Adopting the SystemVerilog Universal Verification Methodology (UVM)

An in-depth examination of what's inside a UVM testbench, and what is needed to be successful at adopting UVM

by Stuart Sutherland

SUTHERLAND HDL

Training Engineers to be SystemVerilog wizards

www.sutherland-hdl.com

Adopting the SystemVerilog Universal Verification Methodology (UVM)

Sutherland HDL Incorporated
22805 SW 92nd Place
Tualatin, OR 97062 USA

phone: +1-503-692-0898
e-mail: info@sutherland-hdl.com
web: www.sutherland-hdl.com

printed 26 January 2016

3

Seminar Objectives...

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- This seminar will:
  - Discuss the benefits of UVM (and some disadvantages, too)
  - Provide an in-depth overview of what is inside a UVM testbench
  - Examine what it takes to adopt UVM in your next project
- The presentation assumes:
  - You are familiar with the requirements of design verification
  - You are familiar with the SystemVerilog language
    - At least enough to recognize the general idea of code examples
- And the "hidden agenda" – we want you to know that …
  - Sutherland HDL is in the business of training engineers to be

SystemVerilog and UVM Wizards!

4

Primary Goals of the UVM Verification Methodology

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- Some of the goals of a standard verification methodology are:
  - Reuse, Reuse, Reuse …
    - Configurable test environments that can be used for many tests
    - Reusable testbench components from project to project
  - Interoperability
    - Verification that works with many types of tools from many vendors
  - Verification Intellectual Property (VIP) models
    - In-house verification code of components that make up a design
    - Commercial verification code for commercial components
  - Separate *stimulus generation* from *stimulus delivery* to the DUT
    - The verification team can develop verification code in parallel
  - Best practices
    - A consistent way of using SystemVerilog at all companies
  - And the reason no one talks about… Sell More Verification IP
    - The major EDA companies make a lot of money selling VIP models

5

In an effort to create a "universal methodology" that would be all things for all designs, the engineers on the UVM standards committee drew from existing methodologies, and created…

6

UVM…
an Ugly Vicious Monster!

UVM is very complex because it is intended to work with any type of design in the entire universe

The complexity of UVM can make it difficult to adopt

7

Good training and expert help can change the Ugly Vicious Monster into a…

Useful Verification Methodology

I found your design bug… Its right here!

```
module controller (...);
   enum {WAIT, LOAD, STORE} State, NextState;
   always @(posedge clock or posedge resetN)
     if (!resetN) State <= WAIT;
     else State <= NextState;

   always_comb
     case (State)
       WAIT:   NextState = LOAD;
       LOAD:   NextState = STORE;
       STORE:  NextState = WAIT;
     endcase
   ...
endmodule
```

8

UVM Origins

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- UVM was created by an organization called "Accellera"
  - A non-profit "think tank" that creates new EDA standards
  - Comprises EDA companies and companies creating electronic chips
- UVM is a summation of verification knowledge and experience
  - 2000 – vAdvisor (Verification Advisor) by Verisity (acquired by Cadence)
  - 2002 – eRM (*e* Re-use Methodology) by Verisity (acquired by Cadence)
  - 2003 – RVM (VERA Reuse Verification Methodology) by Synopsys
  - 2006 – VMM (SystemVerilog Verification Methodology Manual) Synopsys
  - 2006 – AVM (SV Advanced Verification Methodology) by Mentor Graphics
  - 2007 – URM (SV Universal Re-use Methodology) by Cadence
  - 2008 – OVM (SV Open Verification Methodology) by Cadence & Mentor
  - May 2010 – UVM 1.0EA ("early adopter") – very preliminary version
  - Feb 2011 – UVM 1.0 – *major changes* to 1.0EA, still preliminary
  - Jun 2011 – UVM 1.1 – *major changes* to 1.0, stable for 1.1a–1.1d
  - Jun 2014 – UVM 1.2 – *updates* and *major changes* to 1.1
  - 2015/16? – IEEE 1800.2 – anticipate *major changes* to 1.2

UVM draws from all of these!

Rapid evolution of a standard with backward compatibility issues

9

What UVM Provides

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- UVM is SystemVerilog source code that provides:
  - A library of base classes
    - For coding testbench components (drivers, monitors, scoreboards…)
  - A factory
    - For constructing objects and substituting objects
  - Verification phases
    - For synchronizing concurrent processes
  - A reporting mechanism
    - For a consistent way of printing and logging results
  - Transaction-Level Modeling (TLM)
    - For communication between verification components
  - Macros
    - To semi-automate generation of required UVM code

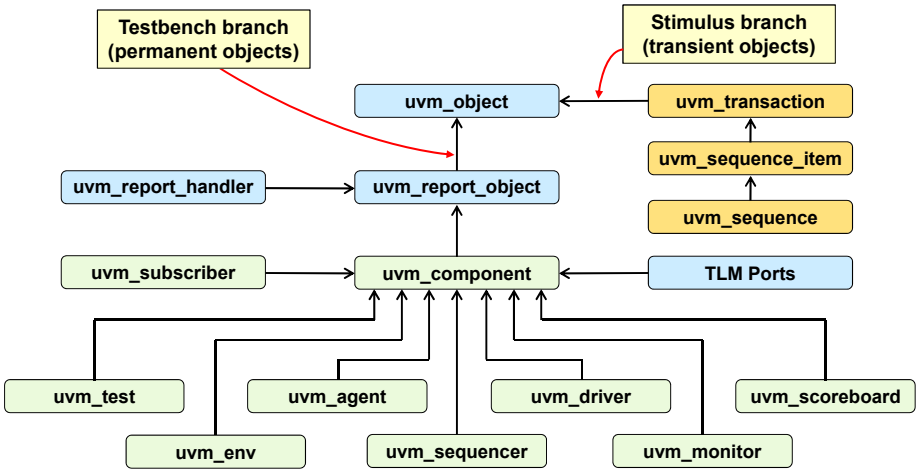
UVM requires advanced-level SystemVerilog and OOP programming skills!

10

UVM Class Hierarchy (“ is a” relationships)

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The UVM library provides a set of base classes
  - Your code will be derived from one of these base classes

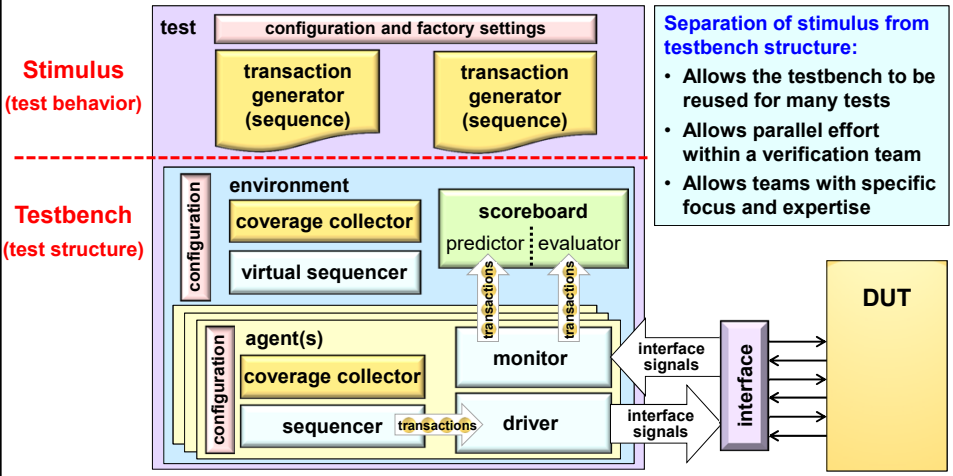


11

UVM Testbench Hierarchy (" has a" relationships)

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A test class has sequence objects and an environment object
- An environment class has component objects to talk to the DUT



Separation of stimulus from testbench structure:
- Allows the testbench to be reused for many tests
- Allows parallel effort within a verification team
- Allows teams with specific focus and expertise

12

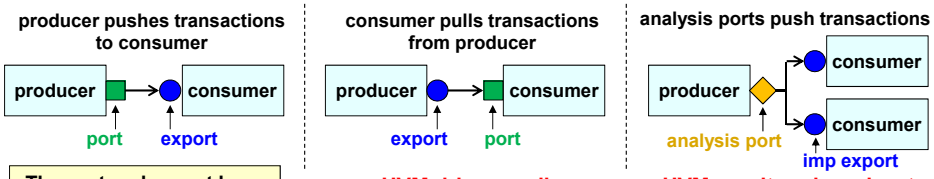
TLM Communication

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

TLM ports pass handles of sequence_item objects

UVM also implements the TLM 2.0 standard

- UVM uses the SystemC TLM 1.0 *"Transaction Level Modeling"* standard for communication between components
  - A "port" or "analysis port" specifies a set of communication methods
  - An "export" or "imp export" implements the port's methods



The port and export have a user-defined name, similar to Verilog ports

UVM drivers pull transactions from a sequencer

UVM monitors broadcast transactions to scoreboards and coverage collectors

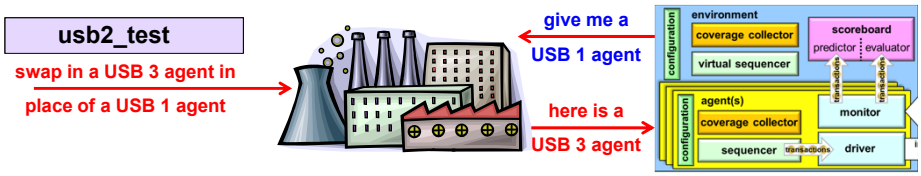
- A port must be paired with *exactly one* export (one-to-one)
- An analysis port is paired with *zero or more* imp exports (one-to-many)

13

The UVM Factory

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The UVM factory is used to build the UVM object hierarchy
  - Constructs the class objects
    - A factory create() method is used instead of directly calling new()
  - Allows tests to swap out testbench components for specific tests
    - Can change drivers, scoreboards or other components
    - Can change method behavior, add constraints, etc.



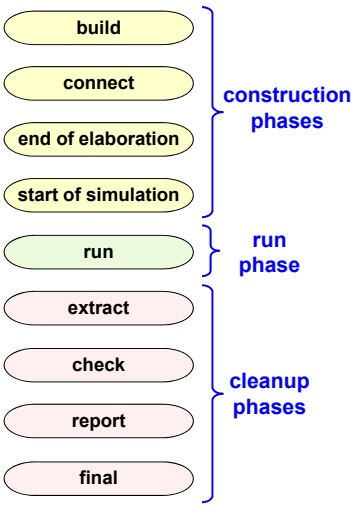
A factory is an advanced-level, complex Object Oriented programming technique – with UVM, the factory is provided in the base classes – your just need to use it!

14

UVM Phases

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- UVM components synchronize with each other using UVM phases



- The *construction phases* are where the testbench is configured and constructed
  - All construction phases execute in zero time and at simulation time zero
- The *run phase* is where simulation time is consumed in running the tests
  - The run phase can be further subdivided into several phases (later slide)
- The *cleanup phases* are where the results of the tests are collected and reported
  - All cleanup phases execute in zero time

15

UVM Phase Tasks and Functions

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- Each UVM component can start activity in any of the phases

```
class my_agent extends uvm_agent;
  ...
  function void build_phase(...);
    ... // construct components
  endfunction

  function void connect_phase(...);
    ... // connect components
  endfunction
endclass: my_agent
```

```
class my_driver extends uvm_driver;
  ...
  task run_phase(...);
    ... // get transaction & drive
  endfunction
endclass: my_driver
```

```
class my_sb extends uvm_scoreboard;
  ...
  function void build_phase(...);
    ... // construct components
  endfunction

  function void connect_phase(...);
    ... // connect components
  endfunction

  task run_phase(...);
    ... // evaluate dut outputs
  endtask

  function void report_phase(...);
    ... // report the pass/fail score
  endfunction
endclass: my_sb
```

Each phase does not end until all activity for that phase type has completed in every component (e.g.: *all* build phases must complete before any connect phase can start)

16

Who Invokes Phase Methods?

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

"Pay no attention to that man behind the curtain!"

This famous quote from the *The Wizard of OZ* movie (1939) applies to UVM!

A virtual "wizard" behind the scenes takes care of running the UVM testbench – all you need to do is tell the wizard which test to run

- The top-level module invokes a "`run_test()`" method
  - This method is "the wizard behind the curtain" that controls the UVM execution, including invoking the phase methods in order

17

Objection Flags Control How Long the Run Phase Actually Runs

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The UVM run phase:
  - Automatically starts running at simulation time 0, after the construction phases have completed
  - Continues to run as long as at least one `run_phase()` task "objects" to the run phase ending

```
class test1 extends uvm_test;
  ...
  task run_phase (uvm_phase phase);
    phase.raise_objection(this);
    ... // do test 1 stuff
    phase.drop_objection(this);
  endtask
endclass
```

```
class sb extends uvm_scoreboard;
  ...
  task run_phase (uvm_phase phase);
    phase.raise_objection(this);
    ... // do verification stuff
    phase.drop_objection(this);
  endtask
endclass
```

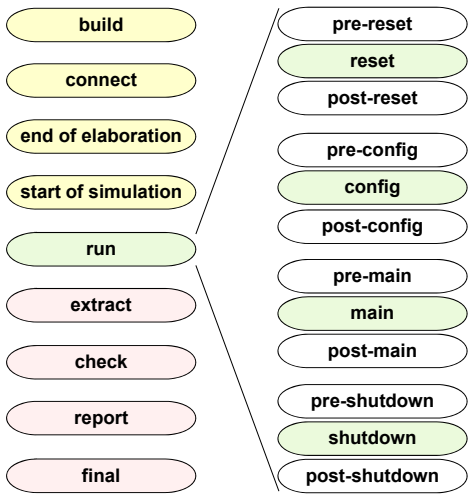
The objection flag can be abused if not careful – two GOTCHAS to watch out for:

- **Simulation will end at time 0 if no run_phase() tasks raise their objection flag**
- **Simulation will never end if a run_phase() task raises its objection flag and then locks up waiting for something to happen (or gets stuck in an infinite loop)**

18

Run Phase Divisions

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com



- **UVM 1.0 and 1.1 allow dividing the run phase into sub-phases**
  - Gives greater control over the order of complex stimulus
  - Allows "phase jumping" forward or backward within the run phase ordering

CAUTION:

- **The UVM 1.0/1.1 standards have problems with the implementation of run phase subdivisions**
- **UVM 1.2 deprecates the UVM 1.0/1.1 subdivided run phases and replaces it with a different mechanism**

RECOMMENDATION:

- **Avoid using the additional run-time phases, unless absolutely needed**

19

A Simple UVM Example

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The UVM example on the following pages verifies a simple design
  - Shows the basic structure and hierarchy of a UVM testbench
  - Shows how to connect UVM testbench components
  - Shows how to start a UVM simulation

NOTE: This seminar focuses on the "BIG PICTURE" – we will avoid getting into too much "nuts and bolts" details

- The design prints the message **"Hello World!"**

Every programming course must have an example that prints "Hello World" – It's the law!

Sutherland HDL's full UVM training course uses a more complex and realistic example than the one shown in this seminar

```
module hello_dut (input  logic clk,
                  input  logic go,
                  output logic done);

  always @(posedge clk)
    if (go) begin
      $display("\nHello World!\n");
      done = 1;
    end
    else done = 0;

endmodule
```

20

Sequence Items (aka "Transactions")

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A sequence_item contains "transaction" data and methods
  - The methods can be hand written or generated using macros

```
class hello_tx extends uvm_sequence_item;

  rand bit   go;    // input to DUT
       logic done;  // output from DUT

  // class constructor (called by factory)
  function new(string name="hello_tx");
    super.new(name);
  endfunction

  // register this class name in the factory
  `uvm_object_utils_begin(hello_tx)

    // automate implementation of methods
    `uvm_field_int(go,   UVM_DEFAULT)
    `uvm_field_int(done, UVM_DEFAULT)

  `uvm_object_utils_end
endclass: hello_tx
```

Extended from a UVM base class

Variables hold the stimulus values and DUT responses

The base class new() constructor has a name argument – SV syntax requires the child call super.new() and pass in a name (which comes from the factory)

User class names are "registered" with the factory using a macro

Macros can be used to automate generation of the standard transaction methods, such as print(), copy() and compare()

21

Sequences – The Transaction Generator

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A UVM sequence contains a body() virtual task that generates one or more sequence_item objects (transactions)
  - Derived from the uvm_sequence base class

```
class hello_sequence extends uvm_sequence #(hello_tx);
  `uvm_object_utils(tx_sequence)
  function new(string name="hello_sequence");
    super.new(name);
  endfunction
  task body();
    tx = hello_tx::type_id::create("tx");
    start_item(tx);
      tx.go = 1'b1;
    finish_item(tx);
    tx = hello_tx::type_id::create("tx");
    start_item(tx);
      tx.go = 1'b0;
    finish_item(tx);
  endtask: body
endclass: hello_sequence
```

Parameter redefinition "specializes" the sequence to a specific sequence_item type

Register class name; call super.new

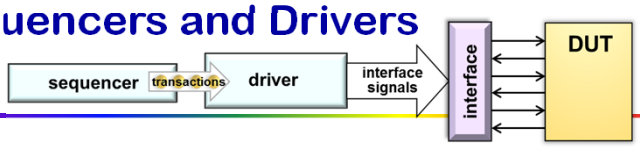
Generating a transaction involves 4 steps:

First transaction:
1) Create a transaction using the factory
2) Wait for driver to ask for a transaction
3) Set the transaction values
4) Pass the transaction to the driver

The next transaction repeats the 4 steps

22

Sequencers and Drivers



SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A sequencer routes transactions to the driver

```
class hello_sequencer extends uvm_sequencer #(hello_tx);
  ... // register this class name in the factory
endclass: hello_sequencer
```

All sequencer functionality is inherited from its base

- The driver sends the stimulus to the Device Under Test (DUT)

```
class hello_driver extends uvm_driver #(hello_tx);
  ... // register this class name in the factory
  function void build_phase(uvm_phase phase);
    if (!uvm_config_db #(virtual hello_if)::get(this, "", "vif", vif))
      `uvm_fatal("NOVIF","No virtual interface")
  endfunction: build_phase
  task run_phase(uvm_phase phase);
    hello_tx tx;
    forever begin
      @(negedge vif.cb) ;     // sync up with inactive edge of DUT clock
      seq_item_port.get(tx); // request a transaction from the sequencer
      vif.cb.go <= tx.go;    // drive the DUT with the stimulus
    end
  endtask: run_phase
endclass: hello_driver
```

Specialized to work with a specific sequence_item type

The virtual interface is retrieved from a database

The driver is a "forever" infinite loop – it runs until the test drops its objection flag

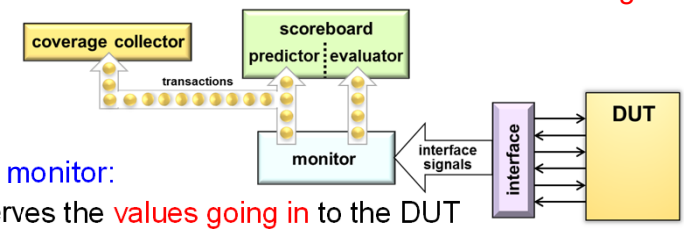
UVM drivers use an interface clocking block to avoid race conditions – enables reuse of UVM testbenches

23

Monitors

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A UVM monitor samples the DUT input and output ports
  - Sampled values are saved in transaction objects (sequence items)
  - The transactions are sent to scoreboards and coverage collectors



- A UVM monitor:
  - Observes the values going in to the DUT
    - Bundles the input values into an *input transaction*
    - Sends a copy of the transaction to a scoreboard predictor
    - Sends a copy of the transaction to a coverage collector
  - Observes the values coming out of the DUT
    - Bundles the output values into an *output transaction*
    - Sends a copy of the transaction to a scoreboard evaluator

1-24

An Example Monitor

SUTHERLAND

```
class hello_monitor extends uvm_monitor;
   ... // register this class name in the factory
   virtual hello_if vif;
   uvm_analysis_port #(hello_tx) dut_in_tx_port;
   uvm_analysis_port #(hello_tx) dut_out_tx_port;
   function void build_phase(uvm_phase phase);
     dut_in_tx_port  = new("dut_in_tx_port", this);
     dut_out_tx_port = new("dut_out_tx_port", this);
     if (!uvm_config_db #(virtual hello_if)::get(this, "", "vif", vif))
       `uvm_fatal("NOVIF","No virtual interface found!")
   endfunction: build_phase
   task run_phase(uvm_phase phase);
     hello_tx tx_in, tx_out;
     fork
       forever @(vif.cb) begin // monitor DUT inputs
         tx_in = hello_tx::type_id::create("tx_in");
         tx_in.go = vif.cb.go;
         dut_in_tx_port.write(tx_in);
       end
       forever @(vif.cb) begin // monitor DUT outputs
         tx_out = hello_tx::type_id::create("tx_out");
         tx_out.done = vif.cb.done;
         dut_out_tx_port.write(tx_out);
       end
     join
   endtask: run_phase
endclass: hello_monitor
```

Extended from uvm_monitor

Add two TLM analysis ports (one-to-many)

Get virtual interface from database

1. Create a transaction object
2. Capture input values
3. Send to scoreboard/cover
4. Do the loop again, as long as run_phase continues

1. Create a transaction object
2. Capture output values
3. Send to scoreboard
4. Do the loop again

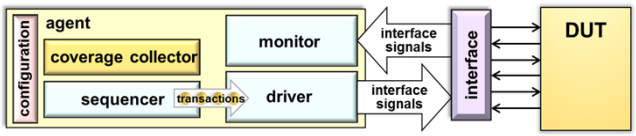
Like drivers, UVM monitors use an interface clocking block to avoid race conditions – enables reuse

25

UVM Agents

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- An agent is a reusable and configurable building block



An agent contains a sequencer, driver, monitor and coverage collector for a specific bus protocol (e.g. USB, AXI, …)

```
class hello_agent extends uvm_agent;
  ... // register this class name in the factory
  hello_sequencer sqr;
  hello_driver    drv;
  hello_monitor   mon;
  function void build_phase(uvm_phase phase);
    sqr = hello_sequencer::type_id::create("sqr", this);
    drv = hello_driver::type_id::create("drv", this);
    mon = hello_monitor::type_id::create("mon", this);
  endfunction: build_phase
  function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction: connect_phase
endclass: hello_agent
```

Agents are extended from the uvm_agent base class

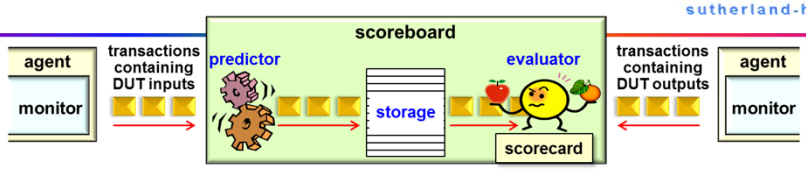
The build phase constructs the agent's components using the factory

The connect phase connects the driver to it's sequencer (see Section 4 for details)

26

Scoreboards

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com



- A UVM ***scoreboard*** receives transactions from a monitor
  - ✓ ***Predicts*** what DUT outputs should occur based on DUT inputs
    1. Receives transactions containing the values driven into the DUT
    2. Calculates what the DUT outputs should be for those inputs
       - Can be done using SystemVerilog, an abstract "reference" model (perhaps in SystemC), reading from a file, etc.
  - ✓ ***Compares*** the predicted results to the actual DUT outputs
    1. Stores the predicted results until the DUT outputs are available
    2. Receives transactions containing the values of the DUT outputs
    3. Compares the actual outputs to the predicted outputs
       - Can use the sequence_item compare() method or a more complex algorithm written in SystemVerilog code

27

An Example Scoreboard

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The scoreboard class takes care of the output verification

```
class hello_scoreboard extends uvm_scoreboard;
  ... // register this class name in the factory
  hello_predictor  predictor;
  hello_evaluator  evaluator;

  // declare TLM communication ports
  uvm_analysis_export #(hello_tx) dut_in_pass_port;
  uvm_analysis_export #(hello_tx) dut_out_pass_port;

  function void build_phase(uvm_phase phase);
    dut_in_pass_port  = new("dut_in_pass_port", this);
    dut_out_pass_port = new("dut_out_pass_port", this);
    predictor = alu_predictor::type_id::create("predictor", thi
    evaluator = alu_evaluator::type_id::create("evaluator", thi
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    dut_in_pass_port.connect(predictor.analysis_export);
    dut_out_pass_port.connect(evaluator.actual_export);
    predictor.expected_port.connect(evaluator.expected_export);
  endfunction: connect_phase
endclass: hello_scoreboard
```

Handles for the predictor and evaluator

TLM ports connect the predictor and evaluator

Construct the ports and the predictor / evaluator blocks

Connect up the predictor and evaluator

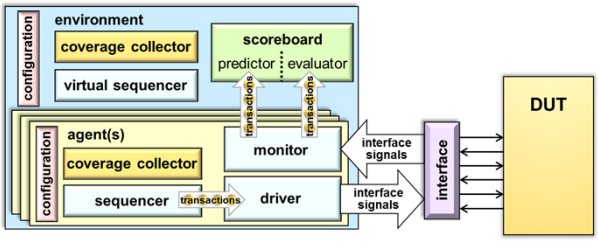
The code for the predictor and evaluator components are not shown for this example

28

The Environment is the Testbench

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The environment encapsulates agents and scoreboards:



An environment can contain any number of agents, one for each bus protocol (e.g. USB, AXI, …)

```
class hello_env extends uvm_env;
  ... // register this class name in the factory
  hello_agent agent;
  hello_scoreboard scorebd;
  function void build_phase(uvm_phase phase);
    agent = hello_agent::type_id::create("agent", this);
    scorebd = hello_scoreboard::type_id::create("scorebd", this);
  endfunction
  ...  // connect phase connects components
endclass: hello_env
```

Environments are extended from the uvm_env base class

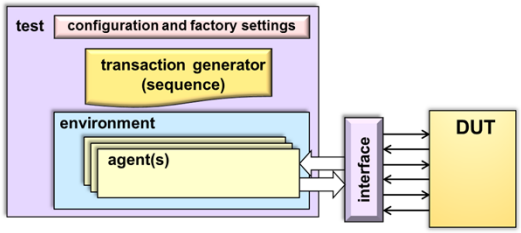
The UVM factory is used to construct UVM components

29

Tests and Stimulus

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- The test encapsulates sequences (stimulus) and the environment




```
class hello_test extends uvm_test;
  ... // register this class name in the factory
  function void build_phase(uvm_phase phase);
    env = hello_env::type_id::create("env", this);
  endfunction
  task run_phase(uvm_phase phase);
    hello_sequence seq;
    phase.raise_objection(this);
    seq = hello_sequence::type_id::create("seq", this);
    seq.start(env.agent.sqr);
    phase.drop_objection(this);
  endtask
endclass: hello_test
```

The test's run phase:
1. Creates a sequence using the factory
2. Starts the sequence
3. Objects to the run phase completing until the sequence has finished

30

Hooking Everything Up and Starting UVM

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A module connects the interface to the DUT and starts UVM

Structural Construction

Object Construction

```
module top;
  logic clk;
  hello_if io (clk);
  hello_dut dut (.clk, .go(io.go), .done(io.done));

  initial begin
    clk <= 0;
    forever #5ns clk = ~clk;
  end

  import uvm_pkg::*;
  import my_components_pkg::*;

  initial
    uvm_config_db #(virtual hello_if)::set(null, "*", "vif", io);

  initial begin
    run_test();
  end
endmodule
```

interface and DUT instances, connect interface to DUT ports

clock generator (or clock module instance)

import the UVM class library

import the testbench classes

store interface instance name in configuration database

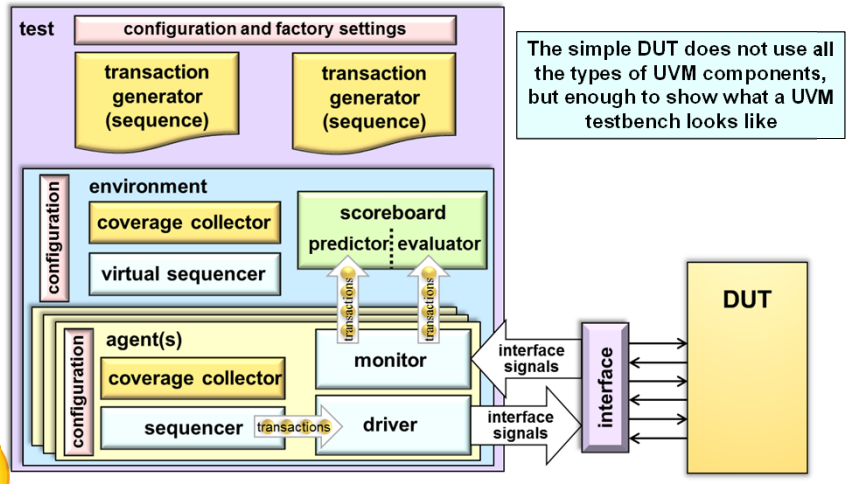
the run_test() task constructs a test class and starts the UVM phases (and other UVM magic)

The test class that run_test() creates is specified using an invocation option (e.g. +UVM_TESTNAME=hello_test)

1-31

And That's The Big Picture!

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com



There's just a few miscellaneous topics to discuss...

32

The UVM Methodology Needs a Methodology!

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- A UVM testbench can involve several dozen files
  - For tests, environments, agents, sequencers, sequences, drivers, transactions, monitors, scoreboards, predictors, configurations, …
  - For VIP source code, documentation, simulation logs, coverage, …
- A methodology is needed to manage the UVM methodology
  - Can be a home-grown approach developed within a company
  - Can use commercial tools
    - There are commercial tools, such as the Paradigm Works VerificationWorks™ that can:
      1) Create a complete UVM testbench directory structure
      2) Create skeleton classes that have all the boilerplate code filled in
         - With some tools, the skeleton testbench will run even before engineers begin filling in design-specific details

33

Additional Resources…

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- There are many more advanced UVM features, including:
  - Interrupt driven sequences and UVM callbacks
  - Virtual sequences and sequencers
  - The Register Abstraction Layer (RAL)
  - Using UVM Verification IP
  - … (and much more)

- Some great resources for many of these advanced topics include:
  - Getting Started with UVM: A Beginner's Guide, Vanessa Cooper, 2013
  - The UVM Primer, Ray Salemi, 2013
  - A Practical Guide to Adopting the Universal Verification Methodology (UVM), 2nd Edition, Rosenberg and Meade, 2013
  - Mentor's UVM Cookbook (verificationacademy.com/cookbook)
  - Accellera's UVM World (www.accellera.org/community/uvm)
  - DVCon and SNUG conference papers

34

Summary: UVM Pros and Cons

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- Some advantages of UVM
  - Testbenches can be highly reusable – can ultimately save time
  - Verification IP is available from a wide variety of sources
  - EDA companies provide tools specific for UVM coding and debug
  - Knowledgeable consultants are available
  - Engineers with prior UVM experience can be recruited
  - Online forums and other resources are readily available
- Some disadvantages of UVM
  - UVM is massive – can be overkill for many verification projects
  - UVM is complex – can have a steep learning curve
  - UVM is evolving – it is not a completely stable standard
    - UVM 1.1 is good, but UVM 1.2 is not backward compatible and will likely be changed as the IEEE make it a standard
  - UVM will let engineers do things wrong – limits VIP and reuse

35

Successfully Adopting UVM

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

Three things that are important for successfully adopting UVM…

1) The best software tools
   - Tools that organize the many files that make up a UVM testbench
   - Tools that generate skeleton UVM components
2) Top-notch consultants
   - An experienced expert on the team is essential for success
   - A knowledgeable consultant shortens the first UVM project
3) Really good training
   - UVM uses advanced SV Object Oriented Programming techniques
   - UVM will let engineers go the wrong direction if not careful
   - Expert training is an investment with continuous pay back

36

And That's a Wrap!

SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

- **Two-line Summary…**
  - **UVM really works and has major advantages**
  - **UVM is very complex with a steep learning curve**

Any questions, comments, or real-life experiences to share?

Sutherland HDL provides world-wide
on-site and *eTutored™-live* online training
www.sutherland-hdl.com – stuart@sutherland-hdl.com