Getting Started with UVM

Vanessa Cooper
Verification Consultant

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

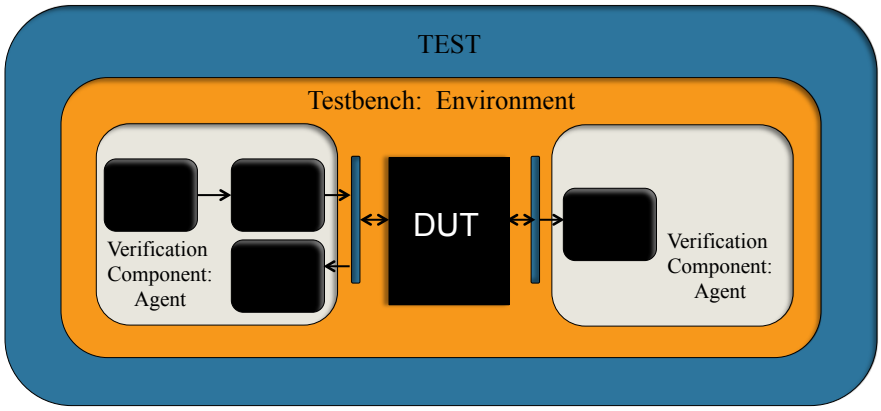
What is UVM?

- NOT a new language
- Library of base classes
- Based off previous methodologies
- Accellera standard

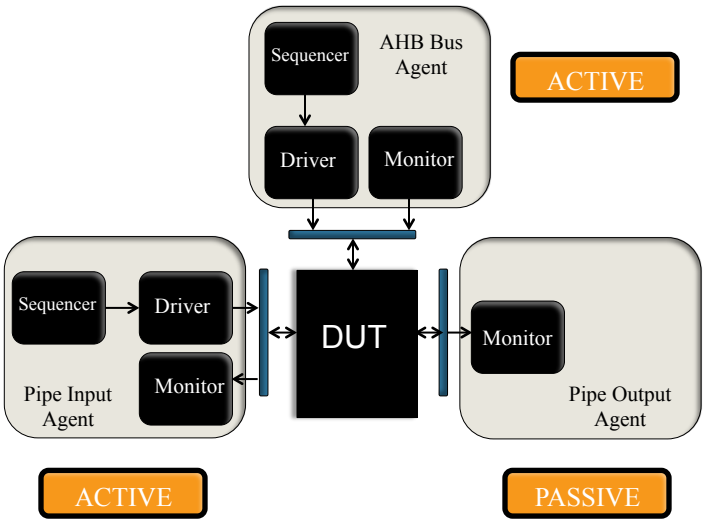
Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Testbench Architecture



Testbench Architecture



Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Phases

build_phase	Builds components top-down
connect_phase	Connects the components in the environment
end_of_elaboration_phase	Post elaboration activity
start_of_simulation_phase	Configuration of components before the simulation begins
run_phase	Test Execution
extract_phase	Collects test details after run execution
check_phase	Checks simulation results
report_phase	Reporting of simulation results

verilab

Phases

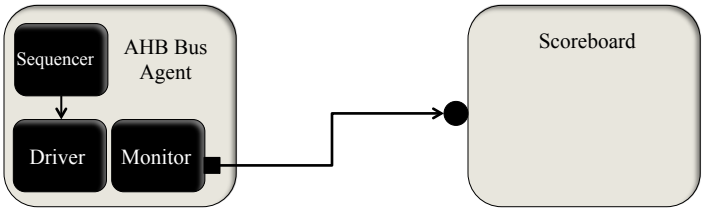
```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(is_active == UVM_ACTIVE) begin
     sequencer = pipe_sequencer::type_id::create("sequencer", this);
     driver = pipe_driver::type_id::create("driver", this);
  end
  monitor = pipe_monitor::type_id::create("monitor", this);
  `uvm_info(get_full_name( ), "Build phase complete", UVM_LOW)
endfunction: build_phase
```

- Call *super.build_phase* first
- Configure before creating
- Create the component

verilab

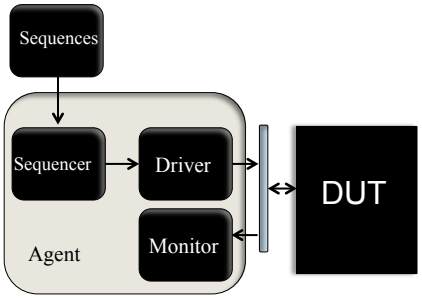
Phases

```
virtual function void connect_phase(uvm_phase phase);
   ahb_env.agent.monitor.ic_port.connect(sboard.pkts_coll.analysis_export);
   ahb_env.agent.monitor.ic_port.connect(coverage.analysis_export);
   `uvm_info(get_full_name( ), "Connect phase complete", UVM_LOW)
endfunction: connect_phase
```



verilab

Phases



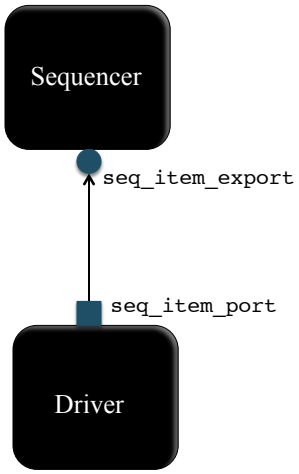

```
virtual task run_phase(uvm_phase phase);
   fork
      ...
      get_and_drive( );
      ...
   join
endtask: run_phase
```

verilab

Phases

```
virtual task get_and_drive( );
   forever begin
      @(posedge vif.rst_n);
      while(vif.rst_n != 1'b0) begin
         seq_item_port.get_next_item(req);
         drive_packet(req);
         seq_item_port.item_done( );
      end
   end
endtask: get_and_drive
```

```
virtual task drive_packet(data_packet pkt);
   @(posedge vif.clk);
   vif.data = pkt.data;
endtask: drive_packet
```



verilab

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Sequence Items

```
class data_packet extends uvm_sequence_item;
   rand bit [15:0] data;

   `uvm_object_utils_begin(data_packet)
      `uvm_field_int(data, UVM_DEFAULT)
   `uvm_object_utils_end

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

verilab

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Macros and the Factory

- Macros
  - Utility macros registers the class with the factory and gives access to the create method: *`uvm_object_utils and `uvm_component_utils*
  - Field automation macros give access to common functions such as copy( ) and clone( ) : *`uvm_field_int*
- Factory
  - Substitute components
  - Defer to run-time for object allocation

Macros and the Factory

```
`uvm_object_utils_begin(data_packet)
   `uvm_field_int(data, UVM_DEFAULT)
`uvm_object_utils_end
```

Macros

```
class monitor extends uvm_monitor;
   data_packet pkt;
   ...
   pkt = new("pkt");
   pkt = data_packet::type_id::create("pkt", this);
   ...
endclass
```

Factory

Macros and the Factory

All

```
object::type_id::set_type_override(derived_obj::get_type( ));

data_packet::type_id::set_type_override(short_pkt::get_type( ));
```

Inst

```
object::type_id::set_inst_override(derived_obj::get_type( ),
"path");

data_packet::type_id::set_inst_override(short_pkt::get_type( ),
"env.agent.*");
```

verilab

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Configuration Database

```
uvm_config_db#(TYPE)::set(uvm_root::get( ), "*.path", "label", value);
```

"dut_intf"	vif
"retry_count"	rty_cnt
"my_env_cfg"	env_cfg

```
uvm_config_db#(TYPE)::get(this, "", "label", value);
```

"retry_count"	rty_cnt

verilab

Configuration Database

Top

```
dut_if vif(.clk(clk),.rst_n(rst_n));

initial begin
   ...
   uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                       "dut_intf", vif);
   ...
end
```

```
static function void set(uvm_component cntxt,
                         string        inst_name,
                         string        field_name,
                         T             value)
```

verilab

Configuration Database

```
uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                    vif);
```

Monitor

```
static function bit get(    uvm_component cntxt,
                            string        inst_name,
                            string        field_name,
                            ref T         value)
```

verilab

Configuration Database

```
uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                    vif);
```

Monitor

```
if(!uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                        vif))
   `uvm_fatal("NOVIF", {"virtual interface must be set for:
                        ", get_full_name( ), ".vif"});
```

verilab

Configuration Database

```
class ahb_agent extends uvm_agent;
   uvm_active_passive_enum is_active = UVM_ACTIVE

   `uvm_component_utils_begin(ahb_agent)
      `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
   `uvm_component_utils_end

   ...
endclass
```

Configure in env in test

```
uvm_config_db#(int)::set(this, "env.agent", "is_active", UVM_PASSIVE);
```

Create env in test

```
env = ahb_env::type_id::create("env", this);
```

Create agent in env

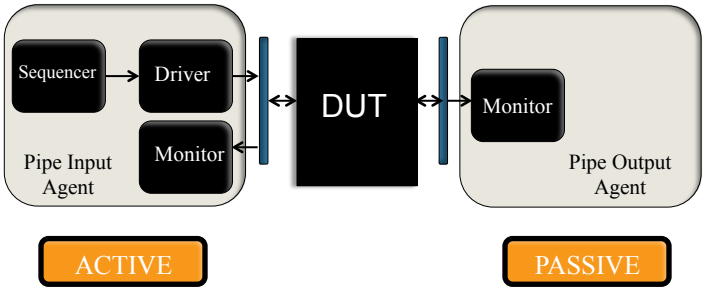
```
agent = ahb_agent::type_id::create("agent", this);
```

verilab

Configuration Database

PROBLEM

- Reuse Monitor and Interface for Input and Output
- Ensure Monitor selects correct Interface



verilab

Configuration Database

Top

```
dut_if dut_ivif(.clk(clk), .rst_n(rst_n));
dut_if dut_ovif(.clk(clk), .rst_n(rst_n));

initial begin
   ...
   uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                       "input_dut_intf",dut_ivif);

   uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                      "output_dut_intf", dut_ovif);
   ...
end
```


Configuration Database

```
class dut_monitor extends uvm_monitor;
   virtual dut_if vif;
   string monitor_intf;
   ...
endclass: dut_monitor
```

Monitor

```
uvm_config_db#(string)::set(this,"input_env.agent.monitor",
                            "monitor_intf", "input_dut_intf");
```

ENV

```
uvm_config_db#(string)::set(this, "output_env.agent.monitor",
                            "monitor_intf",  "output_dut_intf");
```

Configuration Database

"*"	"input_dut_intf"	dut_ivif
"*"	"output_dut_intf"	dut_ovif
"input_env.agent.monitor"	"monitor_intf"	"input_dut_intf"
"output_env.agent.monitor"	"monitor_intf"	"output_dut_intf"

verilab

Configuration Database

```
class dut_monitor extends uvm_monitor;
   virtual dut_if vif;
   string monitor_intf;

   uvm_config_db#(string)::get(this, "", "monitor_intf",
                                          monitor_intf);

   uvm_config_db#(virtual dut_if)::get(this, "",
                                       monitor_intf, vif);
   ...
endclass: dut_monitor
```

Monitor

"input_env.agent.monitor"	"monitor_intf"	"input_dut_intf"
"*"	"input_dut_intf"	dut_ivif

verilab

Configuration Database

```
uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                    "input_dut_intf",dut_ivif);
```

Top

```
uvm_config_db#(virtual dut_if)::set(uvm_root::get( ),
        "*.dut_agent.monitor", "input_dut_intf",dut_ivif);
```

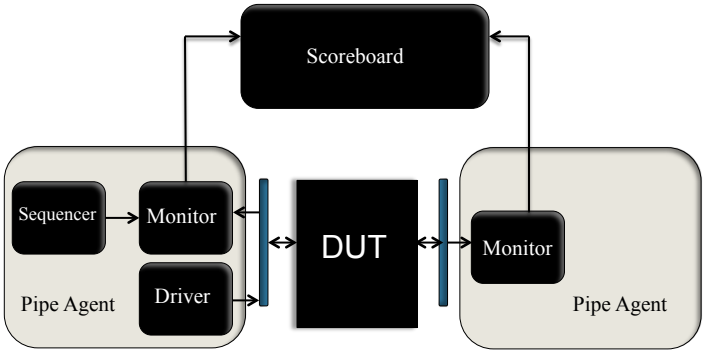

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Connecting a Scoreboard

PROBLEM

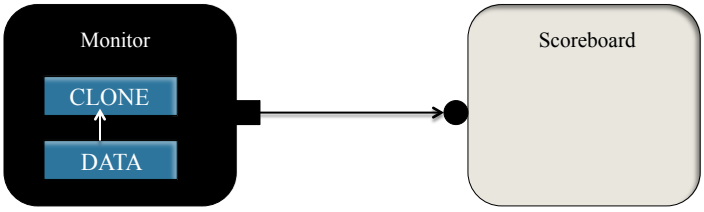
- What is a simple way to connect the monitors to the scoreboard?



Connecting a Scoreboard

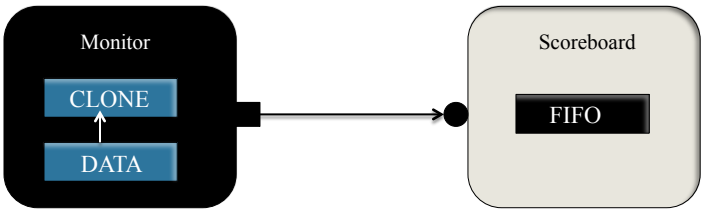
```
class dut_monitor extends uvm_monitor;
   ...
   uvm_analysis_port #(data_packet) items_collected_port;
   data_packet data_collected;
   data_packet data_clone;
   ...

endclass: dut_monitor
```



Connecting a Scoreboard

```
class dut_monitor extends uvm_monitor;
   ...
   virtual task collect_packets;
      ...
      $cast(data_clone, data_collected.clone( ));
      items_collected_port.write(data_clone);
   endtask: collect_packets
   ...
endclass: dut_monitor
```



Connecting a Scoreboard

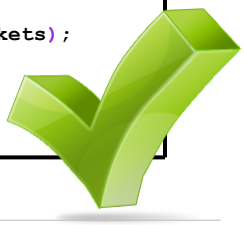
```
class dut_scoreboard extends uvm_scoreboard;
   ...
   uvm_tlm_analysis_fifo #(data_packet) input_packets_collected;
   uvm_tlm_analysis_fifo #(data_packet) output_packets_collected;
   ...

   virtual task watcher( );
      forever begin
         @(posedge top.clk);
         if(input_packets_collected.used( ) != 0) begin
            ...
         end
      end
   endtask: watcher
endclass: dut_scoreboard
```

Connecting a Scoreboard

```
virtual task watcher( );
   forever begin
      @(posedge top.clk);
      if(input_packets_collected.used( ) != 0) begin
         ...
      end
   end
endtask: watcher
```

```
virtual task watcher( );
   forever begin
      input_packets_collected.get(input_packets);
         ...
      end
   end
endtask: watcher
```

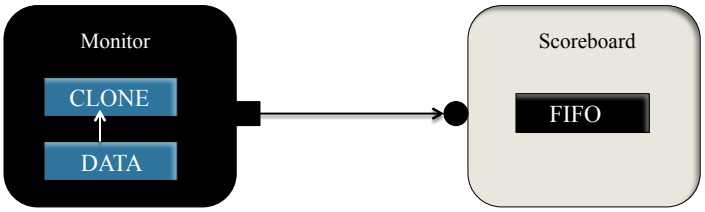


Connecting a Scoreboard

```
input_env.agent.monitor.items_collected_port.connect
      (scoreboard.input_packets_collected.analysis_export);

ENV

output_env.agent.monitor.items_collected_port.connect
      (scoreboard.output_packets_collected.analysis_export);
```



Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

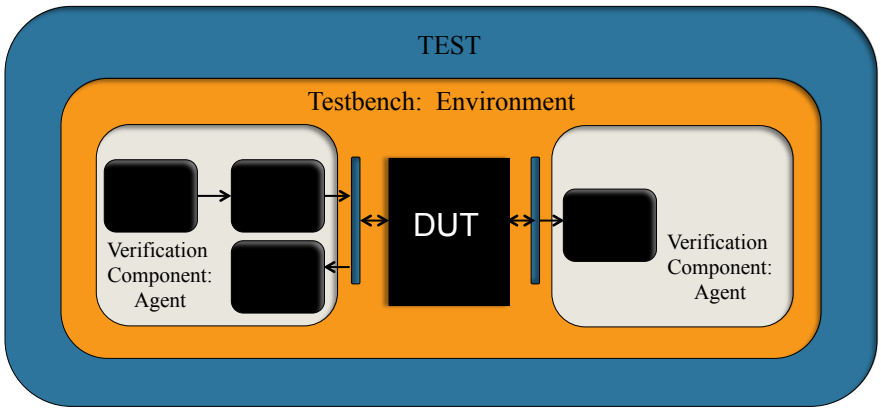
verilab

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Test Structure



verilab

Test Structure

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

   dut_env            env;
   dut_env_cfg        env_cfg;

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

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env_cfg = dut_env_cfg::type_id::create("env_cfg");

      uvm_config_db#(dut_env_cfg)::set(this, "*", "dut_env_cfg",
                                                  env_cfg);

      env = dut_env::type_id::create("env", this);
   endfunction

endclass: base_test
```

verilab

Test Structure

```
class test extends base_test;
   `uvm_component_utils(test)
   random_sequence seq;

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

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

   virtual task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      seq = random_sequence::type_id::create("seq");
      seq.start(env.agent.sequencer);
      phase.drop_objection(this);
   endtask
endclass: test
```

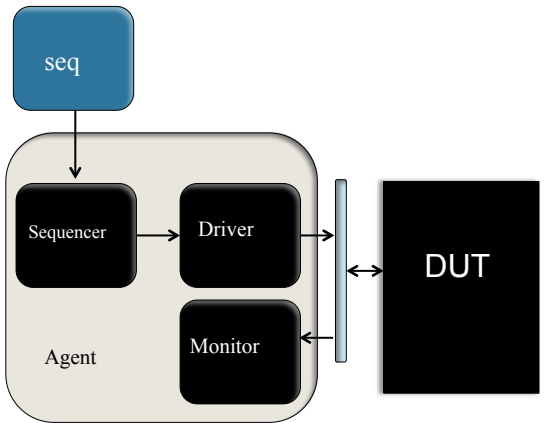
verilab

Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Sequences



verilab

Sequences

```
class random_sequence extends uvm_sequence #(data_packet);
   `uvm_object_utils(random_sequence)

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

   virtual task body( );
      `uvm_do(req);
   endtask

endclass: random_sequence
```

- *req* is a member of uvm_sequence that is parameterized as *data_packet*
- *`uvm_do* creates the transaction, randomizes it, and sends it to the sequencer

verilab

Sequences

```
class constrained_seq extends uvm_sequence #(data_packet);
   `uvm_object_utils(constrained_seq)

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

   virtual task body( );
     `uvm_do_with(req, {req.data < 16'ha;})
   endtask

endclass: constrained_seq
```

- *`uvm_do_with* further constrains sequence_item members

Getting Started with UVM

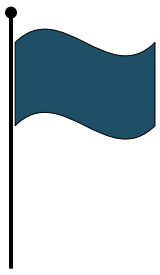
- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Objections

- Used to communicate when it is safe to end
- Components or Sequences can raise or drop objections
- Objections must be raised at start of a phase
- Phase persists until all objections are dropped

Objections

```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    seq = random_sequence::type_id::create("seq");
    seq.start(env.agent.sequencer);
    phase.drop_objection(this);
endtask
```



Getting Started with UVM

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Execution

```
module top;
   ...
   dut_if intf(.clk(clk), .rst_n(rst_n));

   initial begin
      uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                          "dut_intf", intf);

      run_test( );
   end
endmodule: top
```

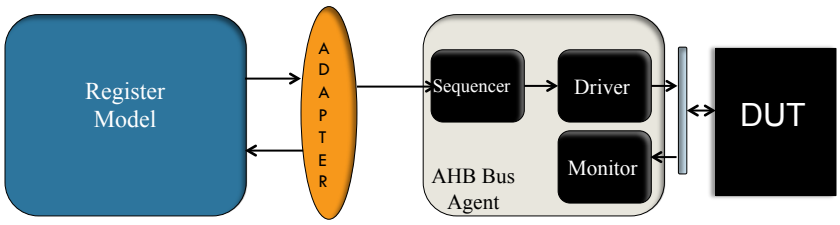
+UVM_TESTNAME="test1"

Getting Started with UVM

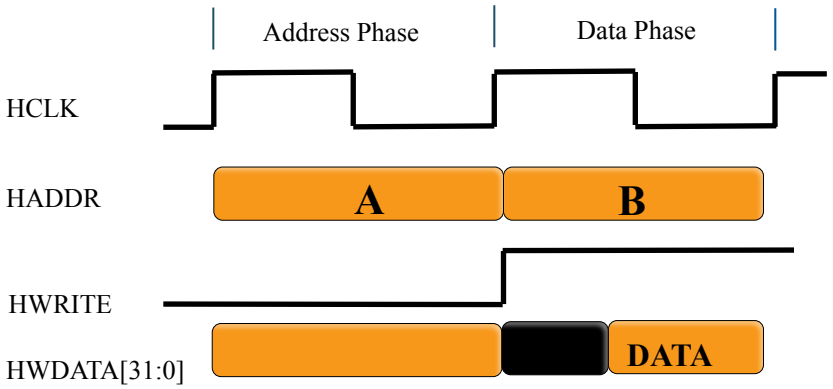
- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

Register Model

PROBLEM
- Use the Register Model with the pipeline AHB bus
- Capture read data accurately

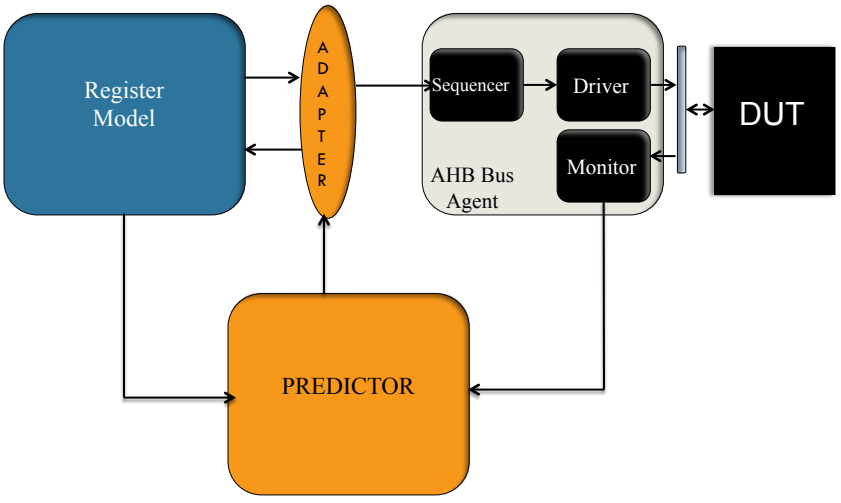


Register Model



verilab

Register Model



verilab

Register Model

- build_phase
  - Create the predictor with the bus uvm_sequence_item parameter in your env
- connect_phase
  - Set the predictor map to the register model map
  - Set the predictor adapter to the register adapter
  - Connect the predictor to the monitor

Register Model

ENV

Declare

```
uvm_reg_predictor#(ahb_transfer) reg_predictor;
```

Create

```
reg_predictor = uvm_reg_predictor#(ahb_transfer)::
                type_id::create("reg_predictor", this);
```

Map

```
reg_predictor.map = master_regs.default_map;
reg_predictor.adapter = reg2ahb_master;
```

Connect

```
ahb_env.agent.monitor.item_collected_port.
                        connect(reg_predictor.bus_in);
```

Register Model

```
master_regs.default_map.set_auto_predict(0);
```

Implicit **Explicit** **Passive**

verilab

Getting Started with UVM

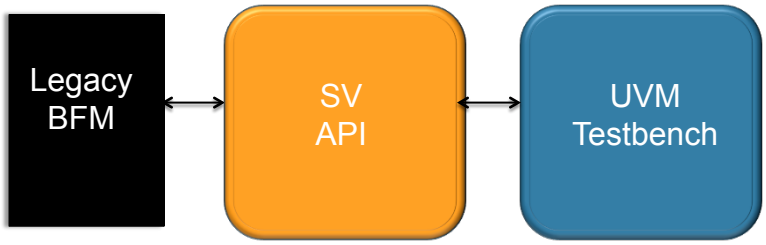
- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code

verilab

Hooking to Legacy Code

- Don't Touch the BFM
- Reusable access to items in the instantiated hierarchy and module items
- Informing the SV testbench about the legacy BFM

Hooking to Legacy Code



Hooking to Legacy Code

```
module SAMPLE_BFM(ports to connect to DUT signals);
   logic L;
   task T( ); ... endtask
endmodule
```

- Need access to signal *L* and task *T*

Hooking to Legacy Code

- Define a API
  - Access to the UVM testbench
  - Access to the legacy BFM
- Base class with pure virtual methods
  - Define in a package
  - Derived class will implement functions

Hooking to Legacy Code

```
package SAMPLE_BFM_uvm_wrapper;
   import uvm_pkg::*;

   virtual class SAMPLE_BFM_access extends uvm_object;
      function new(string name);
         super.new(name);
      endfunction

      pure virtual function logic get_L( );
      pure virtual task run_T( );
   endclass
endpackage
```


Hooking to Legacy Code

- Derive a child class and implement functions
- Physical hook to Legacy BFM
- Create class in interface

Hooking to Legacy Code

```
interface SAMPLE_BFM_uvm_gasket;
   import SAMPLE_BFM_uvm_wrapper::*;

   //This class lives in the hierarchy so it can access
   //signals and task by XMR.
   class concrete_access extends SAMPLE_BFM_access;
      function new(string name);
         super.new(name);
      endfunction

      function logic get_L( );
         return SAMPLE_BFM.L;
      endfunction

      task run_T( );
         SAMPLE_BFM.T( );
      endtask

   endclass
   ...
```

Upwards XMR

Hooking to Legacy Code

- Assume that the interfaces will be instantiated within an instance of the legacy BFM.
- Any XMR in this interface that begins with the module name SAMPLE_BFM will become and upwards XMR.

Hooking to Legacy Code

```
   ...
   concrete_access ACCESS;

    //Provide a function that will return the
    //concrete_access object, constructing it on demand.
    function automatic SAMPLE_BFM_access get_access( );
       if (ACCESS == null)
          ACCESS = new($sformatf("%m.ACCESS"));
       return ACCESS;
    endfunction
 endinterface
```

- signal and task access straightforward since embedded class is in the scope of the interface
- get_access( ) provides easy access to embedded UVM object

Hooking to Legacy Code

Create an instance of the interface in the hierarchy

```
bind SAMPLE_BFM SAMPLE_BFM_uvm_gasket uvm_gasket_instance( );
```

Reference probe class and pass to UVM testbench

```
module top;
   import uvm_pkg::*;

   initial begin
      uvm_config_db#(SAMPLE_BFM_uvm_wrapper::SAMPLE_BFM_access)::
      set(uvm_root::get( ), "*",
      "access_gasket", BFM_inst.uvm_gasket_instance.get_access( ));

      run_test( );
   end
endmodule: top
```

Questions?

- What is UVM?
- Building a Testbench
  - Testbench Architecture
  - Phases
  - Interfaces and Sequence Items
  - Macros and the Factory
  - Configuration Database
  - Connecting a Scoreboard
- Creating Tests
  - Test Structure
  - Sequences
  - Objections
  - Execution
- Register Model
- Hooking to Legacy Code