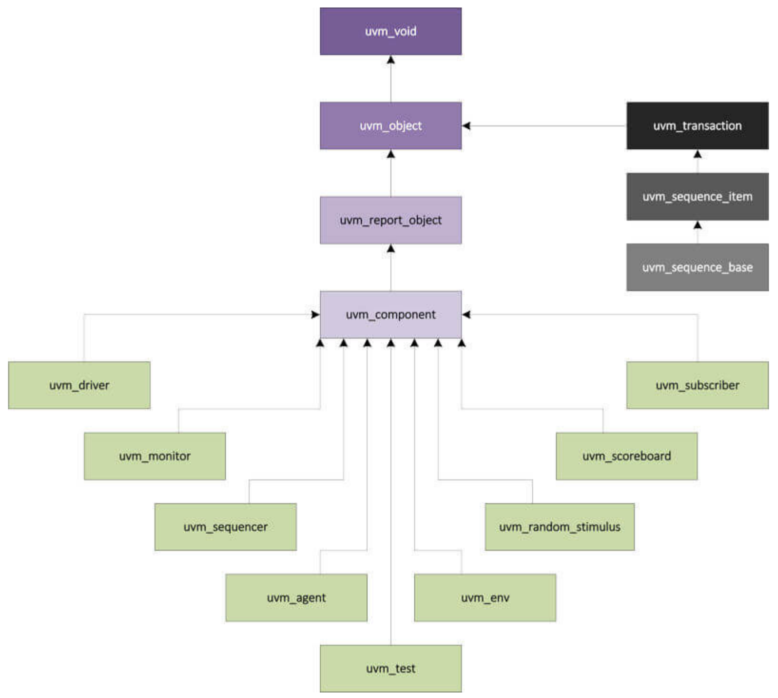
TESTING MULTIPLE INTERFACES

- The UVM framework is designed to allow testing of multiple endpoints or interfaces through Agents
- A single Scoreboard is used to validate data and report on errors in the design.
- Agents send data to Scoreboard via Transaction Level Modeling (TLM) analysis port, and are validated independently.



UVM CLASS HIERARCHY

- UVM provides a set of base classes from which more complex classes can be built by inheritance.
- Two branches in the hierarchy:
  - Verification components underneath uvm_report_object.
  - Data objects consumed and operated on under uvm_transaction.



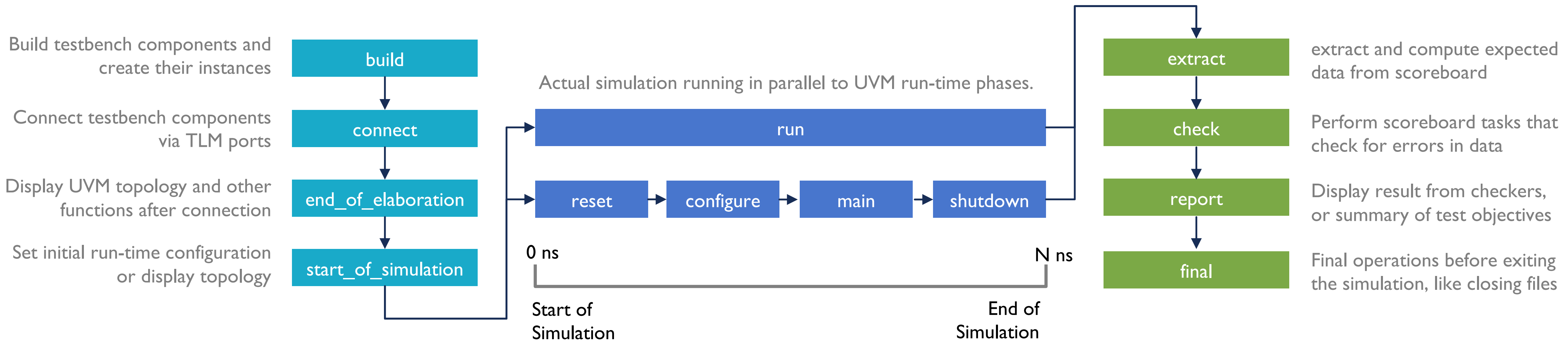
Class	Description
uvm_object	Define methods for common operations like copy, compare and print.
uvm_component	All testbench components like driver, monitor, scoreboards, etc
uvm_sequence_item	All sequence items that need to be sent to a driver to be driven onto the bus.

Component	Purpose
uvm_driver	Drive signals to DUT
uvm_monitor	Monitor signals at DUT output port
uvm_sequence	Create different test patterns
uvm_agent	The Sequencer, Driver and Monitor
uvm_env	All other verification components
uvm_scoreboard	Determines if test Passed/Failed
uvm_subscriber	Subscribes to activities of other components

UVM PHASES

- UVM testbench components are derived from uvm_component base class
  - Each component goes through a pre-defined set of phases
  - Each component cannot proceed to the next phase until all components finish their execution in the current phase.
- UVM phases act as a synchronizing mechanism in the life cycle of a simulation
  - Phases are defined as callbacks
  - classes derived from uvm_component can perform useful work in the callback phase method
- All phases can be grouped into 3 categories
  - Build time phases
  - Run time phases
  - Clean-Up phases

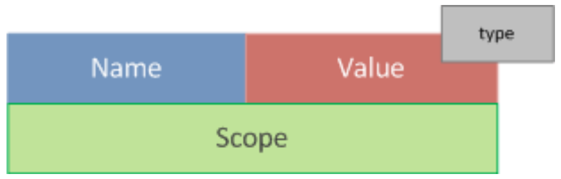
UVM PHASES

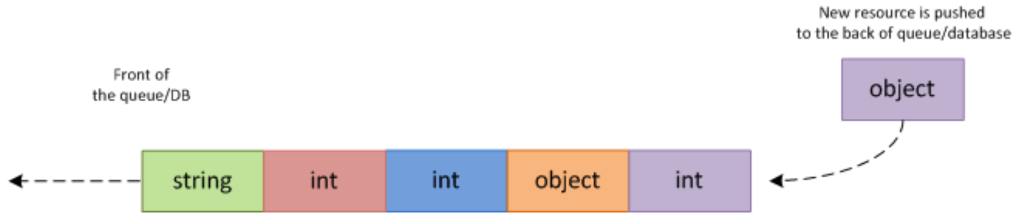

Build testbench components and create their instances
build
Connect testbench components via TLM ports
connect
Display UVM topology and other functions after connection
end_of_elaboration
Set initial run-time configuration or display topology
start_of_simulation
Actual simulation running in parallel to UVM run-time phases.
run
reset
configure
main
shutdown
0 ns
N ns
Start of Simulation
End of Simulation
extract
extract and compute expected data from scoreboard
check
Perform scoreboard tasks that check for errors in data
report
Display result from checkers, or summary of test objectives
final
Final operations before exiting the simulation, like closing files

UVM RESOURCE DATABASE

- UVM has an internal database table in which we can store values under a given name and can be retrieved later by some other component.
- A resource is a parameterized container that holds any data.
- Used to configure components, supply data to sequences, or enable sharing of information across disparate parts of the testbench.
- You can put any data type into the resource database,
- Other components can retrieve resources at any point in simulation
- Scoping information can constrain visibility across the testbench.
- The global resource database has both a **name** table and a **type** table into which each resource is entered.
- The uvm_config_db class simplifies the basic interface for components

Resource Data

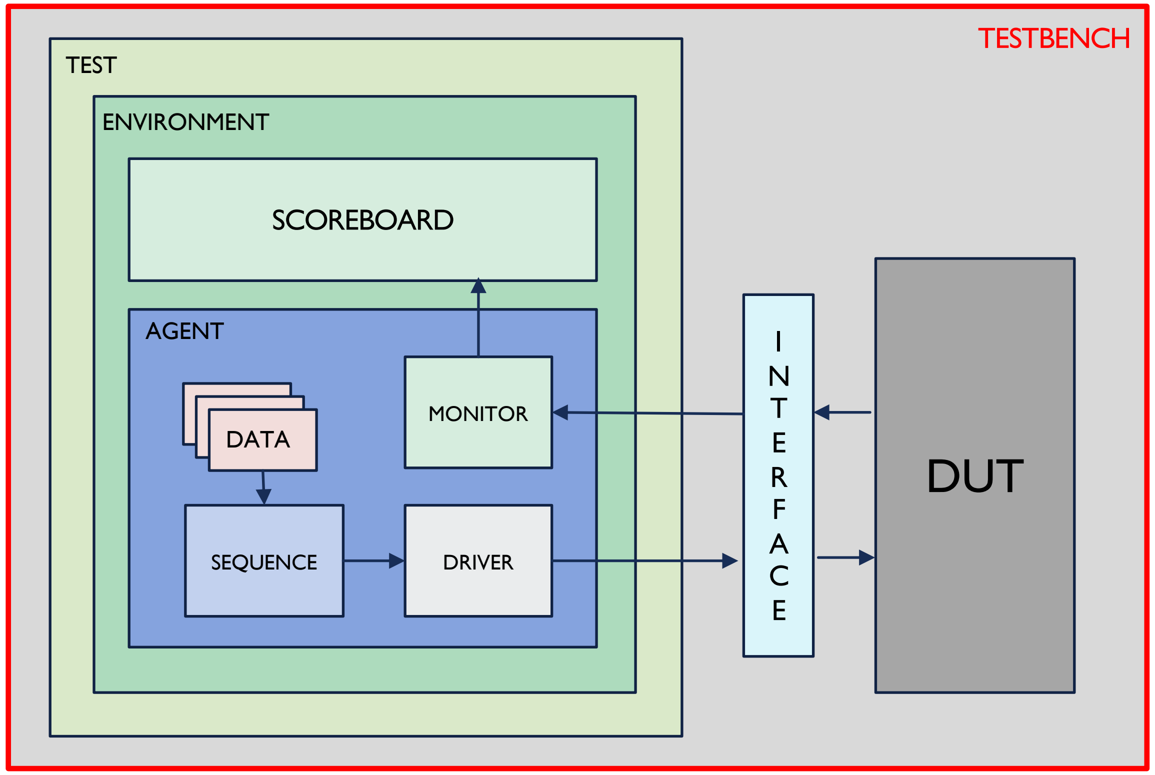




Name	The "name" by which this resource is stored in the database.
Value	The value that should be stored in the database for the given "name".
Scope	Regular expression that specifies the scope of visibility for components
Type	The object data type contained in the resource. Includes: string, int, virtual interface, class object, or any valid SystemVerilog data-type.

UVM TESTBENCH

- All verification components, interfaces and DUT are instantiated in a top level module called testbench.
- It is a static container to hold everything required to be simulated and becomes the *root* node in the hierarchy.
- The testbench generally includes the following:
  - Clock process
  - Reset process
  - DUT instance
  - Interfaces
  - UVM test (invoked by run_test method)



UVM TESTBENCH EXAMPLE

```
import uvm_pkg::*;
import my_uvm_package::*;

`include "my_uvm_if.sv"

`timescale 1 ns / 1 ns

module my_uvm_tb;

my_uvm_if vif();

grayscale_top #(
   .WIDTH(IMG_WIDTH),
   .HEIGHT(IMG_HEIGHT)
) grayscale_inst (
   .clock(vif.clock),
  .reset(vif.reset),
  .in_full(vif.in_full),
  .in_wr_en(vif.in_wr_en),
  .in_din(vif.in_din),
  .out_empty(vif.out_empty),
  .out_rd_en(vif.out_rd_en),
  .out_dout(vif.out_dout)
);
```

Interface instance

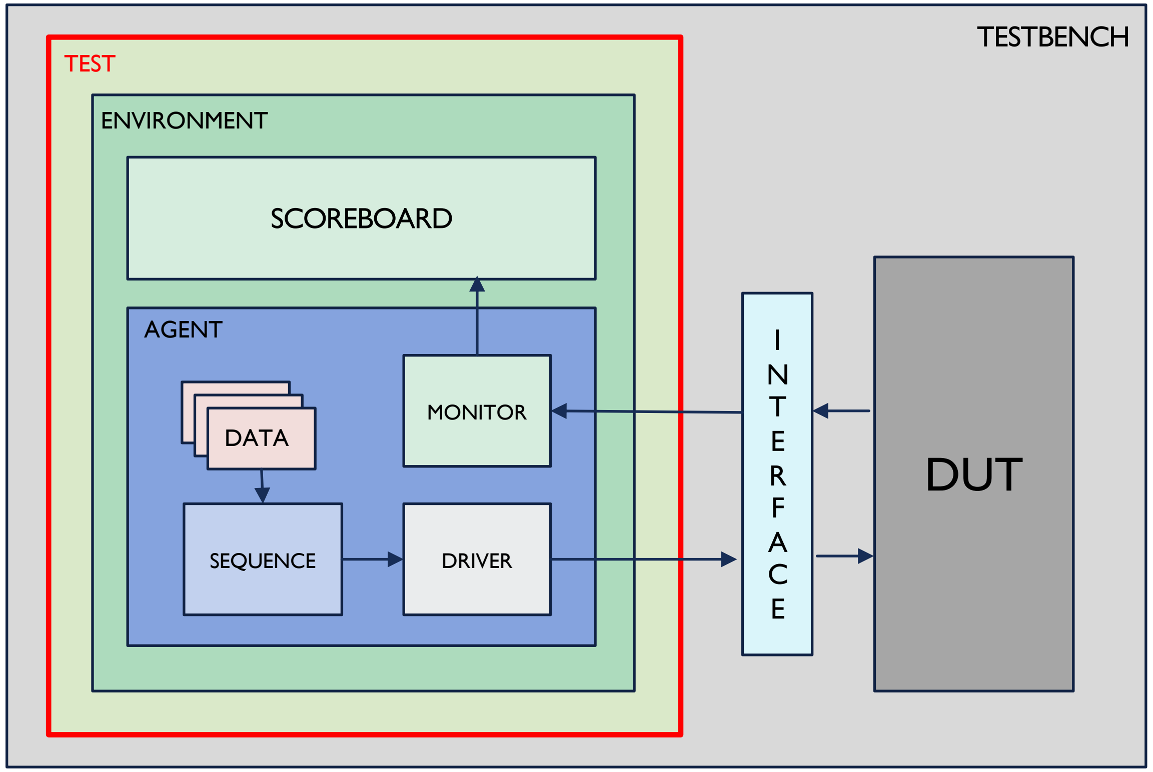
```
initial begin
  // store the virtual interface so it can be retrieved by the driver & monitor
  uvm_resource_db#(virtual my_uvm_if)::set(.scope("ifs"), .name("vif"), .val(vif));

  // run the test
  run_test("my_uvm_test");
end

// reset
initial begin
  vif.clock <= 1'b1;
  vif.reset <= 1'b0;
  @(posedge vif.clock);
  vif.reset <= 1'b1;
  @(posedge vif.clock);
  vif.reset <= 1'b0;
end

// 10ns clock
always
  #CLOCK_PERIOD vif.clock = ~vif.clock;
endmodule
```

Run the simulation test

UVM TEST

- The UVM Test module is responsible for implementing the testcase patterns to check and verify specific features and functionalities of a design.
- The UVM Test module instantiates the environment and configures the system architecture.
- In the run_phase, the sequence is initiated to test the various features.
- A test is usually started within testbench by a task called **run_test**.

```
// This is a global task that gets the UVM root instance and  starts the test
task run_test (string test_name="");
   uvm_root top;
   uvm_coreservice_t cs;
   cs = uvm_coreservice_t::get();
   top = cs.get_root();
   top.run_test(test_name);
endtask
```



UVM TEST EXAMPLE

```
import uvm_pkg::*;

class my_uvm_test extends uvm_test;

`uvm_component_utils(my_uvm_test)

my_uvm_env env;

function new(string name, uvm_component parent);
   super.new(name, parent);
endfunction: new

virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   env = my_uvm_env::type_id::create(.name("env"), .parent(this));
endfunction: build_phase
```

Create the environment

Run_phase creates a sequence to write data to the DUT

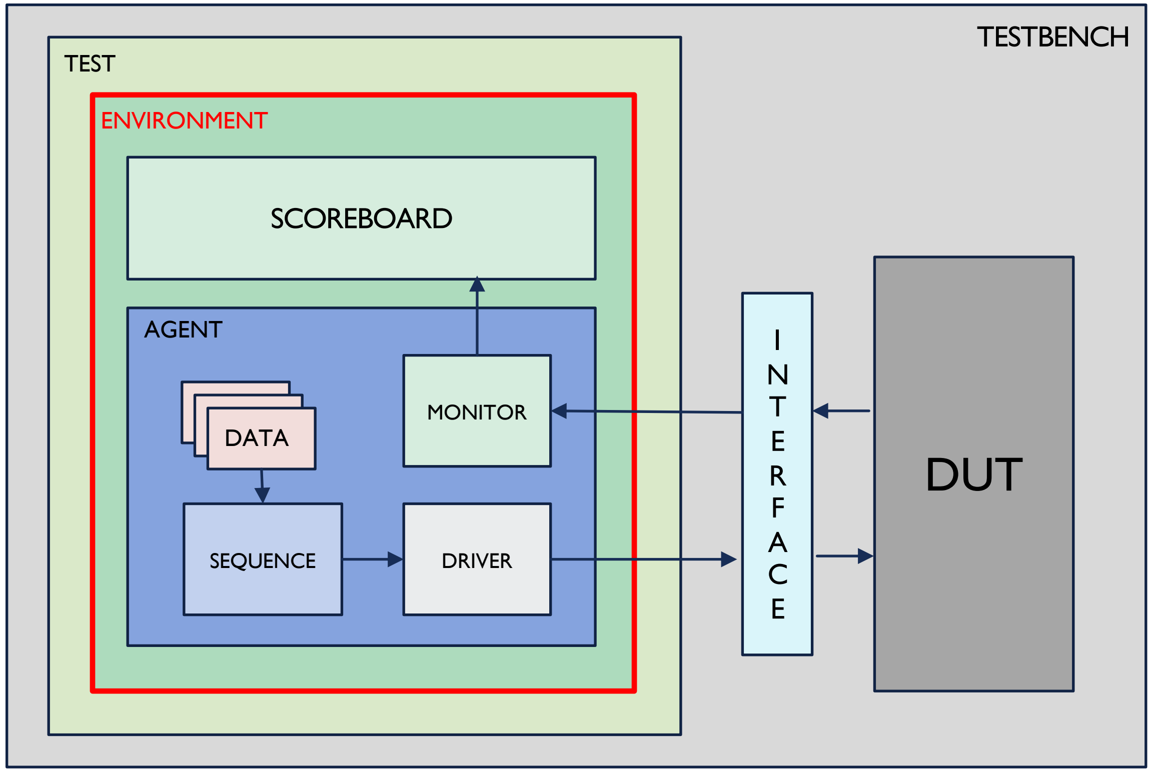
```
virtual function void end_of_elaboration_phase(uvm_phase phase);
    uvm_top.print_topology();
endfunction: end_of_elaboration_phase

virtual task run_phase(uvm_phase phase);
  my_uvm_sequence seq;
  phase.raise_objection(.obj(this));
  seq = my_uvm_sequence::type_id::create( .name("seq"),
                                          .contxt(get_full_name()));
  seq.start(env.agent.seqr);
  phase.drop_objection(.obj(this));
endtask: run_phase

endclass: my_uvm_test
```

Print the topology

IMPORTANT: Simulation terminates once all raised objections have been dropped.

UVM ENVIRONMENT

- A UVM Environment contains multiple, reusable verification components and defines their default configuration as required by the application.
- Instead of writing the same code for different testcases, we use the same environment with a different configuration for each test.
- UVM Environment will generally contain
  - multiple agents for different interfaces
  - a common scoreboard
  - data connections between agents and scoreboard
  - a functional coverage collector
  - additional checkers



UVM ENVIRONMENT EXAMPLE

```
import uvm_pkg::*;

class my_uvm_env extends uvm_env;
  `uvm_component_utils(my_uvm_env)

  my_uvm_agent agent;
  my_uvm_scoreboard sb;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agent = my_uvm_agent::type_id::create(.name("agent"), .parent(this));
    sb = my_uvm_scoreboard::type_id::create(.name("sb"), .parent(this));
  endfunction: build_phase

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    agent.agent_ap_output.connect(sb.sb_export_output);
    agent.agent_ap_compare.connect(sb.sb_export_compare);
  endfunction: connect_phase
endclass: my_uvm_env
```

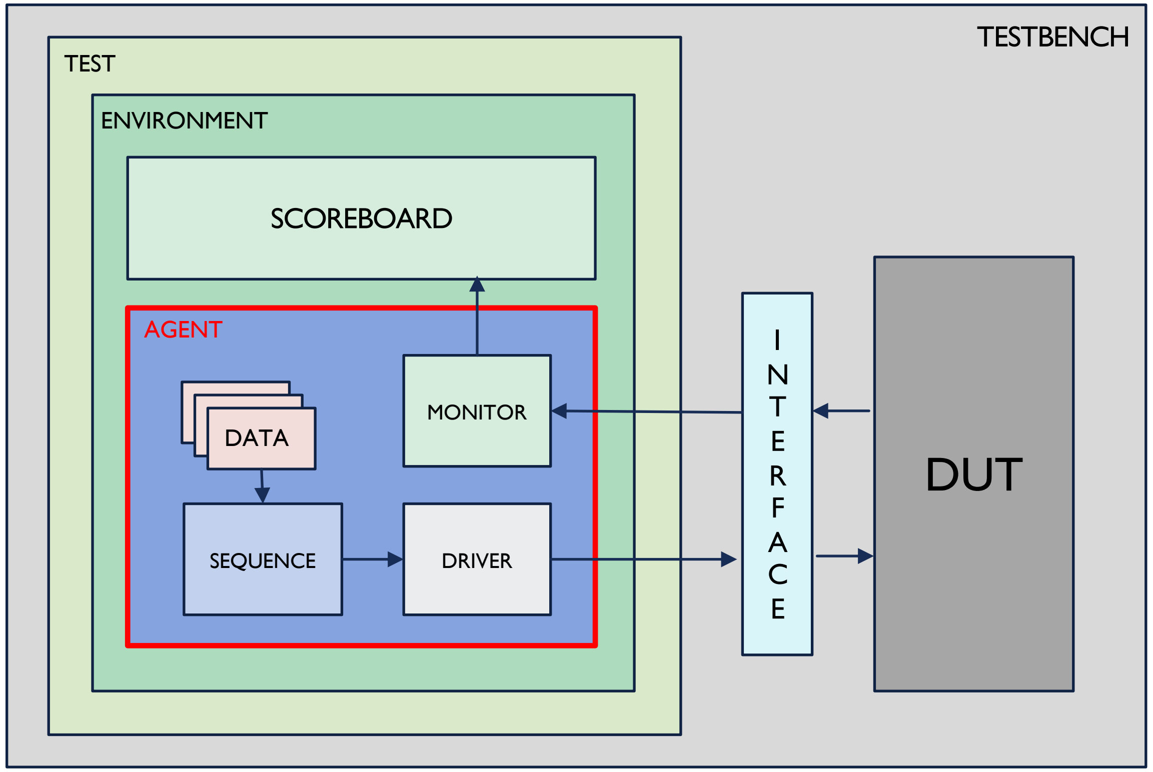
Create the agent

Create the scoreboard

Agent output data + compare data are connected to the scoreboard.

UVM AGENT

- UVM Agents provide protocol specific tasks by instantiating:
  - **Sequencer** to read input data and sends as individual transactions to the Driver
  - **Driver** that utilizes a protocol transactions to send data to the DUT via a specific interface.
  - **Monitor** reads data from DUT, and sends to the Scoreboard for validation via analysis port
- Active Agents
  - Instantiates all three components (Sequencer, Driver, Monitor)
  - Enables data to be driven to DUT via driver
  - Agents are active by default
  - `uvm_config_db #(int)::set(this, "path_to_agent", "is_active", UVM_ACTIVE);`
- Passive Agents
  - Only instantiate the monitor
  - Used for checking and coverage only
  - Useful when there's no data item to be driven to DUT
  - `uvm_config_db #(int)::set(this, "path_to_agent", "is_active", UVM_PASSIVE);`



UVM AGENT EXAMPLE

```
import uvm_pkg::*;

class my_uvm_agent extends uvm_agent;

  `uvm_component_utils(my_uvm_agent)

  uvm_analysis_port#(my_uvm_transaction) agent_ap_output;
  uvm_analysis_port#(my_uvm_transaction) agent_ap_compare;

  my_uvm_sequencer seqr;
  my_uvm_driver drvr;
  my_uvm_monitor_output mon_out;
  my_uvm_monitor_compare mon_cmp;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new
```

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  agent_ap_output = new(.name("agent_ap_output"), .parent(this));
  agent_ap_compare = new(.name("agent_ap_compare"), .parent(this));

  mon_out = my_uvm_monitor_output::type_id::create(.name("mon_out"), .parent(this));
  mon_cmp = my_uvm_monitor_compare::type_id::create(.name("mon_cmp"), .parent(this));
  seqr = my_uvm_sequencer::type_id::create(.name("seqr"), .parent(this));
  drvr = my_uvm_driver::type_id::create(.name("drvr"), .parent(this));
endfunction: build_phase

virtual function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  drvr.seq_item_port.connect(seqr.seq_item_export);
  mon_out.mon_ap_output.connect(agent_ap_output);
  mon_cmp.mon_ap_compare.connect(agent_ap_compare);
endfunction: connect_phase

endclass: my_uvm_agent
```

Agent creates monitors for the DUT output data and compare data.

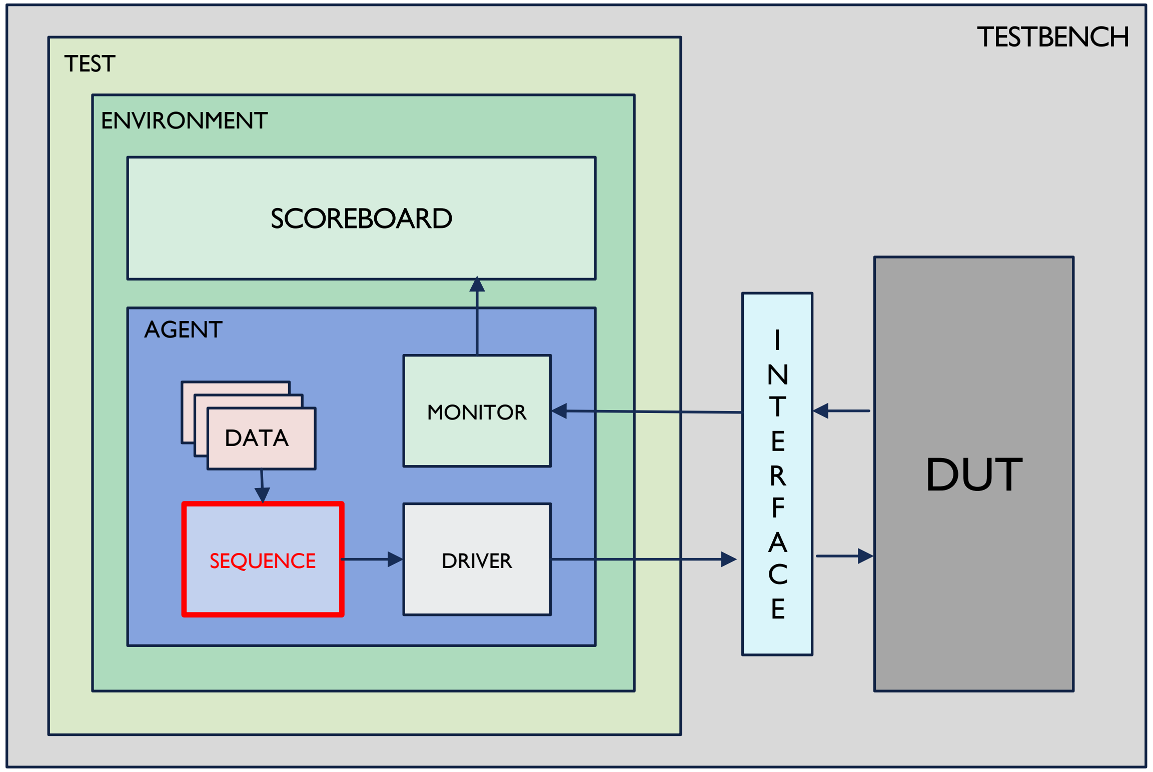
Agent creates and connects sequencer and driver for input data.

UVM SEQUENCE ITEM

- UVM Sequence Item is used to define the transaction data that's passed from the sequencer to the driver.

```
class my_uvm_transaction extends uvm_sequence_item;
  logic [23:0] image_pixel;

  function new(string name = "");
    super.new(name);
  endfunction: new

  `uvm_object_utils_begin(my_uvm_transaction)
     `uvm_field_int(image_pixel, UVM_ALL_ON)
  `uvm_object_utils_end
endclass: my_uvm_transaction
```

Sequence Item defines the transaction data to be sent to the driver

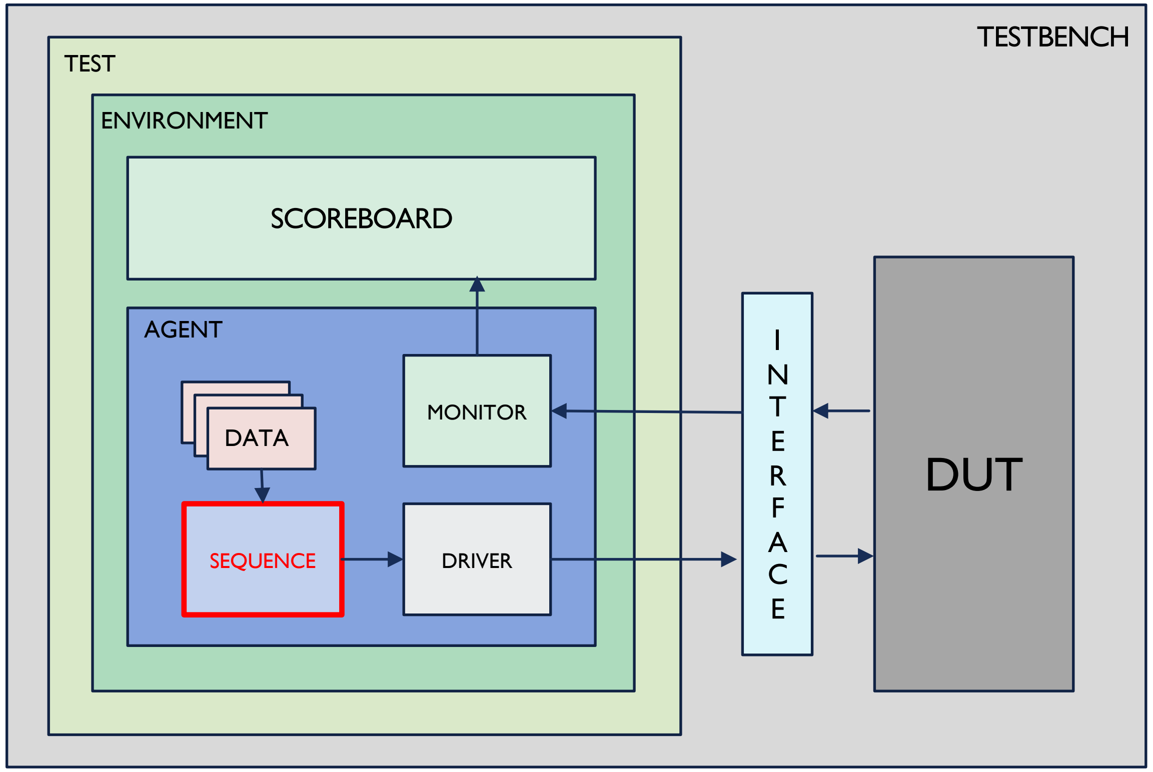


UVM SEQUENCE

- UVM **Sequence** generates data transactions as class objects (UVM Sequence Item) and sends it to the Driver
- Transaction data is implemented as a queue.
- Use start_item() and finish_item() calls to initiate the transaction.
- While creating transactions, it's useful to print information, for example:

  -
    ```
    `uvm_info("SEQ_RUN", $sformatf("Loading file %s...",
             IMG_IN_NAME), UVM_LOW);
    ```

  -
    ```
    `uvm_fatal("SEQ_RUN", $sformatf("Failed to open file %s...",
             IMG_IN_NAME));
    ```



UVM SEQUENCE EXAMPLE

```
class my_uvm_sequence extends uvm_sequence#(my_uvm_transaction);
  `uvm_object_utils(my_uvm_sequence)

  function new(string name = "");
    super.new(name);
  endfunction: new

  task body();
    my_uvm_transaction tx;
    int in_file, n_bytes=0, i=0;
    logic [7:0] bmp_header [0:BMP_HEADER_SIZE-1];
    logic [23:0] pixel;

    `uvm_info("SEQ_RUN", $sformatf("Loading file %s...",
            IMG_IN_NAME), UVM_LOW);

     in_file = $fopen(IMG_IN_NAME, "rb");
     if ( !in_file ) begin
       `uvm_fatal("SEQ_RUN", $sformatf("Failed to open file %s...",
            IMG_IN_NAME));
     end
```

Read image file and write pixels as transaction data

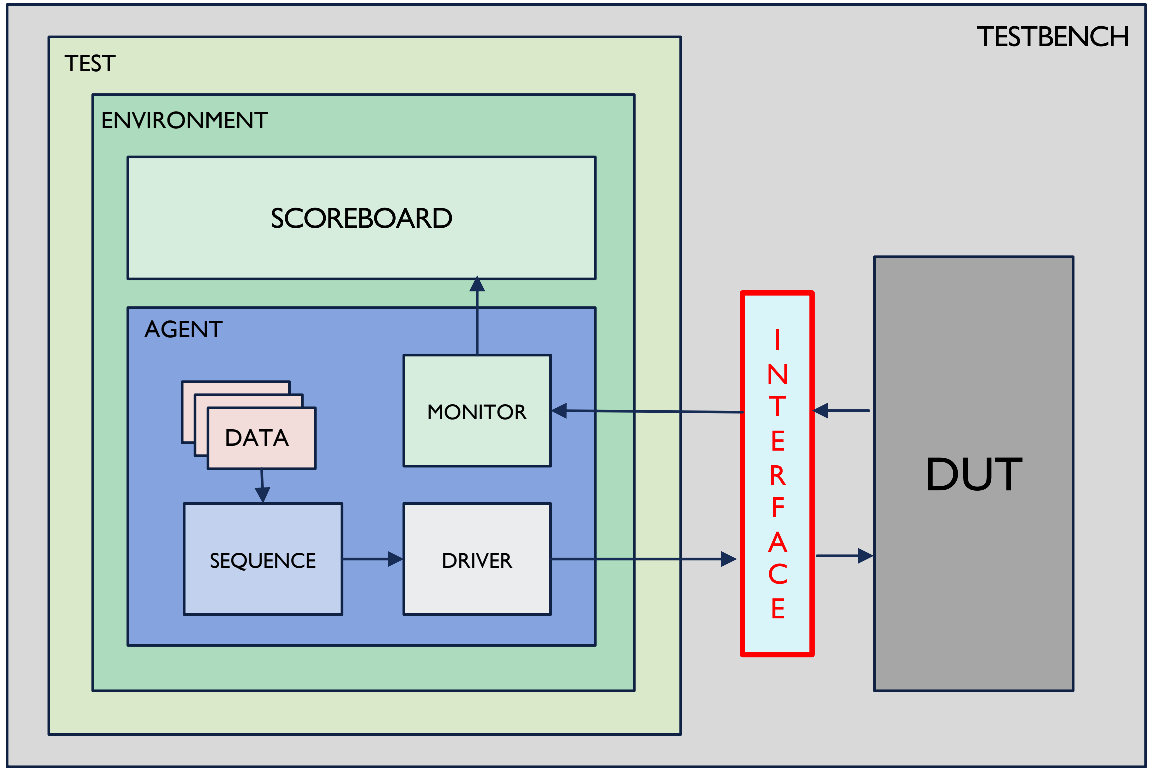
```
    // read BMP header
    n_bytes = $fread(bmp_header, in_file, 0, BMP_HEADER_SIZE);
    if ( !n_bytes ) begin
      `uvm_fatal("SEQ_RUN",
          $sformatf("Failed read header data from %s...", IMG_IN_NAME));
    end

    while ( !$feof(in_file) ) begin
      tx = my_uvm_transaction::type_id::create(.name("tx"),
                                                .contxt(get_full_name()));
      start_item(tx);
      n_bytes = $fread(pixel, in_file, BMP_HEADER_SIZE+i, BYTES_PER_PIXEL);
      tx.image_pixel = pixel;
      //`uvm_info("SEQ_RUN", tx.sprint(), UVM_LOW);
      finish_item(tx);
      i += BYTES_PER_PIXEL;
    end

    `uvm_info("SEQ_RUN", $sformatf("Closing file %s...", IMG_IN_NAME),
            UVM_LOW);
    $fclose(in_file);
  endtask: body
endclass: my_uvm_sequence
```

Transaction queues are demarcated by start_item() and finish_item() calls.

UVM INTERFACE

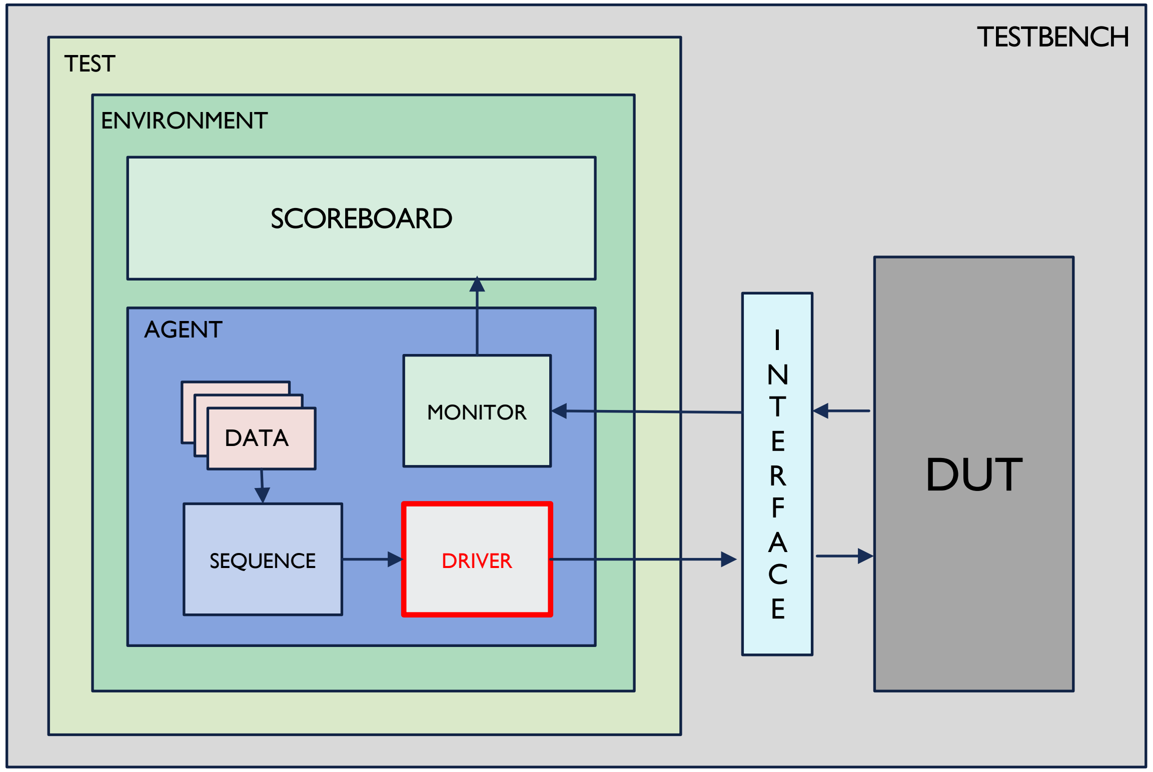
- Define a UVM Interface for each DUT endpoint
- Driver and Monitor pass data to and from DUT via the Interface
- Interfaces may include
  - FIFOs
  - Block RAMs
  - Bus architectures
  - Peripherals
  - Clock / Reset

```
import uvm_pkg::*;

interface my_uvm_if;
  logic clock;
  logic reset;
  logic in_full;
  logic in_wr_en;
  logic [23:0] in_din;
  logic out_empty;
  logic out_rd_en;
  logic [7:0] out_dout;
endinterface
```



UVM DRIVER

- UVM driver is an active entity that drives signals to a particular interface of the DUT.
- It reads transaction data from the queue and sends data to the DUT via a particular interface.
- Handshaking between Driver and Sequencer
  - get_next_item() - Blocks until a request item is available from the sequencer. This should be followed by item_done call to complete the handshake.
  - try_next_item() - Non-blocking method which will return null if a request object is not available from the sequencer. Else it returns a pointer to the object.
  - item_done() - Non-blocking method which completes the driver-sequencer handshake. This should be called after get_next_item or a successful try_next_item call.



UVM DRIVER EXAMPLE

```
import uvm_pkg::*;

class my_uvm_driver extends uvm_driver#(my_uvm_transaction);

  `uvm_component_utils(my_uvm_driver)

  virtual my_uvm_if vif;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(virtual my_uvm_if)::read_by_name(
            .scope("ifs"), .name("vif"), .val(vif)));
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      drive();
   endtask: run_phase
```

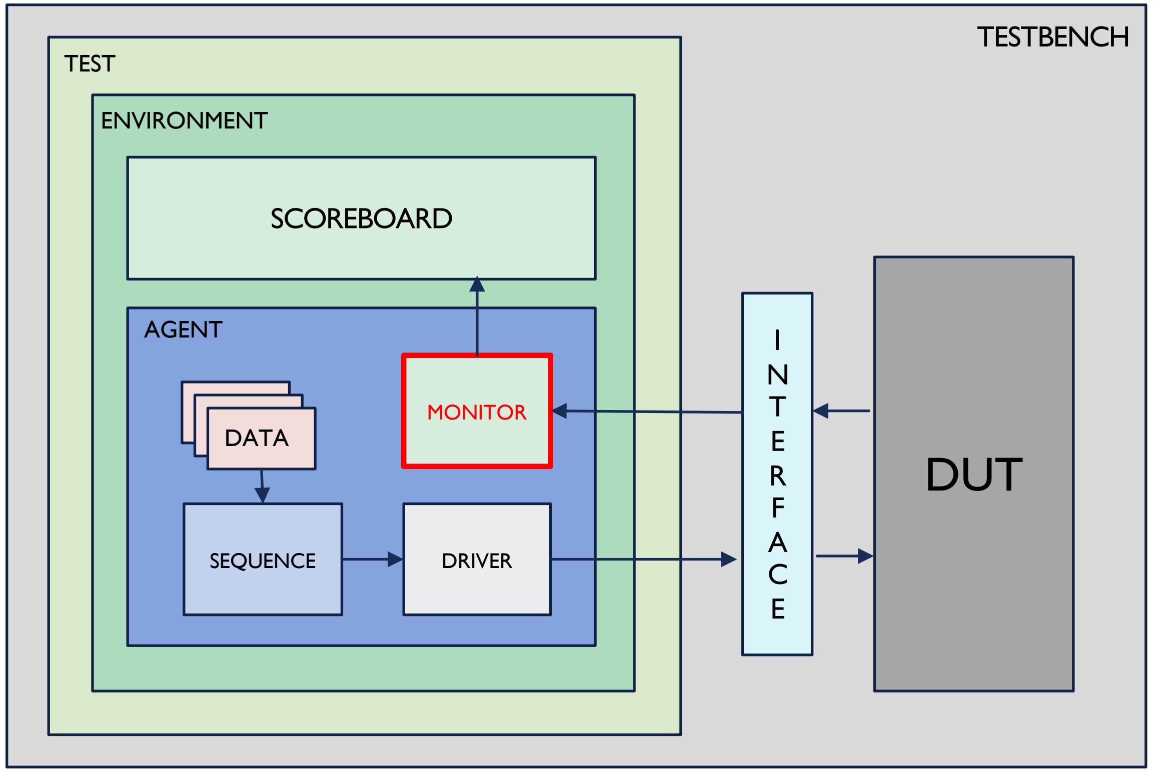
FIFO interface used to pass data to DUT

```
  virtual task drive();
    my_uvm_transaction tx;

    // wait for reset.
    @(posedge vif.reset)
    @(negedge vif.reset)

    vif.in_din = 24'b0;
    vif.in_wr_en = 1'b0;

    forever begin
      @(negedge vif.clock)
      begin
        if (vif.in_full == 1'b0) begin
          seq_item_port.get_next_item(tx);
          vif.in_din = tx.image_pixel;
          vif.in_wr_en = 1'b1;
          seq_item_port.item_done();
        end else begin
          vif.in_wr_en = 1'b0;
          vif.in_din = 24'b0;
        end
      end
    end
  endtask: drive

endclass
```

Get next transaction data from Sequencer and write to FIFO

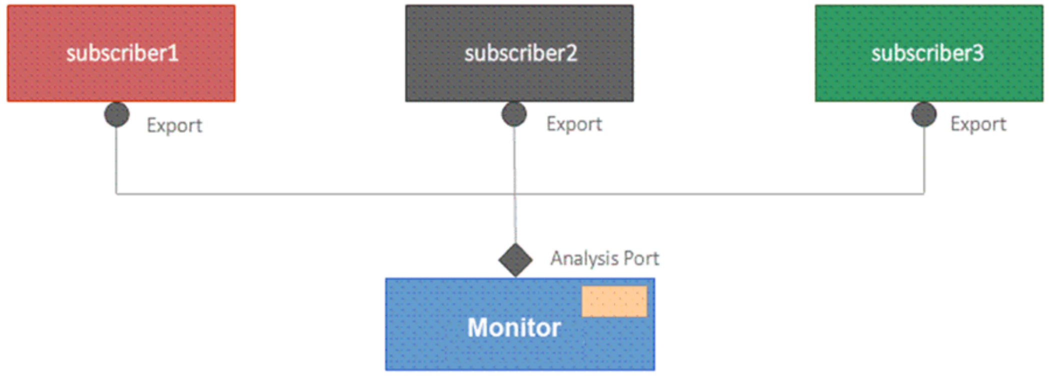
UVM MONITOR

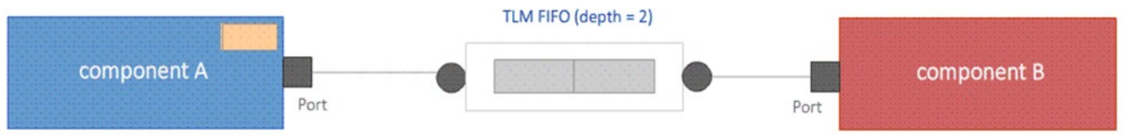
- UVM Monitor is responsible for:
  - Capturing signal data from the DUT via a virtual interface
  - Collected data used for protocol checking and coverage
  - Collected data is exported via a TLM analysis port
- Monitor translates DUT data into transaction level data objects that can be sent to other component.



TLM ANALYSIS PORTS

- ***T***ransaction ***L***evel ***M***odeling (TLM) provides a higher level of abstraction for building transaction models between components and systems.
- Data is represented as transactions (class objects) which flow in and out of different components via special ports called **TLM** interfaces.
- UVM provides a set of transaction-level communication interfaces that can be used to connect between components such that data packets can be transferred between them
  - TLM Analysis Ports are used by Monitors to pass data to other components (subscribers) and the scoreboard.
  - TLM FIFOs provide a queue for storing packets between sender and the receiver to independently operate.





UVM MONITOR – DUT INTERFACE EXAMPLE

```
// Reads data from output fifo to scoreboard
class my_uvm_monitor_output extends uvm_monitor;
    `uvm_component_utils(my_uvm_monitor_output)

    uvm_analysis_port#(my_uvm_transaction) mon_ap_output;

    virtual my_uvm_if vif;
    int out_file;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction: new

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        void'(uvm_resource_db#(virtual my_uvm_if)::read_by_name(
                .scope("ifs"), .name("vif"), .val(vif)));
        mon_ap_output = new(.name("mon_ap_output"), .parent(this));

        out_file = $fopen(IMG_OUT_NAME, "wb");
        if ( !out_file ) begin
            `uvm_fatal("MON_OUT_BUILD", $sformatf("Failed to open
                output file %s...", IMG_OUT_NAME));
        end
    endfunction: build_phase
```

BMP Header data is previously stored to the DB and now retrieved for file write

```
virtual task run_phase(uvm_phase phase);
    int n_bytes;
    logic [0:BMP_HEADER_SIZE-1][7:0] bmp_header;
    my_uvm_transaction tx_out;

    // wait for reset
    @(posedge vif.reset)
    @(negedge vif.reset)

    tx_out = my_uvm_transaction::type_id::create(
            .name("tx_out"), .contxt(get_full_name()));

    // get the stored BMP header as packed array
    if ( !uvm_config_db#(logic[0:BMP_HEADER_SIZE-1][7:0])
        ::get(null, "*", "bmp_header", bmp_header) )
    begin
        `uvm_fatal("MON_OUT_RUN", $sformatf("Failed to
            retrieve BMP header data for file %s...",
            IMG_CMP_NAME));
    end

    // copy the BMP header to the output file
    for (int i = 0; i < BMP_HEADER_SIZE; i++) begin
        $fwrite(out_file, "%c", bmp_header[i]);
    end

    vif.out_rd_en = 1'b0;
```

```
forever begin
  @(negedge vif.clock)
  begin
    if (vif.out_empty == 1'b0) begin
      $fwrite(out_file, "%c%c%c",
        vif.out_dout, vif.out_dout, vif.out_dout);
      tx_out.image_pixel = {3{vif.out_dout}};
      mon_ap_output.write(tx_out);
      vif.out_rd_en = 1'b1;
    end else begin
      vif.out_rd_en = 1'b0;
    end
  end
end
endtask: run_phase

virtual function void final_phase(uvm_phase phase);
  super.final_phase(phase);
  `uvm_info("MON_OUT_FINAL", $sformatf(
    "Closing file %s...", IMG_OUT_NAME), UVM_LOW);
  $fclose(out_file);
endfunction: final_phase

endclass: my_uvm_monitor_output
```

Read from FIFO, write to monitor

UVM MONITOR – COMPARE DATA EXAMPLE

```
// Reads data from compare file to scoreboard
class my_uvm_monitor_compare extends uvm_monitor;
  `uvm_component_utils(my_uvm_monitor_compare)

  uvm_analysis_port#(my_uvm_transaction) mon_ap_compare;
  virtual my_uvm_if vif;
  int cmp_file, n_bytes;
  logic [7:0] bmp_header [0:BMP_HEADER_SIZE-1];

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(virtual my_uvm_if)::read_by_name(
        .scope("ifs"), .name("vif"), .val(vif)));
    mon_ap_compare = new(.name("mon_ap_compare"), .parent(this));

    cmp_file = $fopen(IMG_CMP_NAME, "rb");
    if ( !cmp_file ) begin
      `uvm_fatal("MON_CMP_BUILD", $sformatf("Failed to open file %s...", IMG_CMP_NAME));
    end

    // store the BMP header as packed array
    n_bytes = $fread(bmp_header, cmp_file, 0, BMP_HEADER_SIZE);
    uvm_config_db#(logic[0:BMP_HEADER_SIZE-1][7:0])::
      set(null, "*", "bmp_header", {>>8{bmp_header}});
    endfunction: build_phase
```

BMP Header data is stored as packed data to the DB and so it may be retrieved for file writing

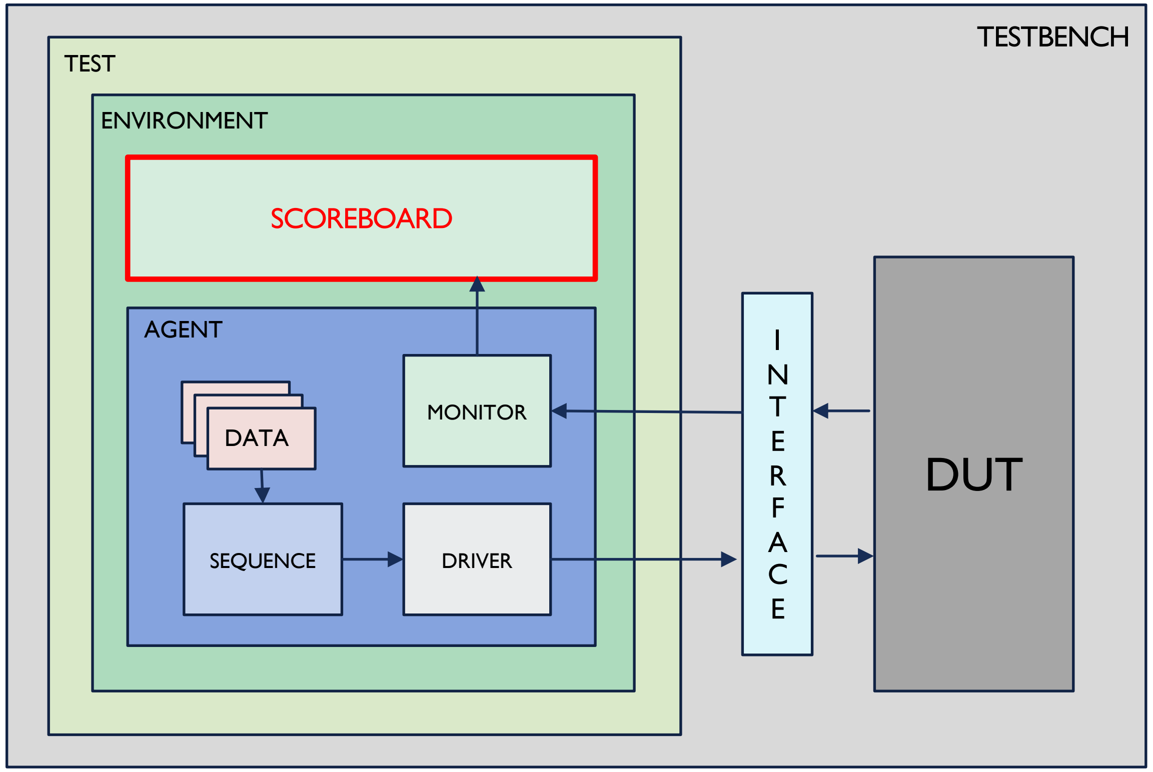
```
virtual task run_phase(uvm_phase phase);
  int n_bytes=0, i=0;
  logic [23:0] pixel;
  my_uvm_transaction tx_cmp;

  // wait for reset
  @(posedge vif.reset)
  @(negedge vif.reset)

  tx_cmp = my_uvm_transaction::type_id::create(.name("tx_cmp"), .contxt(get_full_name()));

  // syncronize file read with fifo data
  while ( !$feof(cmp_file) ) begin
    @(negedge vif.clock)
    begin
      if ( vif.out_empty == 1'b0 ) begin
        n_bytes = $fread(pixel, cmp_file, BMP_HEADER_SIZE+i, BYTES_PER_PIXEL);
        tx_cmp.image_pixel = pixel;
        mon_ap_compare.write(tx_cmp);
        i += BYTES_PER_PIXEL;
      end
    end
  end
endtask: run_phase

virtual function void final_phase(uvm_phase phase);
  super.final_phase(phase);
  `uvm_info("MON_CMP_FINAL", $sformatf("Closing file %s...", IMG_CMP_NAME), UVM_LOW);
  $fclose(cmp_file);
endfunction: final_phase

endclass: my_uvm_monitor_compare
```

Read data from file and create a transaction to broadcast to subscribers

Close file after reading is completed

UVM SCOREBOARD

- UVM **scoreboard** is a verification component that contains checkers and verifies the functionality of a design.
- It receives transaction level objects captured from the interfaces of a DUT via TLM analysis ports
- Scoreboard receives data packets of the expected value and actual DUT data, and compares to see if they match.
- 2 ways to pass actual data:
  - Allow Monitor to read & write expected values
  - Create Passive Agents to read & write expected values



UVM SCOREBOARD EXAMPLE

```
import uvm_pkg::*;

`uvm_analysis_imp_decl(_output)
`uvm_analysis_imp_decl(_compare)

class my_uvm_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(my_uvm_scoreboard)

  uvm_analysis_export #(my_uvm_transaction) sb_export_output;
  uvm_analysis_export #(my_uvm_transaction) sb_export_compare;

  uvm_tlm_analysis_fifo #(my_uvm_transaction) output_fifo;
  uvm_tlm_analysis_fifo #(my_uvm_transaction) compare_fifo;

  my_uvm_transaction tx_out;
  my_uvm_transaction tx_cmp;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    tx_out = new("tx_out");
    tx_cmp = new("tx_cmp");
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    sb_export_output = new("sb_export_output", this);
    sb_export_compare = new("sb_export_compare", this);

    output_fifo = new("output_fifo", this);
    compare_fifo = new("compare_fifo", this);
  endfunction: build_phase
```

FIFO-based transaction queues and analysis ports created to compare the DUT data and file data

```
  virtual function void connect_phase(uvm_phase phase);
    sb_export_output.connect(output_fifo.analysis_export);
    sb_export_compare.connect(compare_fifo.analysis_export);
  endfunction: connect_phase

  virtual task run();
    forever begin
      output_fifo.get(tx_out);
      compare_fifo.get(tx_cmp);
      comparison();
    end
  endtask: run

  virtual function void comparison();
    if (tx_out.image_pixel != tx_cmp.image_pixel) begin
      `uvm_info("SB_CMP", tx_out.sprint(), UVM_LOW);
      `uvm_info("SB_CMP", tx_cmp.sprint(), UVM_LOW);
      `uvm_fatal("SB_CMP", $sformatf("Test: Failed! Expecting: %08x, Received: %08x",
           tx_cmp.image_pixel, tx_out.image_pixel))
    end
  endfunction: comparison

endclass: my_uvm_scoreboard
```

Run() function reads FIFO data from each analysis port , compares values, and reports errors.

HOW DOES THE UVM SIMULATION END?

- UVM provides an objection mechanism to allow hierarchical status communication among components which is helpful in deciding the end of test.
- There is a built-in objection for each phase, which provides a way for components and objects to synchronize their testing activity and indicate when it is safe to end the phase and, ultimately, the test end.
- A component or sequence may:
  - **raise a phase objection** at the beginning of an activity that must be completed before the phase stops
  - **drop a phase objection** at the end of that activity
- Once all of the raised objections are dropped, the phase terminates
- The simulation terminates after all objections have been dropped

```
virtual task run_phase(uvm_phase phase);
  my_uvm_sequence seq;

  // notify that run_phase has started
  // simulation ends once all objections are dropped
  phase.raise_objection(.obj(this));

  seq = my_uvm_sequence::type_id::create(.name("seq"),
                                         .contxt(get_full_name()));
  seq.start(env.agent.seqr);

  // notify that run_phase has completed
  phase.drop_objection(.obj(this));
endtask: run_phase
```

UVM PACKAGE

- Create a UVM Package that includes all of the UVM files.
- DUT files should not be included.
- It's a good idea to also create a global file to include all global variables declarations.

```
package my_uvm_package;

import uvm_pkg::*;

// UVM files
`include "uvm_macros.svh"
`include "my_uvm_globals.sv"
`include "my_uvm_sequencer.sv"
`include "my_uvm_monitor.sv"
`include "my_uvm_driver.sv"
`include "my_uvm_agent.sv"
`include "my_uvm_scoreboard.sv"
`include "my_uvm_config.sv"
`include "my_uvm_env.sv"
`include "my_uvm_test.sv"

endpackage
```

SIMULATING UVM MODELS

```
setenv LMC_TIMEUNIT -9
vlib work
vmap work work

# grayscale architecture
vlog -work work "../sv/fifo.sv"
vlog -work work "../sv/grayscale.sv"
vlog -work work "../sv/grayscale_top.sv"

# uvm library
vlog -work work +incdir+$env(UVM_HOME)/src $env(UVM_HOME)/src/uvm.sv
vlog -work work +incdir+$env(UVM_HOME)/src $env(UVM_HOME)/src/uvm_macros.svh
vlog -work work +incdir+$env(UVM_HOME)/src $env(MTI_HOME)/verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv

# uvm package
vlog -work work +incdir+$env(UVM_HOME)/src "../uvm/my_uvm_pkg.sv"
vlog -work work +incdir+$env(UVM_HOME)/src "../uvm/my_uvm_tb.sv"

# start uvm simulation
vsim -classdebug -voptargs=+acc +notimingchecks -L work work.my_uvm_tb -wlf my_uvm_tb.wlf -sv_lib lib/uvm_dpi -dpicpppath /usr/bin/gcc +incdir+$env(MTI_HOME)/verilog_src/questa_uvm_pkg-1.2/src/

# start basic simulation
#vsim -voptargs=+acc +notimingchecks -L work work.grayscale_tb -wlf grayscale_tb.wlf

do grayscale_wave.do
run -all
```

```
source /vol/eecs392/env/questasim.env

mkdir -p lib

make -f Makefile.questa dpi_lib32 LIBDIR=lib

vsim -do grayscale_sim.do
```

NEXT…

- HW #4: Edge Detection using UVM