

Introduction to the Universal Verification Methodology

Developed &
Presented by



Version 3.5 Copyright (c) 2011-2014
Willamette HDL, Inc.
For use with UVM 1.1d

This entire document is copyright Willamette HDL Inc. Content may not be copied or redistributed in whole or in part.
All trademarks used in this document are the property of their respective owners.

Willamette HDL, Inc.
6107 SW Murray Blvd. #407
Beaverton, OR 97008
info@whdl.com
www.whdl.com
503.590.8499

NOTE

This document is copyright material of WHDL Inc.
It may not be copied or reproduced for any reason.

Part Number:072797

Table of Contents -1

Daily Outline	5	Analysis	144
Introduction	6	TLM Analysis Elements	147
UVM Reporting Facilities	28	Scoreboards	158
Transaction-level Communication	40	Coverage collectors	168
UVM Transactions	45	Exercise	177
Core Utility Functions	49	Hierarchical Testbenches	181
Core Utility Function Implementation	51	UVM Component and hierarchical management	183
Transaction Example	59	Agents	186
Exercise	62	Exercise	192
Transactions reference	65	Test Class	197
UVM Components	71	Factory Overrides	202
UVM Component	76	Override Mechanics	204
UVM Environment	78	Debugging Overrides	210
Phases	79	Exercise	215
Components Example	95	Factory Overrides Reference	218
Exercise	100	Configuration	222
Creating Components & Running the Simulation	104	UVM Resources	223
Factory Registration	106	config_db	228
Creating components using the factory	109	Precedence	236
Starting the Test	116	Debugging Resources	245
Ending the Test	119	Exercise	249
Exercise	125	Resources Reference	254
Factory Registration Reference	129	Introduction to Sequences	264
Connection to the DUT	133	Sequence Elements	270
Transactors	135	Sequence Items	274
		Sequences	276
		Sequencers	286
		Drivers	287
		Sequencer – Driver Connection	292
		Sequence – Sequencer Connection	294
		Exercise	299
		Sequences Reference	306



uvm_intro_3.5

©Willamette HDL Inc.

3

Notes:

Table of Contents -2

More Sequence Topics	308
Virtual Sequences	313
Prioritized Item Selection and Arbitration	322
Responses	327
Exercise	334
Responses Reference	337
UVM Registers	343
The Register Model	346
Creating Register Models	354
Integrating Register Models	368
Exercise	397
Using Register Models	405
Register Access Methods	410
Memories	438
Backdoor Access	448
Quirky Register	453
Built-in Sequences	455
Convenience methods	457
Advanced UVM class topics	459
Exercise	460
Example Testbenches	469
MAC DUT with WISHBONE bus and MM interfaces	
DMA Controller on SoC bus using register package	
Crossbar Router DUT	
SPI DUT with APB interface	
Memory with AHB interface	
Appendix A - UVM Reporting Facilities	482
Sample Solutions	501
uvm_pkg	Last Page – 2
SystemVerilog Simulation Steps	Last Page – 1
Symbol Key	Last Page
uvm_intro_3.5	© Willamette HDL Inc.



Notes:

Course Outline

Day 1

- Messaging
- TLM Communication
- Transactions
- Testbench Components
- Starting and Stopping Simulation
- Construction with the Factory

Day 3

- Configuration
- Stimulus Generation
- Introduction to Sequences
- More on Sequence

Day 2

- DUT Connection
- Analysis
- Hierarchical Testbenches
- Test Class
- Factory Overrides

Day 4

- Registers
- Creating Register Models
- Integrating Register Models
- Using Register Models



UVM_intro_3.5

© Willamette HDL Inc.

5

Notes:

Introduction

In this section



Universal Verification Methodology
UVM Capabilities
UVM Concepts



uvm_intro_3.5

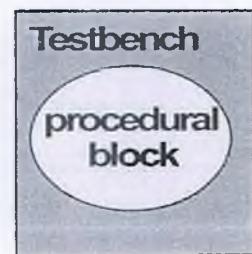
©Willamette HDL Inc.

6

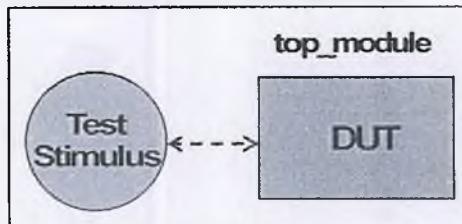
Notes:

Testbench Starting point

- Test is implemented either
 - as a group of procedural blocks (e.g. initial)
 - as a module containing procedural blocks
- Stimulus only
 - Little or no active verification



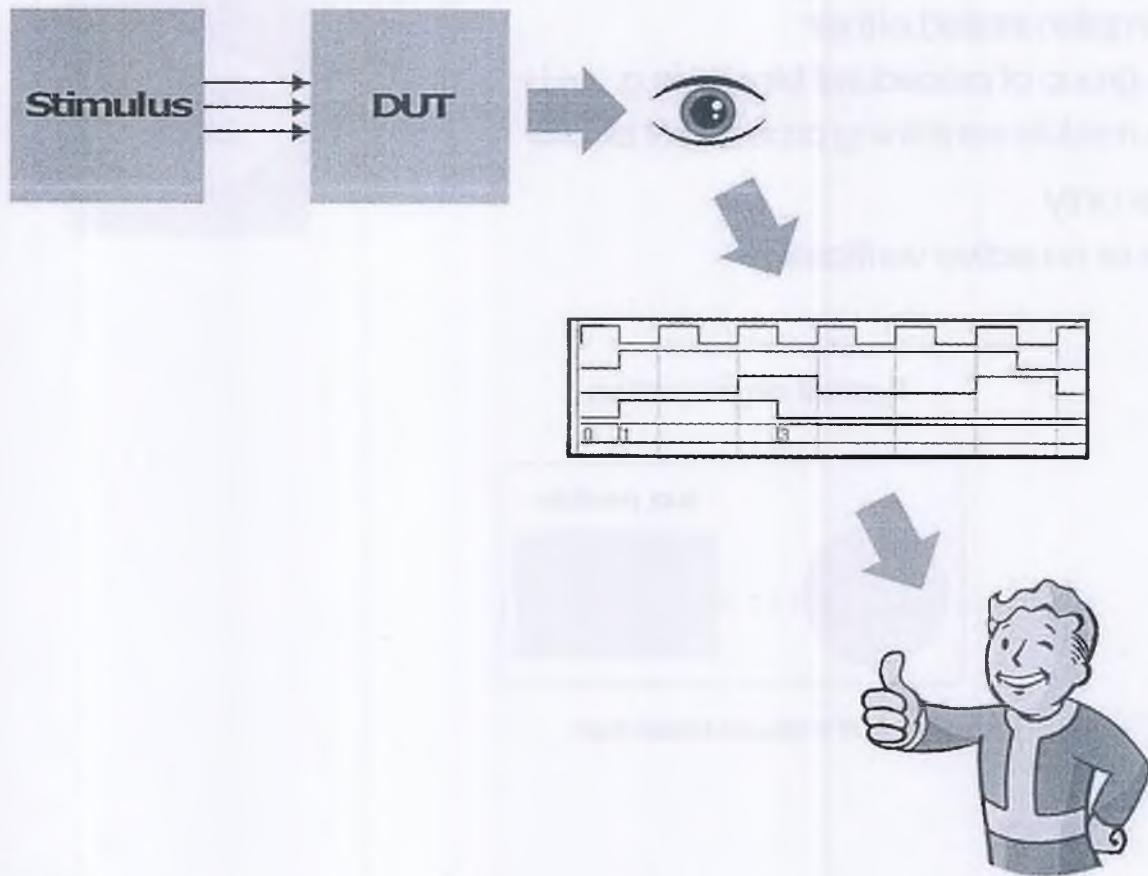
Typical organization



DUT = Device Under Test

Notes:

Traditional Verification Flow



Notes:

Problems With the Traditional Approach

- The traditional approach has fundamental problems
 - Nothing is *really* being verified
 - Stimulus is limited to imagination/experience of designer
 - No documented "proof" of what parts of DUT has been tested/exercised
- It also has these practical problems
 - Tests are hard or impossible to reuse
 - No standardized structure - Hard to maintain
 - ◆ Only the person who wrote it can understand it
- In the past, designs were simple enough that these problems were manageable or could just be ignored
- Not so any more...

Notes:

Why SV & UVM?

- Need to adapt testbenches to increasing complexity in designs
- Industry answer is SystemVerilog (SV) and the Universal Verification Methodology (UVM) library



uvm_intro_3.5

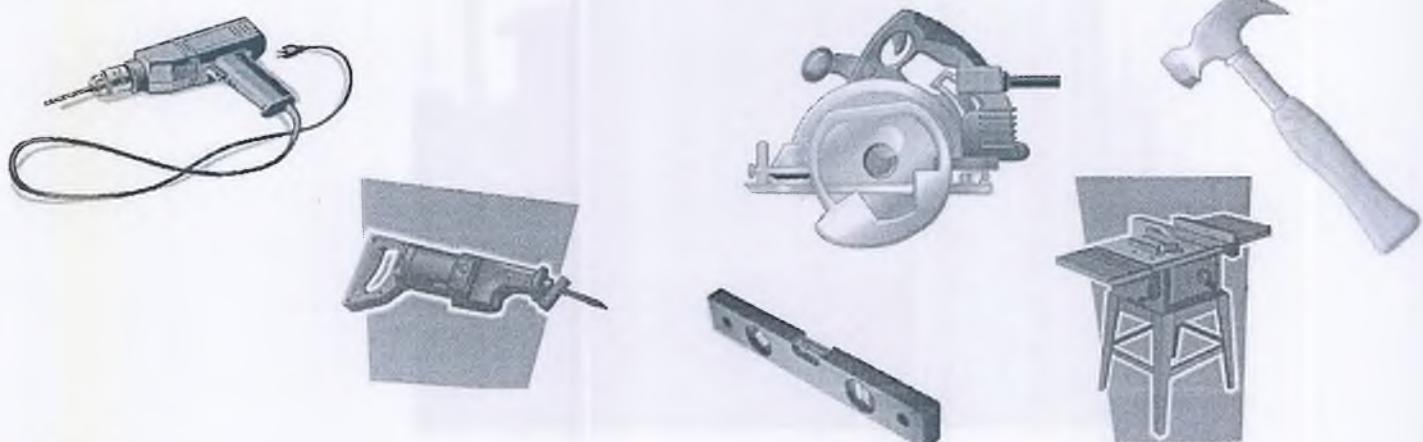
©Willamette HDL Inc.

10

Notes:

SystemVerilog – Tools of the (Verification) Trade

- SystemVerilog provides technology (capability) that doesn't exist in Verilog or VHDL – i.e. additional tools to get the job done
 - ◆ Constrained random number generation
 - ◆ Functional coverage
 - ◆ Classes (object oriented programming)
 - ◆ Assertions
- Learning SystemVerilog is like learning how to use the tools of a trade like carpentry



Notes:

But.. Do You Know How to Build a House (Testbench)

- We know in general what needs to be done
 - Generate stimulus, verify results etc.
- But *how* requires following proven methodologies else we get less than optimal results



Notes:

UVM Provides Testbench Methodology

- UVM provides testbench methodology – i.e. how to use the SV tools to create effective testbenches
 - Testbenches that not only allow the application of SV technology but have
 - ◆ Reusability
 - ◆ Maintainability
 - ◆ Scalability



uvm_intro_3.5

© Willamette HDL Inc.

13

Notes:

Universal Verification Methodology (UVM)

- System-level-to-RTL verification methodology
 - Provides an environment that can take advantage of advanced verification techniques
 - ◆ Functional coverage, constrained random stimulus generation, assertions etc.
- Provides libraries of classes (and macros)
 - Open source
 - Multiple vendor support
 - No requirements beyond standard SystemVerilog
- We will use this symbol to indicate code from the UVM library (as opposed to user code)



Notes:

UVM History

- 2009: Accellera sets up a committee to develop a single standard verification methodology (UVM)
- Dec 2009: Committee adopts OVM v2.1 (over VMM) as the baseline for UVM
 - Mar 2010: Committee adopts OVM v2.1.1 as the baseline
- Apr 2010: UVM Early Adopter Kit (UVM EA 1.0) release
 - OVM 2.1.1 with "o" changed to "u" everywhere
- DVCon 2011: uvm-1.0.p1 release
 - Initial public release
- DAC 2011: uvm-1.1 release
 - Bug fixes
 - Update the User's Guide and Reference Manual
 - Determined there is problems with Phasing as adopted.
- Various bug fix releases to 1.1 since



uvm_intro_3.5

© Willamette HDL Inc.

15

Notes:

UVL Infrastructure Features

- UVL has the following infrastructure features to help develop a reusable test environment:
 - **Component Hierarchy**
 - ◆ Provides structural "skeleton" for the testbench
 - **Message Reporting**
 - ◆ Can filter or specify actions based on verbosity, severity or classification of messages
 - **Configuration functions (Resources)**
 - ◆ Used as a shared area for storing and retrieving values
 - **Transaction-level (TLM) Communication**
 - ◆ Supports industry-standard API
 - ◆ Efficient and reusable
 - **Factory**
 - ◆ The actual type of an object being created is set at run-time
 - Delays construction of objects to enable...
 - ◆ **Factory Overrides**
 - Used to dynamically change the type of object that the factory constructs

Notes:

UVM Methodology Features

- In addition to the infrastructure features, UVM provides classes to develop:
 - **Communication Channels for Analysis**
 - ◆ Provides a methodology for observing information about DUT activity and recording it for coverage analysis
 - **Reusable Stimulus Generation Sequences**
 - ◆ Methodology for scalable, reusable stimulus generation
 - ◆ Test developers do not need intimate knowledge of test environment
 - **Registers**
 - ◆ Mirroring of registers in Hardware
 - ◆ RW of registers & memories with abstract names
 - ◆ Built in sequences for testing



uvm_intro_3.5

©Willamette HDL Inc

17

Notes:

UVM Prime Directive: It's All About Reuse!

- More time spent developing verification environment than developing the DUT
- Need to reuse all that work as much as possible
- From the start, need to design the test environment with reuse in mind
- We will point out the features of UVM and promote styles that facilitate reuse throughout the class



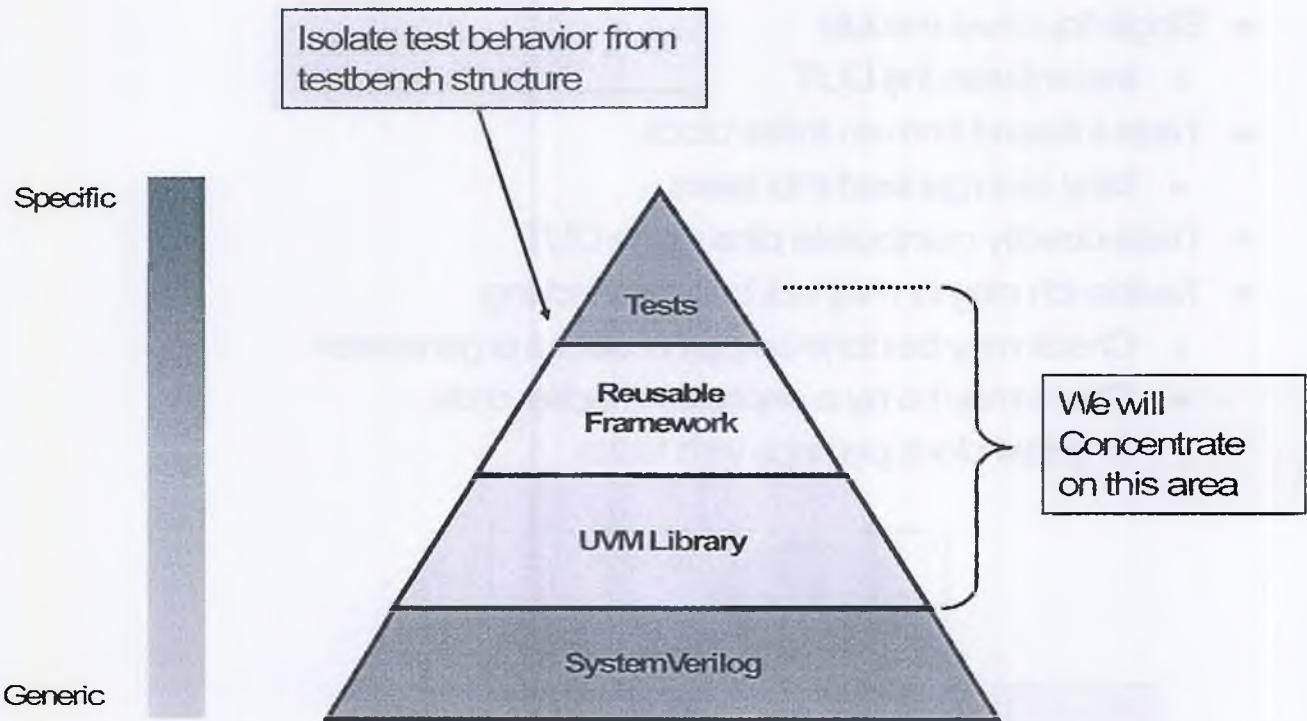
uvm_intro_3.5

©Willamette HDL Inc.

18

Notes:

Building on a Reusable Foundation



uvm_intro_3.5

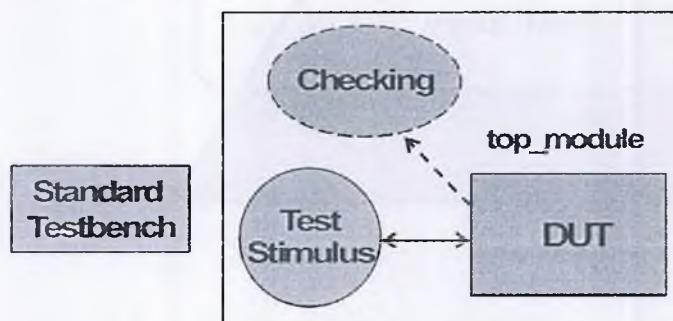
©Wilamette HDL Inc.

19

Notes:

Traditional or Standard Testbench

- Traditional or standard testbench
 - Single top level module
 - ◆ Instantiates the DUT
 - Tests initiated from an initial block
 - ◆ May be organized into tasks
 - Tests directly manipulate pins of the DUT
 - Testbench may or may not be self-checking
 - ◆ Check may be done as part of stimulus generation
 - ◆ Check may be by a separate checker code
 - Initial block perhaps with tasks

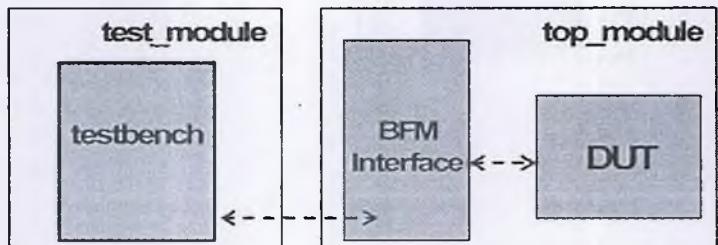
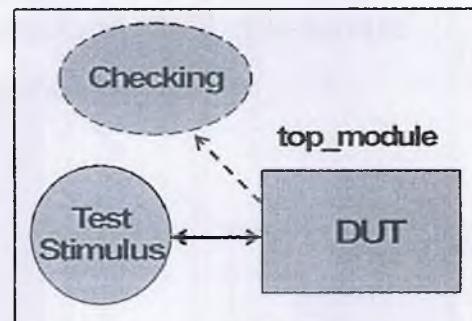


© Willamette HDL Inc.

Notes:

Moving to a UVM (OOP) Testbench Structure

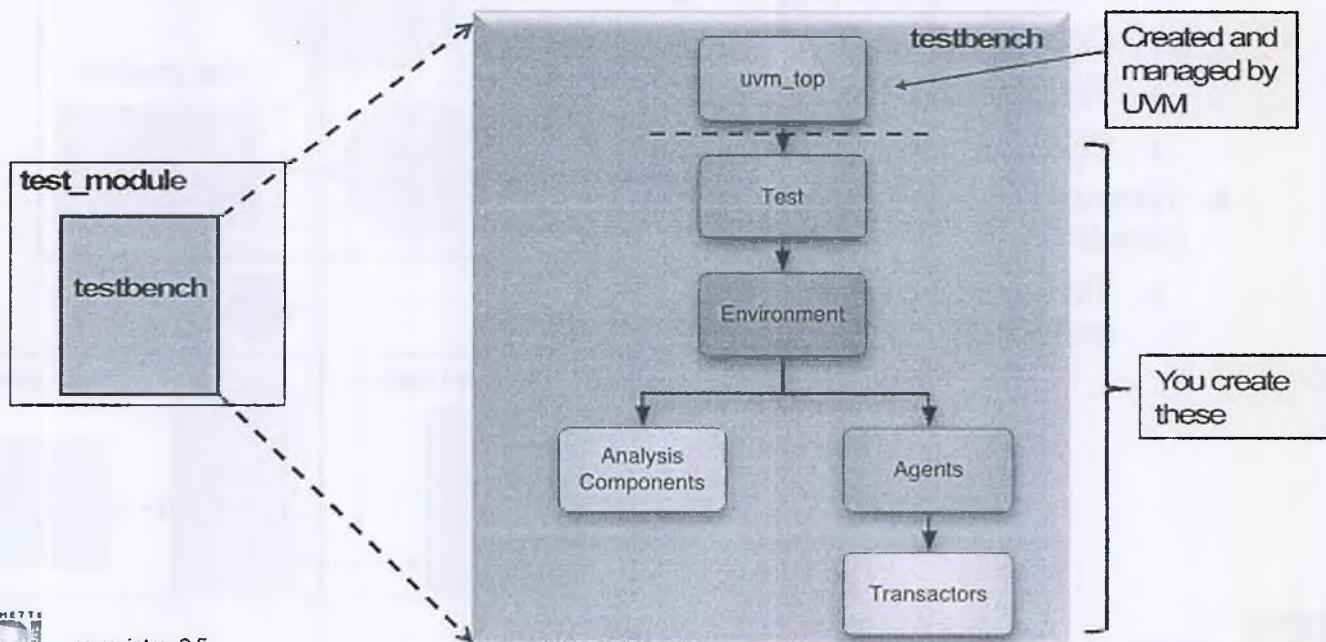
- Stimulus generation and checking become class based (OOP)
- Separate module for testbench (dual-top) organization
- Interface used for DUT-testbench connection
 - Virtual interface connection
 - **BFM Interface (recommended)**
 - ◆ Contains all DUT timing information
 - ◆ Tasks for DUT interaction
 - ◆ Style supported by emulation
 - Alternate style interface is signal based
 - ◆ Timing information in the testbench



Notes:

UVM Testbench Structure

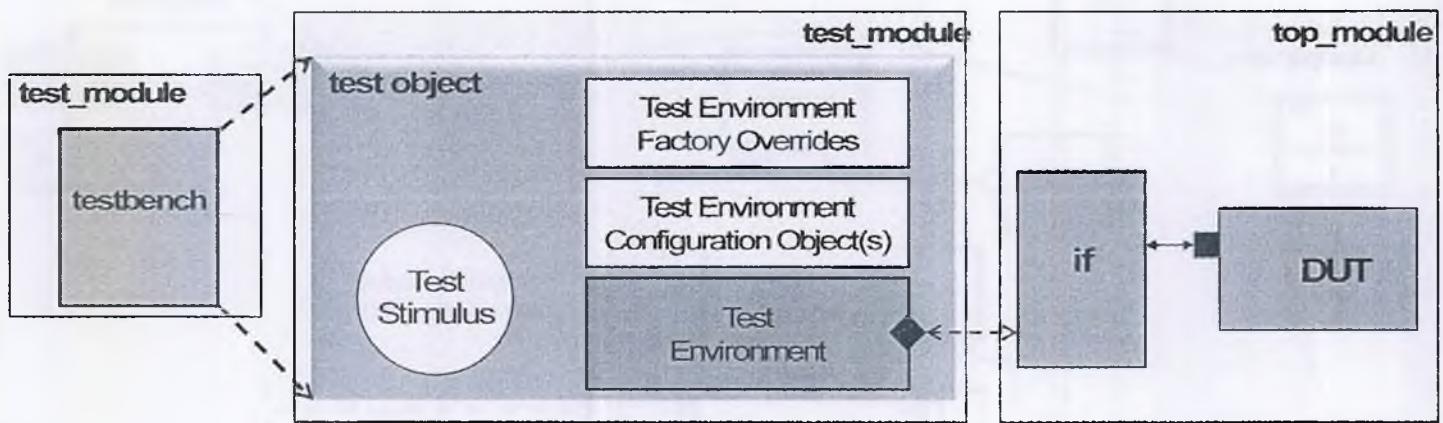
- Testbench consists of a number of class objects that are hierarchically related
 - `uvm_top` is a test component that is created and managed by UVM
 - The test object configures, then instantiates the environment
 - The environment instantiates the testbench components that generate stimulus to DUT and analyze the DUT behavior



Notes:

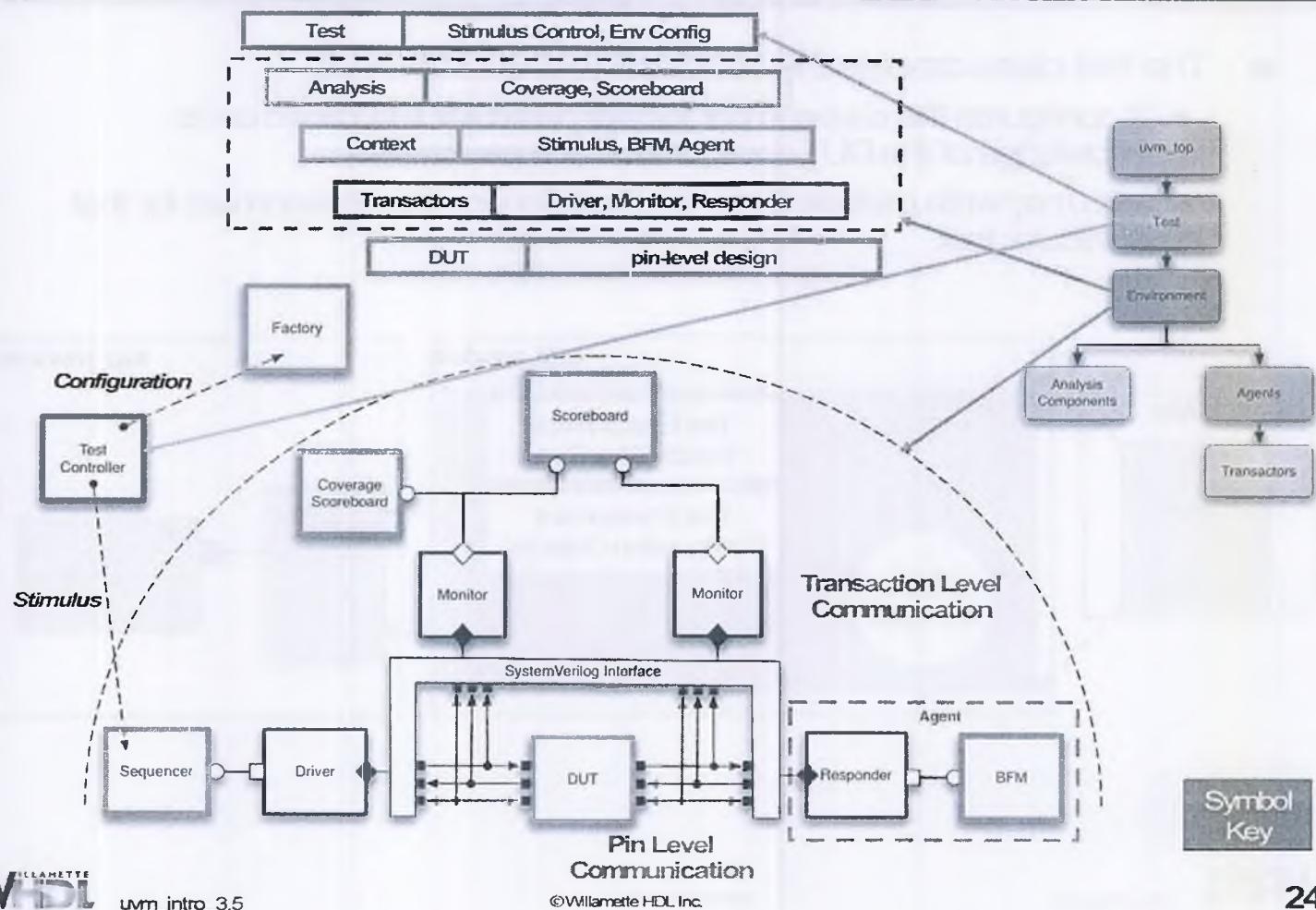
UVM Testbench Structure – Another viewpoint

- The test class creates the structural test environment
 - It configures the environment for that particular test based on its knowledge of the DUT configuration and parameters
 - You may write multiple tests, each configuring the environment for that particular test



Notes:

UVM Layers and Structure



Notes:

UVL Library Package

- UVM library is defined within a package called `uvm_pkg`
 - Class declarations
 - Static singletons
 - Convenience methods
- Test bench components typically
 - Inherit from base classes in `uvm_pkg`
 - Instantiate classes from `uvm_pkg`
 - Access static singletons in `uvm_pkg`
 - Call convenience methods in `uvm_pkg`
- Sometimes these singletons and convenience methods are referred to as "global objects" or "global methods"
 - Since `uvm_pkg` is imported everywhere it appears as if they are "global" while technically their scope is the `uvm_pkg`

Notes:

Introduction UVM class & Advanced UVM class

■ Introduction UVM class (this class)

- "Steers" a path through the UVM library
 - ◆ Covers the "need to know" classes for developing basic testbenches
 - ◆ You will be able to begin putting testbenches together
 - ◆ You will know all the basic and common things to know
- You will take away multiple full example testbenches

■ Advanced UVM class

- Addresses issues and situations
 - ◆ Not standard or basic stuff you sometimes run into as you put together testbenches
- Covers classes for developing advanced testbench techniques
- Take away several full example testbenches



uvm_intro_3.5

© Willamette HDL Inc.

26

Notes:

Introduction UVM Class Flow

- Flow of the class follows the flow in moving from a traditional testbench to a full UVM testbench
 - 1. Testbench Infrastructure
 - ◆ Needed to add Analysis and stimulus components
 - 2. Analysis
 - ◆ SystemVerilog Coverage
 - ◆ Scoreboarding
 - 3. Stimulus generation
 - ◆ Constrained random generation
 - 4. Layering
 - ◆ Register layer
 - Provides for abstraction in stimulus generation and analysis



uvm_intro_3.5

© Willamette HDL Inc.

27

Notes:

UVM Reporting Facilities

In this section



Reporting concepts
Reporting methods
Reference: Reporting levels and actions

Notes:

Reporting Requirements

- Testbenches may produce many different messages
 - Many different severity levels possible
 - ◆ warning, information (status, debugging), error, fatal
 - Easy to be overwhelmed by messages generated by complex testbenches
- Reporting requirements
 - Separation and/or enabling by relevance
 - Filtering by verbosity
 - Separation and/or enabling by severity
 - Separation by function (debugging from errors etc.)
 - Logging in files and/or sent to standard output
 - ◆ Perhaps in relevant subsets



uvm_intro_3.5

©Willamette HDL Inc.

29

Notes:

UVM Reporting

- Each testbench component in UVM has its own report handler
- The report handler has an API with methods which provide means for displaying messages and taking actions
 - Each message consists of a message string and also has the following properties:
 - ◆ Severity
 - One of: `UVM_INFO`, `UVM_WARNING`, `UVM_ERROR`, `UVM_FATAL`
 - ◆ Category (ID)
 - An arbitrary string that you provide with each message call (e.g. "BUS", "DEBUG", "COMPARE")
 - ◆ Verbosity
 - A positive integer, lower values denote higher "importance"
 - The reporting API allows you to use these properties to manage the messages that are generated.

Notes:

UVM Reporting

- You can control verbosity by setting a threshold value
 - Any messages with verbosity greater than the threshold will be ignored
- Each message can trigger actions based on the severity, the ID, or both
 - Displaying to the screen
 - Logging to a file
 - Calling a "hook" function



uvm_intro_3.5

©Willamette HDL Inc.

31

Notes:

Reporting Macros

- A report is issued by calling one of the reporting macros
- Each method has a built in severity level (`UVM_INFO`, `UVM_WARNING`, `UVM_ERROR`, `UVM_FATAL`)
- Each macro has arguments
 - `ID`: Type `string`, used as a message category
 - ◆ Used to group or separate messages
 - `MESSAGE`: Type `string`, the actual message to print/write
 - `VERBOSITY`: Type `int`, the verbosity level of the message (INFO only)
 - ◆ Message will print only if the verbosity level is lower than or equal to the UVM report setting

```
'uvm_info  ( ID, MESSAGE, VERBOSITY)
'uvm_warning( ID, MESSAGE )
'uvm_error   ( ID, MESSAGE )
'uvm_fatal   ( ID, MESSAGE )
```

Notes:

Verbosity Levels

- **UVM_NONE**
 - Report is always issued - cannot be disabled
- **UVM_LOW**
 - Report is issued if verbosity is set to **UVM_LOW** or above
- **UVM_MEDIUM**
 - Report is issued if verbosity is set to **UVM_MEDIUM** or above (Default)
- **UVM_HIGH**
 - Report is issued if verbosity is set to **UVM_HIGH** or above
- **UVM_FULL**
 - Report is issued if verbosity is set to **UVM_FULL** or above
- **UVM_DEBUG**
 - Report is issued if verbosity is set to **UVM_DEBUG** or above

```
typedef enum {  
    UVM_NONE    = 0,  
    UVM_LOW     = 100,  
    UVM_MEDIUM  = 200,  
    UVM_HIGH    = 300,  
    UVM_FULL    = 400,  
    UVM_DEBUG   = 500  
} uvm_verbosity;
```

UVM

Notes:

Actions

- Besides displaying a message a reporting method may invoke actions
 - **UVM_NO_ACTION**
 - ◆ Do nothing
 - **UVM_DISPLAY**
 - ◆ Display report to standard output
 - **UVM_LOG**
 - ◆ Send report to a file
 - **UVM_COUNT**
 - ◆ Count up to a threshold before exiting
 - **UVM_EXIT**
 - ◆ Exit simulation immediately
 - **UVM_CALL_HOOK**
 - ◆ Call hook method

```
typedef enum uvm_action UVM  
{  
    UVM_NO_ACTION = 5'b00000,  
    UVM_DISPLAY   = 5'b00001,  
    UVM_LOG        = 5'b00010,  
    UVM_COUNT      = 5'b00100,  
    UVM_EXIT       = 5'b01000,  
    UVM_CALL_HOOK  = 5'b10000  
} uvm_action_type;
```

Notes:

Default Severity Actions

- Each severity level has one or more actions associated with it
 - The default actions for each severity level:
 - ◆ `UVM_INFO`
 - `UVM_DISPLAY`,
 - ◆ `UVM_WARNING`
 - `UVM_DISPLAY`,
 - ◆ `UVM_ERROR`
 - `UVM_DISPLAY`, `UVM_COUNT`
 - ◆ `UVM_FATAL`
 - `UVM_DISPLAY`, `UVM_EXIT`
 - The Actions may be combined by using the "|" operator
 - ◆ Example:

```
uvm_action act = UVM_DISPLAY | UVM_EXIT // both actions happen
```

Notes:

Example Reporting

```
if (result_txn.result == expected_result)
    `uvm_info("SB OK", $sformatf("ALU operation %0d %s %0d = %0d passed",
                                    result_txn.val1, result_txn.mode.name(),
                                    result_txn.val2, result_txn.result),
        UVM_FULL );
else
    `uvm_error("SB MISMATCH",
               $sformatf("ALU operation failed: expected %0d, got %0d",
                         expected_result, result_txn.result))
```

Message category (ID)

This message will only print out if the UVM report verbosity is set to UVM_FULL (400) or higher

Notes:

Command Line Reporting

+UVM_VERBOSITY=<verbosity>

- Sets initial verbosity for all components
 - ◆ Value can be one of: UVM_NONE, UVM_LOW, UVM_MEDIUM (default), UVM_HIGH, UVM_FULL

Output with +UVM_VERBOSITY=UVM_FULL

```
# UVM_INFO @ 0: reporter [RNTST] Running test default test...
# UVM_INFO result_checker.svh(30) @ 200: uvm_test_top.env.chk [SB OK] ALU operation 664 DIV 951 = 0 passed
# UVM_INFO result_checker.svh(30) @ 300: uvm_test_top.env.chk [SB OK] ALU operation 766 SUB 414 = 352 passed
# UVM_INFO result_checker.svh(30) @ 400: uvm_test_top.env.chk [SB OK] ALU operation 372 ADD 518 = 890 passed
# UVM_INFO result_checker.svh(30) @ 500: uvm_test_top.env.chk [SB OK] ALU operation 784 MUL 635 = 497840 passed
# UVM_INFO result_checker.svh(30) @ 600: uvm_test_top.env.chk [SB OK] ALU operation 698 MUL 82 = 57236 passed
# UVM_INFO result_checker.svh(30) @ 700: uvm_test_top.env.chk [SB OK] ALU operation 685 DIV 719 = 0 passed
# UVM_INFO result_checker.svh(30) @ 800: uvm_test_top.env.chk [SB OK] ALU operation 644 SUB 23 = 621 passed
# UVM_INFO result_checker.svh(30) @ 900: uvm_test_top.env.chk [SB OK] ALU operation 544 DIV 750 = 0 passed
# UVM_INFO result_checker.svh(30) @ 1000: uvm_test_top.env.chk [SB OK] ALU operation 466 ADD 239 = 705 passed
# UVM_INFO result_checker.svh(30) @ 1100: uvm_test_top.env.chk [SB OK] ALU operation 993 DIV 796 = 1 passed
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 11
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [RNTST] 1
# [SB OK] 10
```



Recommendation: Use UVM_MEDIUM for "important" info messages and UVM_HIGH for general info messages. Set a default verbosity using the +UVM_VERBOSITY plusarg of UVM_MEDIUM for general runs and UVM_HIGH for debugging runs

Notes:

Reporting API

- There are many reporting API methods for customizing how reports are handled
- Most API functions come in two "forms"
 - Non-hierarchical
 - ◆ Effect the component's report handler only
 - Hierarchical
 - ◆ Have an `_hier` appended to the name
 - ◆ Effect the component's report handler *and all its hierarchical descendants*
- Most API functions are not covered in this section
 - See Appendix A for a description of the rest of the reporting API methods
- The following slide gives an overview of the reporting API

Notes:

UVM Reporting API Overview

UVM has powerful report controls for filtering and managing messages. See the reference docs for more info.

UVM Reporting

Methods

Severity	
UVM_INFO	<code>uvm_report_info</code>
UVM_WARNING	<code>uvm_report_warning</code>
UVM_ERROR	<code>uvm_report_error</code>
UVM_FATAL	<code>uvm_report_fatal</code>

Filtering

By Verbosity

`set_report_verbosity_level`

By Severity

`set_report_severity_file`

File Output

By Context (ID)

`set_report_id_file`

`set_report_default_file`

By Both

`set_report_severity_id_file`

Actions

Grouping

UVM_NO_ACTION
UVM_DISPLAY
UVM_LOG
UVM_EXIT
UVM_COUNT
UVM_CALL_HOOK
UVM_STOP

By Severity

`set_report_severity_action`

By Context (ID)

`set_report_id_action`

By Both

`set_report_severity_id_action`

Notes:

Transaction-level (TLM) Communication

In this section

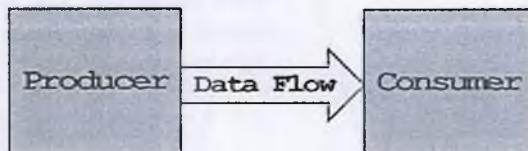


TLM Ports
TLM Exports
TLM fifos

Notes:

TLM Terminology

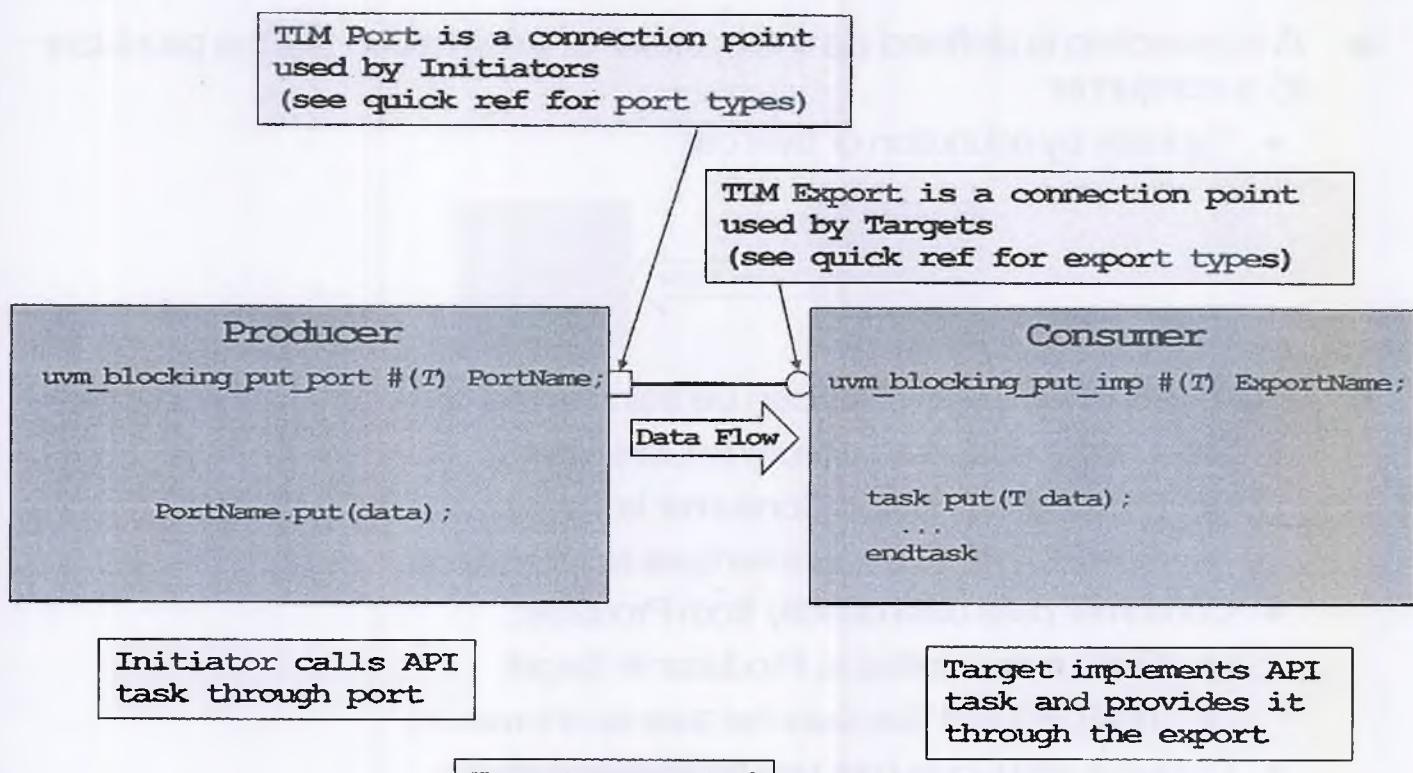
- A transaction is defined as the transfer of information from a producer to a consumer
 - Typically by a function or task call



- There are three ways data can be transferred between components
 - Producer *pushes* data directly to Consumer:
 - ◆ Producer is Initiator, Consumer is Target
 - ◆ TLM PUT API (See Quick Ref Guide for API methods)
 - Consumer *pulls* data directly from Producer:
 - ◆ Consumer is Initiator, Producer is Target
 - ◆ TLM GET API (See Quick Ref Guide for API methods)
 - Producer and Consumer use an intermediate fifo
 - ◆ Both Producer and Consumer are Initiators, fifo is Target for both
 - ◆ TLM FIFO

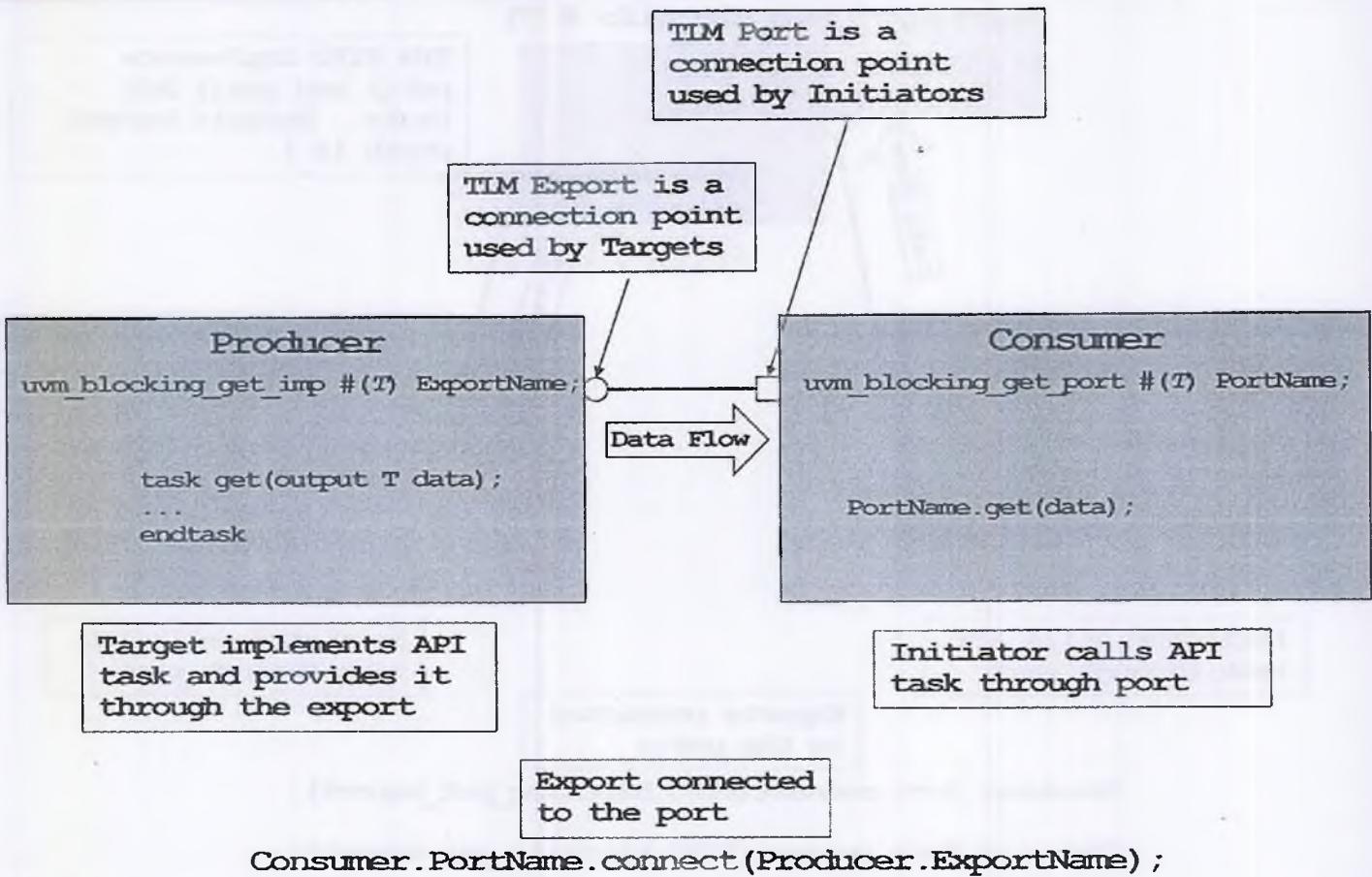
Notes:

TLM API Classes (Push Mode)



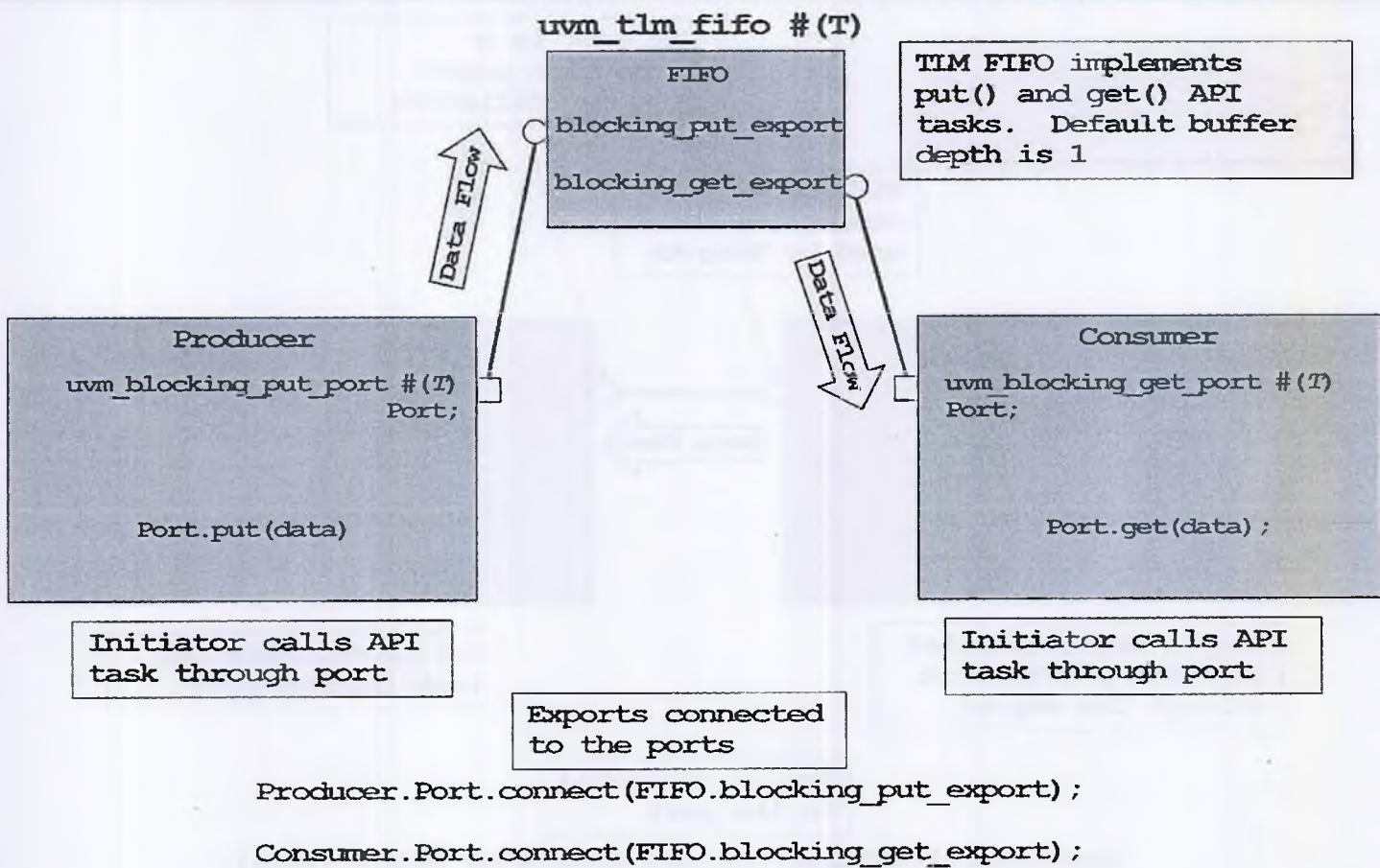
Notes:

TLM API Classes (Pull Mode)



Notes:

TLM API Classes (FIFO Model)



Notes:

UVM Transactions

In this section



The `uvm_sequence_item` class
UVM Core utility functions

Notes:

UMM Transaction Classes

- UMM is a transaction-based methodology
- All stimulus and analysis is conducted with class objects that represent transactions
- UMM transaction classes should inherit from the `uvm_sequence_item` base class
 - UMM transactions have a standardized constructor
 - ◆ Single string argument with a default value
 - ◆ Pass the argument to the base class constructor
 - Also one macro declaration to "register" the class with UMM factory (more later)

```
class alu_txn extends uvm_sequence_item;
  `uvm_object_utils(alu_txn);
  rand op_type_t mode;
  bit [3:0] done;
  rand shortint unsigned val1;
  rand shortint unsigned val2;
  int result;

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

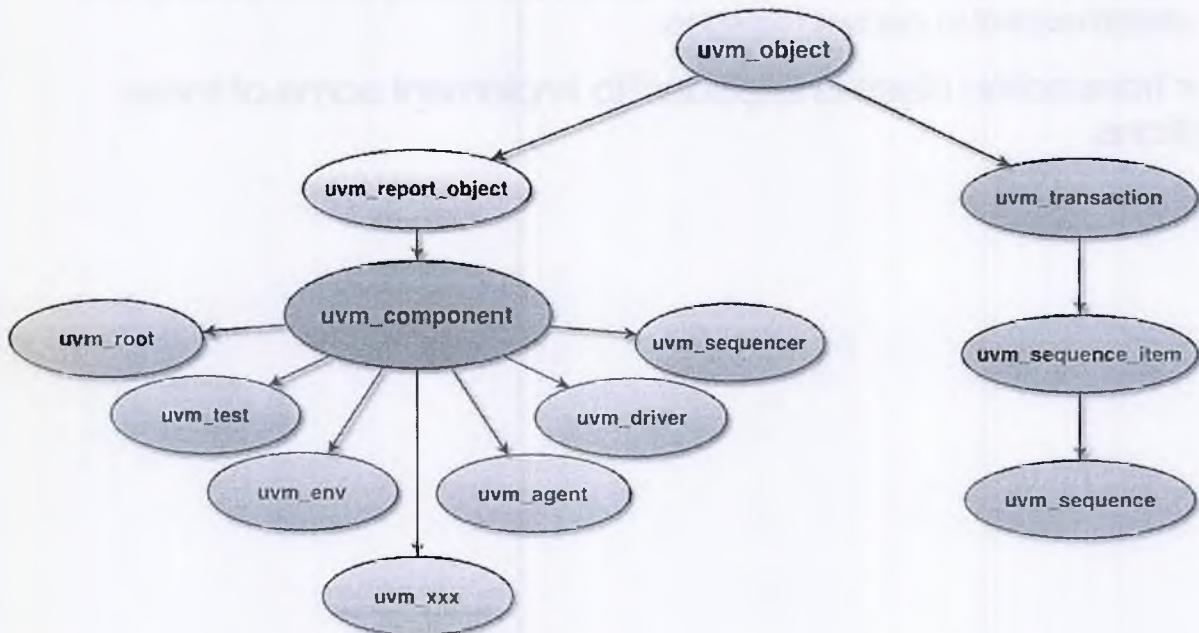
Make sure that the argument to the macro is the same name as your class name.

Deriving from `uvm_sequence_item` requires that the derived class have the string name constructor argument with a default value (can be empty string). That argument must be passed to the base class.

Notes:

UVM Transactions: uvm_sequence_item

- The `uvm_sequence_item` class extends `uvm_transaction` which extends `uvm_object` both of which adds several transaction-related properties and methods



Notes:

UVM_object Class

- Virtual base class in the UVM library
 - Base class for all UVM transaction objects and testbench components
- Provides a number of data access methods and control methods
 - What the methods are and their usage will be explained as they are used/needed in derived classes
- Your transaction class is expected to implement some of these functions



uvm_intro_3.5

©Willamette HDL Inc.

48

Notes:

UVM Object Core Utilities

- Transaction objects, are expected to have certain "core utility" capabilities
 - Provides an API for the structural components that use the transactions
 - Listed in the table below
- Since you define the transaction, you have to implement these functions in a meaningful way for your transaction data fields

uvm_object clone()
string convert2string()
void copy(uvm_object rhs)
void print(uvm_printer printer = null)
string sprint(uvm_printer printer = null)
bit compare(uvm_object rhs, uvm_comparer comparer = null)
int pack(ref bit bitstream[], uvm_packer packer = null)
int unpack(ref bit bitstream[], uvm_packer packer = null)
void record(uvm_recorder recorder = null)

Notes:

Providing Core Utility Function Behavior

- You don't have to override `clone()`. The default behavior works fine (as long as you implement `do_copy()` below...)
- Method you will want to override directly:
 - `convert2string()`
- Others are *template method pattern* functions where you will insert behavior by overriding a provided "hook" method
 - We'll talk about the top three common "do_" methods that you will want to override

Utility Function	Method You Write
<code>copy()</code>	<code>do_copy()</code>
<code>print()</code>	<code>do_print()</code>
<code>compare()</code>	<code>do_compare()</code>
<code>pack()</code> , <code>unpack()</code>	<code>do_pack()</code> , <code>do_unpack()</code>
<code>record()</code>	<code>do_record()</code>

Notes:

Convert2string

UVM virtual function string convert2string();
 return "";
endfunction

- Override this function to create a string representation of your transaction for printing, as part of messages etc.

```
class alu_txn extends uvm_sequence_item;
  ...
  function string convert2string();
    string str1;
    str1 = { "----- Start ALU txn -----`n",
              "ALU txn `n",
              $sformatf(" mode : %s`n", mode.name()),
              $sformatf(" val1 : 'h%h`n", val1),
              $sformatf(" val2 : 'h%h`n", val2),
              $sformatf(" result : 'h%h`n", result),
              "----- End ALU txn -----`n"};
    return (str1);
  endfunction
  ...
endclass
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu_transaction

51

Notes:

Copy

UVM

```
function void copy(uvm_object rhs)
virtual function void do_copy(uvm_object rhs)
```

- Calls the "hook" method `do_copy()`

- You override this function in your class to provide the copy behavior for your class properties
- It is *not* empty in the base class!
 - ◆ Copies various properties used in the `uvm_transaction` and `uvm_sequence_item` base class
 - ◆ Be sure and add `super.do_copy()` if you override `do_copy()`!

Notes:

Example Override of do_copy()

```
class alu_txn extends uvm_sequence_item;
  ...
  rand op_type_t mode;
  bit [3:0] done;
  rand shortint unsigned val1;
  rand shortint unsigned val2;
  int result;
  ...
  function void do_copy(uvm_object rhs);
    alu_txn tmp;
    if(!$cast(tmp, rhs)) // cast so can access the fields
      `uvm_fatal("TypeMismatch", "Type mismatch in copy")
    super.do_copy(tmp); // not empty so include
    mode = tmp.mode;
    done = tmp.done;
    val1 = tmp.val1;
    val2 = tmp.val2;
    result = tmp.result;
  endfunction
  ...
endclass
```

The rhs argument comes in as a uvm_object, so you have to \$cast it to your transaction type in order to access your fields

Also remember that with SystemVerilog, when you override a function, you are not free to choose your own argument names – the names must match the original method argument names.



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/alu_transaction

53

Notes:

Printing

UVM

```
function void print (uvm_printer printer = null)
function string sprint(uvm_printer printer = null)
virtual function void do_print(uvm_printer printer)
```

- Calls the "hook" method `do_print()`

- You override this function in your class to provide the print behavior for the transaction class properties
 - ◆ Easiest implementation is to simply display (`$display()`) the string returned by the `convert2string()` method
 - ◆ Can use auto-formatting functions in the `printer` argument

- `sprint()` is similar but returns a string:

- Since both `print()` and `sprint()` call the same "hook" function, you have to figure out if you should `$display` the data or return it as a string
 - ◆ Use the bit `printer.knobs.sprint` from the input argument
- If called by `sprint()` you should return the string in the `m_string` property of the `printer` argument
 - ◆ In the `do_print()` method, just set this property to the string returned by the `convert2string()` method



uvm_intro_3.5

© Willamette HDL Inc.

54

Notes:

do_print() using convert2string()

```
class alu_txn extends uvm_sequence_item;
  ...
  function void do_print(uvm_printer printer);
    super.do_print(printer); // not empty so include
    if (printer.knobs.sprint) //is this a sprint() call?
      printer.m_string = convert2string(); // set string
    else // nope a print() call
      $display(convert2string()); // print out string
  endfunction
  ...
endclass
```

- Note: There is no need to override `do_print()` in derived classes
 - Simply override the `convert2string()` method in the derived class

do_print() using printer object

- The printer object has various methods for printing out different types of data with automatic formatting

```
class alu_txn extends uvm_sequence_item;  
...  
function void do_print(uvm_printer printer);  
    super.do_print(printer);  
    printer.print_generic("mode", "op_type_t", 1, mode.name());  
    printer.print_generic("done", "bit[3:0]", 4,  
                          $sformatf("%h",done));  
    printer.print_generic("vall", "shortint unsigned", 16,  
                          $sformatf("%h",vall));  
    printer.print_generic("val2", "shortint unsigned", 16,  
                          $sformatf("%h",val2));  
    m_result.print(printer);  
endfunction  
...  
endclass
```

Notes:

Compare

UVM

```
function bit compare(uvm_object rhs, uvm_comparer comparer = null)
virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer)
```

- `compare()` calls the "hook" method `do_compare()`

- ◆ You override this function in your class to provide the compare behavior for your transaction
- ◆ No need to use the comparer object argument



uvm_intro_3.5

©Willamette HDL Inc.

57

Notes:

alu_txn Example-2

```
function void do_copy(uvm_object rhs);
    alu_txn tmp;
    if(!$cast(tmp, rhs)) // cast so can access the fields
        `uvm_fatal("TypeMismatch", "Type mismatch in copy")
    super.do_copy(tmp);
    mode = tmp.mode;
    done = tmp.done;
    val1 = tmp.val1;
    val2 = tmp.val2;
    result = tmp.result;
endfunction

function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    alu_txn tmp;
    do_compare = (
        $cast(tmp, rhs) &&
        super.do_compare(rhs, comparer) &&
        mode == tmp.mode &&
        done == tmp.done &&
        val1 == tmp.val1 &&
        val2 == tmp.val2 &&
        result == tmp.result
    );
endfunction
```

continued...



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu_transaction

60

Notes:

alu_txn Example - 3

```
function void do_print(uvm_printer printer);
    super.do_print(printer);
    if (printer.knobs.sprint) //is this a sprint() call?
        printer.m_string = convert2string();
    else // nope a print() call
        $display(convert2string());
endfunction

/*
function void do_print(uvm_printer printer);
    super.do_print(printer);
    printer.print_generic("mode", "op_type_t", 1, mode.name());
    printer.print_generic("done", "bit[3:0]", 4,
                          $sformatf("%h",done));
    printer.print_generic("val1", "shortint unsigned", 16,
                          $sformatf("%h",val1));
    printer.print_generic("val2", "shortint unsigned", 16,
                          $sformatf("%h",val2));
    printer.print_generic("result", "int", 32,
                          $sformatf("%h",result));
endfunction
*/
endclass
```



uvm_intro_3.5

© Willamette HDL Inc.

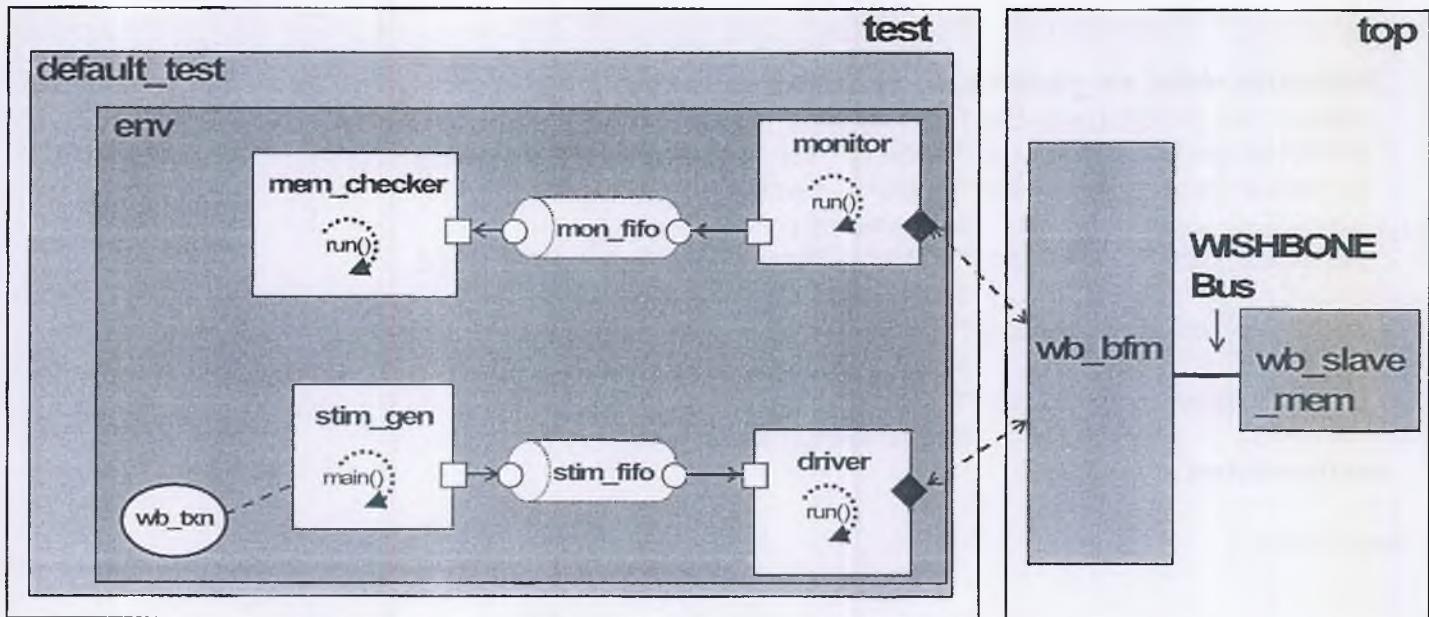
code in examples/alu_transaction

61

Notes:

Lab – Transactions: Overview -1

- Working directory: transactions
- Objective is to provide the `wb_txn` utility methods
- We will ignore all the UVM infrastructure stuff that we haven't talked about yet



WILLAMETTE
HDL

uvm_intro_3.5

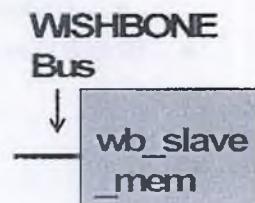
© Willamette HDL Inc.

62

Notes:

Lab – Transactions: WISHBONE bus

- Official Name:
 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores
 - Will refer to it as “WISHBONE bus” or WB or wb
- Available from OpenCores.org
- Features:
 - Multiple Master / Multiple Slave architecture
 - 32 bit Data /Address
 - Read/Write cycle, Block Read/Write cycle, RMW cycle
 - ◆ Synchronous/asynchronous handshake
 - Interrupt structure
- wb_slave_mem
 - 1 Mbyte bus slave memory
 - OpenCores.org
 - Memory mapped to the WISHBONE bus



Notes:

Lab – Transactions: Instructions

- Working directory: transactions
- NOTE: This lab uses test `default_test`
- Edit the file `wb_txn.svh`
 - Extend the `wb_txn` class from `uvm_sequence_item`
 - Write a proper UVM transaction constructor
 - Override the following methods
 - ◆ `do_copy()`
 - ◆ `do_compare()`
 - ◆ `do_print()`
 - Note that `convert2string()` is provided
- Compile & run (should be no errors or print outs of the transactions)
- For errors and to see if `do_print()` and `do_compare()` are working correctly:
 - Compile and run the "bad" memory which sometimes does not operate properly
 - ◆ make bad
 - Should be errors and print outs of the transaction objects

Notes:

Lab – Transactions: Partial Sample Output

```
#UVM_INFO@ 0: reporter [RNTST] Running test default_test...
#
#INFO: WISHBONE MEMORY MODEL INSTANTIATED (top.wb_s_0)
#    Memory Size 18 address lines 262144 words
#
#UVM_INFO@ 210: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 00032449
#UVM_INFO@ 330: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 0001359e
#UVM_INFO@ 450: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 00039c79
#UVM_INFO@ 570: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 00019878
#UVM_INFO@ 690: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 0001235a
#UVM_INFO@ 810: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 0000b408
#UVM_INFO@ 930: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 000396ca
#UVM_INFO@ 1050: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 0001ab3a
#UVM_INFO@ 1170: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 00034a1d
#UVM_INFO@ 1290: uvm_test_top.env.chk [MEM_CHECK] Memory passed at address 000303f3
#
#— UVM Report Summary —
#
# ** Report counts by severity
#UVM_INFO: 11
#UVM_WARNING: 0
#UVM_ERROR: 0
#UVM_FATAL: 0
# ** Report counts by id
#[MEM_CHECK] 10
#[RNTST] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 1290 ns Iteration: 56 Instance: /test
```

Sol



uvm_intro_3.5

© Willamette HDL Inc.

65

Notes:



```
int pack(ref bit bitstream[], uvm_packer packer = null)
int pack_bytes(ref byte bytestream[], uvm_packer packer = null)
int pack_ints(ref int unsigned intstream[], uvm_packer packer = null)
int unpack(ref bit bitstream[], uvm_packer packer = null)
int unpack_bytes(ref byte bytestream[], uvm_packer packer = null)
int unpack_ints(ref int unsigned intstream[], uvm_packer packer = null)
virtual void do_pack(uvm_packer packer)
virtual void do_unpack(uvm_packer packer)
```

- The `pack` functions append the packed contents of an object to the supplied array (of type `bit`, `byte`, or `int`)
 - `pack()` returns the size of the stream array
- The `unpack` functions uses the contents of an array to rebuild a class object
 - `unpack()` returns the number of array elements extracted from the stream
- `pack()` & `unpack()` operate on the automatic fields first
 - Both use the `packer` policy class on the automatic fields
- `pack()`/`unpack()` then call `do_pack`/`do_unpack`
 - Override these methods to provide pack and unpack behavior

Notes:

do_pack / do_unpack Implementation

Reference Page

- The implementation of these methods must be such that unpacking is the exact reverse of packing
- Packing and unpacking require precise concatenation of property values into a bit vector
- An `uvm_packer` singleton policy object named `packer` is used by the pack/unpack methods
 - Only need to know about two properties of this policy object:

```
uvm_pack_bitstream_t m_bits;  
    ◆ Vector that holds the packed stream  
    ◆ Pack to m_bits and unpack from it  
    ◆ uvm_pack_bitstream_t is a typedef for bit [4095:0]  
int count = 0;  
    ◆ Index into m_bits  
    ◆ Increment count when packing and unpacking to index into the  
m_bits vector
```

Notes:

Example Override of do_pack()

Reference Page

```
class alu_txn extends uvm_sequence_item;
  ...
  rand op_type_t mode;
  bit [3:0] done;
  rand shortint unsigned val1;
  rand shortint unsigned val2;
  alu_result m_result; // object that holds the result
  ...
  virtual function void do_pack(uvm_packer packer);
    super.do_pack(packer);
    packer.m_bits[packer.count +: 32] = mode;
    packer.count += 32;
    packer.m_bits[packer.count +: 4] = done;
    packer.count += 4;
    packer.m_bits[packer.count +: 16] = val1;
    packer.count += 16;
    packer.m_bits[packer.count +: 16] = val2;
    packer.count += 16;
    m_result.do_pack(packer);
  endfunction
  ...
endclass
```

```
# Example ALU txn:
#   mode      : MUL
#   done      : 'b0000
#   val1      : 'h01b6
#   val2      : 'h00a9
#   result    : 'h2126
#
#   m_bits =
2126_00a9_01b6_0_00000002
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu_transaction

68

Notes:

Example Override of do_unpack()

Reference Page

```
class alu_txn extends uvm_sequence_item;
  ...
  rand op_type_t mode;
  bit [3:0] done;
  rand shortint unsigned val1;
  rand shortint unsigned val2;
  alu_result m_result; // object that holds the result
  ...
  virtual function void do_unpack (uvm_packer packer);
    super.do_unpack(packer);
    mode = op_type_t'(packer.m_bits[packer.count +: 32]);
    packer.count += 32;
    done = packer.m_bits[packer.count +: 4];
    packer.count += 4;
    val1 = packer.m_bits[packer.count +: 16];
    packer.count += 16;
    val2 = packer.m_bits[packer.count +: 16];
    packer.count += 16;
    m_result.do_unpack(packer);
  endfunction
```

Notes:



```
function void record(uvm_recorder recorder = null)
virtual function void do_record(uvm_recorder recorder)
```

- The `record()` function records transactions for analysis
- Records properties such as address, data, start time, end time
 - A vendor-neutral interface
 - ◆ The backing implementation is vendor-specific and is implemented in the `uvm_recorder` policy class
- After recording automated fields, calls `do_record()`
- `record` uses an `uvm_recorder` policy object
 - Governs how recording takes place
 - Has methods for recording objects and fields
 - Predefined policy object `uvm_default_recorder` is used by default
 - ◆ Provides hooks to vendor-dependent recording mechanism

Notes:

UVM Components

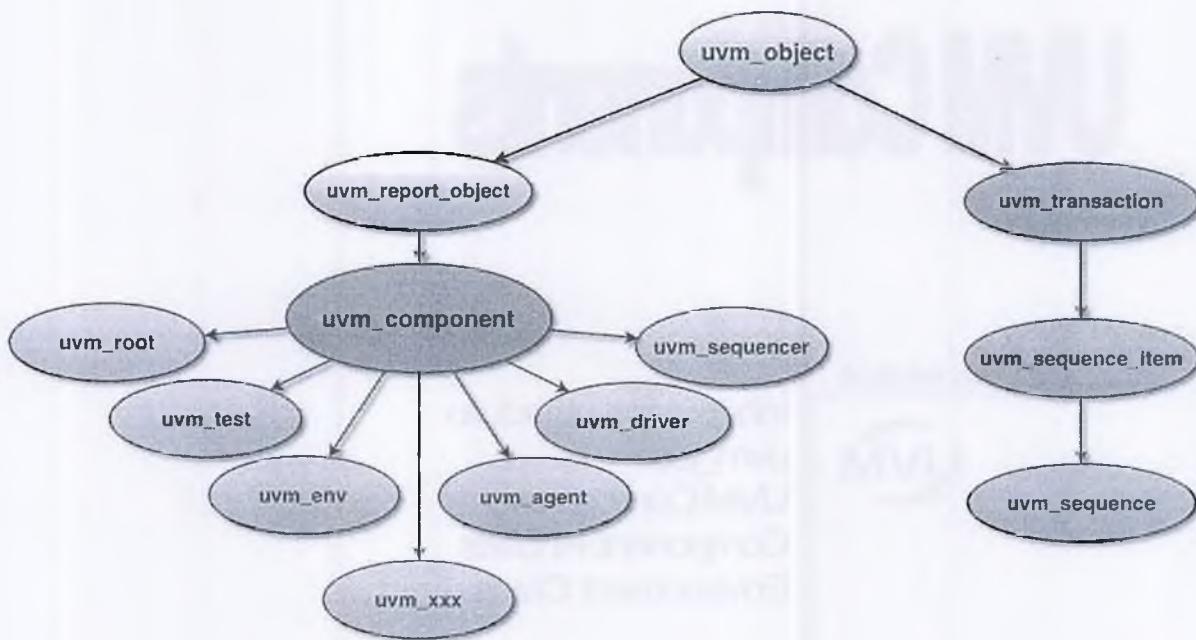
In this section



- Inheritance structure
- `uvm_top`
- UVM Component
- Component Phases
- Environment Class

Notes:

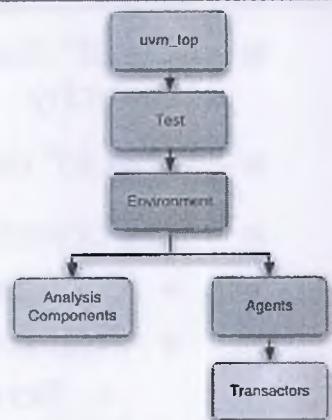
UVM Base Classes



Notes:

UVM_top

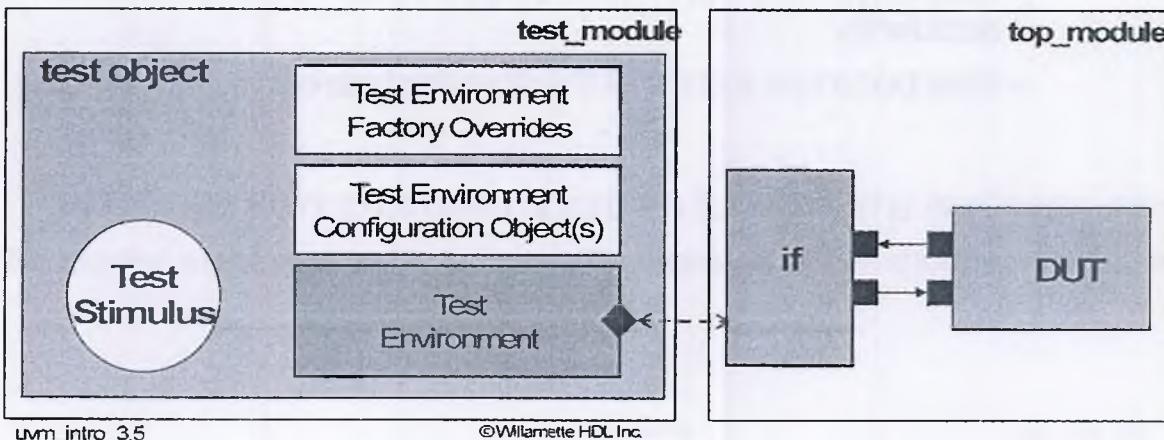
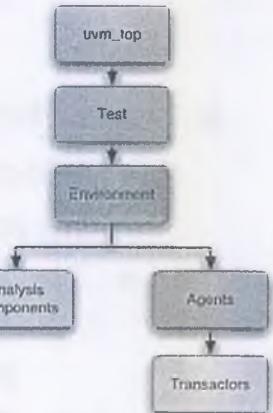
- **uvm_top** is a singleton of **uvm_root**
 - It is created and managed by UVM
 - Its scope is the **uvm_pkg**
 - The top-level component of all test components
 - ◆ All test bench components are created with a parent argument
 - ◆ If a component is created with the parent argument as "null" then it becomes a child of **uvm_top**
 - This is typically not a good thing and will cause problems if done accidentally
 - Else becomes a child of the specified parent
- **uvm_top** has a number of methods used to control simulation
 - These methods and their usage will be explained as they are used/needed



Notes:

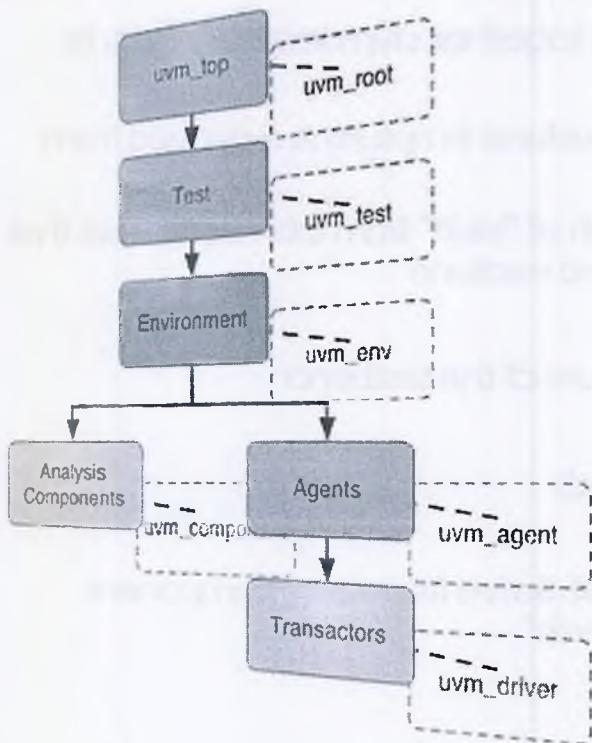
uvm test Class

- A "test" class is the top level of your class instance hierarchy
- "Should" derive from `uvm_test`
- Test object
 - Configures, then instantiates the environment
 - Instantiates and runs the test stimulus (a sequence)
 - ◆ Sometimes referred to as the test controller
- We will cover more about the test class, configurations, sequences, the factory and factory overrides later

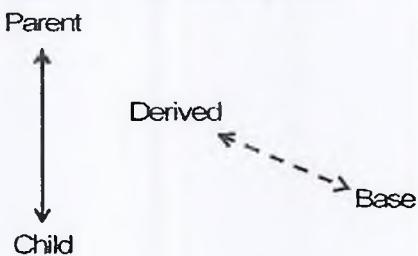


Notes:

Structural Hierarchy vs. Inheritance Hierarchy



- There are 2 orthogonal "hierarchy" concepts in UVM.
- When we talk about "hierarchy" in this training class, we usually mean structural hierarchy, where "parent" means the containing object closer to `uvm_top`
- Sometimes this is confused with inheritance hierarchy, where "parent" can mean the base class from which a class is derived
- We will try to reserve the term "parent" and "child" for structural hierarchy and use the terms "base" and "derived" for inheritance



Notes:

uvm_component Class

- The main UVM base class for putting together uvm testbenches is **uvm_component**
 - All the test bench objects that are structural in nature are derived from this class
 - Structural components inherit a bunch of "stuff" from this base class that will be covered in upcoming slides and sections
 - ◆ Phasing of the simulation
 - ◆ Managing the hierarchical structure of the testbench
 - ◆ Configuration
 - ◆ Reporting object (already covered)
 - ◆ Factory for creating components
 - We will call the classes we create that derive from **uvm_component** "components" or "structural components"

Notes:

Component Constructor

```
function new( string name , uvm_component parent);
```

UVM

Two required arguments!

- **string name**
 - ◆ The local instance name of the component
 - Used in its hierarchical path name
 - » Assembled automatically
 - Must be unique at the level it exists
- **uvm_component parent**
 - ◆ The component's *structural* parent
 - If a component is instantiated inside another component it has a parent
 - » Set argument value to this when creating child components

We will look at how Hierarchical names and connections are made and handled later

Willamette HDL

uvm_intro_3.5

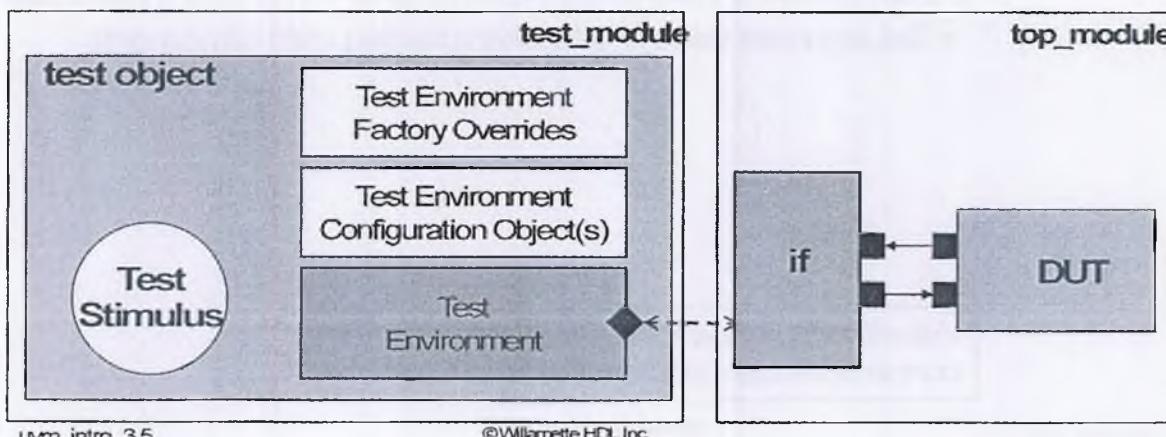
©Willamette HDL Inc.

77

Notes:

uvm_env Class

- Inherits from `uvm_component`
- The top level container class that contains all the structural components is derived from the `uvm_env` class
 - We'll refer to this class as the "*environment class*"
 - There may be multiple environment classes for a single simulation
 - All the structural verification components of the testbench are contained within environment classes
 - The environment class is usually instantiated inside a test object
 - Usually, but not always, there is an environment for each DUT interface



78

Notes:

UVL Phases

- All components go through a series of three stages during their lifetime
 - Construction
 - ◆ At the beginning of simulation, components create the hierarchical test environment
 - Run-time
 - ◆ Components run the test stimulus and measure the DUT response
 - Cleanup
 - ◆ Components decide and report the outcome of the test
- Rather than writing a component's lifetime behavior in one large task
 - Stages are split into separate tasks and functions.
- These stages are regulated by UVM, and are called phases

Notes:

Phase Management

- Fortunately we don't have define and create our own phases!
 - UVM pre-defines and declares a set of phases
- For components phase behavior is implemented in specific functions and tasks
- The general form of the phase method is:
`phasename_phase(uvm_phase phase)`
 - ◆ Phase functions and tasks have a single argument of type `uvm_phase`
 - ◆ Phase methods are inherited from `uvm_component`
 - Do nothing by default
 - Components "participate" in a phase by overriding the inherited phase method

Notes:

Phasing

- The component phases are synchronized by UVM
 - All components finish one phase before the next phase is started
- During a phase, UVM will progress through all components in an orderly fashion
 - Each component will get a chance to execute its behavior for the phase
 - Depending on the phase, the order can be top-down, bottom-up, or concurrent (i.e. no order)



uvm_intro_3.5

©Willamette HDL Inc.

81

Notes:

Phase Schedules

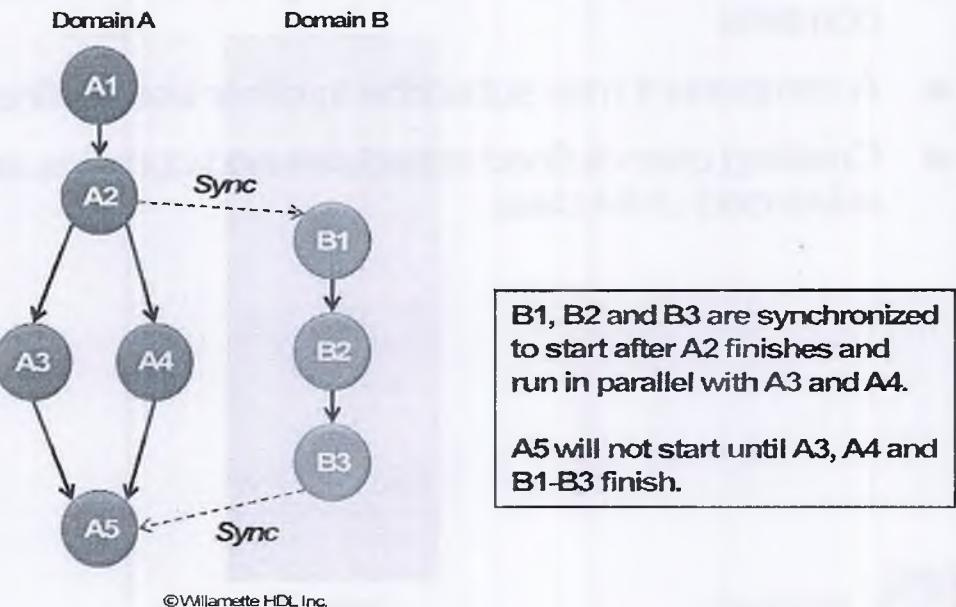
- A schedule is a group of one or more phase/state nodes linked together by a graph structure
 - Nodes in a schedule are linked together as a Directed Acyclic Graph (DAG)
 - ◆ Like a flow chart showing sequential and parallel relationships
 - Each schedule node points to a phase and holds the execution state of that phase
 - Executed by stepping through the nodes in the graph order



Notes:

Phase Domains

- Phase schedules are grouped into **domains**
- You can have several copies of a schedule in different domains
 - Schedule copies in different domains are executed independently (i.e. concurrently)
 - Phases in different domains can be synchronized
- Domains can be used for separate power and reset domains in the DUT



Notes:

Pre-defined Domains

- UVM pre-defines two domains
 - `common`
 - ◆ This is the OVM backward compatible domain
 - `uvm`
 - ◆ Has a default schedule named `uvm_sched`
- All components are automatically subscribed to the `common` and `uvm` domains
- A component may subscribe to other user defined domains
- Creating user-defined schedules and domains are covered in the advanced UVM class



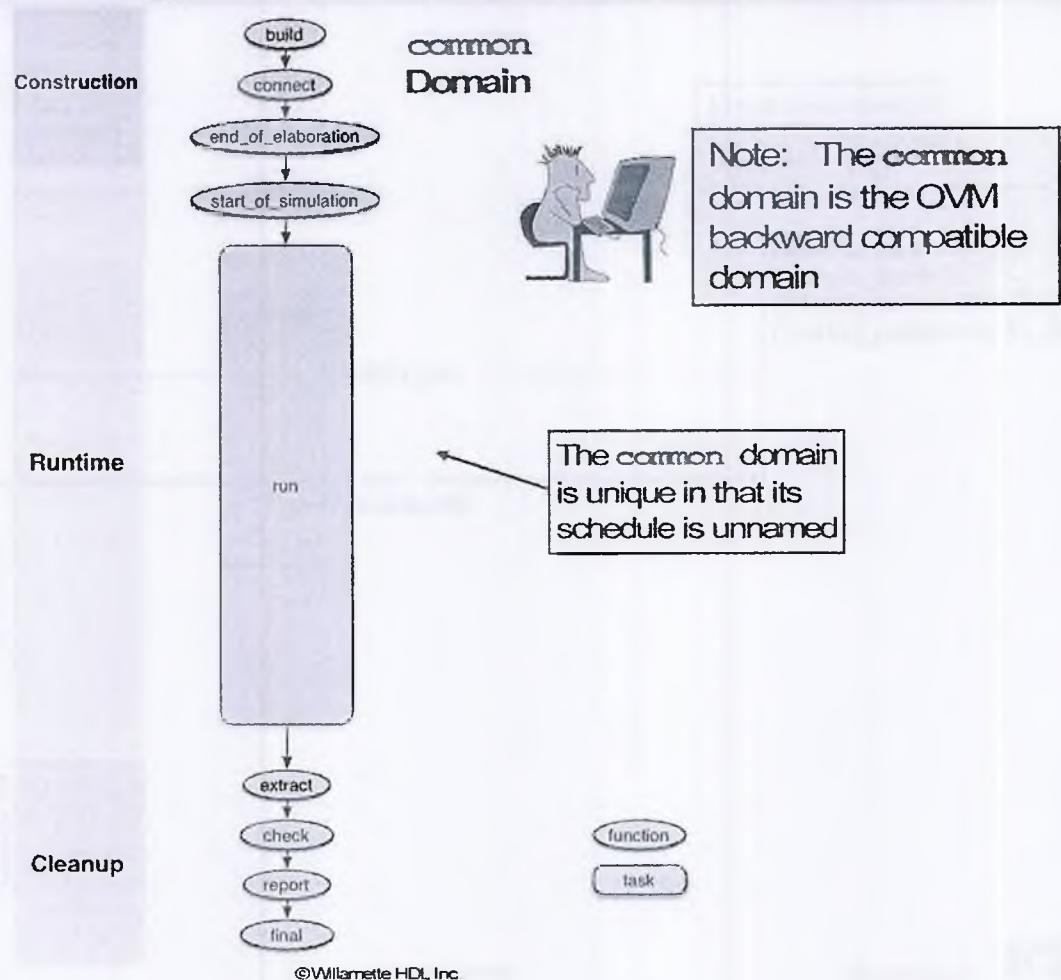
uvm_intro_3.5

©Willamette HDL Inc.

84

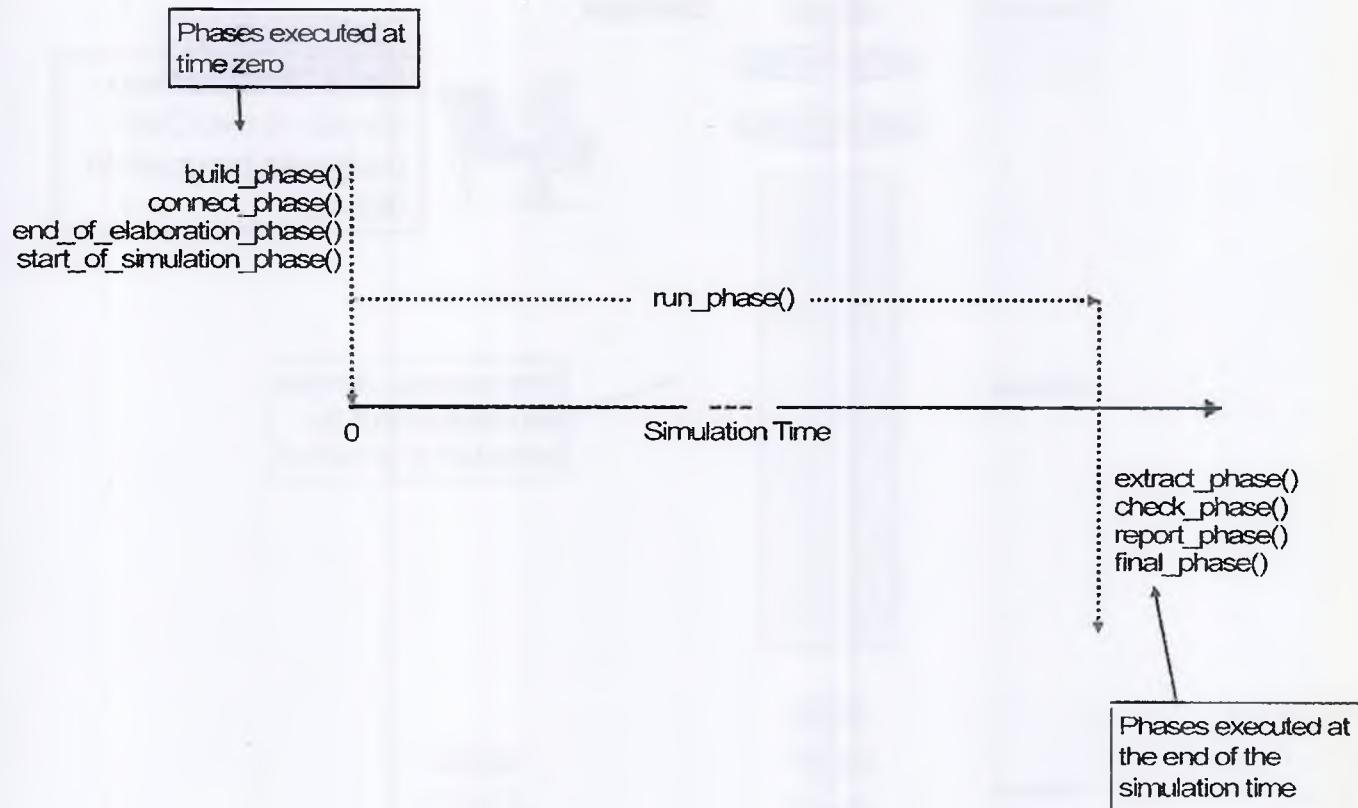
Notes:

common Domain Schedule & Phases



Notes:

UVM Phase Timeline for common Domain



Notes:

Component Phasing Methods (common domain)

- Override these inherited methods in your components:

```
virtual function void build_phase(uvm_phase phase)
```

- For constructing child components and other structural things

```
virtual function void connect_phase(uvm_phase phase)
```

- For connecting up the ports of components etc.

```
virtual function void end_of_elaboration_phase(uvm_phase phase)
```

- Set up after components are built and constructed

```
virtual function void start_of_simulation_phase(uvm_phase phase)
```

- Set up before simulation starts

```
virtual task run_phase(uvm_phase phase)
```

- Run behavior of the component – generate stimulus and check results

```
virtual function void extract_phase(uvm_phase phase)
```

- ◆ Retrieve and process information from scoreboards, monitors etc

```
virtual function void check_phase(uvm_phase phase)
```

- ◆ Check that DUT behavior is correct and identify testbench execution errors

```
virtual function void report_phase(uvm_phase phase)
```

- ◆ Display or log simulation results

```
virtual function void final_phase(uvm_phase phase)
```

- ◆ Complete any outstanding testbench actions



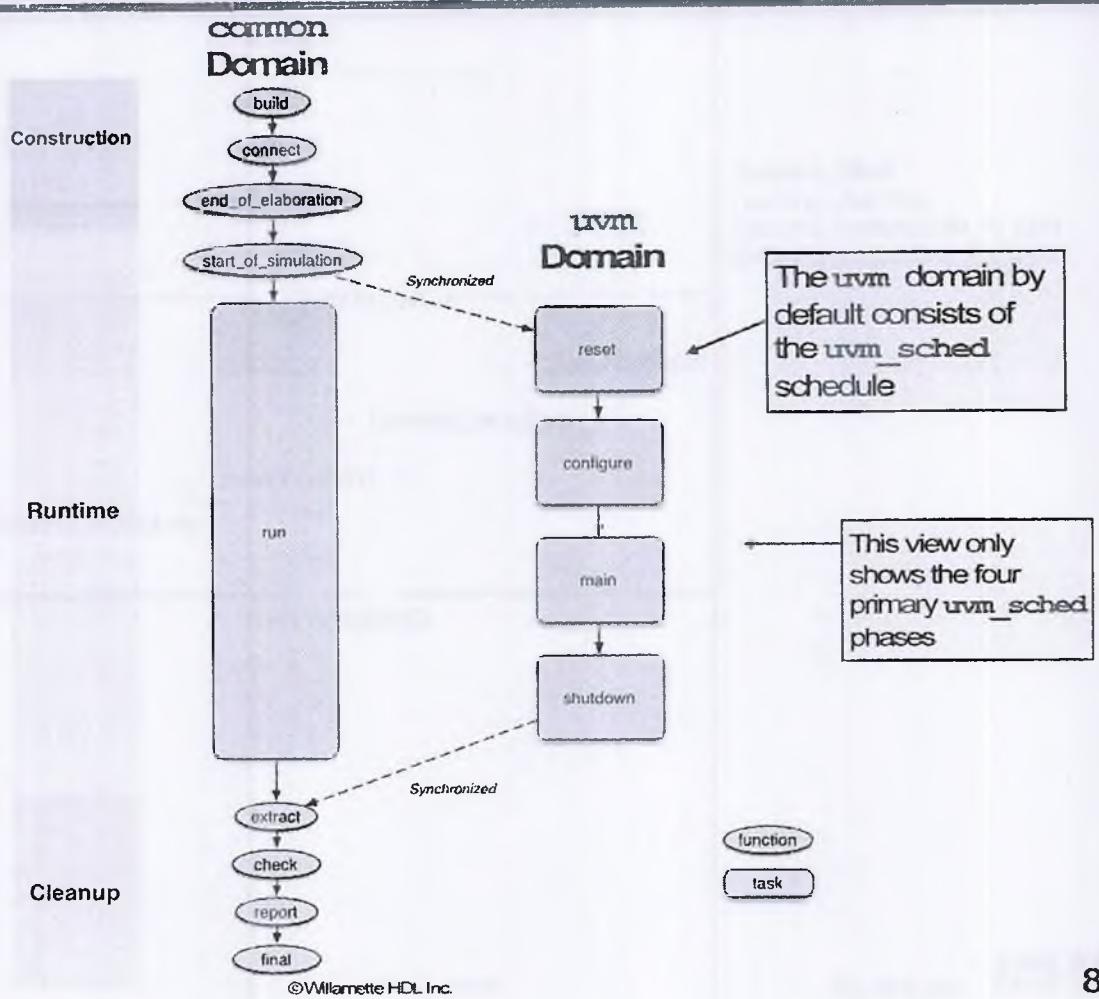
Notes:

Run-Time Activities

- In general, four (serial) activities occur during run-time
 - DUT and component initialization (reset)
 - DUT configuration (registers etc.)
 - Stimulus generation & checking
 - Drain of stimulus through the DUT and scoreboards
- The `common` domain only has a single task phase (`run_phase`)
 - Must manually synchronize the above mentioned activities between processes in different components
 - ◆ Use barriers, objections (Advanced class topics) and events for synchronization
- `uvm` domain
 - Alternate solution
 - Multiple task phases
 - ◆ Run in parallel with the `run_phase()` of the `common` domain

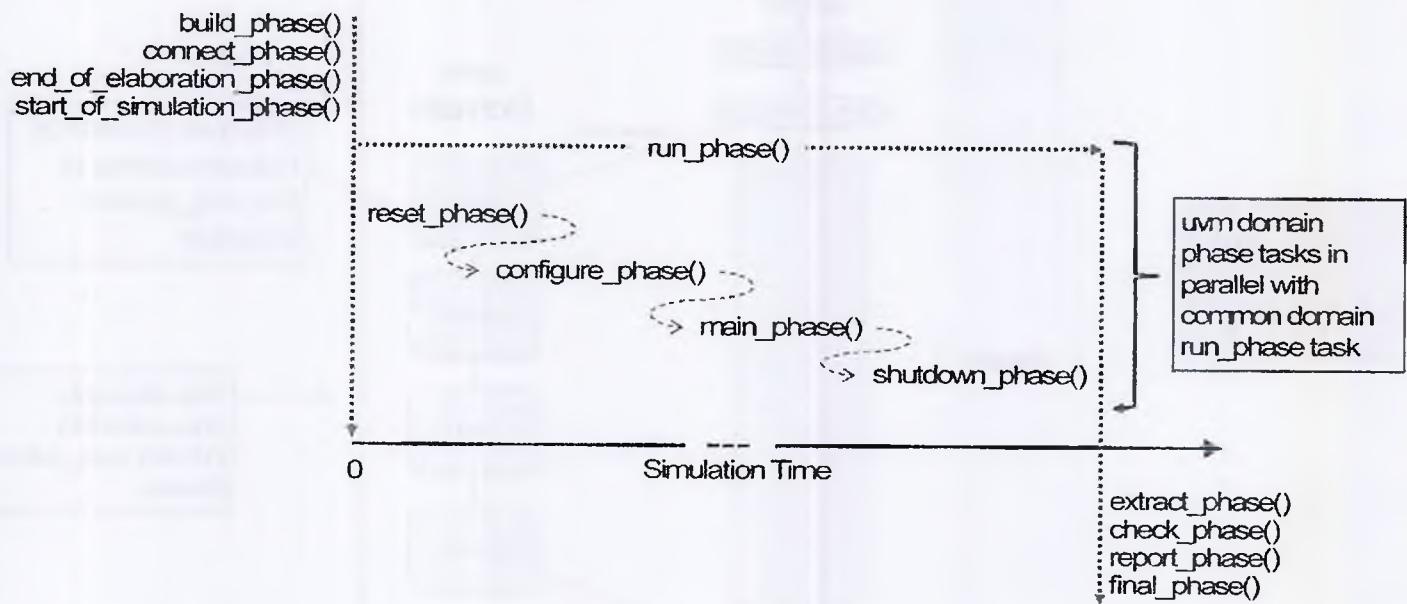
Notes:

uvm Domain Phases



Notes:

UVM Phase Timeline with common & uvm Domains



Notes:

Component Phasing Methods [uvm_domain]

```
virtual task pre_reset_phase(uvm_phase phase)
virtual task reset_phase(uvm_phase phase)
virtual task post_reset_phase(uvm_phase phase)
```

- Set component to an initial state

uvm
Domain

pre_reset

reset

post_reset

pre_configure

configure

post_configure

pre_main

main

post_main

pre_shutdown

shutdown

post_shutdown

```
virtual task pre_configure_phase(uvm_phase phase)
virtual task configure_phase(uvm_phase phase)
virtual task post_configure_phase(uvm_phase phase)
```

- Program DUT registers to prepare for test

```
virtual task pre_main_phase(uvm_phase phase)
virtual task main_phase(uvm_phase phase)
virtual task post_main_phase(uvm_phase phase)
```

- Test stimulus is generated and applied to the DUT

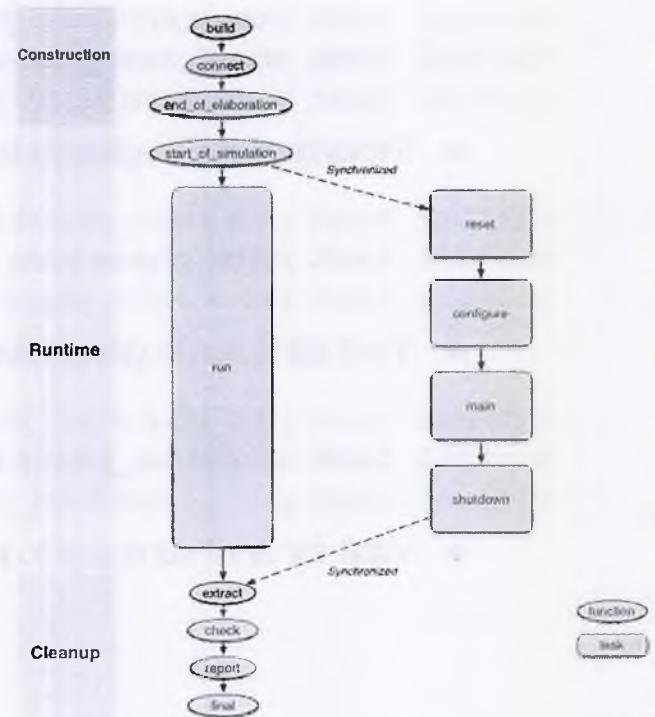
```
virtual task pre_shutdown_phase(uvm_phase phase)
virtual task shutdown_phase(uvm_phase phase)
virtual task post_shutdown_phase()
```

- Wait for final stimulus to propagate and drain through the DUT

Notes:

m_run_phases() - UVM Phases

- The phases are called by the `m_run_phases()` task
 - More later on when & how this task is called
 - May be considered a "phaser machine" which traverses the master schedule graph executing the phases
- It starts by executing the `build_phase` in the `common` domain



Notes:

m_run_phases() - UVM Phases

- All function phases except build_phase are bottom up
 - Bottom up means the child is phased (phase method called) before the parent
 - build_phase is a top down phase
- All task based phases are concurrent



uvm_intro_3.5

© Willamette HDL Inc.

93

Notes:

Additional UVM Phase Features

The additional features below are defined in the UVM specification but . . .

- There are issues/problems in the implementation
- UVM committee has a sub-committee addressing issues
- Very likely some of these features will be deprecated
- Recommendation for now is "don't use"

■ Additional features:

- You can jump from one phase to any other phase, backward or forward
- You can define your own
 - ◆ Schedules
 - ◆ Domains
 - ◆ Phases (you can add them to new or existing schedules)
- Phases in different schedules and domains can be synchronized
 - ◆ Predecessor
 - ◆ Successor
 - ◆ Parallel / Concurrent



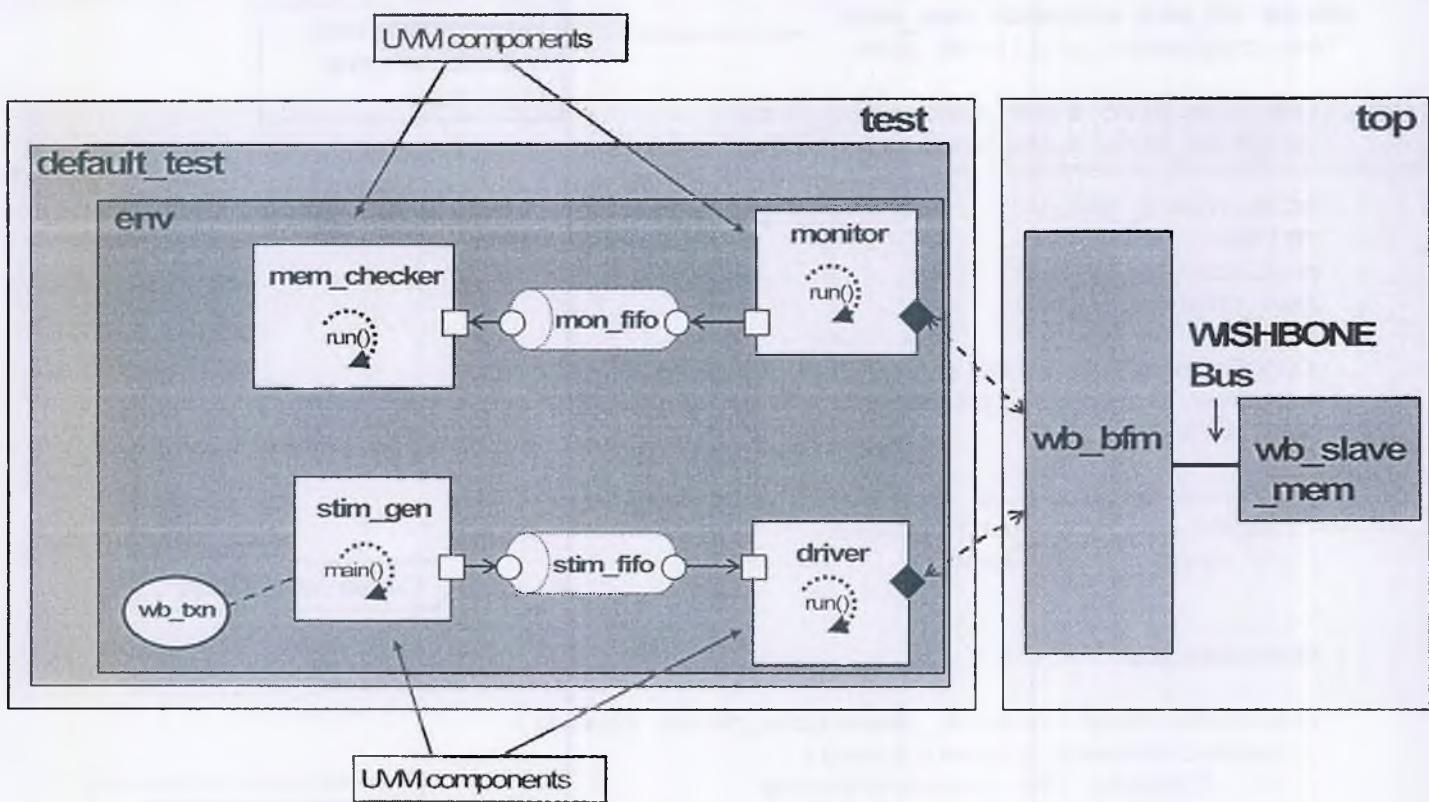
uvm_intro_3.5

©Wilamette HDL, Inc.

94

Notes:

Example: Wishbone bus Memory Testbench



Notes:

Example Environment Class: wb_env

```
class wb_env extends uvm_env;
`uvm_component_utils(wb_env)

  uvm_tlm_fifo #(wb_txn) stim_fifo;
  uvm_tlm_fifo #(wb_txn) mon_fifo;

  stim_gen s_gen;
  driver   drv;
  monitor  mon;
  mem_checker chk;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // create components
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // Connect the ports/exports
  endfunction
endclass
```

Environment class
extends uvm_env
base class

Declare components

Create components

Connect components

Notes:

Example Component Class - 1: stim_gen

```
class stim_gen extends uvm_component;
`uvm_component_utils(stim_gen)

uvm_blocking_put_port #(wb_txn) stim_p;

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

ALL component constructors should start out exactly like this (i.e. these three lines)

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    stim_p = new("stim_p", this);
endfunction
```

Ports are instantiated in build_phase() with name and parent

```
task main_phase(uvm_phase phase);
    wb_txn txn;
    bit[31:0] address;
    bit[31:0] data;
    ...
    repeat(num_ops) begin
        address = $urandom_range(0, 32'h0003ffff);
        data = $urandom();
        txn = wb_txn::type_id::create("txn");
        txn.init(WRITE, address, data);
        stim_p.put(txn);
    ...
endtask
endclass
```

Runtime behavior of the component is placed here

```
This component has no internal connectivity, so no connect_phase() function is needed
```

Notes:

Example Component Class - 2: driver

```
class driver extends uvm_component;
  `uvm_component_utils(driver)
  uvm_blocking_get_port #(wb_txn) stim_p;
  virtual wishbone_bus_syscon_if v_wb_bfm;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    stim_p = new("stim_p", this);
    ...
  endfunction

  task run_phase(uvm_phase phase);
    wb_txn txn;
    logic [31:0] read_data[];
    forever begin
      ...
    end
  endtask

endclass
```



This component's runtime behavior does not need separate reset, configure, main, and shutdown behaviors, so runtime behavior is modeled in `run_phase()`. Indicator is that it is implemented with an infinite loop.



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_comp

98

Notes:

File Organization

■ Recommendations

- Classes:
 - ◆ One class per file, file name same as class name
 - ◆ `class_name.svh`
 - All .svh files have compiler directives to prevent multiple include errors
- Packages
 - ◆ One package per file, file name same as package name
 - ◆ `packagename_pkg.sv`
 - Packages have imports, global variables and `include class files
 - `import uvm_pkg::*;

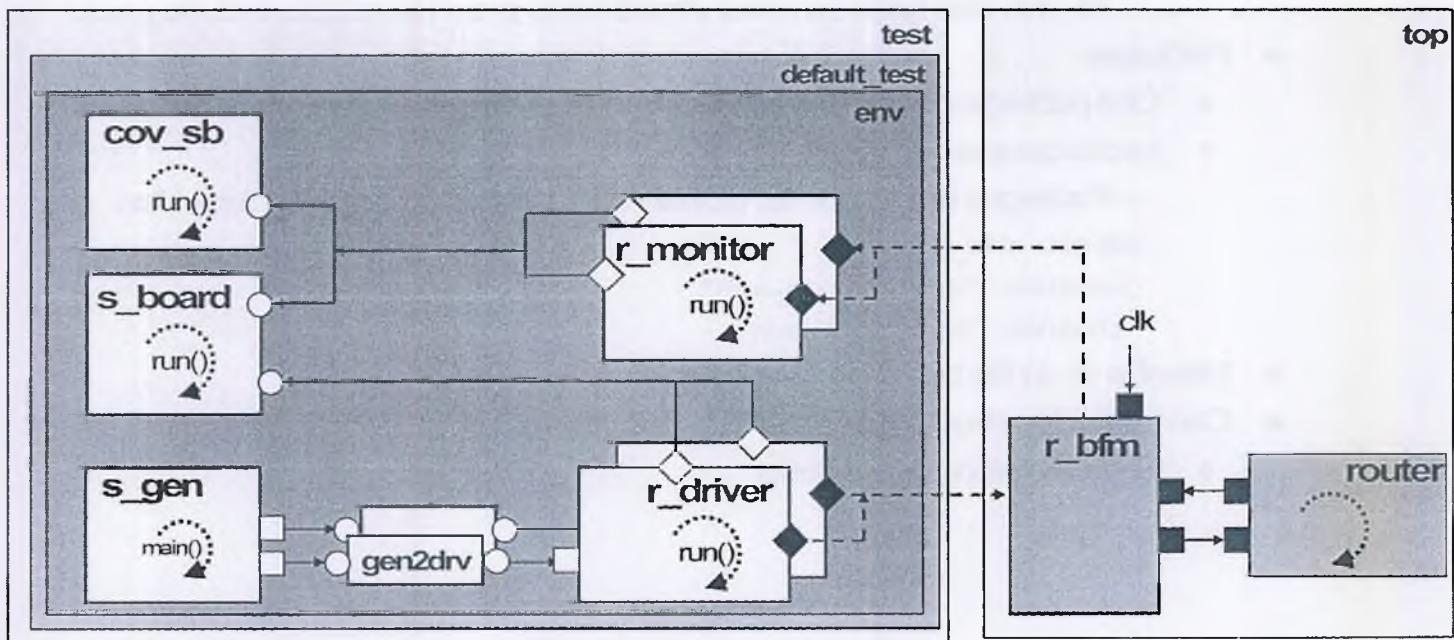
`include "uvm_macros.svh"

`include "my_class.svh"`
- Makefile or do file compiles .sv files
- Can be flat file structure or have sub-directories
 - ◆ Organize along agent basis

Notes:

Lab – UVM Components: Overview

- Working directory: components



Notes:

Lab – UVM Components: Instructions – 1

- Working directory: **components**
- Instructions
 - Edit the file **stim/stim_gen.svh**
 - ◆ Create the **stim_gen** component
 - Extend **uvm_component**
 - Declare the port
 - port array: **out_p[`ROUTER_SIZE]**
 - Blocking put port of type **Packet**
 - Write a component constructor
 - Create a **build_phase()**
 - Create the ports
 - HINT:
 - Components at the same level must have unique names
 - Use the **\$sformatf()** function to create a string with embedded numbers:
 - string name = \$sformatf("port_name_%0d", i);
 - Create a **main_phase()**
 - The body of the **main_phase()** is provided
 - Simply include the provided code in your **main_phase()**
 - Continued...

Notes:

Lab – UVM Components: Instructions – 2

- Edit the file `xactors/rtr_driver.svh`
 - ◆ Create the driver component
 - Extend from `uvm_driver`
 - Declare the port
 - `in_p` a blocking get port of type `Packet`
 - Write a component constructor
 - Create a `build_phase()`
 - Create the port
 - Note: the other components are *already* modified
- Compile & run

Notes:

Lab – UVM Components: Partial Sample Output

```
# UVM_INFO@0: reporter [RNTST] Running test default_test...
# Router size = 8x8
# UVM_INFO@1972300: uvm_test_top.env cov_sb [COVERAGE] ****
# UVM_INFO@1972300: uvm_test_top.env cov_sb [COVERAGE] Final Coverage = 100.000000%
# UVM_INFO@1972300: uvm_test_top.env cov_sb [COVERAGE] 100% Coverage met with 266 transactions
# UVM_INFO@1972300: uvm_test_top.env cov_sb [COVERAGE] ****
#
# UVM_INFO@1972300: uvm_test_top.env.s_board [Packet_Comparator] Matches: 991
# UVM_INFO@1972300: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
# UVM_INFO@1972300: uvm_test_top.env.s_board [Packet_Comparator] Missing: 1
#
#
#— UVM Report Summary —
#
# ** Report counts by severity
# UVM_INFO: 8
# UVM_WARNING: 0
# UVM_ERROR: 0
# UVM_FATAL: 0
# ** Report counts by id
#[COVERAGE] 4
#[Packet_Comparator] 3
#[RNTST] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 1972300 ns Iteration: 100 Instance: /test
```

Sol



uvm_intro_3.5

© Willamette HDL Inc.

103

Notes:

Creating Components & Running the Simulation

In this section



Creating Components Using the Factory
Starting and Stopping the Test

Reference
Registry class

Notes:

Introducing the UVM Factory

- A well-established object-oriented best practice, a creation pattern is called the **Factory Pattern**
- The **factory** is singleton of the `uvm_factory` class
 - Used to create objects dynamically
 - Its main benefit is that the type of object that is constructed can be specified at *runtime* for maximum flexibility
 - ◆ a.k.a. "*Polymorphic Construction*"
- The **factory** itself is not a part of the testbench component hierarchy
 - Its scope is the `uvm_pkg`
- The **factory** can create any kind of `uvm_object`
 - Testbench components or transaction objects
- The **factory** provides an **override** mechanism
 - Helps when making modifications to an existing testbench
 - More on overrides in a later section

Notes:

Registering Types with the Factory

- For the factory to be able to create a testbench component or transaction object
 - Must first *register* that type with the factory
 - Limitations:
 - ◆ Class may have extra constructor arguments besides name & parent
 - **BUT** extra arguments must have default values and the factory will create using the default values
- Registration is done by using macros:
 - ◆ `uvm_component_utils(*typename*)
 - ◆ `uvm_object_utils(*typename*)
- Macros do the following:
 - ◆ Declare a `typedef` of a "wrapper" type in the class definition
 - The `typedef` is named `type_id`
 - Uses static initialization technique to register type with factory
 - ◆ Define a `get_type()` function that returns `type_id::get()`
 - ◆ Define a `get_type_name()` function that returns the type name as a string

static
Initialization



Sometimes you need to do the registration "by hand".
Reference section has examples on how to do this

Notes:

Example Registering Component with a Macro

```
'include "uvm_macros.svh"
```

Need to include this file.
Most of the time it is
included in the package

```
class stim_gen extends uvm_component;  
  `uvm_component_utils(stim_gen)  
  ...  
endclass
```

Argument to macro is the
name of the class

Macro declares the `type_id` registry `typedef`
for you and generates the `get_type()` and
`get_type_name()` functions

Note: no semicolon!

Notes:

Example Registering Object with a Macro

```
'include "uvm_macros.svh"
```

Need to include this file.
Most of the time it is
included in the package

```
class wb_txn extends uvm_sequence_item;  
`uvm_object_utils(wb_txn)
```

```
...  
endclass
```

Argument to macro is the
name of the class

Macro declares the `type_id` registry `typedef`
for you and generates the `get_type()` and
`get_type_name()` functions

Note: no semicolon!

Notes:

Factory – Creating Testbench Components

- To create a component, call the static `create` function on the type's `type_id`
 - Do this in the `build_phase()` function in the parent class

UVM

```
static function T create (string name,  
                         uvm_component parent, string context="")
```

- `string name`
 - Required leaf instance name of component to create
- `uvm_component parent`
 - Required handle of parent component (usually `this`)
- `string context`
 - Hierarchical path as a string
 - Leave blank - automatically generated inside `create()`

WILLAMETTE
HDL

uvm_intro_3.5

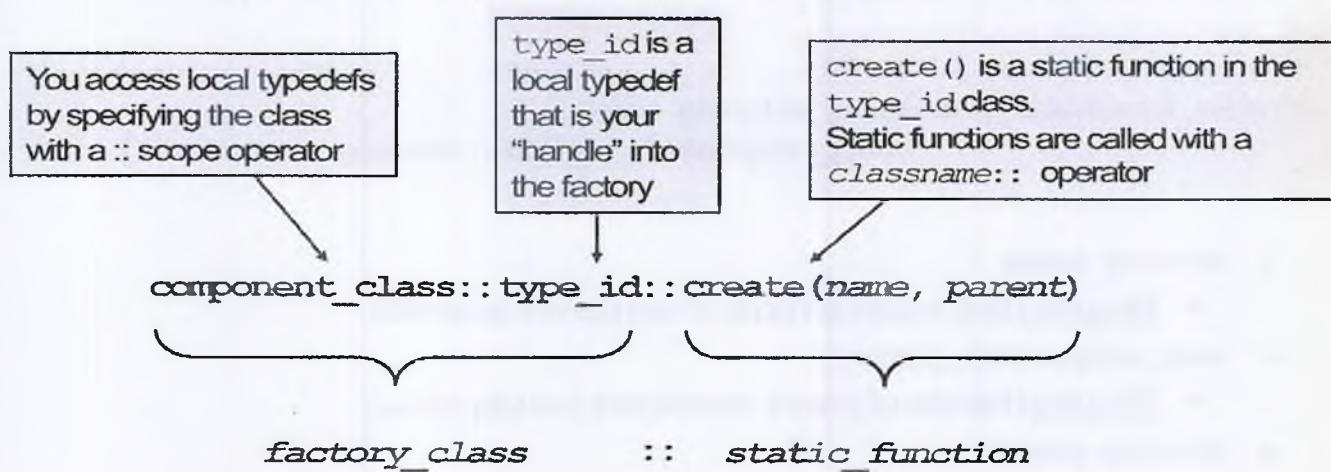
© Willamette HDL Inc.

109

Notes:

Factory – Creating Testbench Components

- To create a component, call the static `create` function on the type's `type_id`
 - Do this in the `build_phase()` function in the parent class



Notes:

Example Using a Factory to Create a Component

```
class wb_env extends uvm_env;
`uvm_component_utils(wb_env)
  uvm_tlm_fifo #(wb_txn) stim_fifo;
  uvm_tlm_fifo #(wb_txn) mon_fifo;

  stim_gen s_gen;
  driver   drv;
  monitor  mon;
  mem_checker chk;
  ...
function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  s_gen = stim_gen::type_id::create("s_gen", this);
 drv = driver::type_id::create("drv", this); ←
  mon = monitor::type_id::create("mon", this);
  chk = mem_checker::type_id::create("chk", this);

  stim_fifo = new("stim_fifo", this);
  mon_fifo = new("mon_fifo", this);
endfunction
...
endclass
```

These components are registered in the factory

Match to handle name is optional but recommended

this object is the parent – Required!!

stim_gen::type_id, driver::type_id etc. are the registry types. You are calling the static create() function in the registry



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/Wb_mem_comp

111

Notes:

Factory – Creating Transaction Objects

- To create a transaction object

- Similar to a component – call the static `create` function on the type's `type_id`

UVM

```
static function T create (           string name="",
                           uvm_component parent=null,
                           string context="" )
```

- `string name`
 - ◆ Optional leaf instance name of component to create
- `uvm_component parent`
 - ◆ Optional handle of parent component (Usually leave empty)
- `string context`
 - ◆ Hierarchical path as a string
 - ◆ Leave blank - automatically generated inside `create()`

WHDL

uvm_intro_3.5

© Willamette HDL Inc.

112

Notes:

Example Using a Factory to Create an Object

```
class stim_gen extends uvm_component;
`uvm_component_utils(stim_gen)

uvm_blocking_put_port #(wb_txn) stim_p;
...

task main_phase(uvm_phase phase);
    wb_txn txn;
    bit[31:0] address;
    bit[31:0] data;
    phase.raise_objection(this);
    repeat (num_ops) begin
        address = $urandom_range(0, 32'h0000ffff);
        data    = $urandom();
        txn = wb_txn::type_id::create("txn");
        txn.init(WRITE, address, data);
        stim_p.put(txn);
        txn = wb_txn::type_id::create("txn");
        txn.init(READ, address);
        stim_p.put(txn);
    end
    phase.drop_objection(this);
endtask
endclass
```

Parent handle is optional for objects.
Typically you don't supply one

String name is optional



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_comp

113

Notes:

Registering Parameterized Classes - Components

```
'include "uvm_macros.svh"
```

```
class stim_gen #(int N = 1) extends uvm_component;  
  `uvm_component_param_utils(stim_gen #(N))  
  ...  
endclass
```

Use the *param_utils* version of the macro

You must specify the parameter specialization as the macro argument

Declare specialization

Example:

```
stim_gen #(4) s_gen;
```

...

Create component of specialization type

```
s_gen = stim_gen #(4)::type_id::create("s_gen", this);
```

Notes:

Registering Parameterized Classes - Objects

```
'include "uvm_macros.svh"

class wb_txn #(int N = 1) extends uvm_sequence_item;
  `uvm_object_param_utils(wb_txn #(N) )
  ...
endclass
```

Use the *param_utils* version of the macro

You must specify the parameter specialization as the macro argument

Declare specialization

Example:

```
wb_txn #(10) txn;
```

...

```
txn = wb_txn #(10)::type_id::create("txn");
```

Create object of specialization type

Notes:

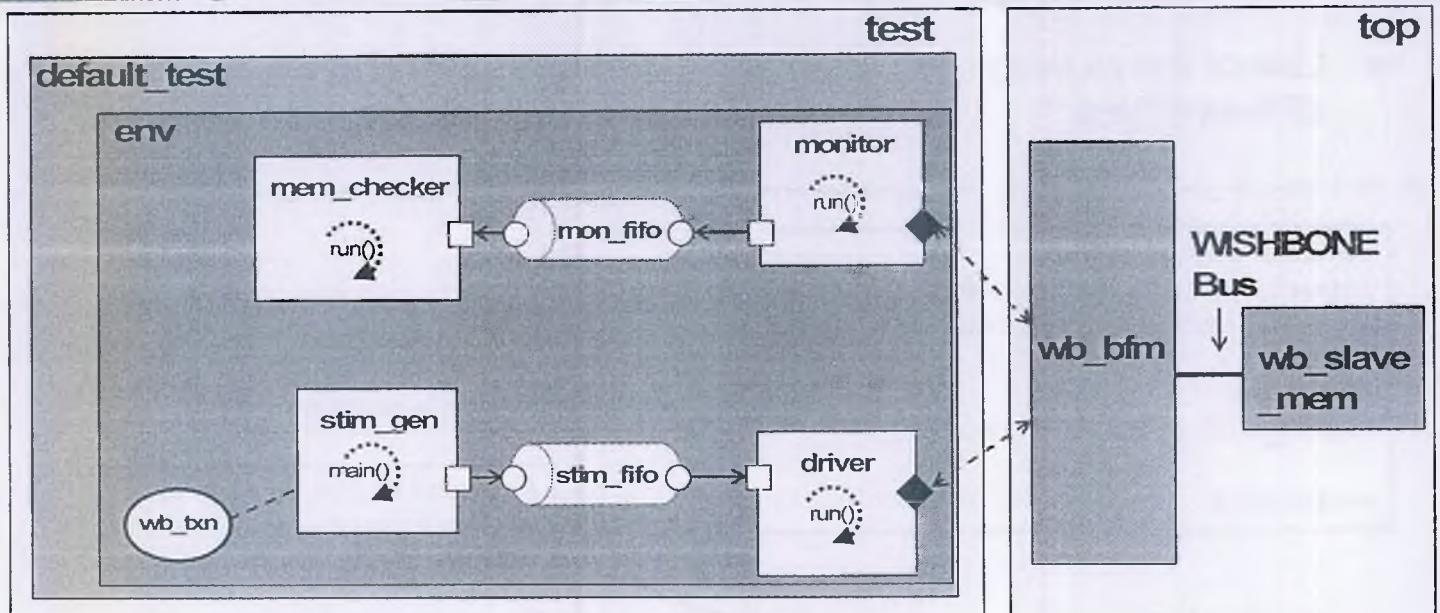
Starting Up the Test: `run_test()`

UVM `task run_test(string testname = "")`

- To start the testbench call the task `run_test()`
 - This method is defined in `uvm_pkg`
 - Call `run_test()` in an initial block of the test module
- `run_test()` uses the argument `testname` to choose which class it creates as the top level class in the testbench
 - Uses the factory to create the test object
 - Handle (scope is `uvm_pkg` i.e. global) `uvm_test_top`
 - Name: "`uvm_test_top`"
 - Parent: `null`
 - This can be overridden with the plusarg `UVM_TESTNAME`
- After `run_test()` creates the top level class it starts the phases
 - Does so by calling a task called `m_run_phases()`
- After the phases are complete `run_test()` calls `$finish()` to end the simulation

Notes:

Example Starting the Test: test[nm_test arg]



You must import `run_test()` from `uvm_pkg`, and import the test class from its package

```

module test;
import uvm_pkg::run_test;
import wishbone_pkg::default_test;

initial
    run_test("default_test");
endmodule

```

Creates `default_test` type with name of "uvm_test_top" and assigns to the handle `uvm_test_top`

Notes:

Example Starting the Test: test (plusarg)

- Use of the plusarg UVM_TESTNAME allows for selection at run-time of different "tests"

```
module test;
import uvm_pkg::run_test;
import wishbone_pkg::default_test;

initial
  run_test("default_test");

endmodule
```

You could think of the argument to run_test() as the default test for when there is no plusarg

Argument to run_test() is ignored if the plusarg exists on vsim command line

```
% vsim +UVM_TESTNAME=test1 top test
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_comp

118

Notes:

Ending the Test

- In UVM in order to keep the task based phases (`run_phase`, `main_phase` etc.) alive we need to raise an objection
 - Otherwise phase will terminate!
 - i.e. UVM wants to end the phase *as soon as it possibly can*
 - ◆ Objection must be raised immediately
- Think of raising an objection as saying "I object to stopping, I'm not done yet!"
 - Or "don't stop yet, I haven't even got started!"
- When you wish the phase to end, drop the objection
 - When all the objections to the phases are dropped then the phase will terminate
- Not all components need to use objections
 - All components continue in their task based phases until all objections are dropped (even if they did not raise an objection)

Notes:

Objecting to the End of a Phase

- Phases have convenience methods for raising and dropping objections

```
virtual function void raise_objection (uvm_object obj,  
                                     string description="", int count=1);  
    phase_done.raise_objection(obj,description,count);  
endfunction
```



```
virtual function void drop_objection (uvm_object obj,  
                                     string description="", int count=1);  
    phase_done.drop_objection(obj,description,count);  
endfunction
```

Notes:

Example Objection in Phase: stim_gen

```
class stim_gen extends uvm_component;
`uvm_component_utils(stim_gen)

uvm_blocking_put_port #(wb_txn) stim_p;
***

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    stim_p = new("stim_p", this);
endfunction

task main_phase(uvm_phase phase);
    wb_txn txn;
    bit[31:0] address;
    bit[31:0] data;
    phase.raise_objection(this, "Generate Stimulus");
    repeat (num_ops) begin
        address = $urandom_range(0, 32'h0003ffff);
        data = $urandom();
        txn = wb_txn::type_id::create("txn");
        txn.init(WRITE, address, data);
        stim_p.put(txn);
        txn = wb_txn::type_id::create("txn");
        txn.init(READ, address);
        stim_p.put(txn);
    end
    phase.drop_objection(this, "Done");
endtask
endclass
```

Raise and drop the objection
on the provided argument
"phase" of type uvm_phase



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_comp

121

Notes:

Task Based Phases Use Model

■ Recommended use model:

- Components or objects that generate stimulus should use the phases in the `uvm_sched` schedule(`reset`, `configure`, `main`, `shutdown`)
 - ◆ Raise objection before stimulus starts
 - ◆ Lower objection when finished generating stimulus
 - ◆ Do not raise and lower around every stimulus object – performance hit!
- Components or objects that are free running (i.e. have forever loops) such as scoreboards, monitors, drivers should use `run_phase`
 - ◆ Do not raise and lower objections
- This will be discussed more in a later section
- The issue of draining scoreboards etc. is addressed in the Advanced UVM class

Notes:

Stopping the Test – OVM Backward Compatible

- The OVM mechanism for ending the test is deprecated in UVM
 - To simulate with OVM semantics you must set this command line plusarg:
 - ◆ `+UVM_USE_OVM_RUN_SEMANTIC`
 - ◆ If you don't set this your converted OVM code is likely to do nothing!

Notes:

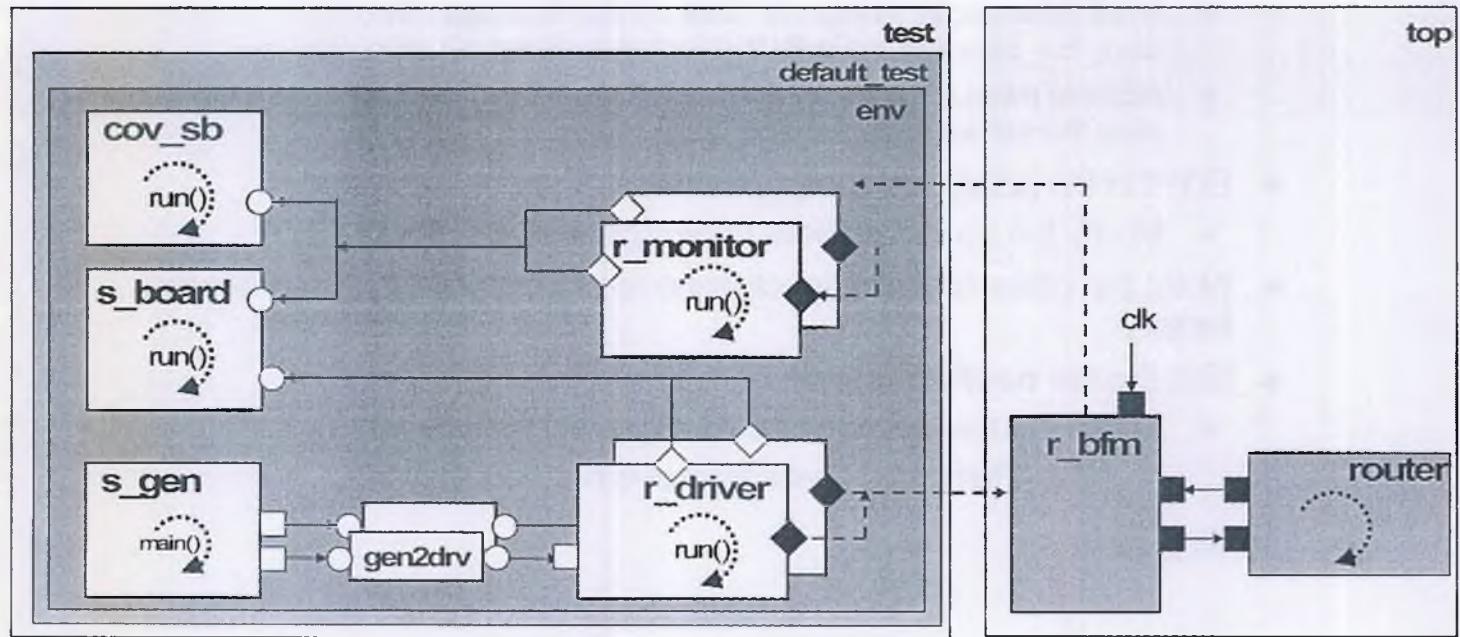
Watch Dog Timer

- There is a watch dog timer for the run phase
 - The `run_phase()` methods will be terminated if they execute longer than the time out value
 - ◆ Default time out value is 0 (0 really means 9200 seconds)
 - Set the timeout value in the `uvm_top` instance
 - ◆ No method, simply set the value directly
 - `uvm_top.phase_timeout = timeout_val`

Notes:

Lab - UVM Environment

- Working directory: environment



Notes:

Lab - UVM Environment: Instructions – 1

- Working directory: **environment**
- Instructions
 - Edit the file **stim/stim_gen.svh**
 - ◆ Modify the `stim_gen` class to be registered with the factory
 - ◆ Modify the `main_phase()` task so that it creates the `txn_to_randomize` with the factory instead of using `new()`
 - ◆ Add the raise and drop objection to the `main_phase()` before and after the stimulus generation respectively
 - Edit the file **analysis/comp_sb.svh**
 - ◆ Modify the `comp_sb` class to be registered with the factory
 - Note: the other component classes are *already* registered with the factory
 - Edit the file **txm/Packet.svh**
 - ◆ Modify the `Packet` class to be registered with the factory
 - **Hint** - Remember that a `Packet` is not a component!
- Continued...

Notes:

Lab-UVM Environment: Instructions-2

■ Instructions (cont.)

- Edit the file ***env/router_env.svh***
 - ◆ Create an environment class `router_env`
 - Register the `router_env` class with the factory
 - Declare component handles for `stim_gen` (`s_gen`), `comp_sb` (`s_board`), `coverage_sb` (`cov_sb`), `rtr_driver` (`r_driver`ROUTER_SIZE`), `rtr_monitor` (`r_monitor`ROUTER_SIZE`)
 - Note the fifo's in the environment are done for you
 - Create a `build_phase()` method to create components using the factory
 - Include the indicated code
 - Create a `connect_phase()` method to connect the components
 - Include the indicated code
- Edit the file ***top_modules/test.sv***
 - ◆ Start the test environment running with "default_test" as the test class

■ Compile & run



uvm_intro_3.5

© Willamette HDL, Inc.

127

Notes:

Lab - UVM Environment: Partial Sample Output

```
# UVM_INFO @ 0: reporter [RNTST] Running test default_test...
# Router size = 8x8
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] ****
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage = 100.000000%
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] 100% Coverage met with 266 transactions
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] ****
#
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Matches: 991
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Missing: 1
#
#
#— UVM Report Summary —
#
# ** Report counts by severity
# UVM_INFO: 8
# UVM_WARNING: 0
# UVM_ERROR: 0
# UVM_FATAL: 0
# ** Report counts by id
# [COVERAGE] 4
# [Packet_Comparator] 3
#[RNTST] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 1972300 ns Iteration: 100 Instance: /test
```

Sol



uvm_intro_3.5

© Willamette HDL Inc

128

Notes:

UVM

```
class uvm_component_registry #(type T, string Tname = "")
```

- **type T**
 - ◆ The type that is being registered
 - **string Tname**
 - ◆ Name used in name-based lookup for this type in the factory
 - By convention this is an actual type name
 - ◆ **Do not** specify this string for parameterized types
- Creates a singleton object that can be retrieved with static function `get()`
 - It is only necessary to declare a specialization `typedef` to trigger creation of the wrapper
 - `Typedef` should be named `type_id`

```
typedef uvm_component_registry #(stim_gen, "stim_gen") type_id;
```

Example Registering a Component

Reference Page

```
class source extends uvm_component;
  //ports
  uvm_blocking_put_port #(Packet) out_p;

  typedef uvm_component_registry #(source, "source") type_id;

  function string get_type_name();
    return "source";
  endfunction

  function type_id get_type();
    return type_id::get();
  endfunction

  // rest of class not shown
endclass
```

Type being registered

Name for string-based lookup in the factory

Function that returns the type name of this class. Must match the string used in the registry declaration

Convenience function that returns the singleton wrapper.

Notes:

UVM

```
class uvm_object_registry #(type T, string Tname = "")
```

- **type T**
 - ◆ The type that is being registered
- **string Tname**
 - ◆ Name used in name-based lookup for this type in the factory
 - By convention this is an actual type name
 - ◆ **Do not** specify this string for parameterized types
- Creates a singleton object that can be retrieved with static function `get()`
- It is only necessary to **declare** a specialization `typedef` to trigger creation of the wrapper
- **Typedef should be named `type_id`**

```
typedef uvm_object_registry #(stim_gen, "stim_gen") type_id;
```



uvm_intro_3.5

© Willamette HDL, Inc.

131

Notes:

Example Registering an Object

Reference Page

```
class Packet extends uvm_sequence_item;
    int pkt_id;
    rand int val1, val2;
    rand op_t mode;
    int result;

    typedef uvm_object_registry #(Packet, "Packet") type_id;
    function type_id get_type();
        return type_id::get();
    endfunction

    function string get_type_name();
        return "Packet";
    endfunction

    //Rest of class not shown

endclass
```

Type being registered

Name to reference this type in the factory



Notes:

Connection to the DUT

In this section



Transactor Types
Signal Style Transactors
BFM Style Transactors



uvm_intro_35

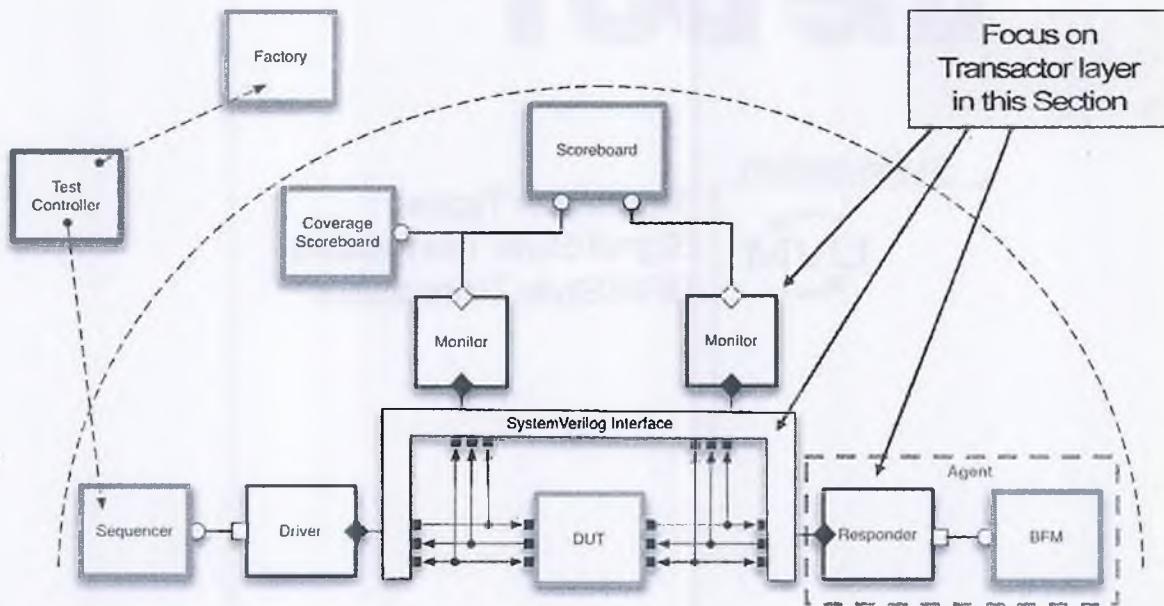
© Willamette HDL Inc

133

Notes:

UVM Class Based Testbenches

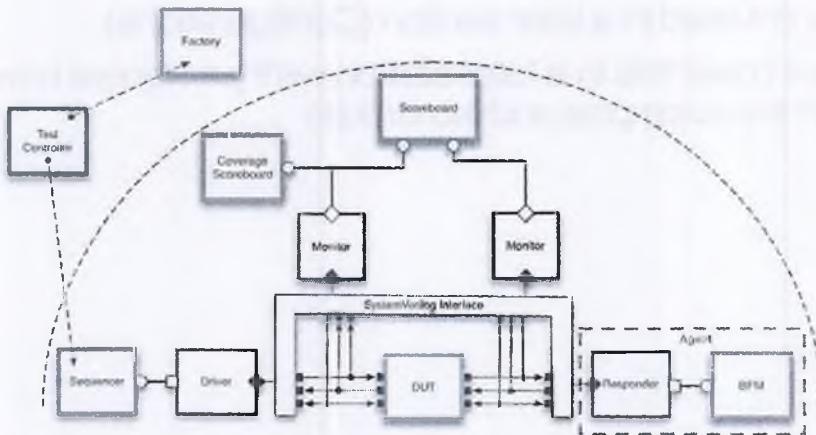
- Given the UVM TLM library and the UVM building block classes
 - We can now start putting together a class based testbench
- We introduced the components of an UVM concentric test bench structure in the overview section at the beginning of the class
 - In this section we will focus on the DUT and Transaction layer verification components and the interface between them



Notes:

Types of Transactors

- Drivers
 - Converts a stream of transactions to pin level activity
 - Example is the driver
- Monitors
 - Watches pin level activity and converts it to streams of transactions
- Responders
 - Responds to pin level activity as opposed to driving or watching



Notes:

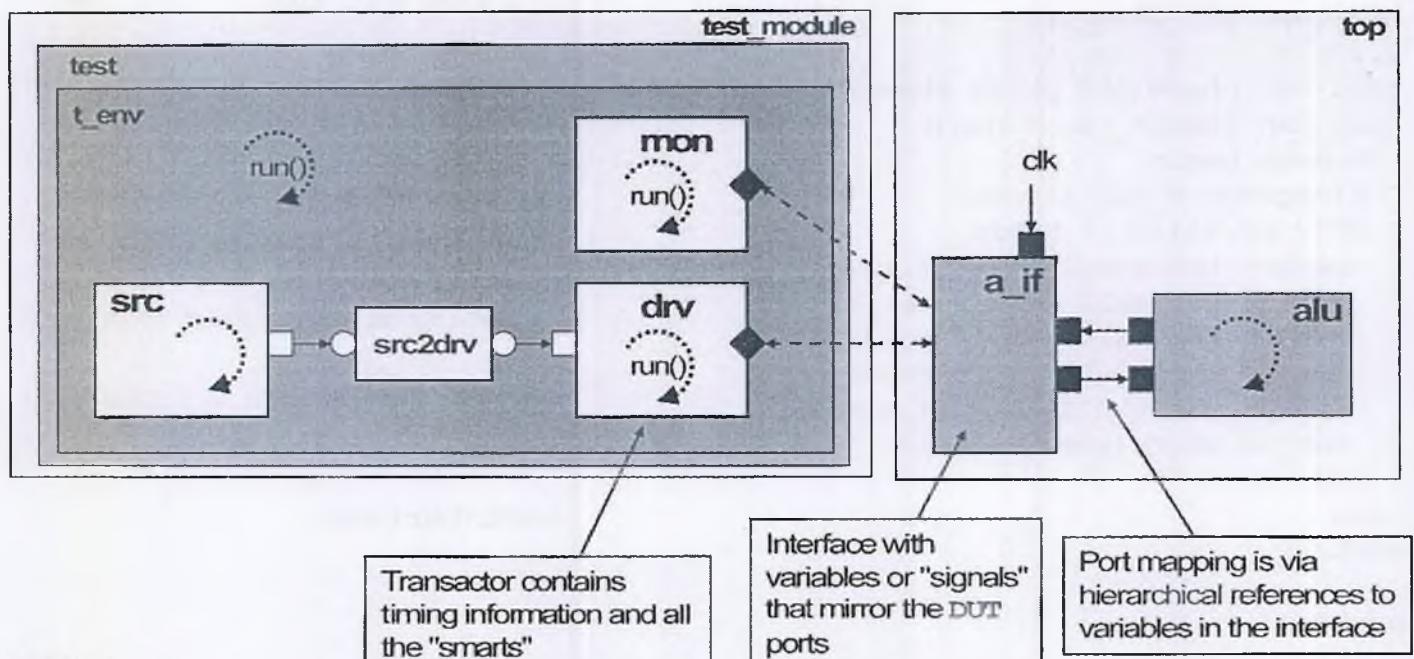
Communicating with the DUT

- Transactors are class-based and communicate with the DUT module through a virtual interface connection
 - The transactor has a virtual interface property that points to the interface connected to the DUT
 - ◆ Virtual interface review on reference slide at the end of this section
- The location of the interface connected to the DUT needs to be provided to the transactor
 - The recommended approach for accomplishing this uses a UVM feature not yet covered
 - ◆ Will be covered in a later section (Configurations)
 - ◆ Until we cover this in a later section we'll just ignore how it is being done in the examples and lab circuits

Notes:

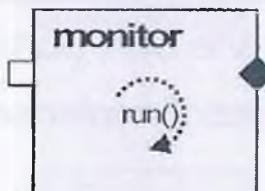
Signal Style Transactor

- Interface contains only variables (signals) that mirror the DUT ports
- Transactors contain timing information and reads/writes the interface signals
- Style does not support emulation



Notes:

Example Signal Style Transactor and Interface



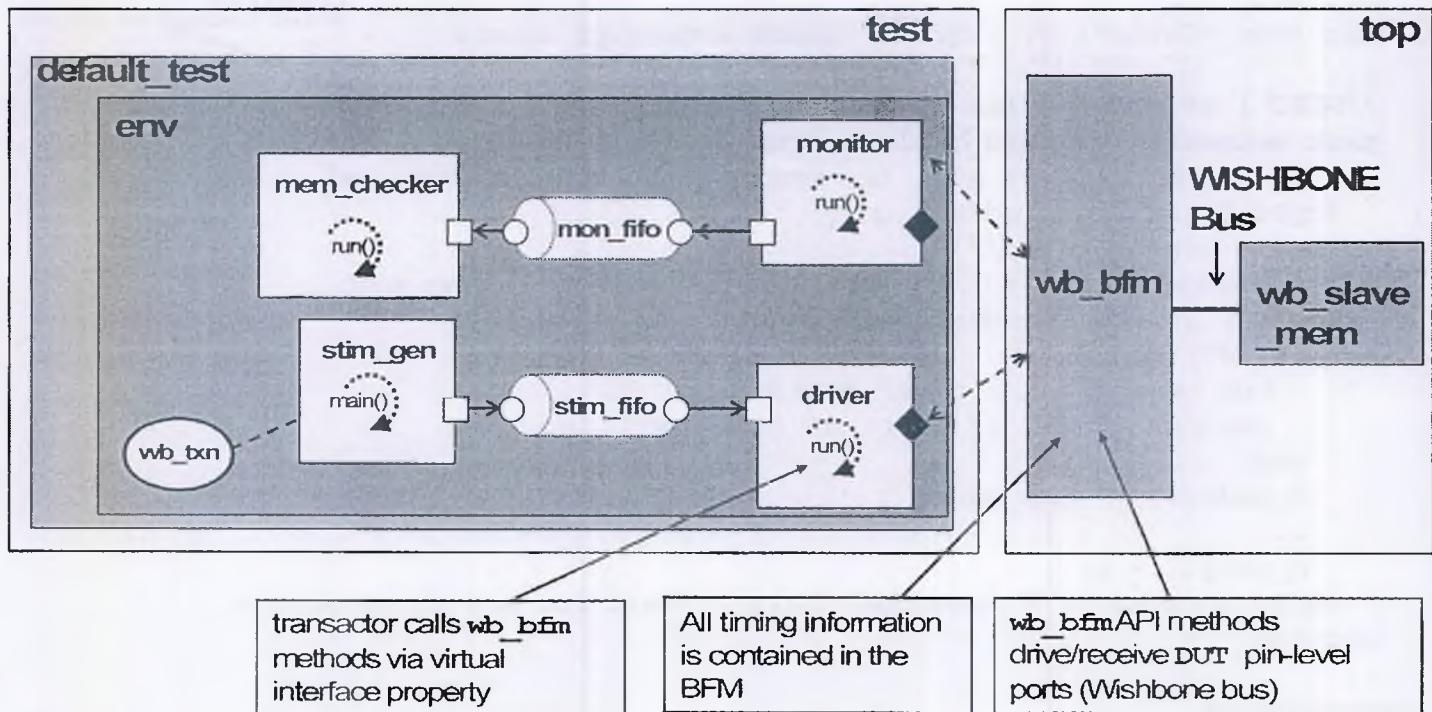
```
interface alu_if(
  input bit clk
);
  // alu ports
  byte unsigned val1, val2;
  bit valid_i;
  op_type_t mode;
  bit valid_o;
  shortint unsigned result;
  // not connected to the alu
  int txn_id;
endinterface
```



Notes:

BFM Style Transactor

- Interface contains tasks that generate DUT/bus transactions
- Transactors call BFM tasks to generate transactions
- Style supports emulation



Notes:

Example BFM Style Transactor

```
interface wishbone_bus_syscon_if #(int num_masters = 8, int num_slaves = 16,
                                         int data_width = 32, int addr_width = 32) ();
    // wishbone common signals
    bit clk;
    bit rst;
    bit [num_slaves-1:0] irq; // slave interrupt vector
    ...
    //READ 1 or more cycles
    task automatic wb_read_cycle(output logic[31:0] data[],
        input bit [31:0] adr, int unsigned count = 1, byte_sel = 4'b1111);
        logic [31:0] temp_addr = adr;
        data = new[count];
        @ (posedge clk) #1; // sync to clock edge + 1 time step
        for(int i = 0; i<count; i++) begin
            if(rst) begin
                bus_reset(); // clear everything
                return; //exit if reset is asserted
            end
            m_addr[1] = temp_addr;
            ...
            m_stb[1] = 0;
            while (m_ack[1]) @ (posedge clk); // Wait for ack to de-assert.
        endtask
    ...
endinterface
```

BFM task

Must be automatic



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_comp

140

Notes:

Example BFM Style Transactor

```
class driver extends uvm_component;
    uvm_blocking_get_port #(wb_txn) stim_p;
    virtual wishbone_bus_syscon_if v_wb_bfm;
    ...
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        stim_p = new("stim_p", this);
        if(!uvm_config_db #(virtual wishbone_bus_syscon_if)::get(
            this, "", "v_wb_bfm", v_wb_bfm))
            `uvm_error("CONFIGURATION", "Could not get virtual interface");
    endfunction

    task run_phase(uvm_phase phase);
        wb_txn txn;
        logic [31:0] read_data[];
        forever begin
            stim_p.get(txn);
            case (txn.txn_type)
                WRITE: v_wb_bfm.wb_write_cycle(txn.data, txn.adr, txn.count);
                READ : v_wb_bfm.wb_read_cycle(read_data, txn.adr, txn.count);
            endcase
        end
    endtask
endclass
```



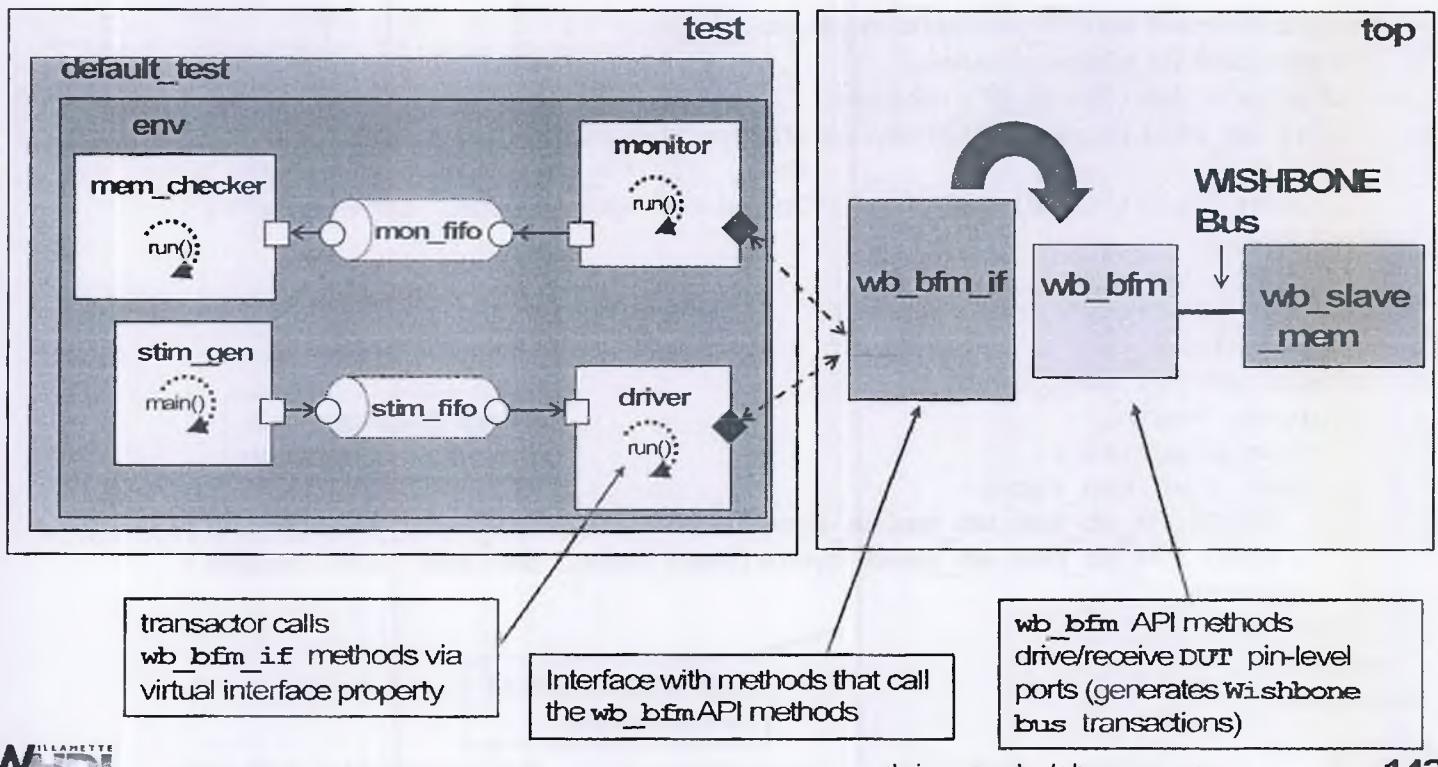
Note that arguments
are *not* a transaction

Call of task inside of
BFM interface

Notes:

BFM Style Transactor – BFM Module

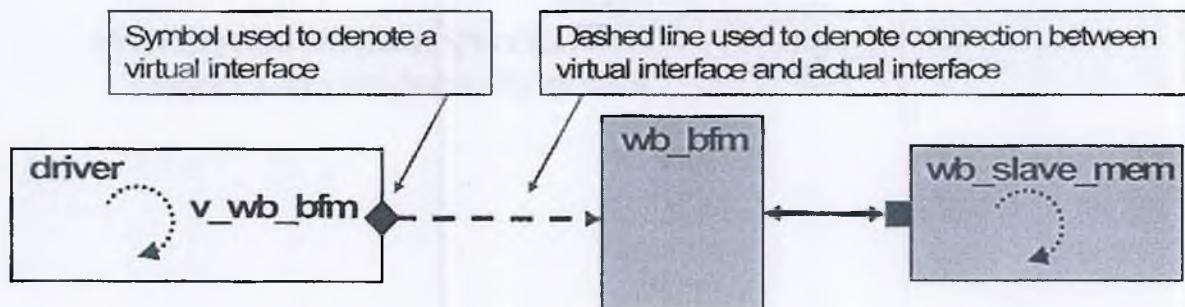
- Sometimes the BFM is a module and cannot be changed
 - A "proxy" interface is used which calls the BFM tasks on behalf of the transactor



Notes:

■ Virtual Interface

- Connection point for a class object to a SystemVerilog interface
- Declared as a class property
 - ◆ Think of it as a "pointer" to the actual interface
 - Set to point to the actual interface
 - ◆ Instead of referring directly (via a hierarchical pathname) to the actual signals or methods in the actual interface the class method manipulates the signals or calls the methods through the virtual interface property



Virtual interface declaration syntax: `virtual interface_type virtual_if_name;`

Notes:

Analysis

In this section



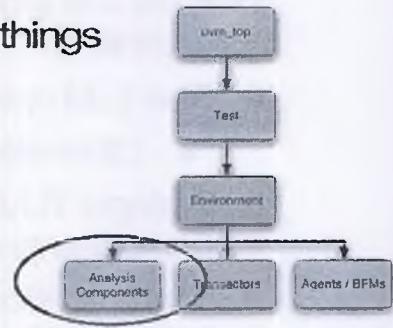
Analysis

- UMM Library analysis components
- Testbench analysis components

Notes:

Analysis Components

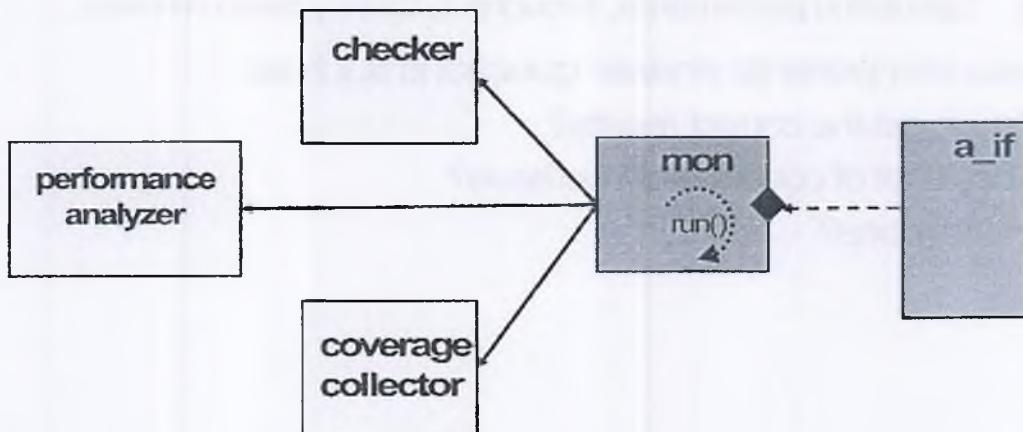
- Analysis-type components receive information about what is going on and make determinations and provide feedback
 - Called scoreboards because they keep track of how things are behaving i.e. the "score"
- Analysis components fall into two categories:
 - Checking
 - ◆ Evaluating functional correctness of DUT behavior
 - Metrics
 - ◆ Keeping track of what activity has occurred
 - ◆ Calculating performance, throughput, latency measurements
- Analysis components answer questions such as:
 - Did we get the correct results?
 - What level of coverage do we have?
 - Are we done?



Notes:

Analysis Components and TLM communication

- Monitors use either push-mode or fifo-mode to communicate with analysis components, but analysis components have three special requirements
 - The TLM operation must not block
 - ◆ Otherwise, the monitor might miss some bus activity
 - A single TLM operation must be able to work with multiple analysis components (i.e. broadcast)
 - The operation must complete without error even if no analysis components are connected



Notes:

UVM Analysis API

- Contains a single method `write()`
 - Broadcast method
 - May not block
 - Must always succeed

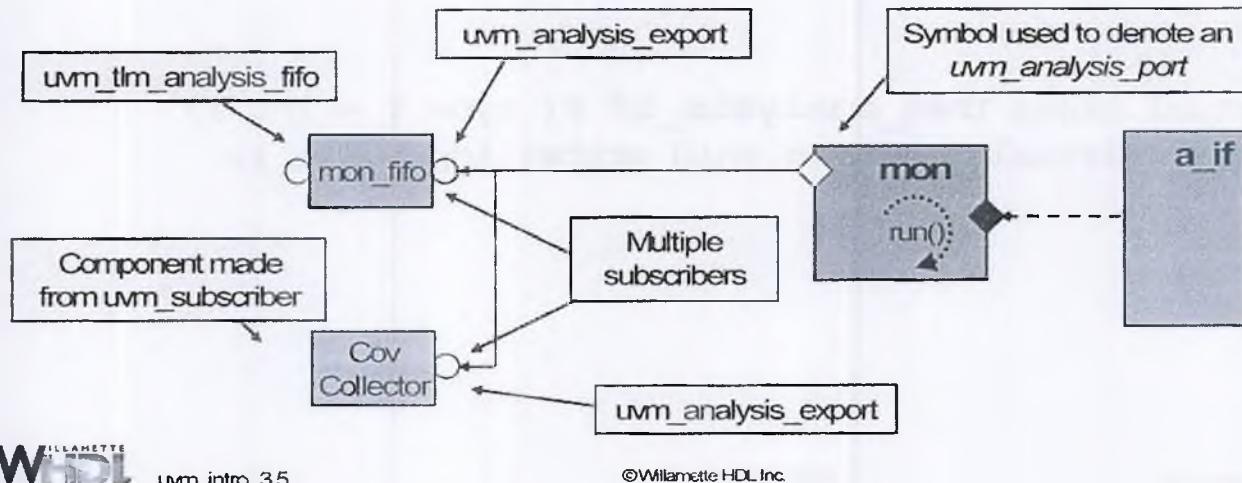
UVM

```
virtual class uvm_analysis_if #( type T = int );
    pure virtual function void write( input T t );
endclass
```

Notes:

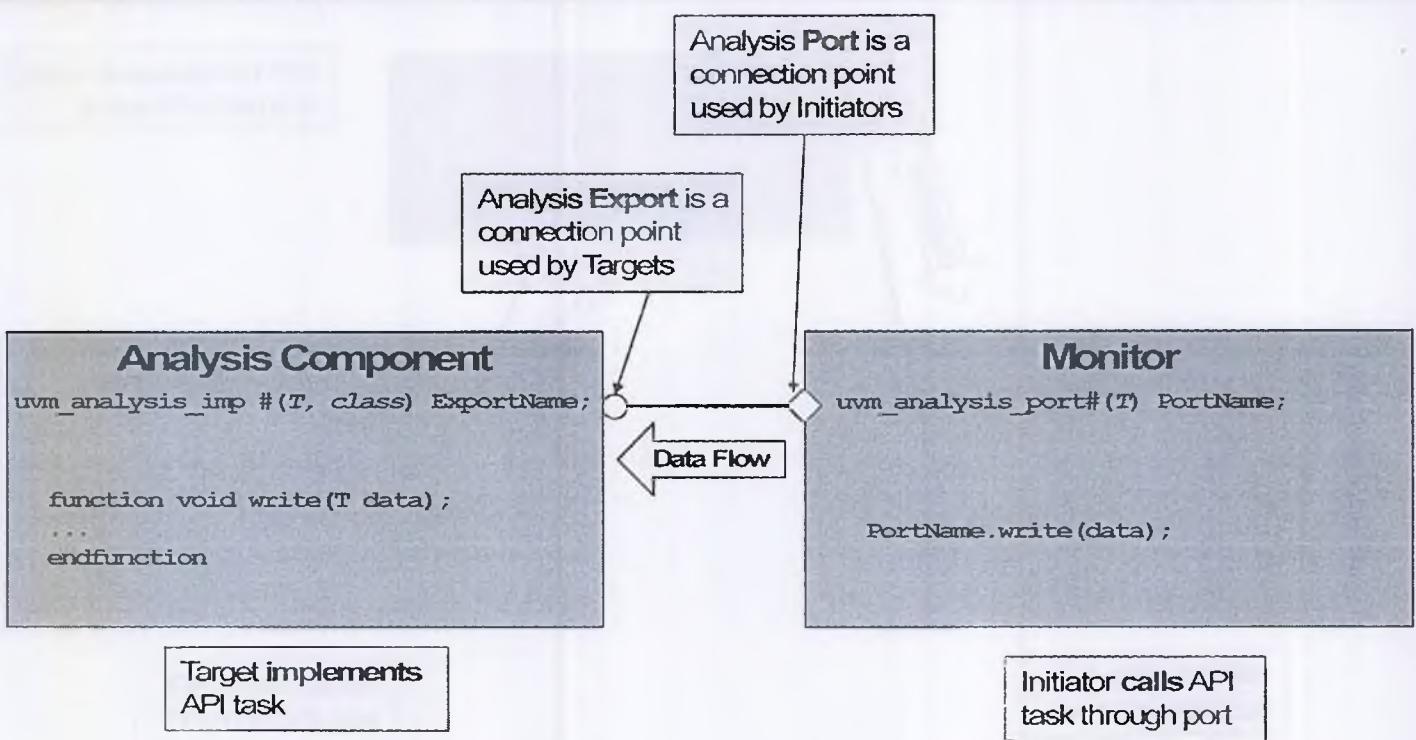
UVM Analysis TLM Elements

- `uvm_analysis_port`
 - Initiator uses port to push data using `write()`
- `uvm_analysis_export`
 - Exports an implementation of `write()`
- `uvm_tlm_analysis_fifo`
 - Unbounded FIFO acts as non-blocking target
- `uvm_subscriber`
 - Class with built-in `uvm_analysis_export`



Notes:

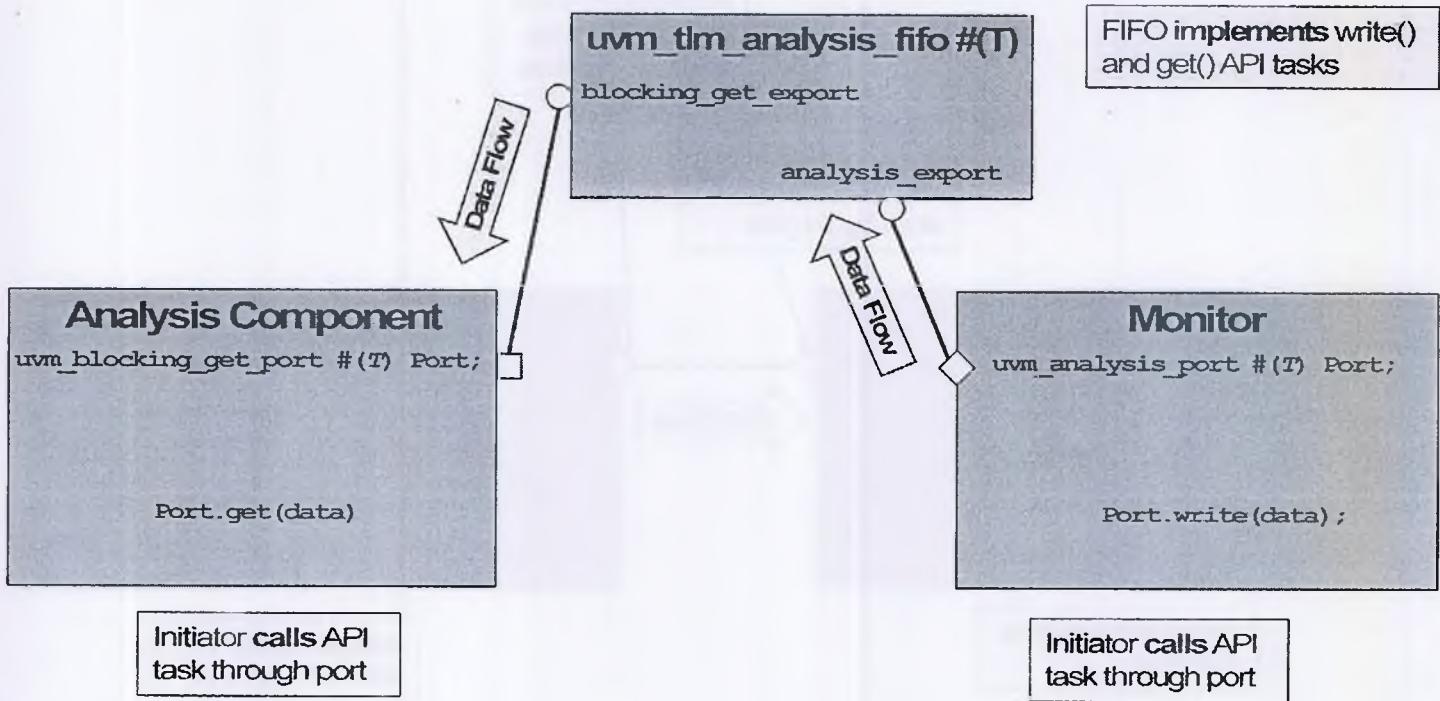
UVM Analysis API Classes (Push Mode)



```
Monitor.PortName.connect(Analysis_Component.ExportName);
```

Notes:

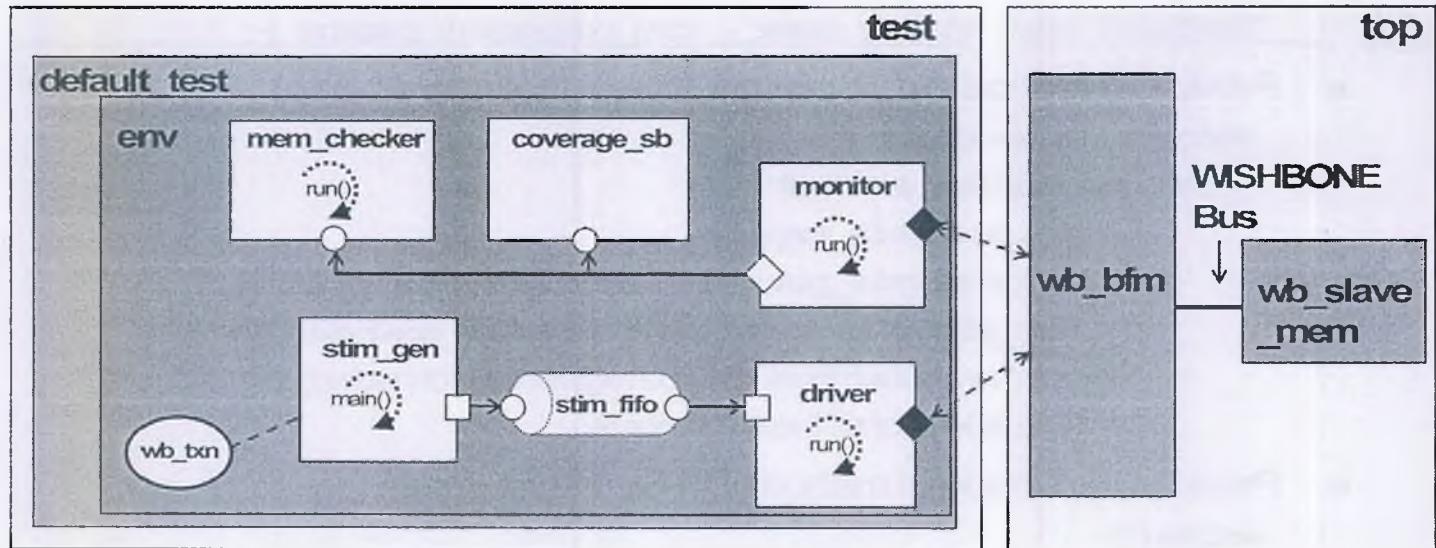
TLM Analysis API Classes (FIFO Model)



```
Monitor.Port.connect(FIFO.analysis_export);  
Analysis_Component.Port.connect(FIFO.blocking_get_export);
```

Notes:

Example Wishbone Memory Testbench with Analysis



Notes:

Port: uvm_analysis_port

- Port for broadcasting (publishing) information that was "snatched"

- Constructor:

```
function new( string name , uvm_component parent );
```

- Provides a method to register subscribers (Observer Pattern)

```
connect( subscriber )
```

- ◆ subscriber is one of:

- uvm_analysis_export

- uvm_analysis_port

- uvm_analysis_export of an uvm_tlm_analysis_fifo

- ◆ Zero or more subscribers may be registered (connected)

- A list is kept of all the subscribers

- Provides the broadcast method of the analysis_if

```
write(T)
```

- ◆ Broadcasts information of type T to every subscriber (Observer Pattern)

- Calls the write() method of every subscriber

Notes:

Example Analysis Port monitor

```
class monitor extends uvm_monitor;
`uvm_component_utils(monitor)

  uvm_analysis_port #(wb_txn) mon_ap;
  virtual wishbone_bus_syscon_if v_wb_bfm;

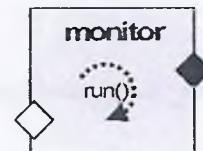
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_ap = new("mon_ap", this);
    if(!uvm_config_db #(virtual wishbone_bus_syscon_if)::get(
      this, "", "v_wb_bfm", v_wb_bfm))
      `uvm_error("CONFIGURATION","Could not get virtual interface");
  endfunction

  task run_phase(uvm_phase phase);
    wb_txn txn;
    forever begin
      txn = wb_txn::type_id::create("txn");
      // get a txn
      v_wb_bfm.monitor_txn(txn.data, txn.adr, txn.txn_type);
      mon_ap.write(txn); // broadcast txn
    end
  endtask

endclass
```

Declaration



use of write() method
to broadcast result_txn



uvm_intro_3.5

© Willamette HDL Inc

code in examples/wb_mem_analysis

153

Notes:

uvm_tlm_analysis_fifo

- "Catcher" of data broadcast from an `uvm_analysis_port`
- An `uvm_tlm_analysis_fifo` is a `uvm_tlm_fifo` that also implements and exports the analysis interface
 - FIFO has a unbounded depth so `write()` always succeeds (FIFO never full)



```
Class uvm_tlm_analysis_fifo #( type T = int ) extends uvm_tlm_fifo #( T );
```

analysis_export
implements the analysis_if

```
    uvm_analysis_imp #(T, uvm_tlm_analysis_fifo #(T)) analysis_export;
```

```
    function new(string name , uvm_component parent = null);
```

```
        super.new(name, parent, 0); // analysis fifo must be unbounded
```

```
        analysis_export = new("analysis_export", this);
```

Underlying `uvm_tlm_fifo`
is unbounded

```
    endfunction
```

```
    function void write( input T t );
```

```
        void'(this.try_put( t )); // unbounded => must succeed
```

```
    endfunction
```

```
endclass
```

write() function uses
`try_put()`



uvm_intro_3.5

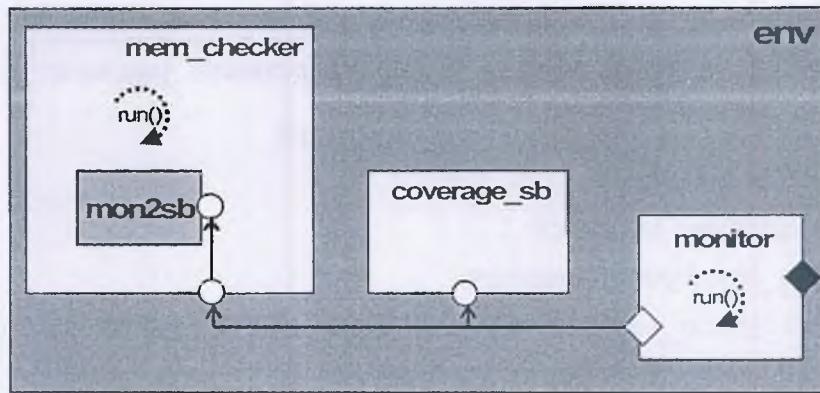
© Willamette HDL Inc.

154

Notes:

Analysis FIFO Inside of Analysis Component

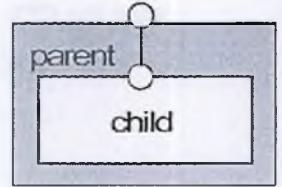
- An architecture choice often made is to encapsulate the analysis fifo inside of a component like a scoreboard



- Results in hierarchical *export to export* TLM connections

`this.parentExport.connect (child.childExport)`

- Connection is done in the context of the parent's `connect_phase()`



Notes:

Export: uvm_analysis_export

- Enables the "re-broadcast" or the "passing along" of the information from the analysis port
- Constructor:

```
function new( string name , uvm_component parent );
```
- Provides a method to register subscribers

```
connect( subscriber )
```

 - ◆ subscriber is one of:
 - uvm_analysis_export
 - analysis_export of a uvm_tlm_analysis_fifo
 - ◆ One or more subscribers may be registered (connected)
 - A list is kept of all the subscribers
- Provides the broadcast method of the analysis_if

```
write(T)
```

 - ◆ Broadcasts information of type T to every subscriber (Observer Pattern)
 - Calls the write() method of every subscriber

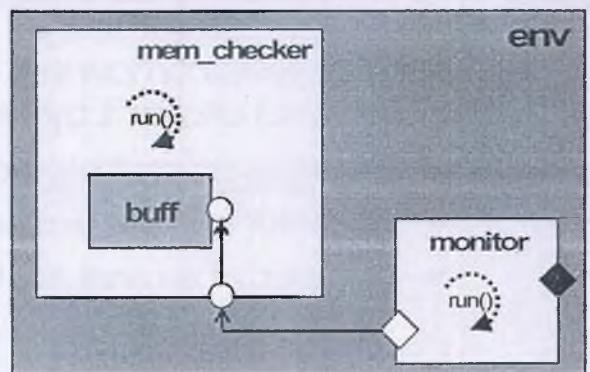
Notes:

Example Analysis FIFO Inside a Scoreboard

```
class mem_checker extends uvm_scoreboard;
  uvm_analysis_export #(wb_txn) mon_exp;
  uvm_tlm_analysis_fifo #(wb_txn) buff;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_exp = new("mon_exp", this);
    buff = new("buff", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    mon_exp.connect(buff.analysis_export); //connect port & buffer
  endfunction

  task run_phase(uvm_phase phase);
    wb_txn txn, expected_txn;
    bit[31:0] data;
    forever begin
      buff.get(txn);
      ...
    endtask
  ...
endclass
```



Notes:

uvm_subscriber

- If an analysis component only has one input an alternative architecture choice is to create it by inheriting from uvm_subscriber
 - Provides an analysis export
 - Behavior is in the write() method instead of the run_phase()
 - No need of an analysis fifo for buffering



```
virtual class uvm_subscriber #(type T=int) extends uvm_component;
    typedef uvm_subscriber #(T) this_type;

    uvm_analysis_imp #(T, this_type) analysis_export; ← Analysis Export

    function new (string name, uvm_component parent);
        super.new(name, parent);
        analysis_export = new("analysis_imp", this);
    endfunction

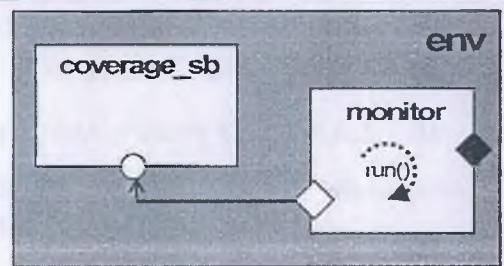
    pure virtual function void write(T t); ← "pure" virtual method that must
                                                be overridden in derived class
                                                with "write" method behavior
    endclass
```

Notes:

Example Scoreboard Using uvm_subscriber

```
class coverage_sb extends uvm_subscriber #(wb_txn);
  `uvm_component_utils(coverage_sb)
  wb_txn txn;
  covergroup cg;
    address_cp: coverpoint txn.adr iff
      (txn.txn_type == READ) {
      bins lowRange = { [0 : 32'h00001000] };
      bins hiRange = { [32'h00030000 : 32'h0003ffff] };
    }
  endgroup
  function new(string name, uvm_component parent);
    super.new(name, parent);
    cg = new();
  endfunction
  function void write(wb_txn t);
    txn = t;
    cg.sample();
    `uvm_info("COVERAGE",
      $sformatf("Current coverage = %f", cg.get_coverage), UVM_HIGH)
  endfunction
  function void report_phase(uvm_phase phase);
    `uvm_info("COVERAGE", "*****", UVM_MEDIUM);
    `uvm_info("COVERAGE", $sformatf("Final coverage = %f", cg.get_coverage),
      UVM_MEDIUM)
    `uvm_info("COVERAGE", "*****\n", UVM_MEDIUM);
  endfunction

```



Notes:

Developing "Comparator" Scoreboards

- Typical behavior includes comparison of actual vs. expected, keeping track of errors, successes etc.
- Develop your own from scratch
 - Inherit from `uvm_component`, `uvm_scoreboard` or `uvm_subscriber`
- There are several classes provided that may be used to help create scoreboards
 - `uvm_in_order_class_comparator`
 - ◆ Compares objects with assumption that the expected and actual results arrive in order
 - `uvm_algorithmic_comparator`
 - ◆ Compares streams of data *before* and *after* a transformation has been applied to the data
 - Not a UVM library component
 - ◆ Contributions in `uvm_world.org` or from other sources
 - `vhdl_coo_comparator`
 - Compares objects with no assumption as to the order of arrival of the expected and actual objects

Notes:

Example Scoreboard-1:mem_checker -1

```
class mem_checker extends uvm_scoreboard;
  `uvm_component_utils(mem_checker)

  uvm_analysis_export #(wb_txn) mon_exp;
  uvm_tlm_analysis_fifo #(wb_txn) buff;

  bit[31:0] shadow_mem[bit[31:0]];
  int txn_count;
  int error_count;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_exp = new("mon_exp", this);
    buff    = new("buff", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    mon_exp.connect(buff.analysis_export); //connect port & buffer
  endfunction
```

For expected results the scoreboard has a memory that "shadows" the DUT memory



uvm_intro_3.5

© Wilamette HDL Inc.

code in examples/wb_mem_analysis

161

Notes:

Example Scoreboard-1:mem_checker -2

```
task run_phase(uvm_phase phase);
    wb_txn txn, expected_txn;
    bit[31:0] data;
    forever begin
        buff.get(txn);
        txn_count++;
        case (txn.txn_type)
            WRITE: shadow_mem[txn.adr] = txn.data[0];
            READ: begin
                expected_txn = new("expected_txn");
                expected_txn.copy(txn);
                expected_txn.data[0] = shadow_mem[txn.adr];
                if (expected_txn.compare(txn)) begin
                    `uvm_info("MEM_CHECK", $sformatf(
                        "Memory passed at address %x", txn.adr), UVM_MEDIUM);
                end else begin
                    `uvm_error("MEM_CHECK", $sformatf(
                        "Memory failed at address %x:\nExpected:\n%s\nActual:\n%s",
                        txn.adr, expected_txn.sprint(), txn.sprint()));
                    error_count++;
                end
            end
        endcase
    end // forever
endtask
```

Comparison behavior
done in the run_phase



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/wb_mem_analysis

162

Notes:

Example Scoreboard-1:mem_checker -3

For end of simulation report

```
function void report_phase(uvm_phase phase);
    `uvm_info("MEM_CHECK", "\n", UVM_MEDIUM);
    `uvm_info("MEM_CHECK", "*****\n", UVM_MEDIUM);
    `uvm_info("MEM_CHECK", $sformatf(" total transactions: %0d", txn_count),
              UVM_MEDIUM);
    `uvm_info("MEM_CHECK", $sformatf(" total errors:      %0d", error_count),
              UVM_MEDIUM);
    `uvm_info("MEM_CHECK", "*****\n", UVM_MEDIUM);
endfunction

endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/wb_mem_analysis

163

Notes:

Example Scoreboard: SB

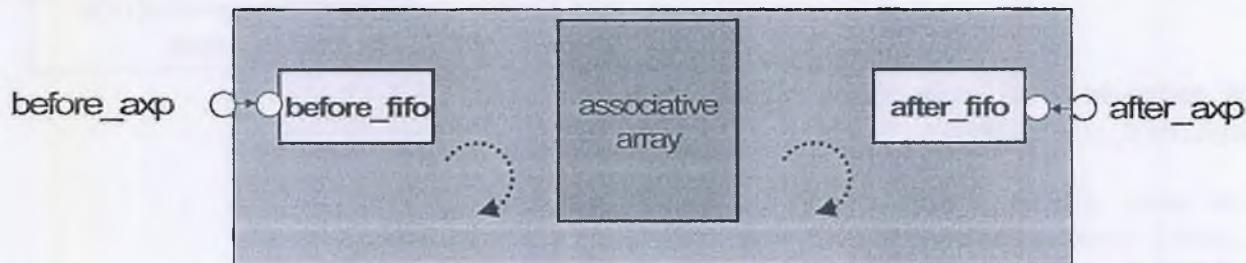
```
class SB extends uvm_in_order_class_comparator #(alu_txn);  
`uvm_component_utils(SB)  
function new( string name,  
            uvm_component parent) ;  
super.new( name , parent );  
endfunction  
  
endclass
```



Example uses an UVM library comparator class

Notes:

whdl_ooo_comparator



```
class whdl_ooo_comparator
  #(type T = int,
    type IDX = int,
    type index_id_type = whdl_class_index_id #(T, IDX) )
  extends uvm_component;

  uvm_analysis_export #(T) before_axp, after_axp;
  protected uvm_tlm_analysis_fifo #(T) before_fifo, after_fifo;

  // API methods
  virtual function int get_matches();
  virtual function int get_mismatches();
  virtual function int get_total_missing();

  ...
endclass
```

Doesn't matter what order things arrive in (on either side)
Despite names it does not matter if objects arrive on before or after arrive first

Requires an `index_id()` method in the transaction object.
Returns an id that is unique to the transaction object which is used to access the associative array

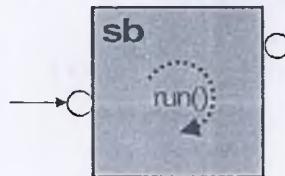
Notes:

Example Scoreboard – 3: SB

Example uses whdl_ooo_comparator class

```
class SB extends whdl_ooo_comparator #(alu_txn,int);  
`uvm_component_utils(SB)  
  
function new( string name , uvm_component parent) ;  
super.new( name , parent );  
endfunction  
  
function void report_phase(uvm_phase phase);  
  uvm_report_info("SB", $sformatf("Matches: %0d", get_matches()));  
  uvm_report_info("SB", $sformatf("Mismatches: %0d", get_mismatches()));  
  uvm_report_info("SB", $sformatf("Missing: %0d \n", get_missing()));  
endfunction  
  
endclass  
  
-----  
class alu_txn extends uvm_sequence_item;  
  
virtual function int index_id();  
  return txn_id;  
endfunction : index_id  
  
// rest not show  
endclass // alu_txn
```

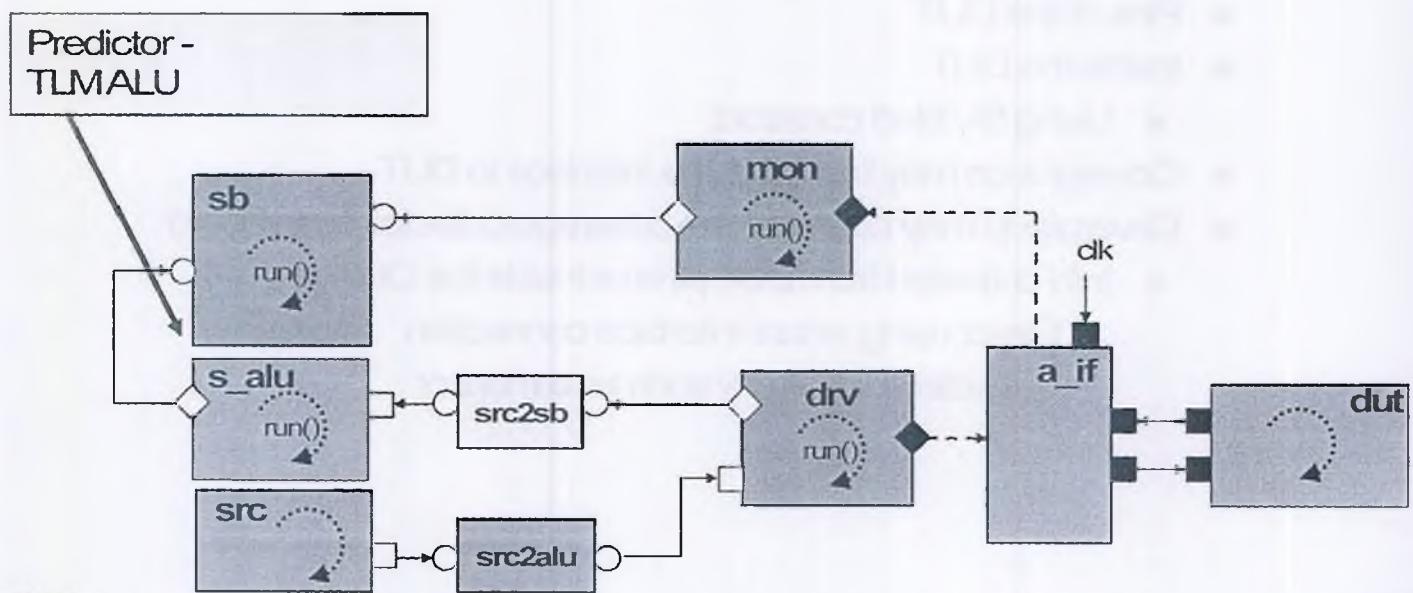
This method added to the alu_txn class so can get the value to index associative array



Notes:

Predictor

- Analysis component which generates expected results for use by other analysis components such as scoreboards
 - Takes the same stimulus as the DUT
 - May be non-SV language (SystemC, C, C++)
- Typically an abstract (TLM) model
 - Minimizes overhead



Notes:

Coverage Collectors

- Analysis component (scoreboard) which gathers metrics
- Typically used to determine completeness of testing
 - Are we done yet?
 - Uses covergroups
- Sources of coverage information
 - Pins of the DUT
 - Inside the DUT
 - ◆ Using SV bind construct
 - Covergroups may be inside the interface to DUT
 - Covergroup may be inside the coverage collector scoreboard
 - ◆ Info collected from DUT pins or inside the DUT
 - Direct using virtual interface connection
 - Collected by a proxy such as a monitor

Notes:

Example Coverage Object: mem_coverage

```
class mem_coverage extends uvm_object;
  `uvm_object_utils(mem_coverage)
  wb_txn txn;

  covergroup cg;
    address_cp: coverpoint txn.adr iff (txn.txn_type == READ) {
      bins lowRange = { [0 : 32'h00001000];
      bins hiRange  = { [32'h00030000 : 32'h0003ffff] };
    }
  endgroup

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

  virtual function void sample(wb_txn t);
    txn = t;
    cg.sample();
  endfunction

  virtual function real get_coverage();
    return cg.get_coverage();
  endfunction
endclass
```

Covergroup

mem_coverage

Sample covergroup

Access Methods

Notes:

Example Coverage Collector: coverage_sb

```
class coverage_sb extends uvm_subscriber #(wb_txn);
  `uvm_component_utils(coverage_sb)

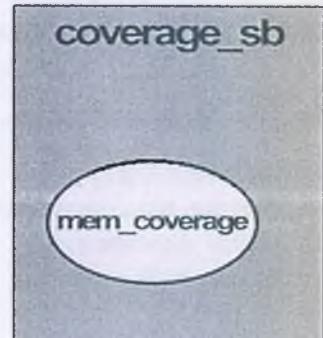
  mem_coverage m_cov; ← Coverage object with
                        coveragegroup inside

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_cov = mem_coverage::type_id::create("r_cov");
  endfunction

  function void write(wb_txn t); ← Sample covergroup
    m_cov.sample(t);
    `uvm_info("COVERAGE",
      $sformatf("Current coverage = %f", m_cov.get_coverage()), UVM_HIGH)
  endfunction

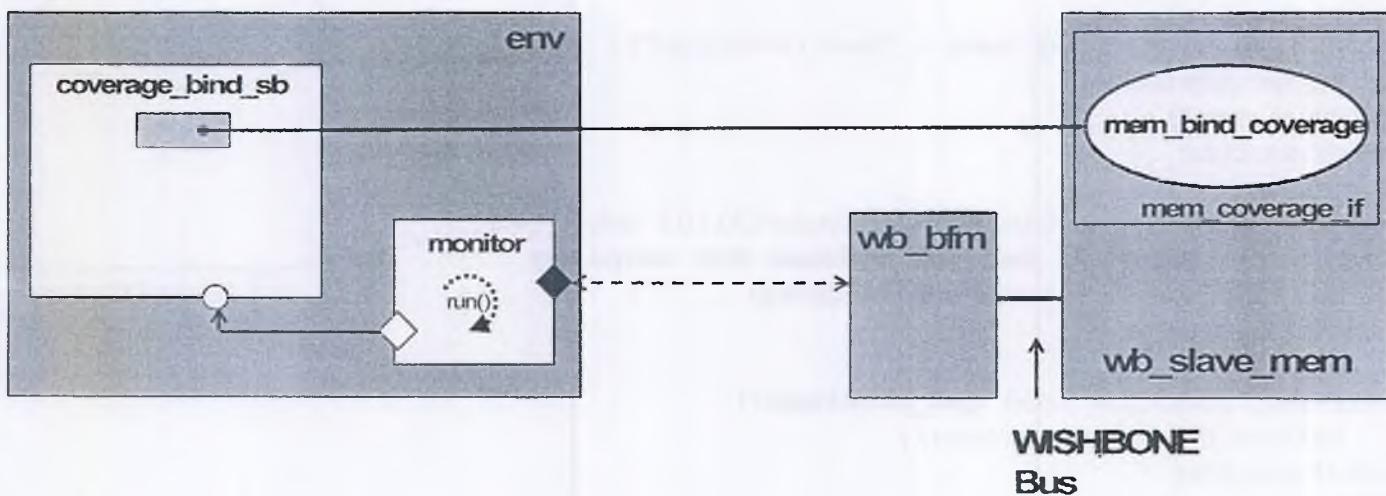
  function void report_phase(uvm_phase phase);
    `uvm_info("COVERAGE", "*****", UVM_MEDIUM);
    `uvm_info("COVERAGE", $sformatf("Final coverage = %f", m_cov.get_coverage()), UVM_MEDIUM)
    `uvm_info("COVERAGE", "*****\n", UVM_MEDIUM);
  endfunction
```



Notes:

Coverage Scoreboard with Covergroup inside DUT

- The coverage scoreboard (`coverage_bind_sb`) in this example gets its coverage information from inside the DUT (`wb_slave_mem`)
 - A covergroup is inside of a coverage object (`mem_bind_coverage`) which is inside a bind instance inside the DUT
 - The coverage scoreboard has a handle to the coverage object that is set through the configuration database



Notes:

Example Coverage Object mem_bind_coverage

```
class mem_bind_coverage extends uvm_object;
  `uvm_object_utils(mem_bind_coverage)

  logic [31:0] m_addr;

  covergroup cg;
    address_cp: coverpoint m_addr {
      bins lowRange = { [0 : 32'h00001000] };
      bins hiRange = { [32'h00030000 : 32'h0003ffff] };
    }
  endgroup

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

  virtual function void sample(logic[31:0] adr);
    m_addr = adr; // set the address for sampling
    cg.sample(); // sample covergroup
  endfunction

  virtual function real get_coverage();
    return cg.get_coverage();
  endfunction
endclass
```

Covergroup has bins for low and high address coverage

mem_bind_coverage

Access Methods

Notes:

Example Container: mem_coverage_if

```
interface mem_coverage_if(input mem_re, clk,
                           input logic[31:0] adr);
    import uvm_pkg::*;
    import analysis_pkg::mem_bind_coverage;

    mem_bind_coverage mem_bind_cov; // coverage object

    always @ (posedge clk)
        if (mem_re)
            mem_bind_cov.sample(adr);

    initial begin
        mem_bind_cov = new(); // create cov object
        // put handle to cov object in config db
        uvm_config_db #(mem_bind_coverage)::set(
            uvm_top, "*", "mem_cov_bind", mem_bind_cov);
    end

endinterface
```

Coverage object



Put handle to
coverage object
into config_db

Notes:

Example Bind top

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

wb_slave_mem # (MEM_SLAVE_0_WD_SIZE, MEM_SLAVE_0_WB_ID, SLAVE_ADDR_SPACE_SZ)
wb_s_0 (
  // inputs
  .clk ( clk ),
  .rst ( rst ),
  ...
);

bind wb_s_0 mem_coverage_if cov_bind(.mem_re, .clk, .adr);
...
endmodule
```

DUT instance

Bind instance

Notes:

Example Coverage Collector: coverage_bind_sb-1

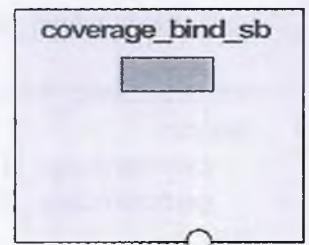
```
class coverage_bind_sb extends uvm_subscriber #(wb_txn);
`uvm_component_utils(coverage_bind_sb)

mem_bind_coverage m_bind_cov; ← Coverage object handle
int txn_cnt;
real current_bind_coverage;
int percentage_100_cnt;
bit percentage_100_met;

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

function void build_phase(uvm_phase phase);
super.build_phase(phase);
// get a handle to the coverage object in the bind instance
if(!uvm_config_db #(mem_bind_coverage)::get(
    this, "", "mem_cov_bind", m_bind_cov))
`uvm_error("CONFIG", "mem_cov_bind not found");

endfunction
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem_bind

175

Notes:

Example Coverage Collector: coverage_bind_sb - 2

```
function void write(wb_txn t);
    txn_cnt++;
    current_bind_coverage = m_bind_cov.cg.get_coverage();

    if(current_bind_coverage == 100 && !percentage_100_met)
    begin
        percentage_100_cnt = txn_cnt;
        percentage_100_met = 1;
    end
    `uvm_info("COVERAGE",
        $sformatf("Current coverage = %f ", current_bind_coverage ), UVM_HIGH)
endfunction

function void report_phase(uvm_phase phase);
    `uvm_info("COVERAGE", "*****", UVM_MEDIUM);
    `uvm_info("COVERAGE", $sformatf(" Final Coverage = %f%%", current_bind_coverage), UVM_MEDIUM );
    if(percentage_100_met)
        `uvm_info("COVERAGE", $sformatf(" 100% Coverage met with %0d transactions", percentage_100_cnt), UVM_MEDIUM);
    `uvm_info("COVERAGE", "*****\n", UVM_MEDIUM);

endfunction

endclass
```

Get coverage using handle to coverage object

coverage_bind_sb



uvm_intro_3.5

© Willamette HDL Inc

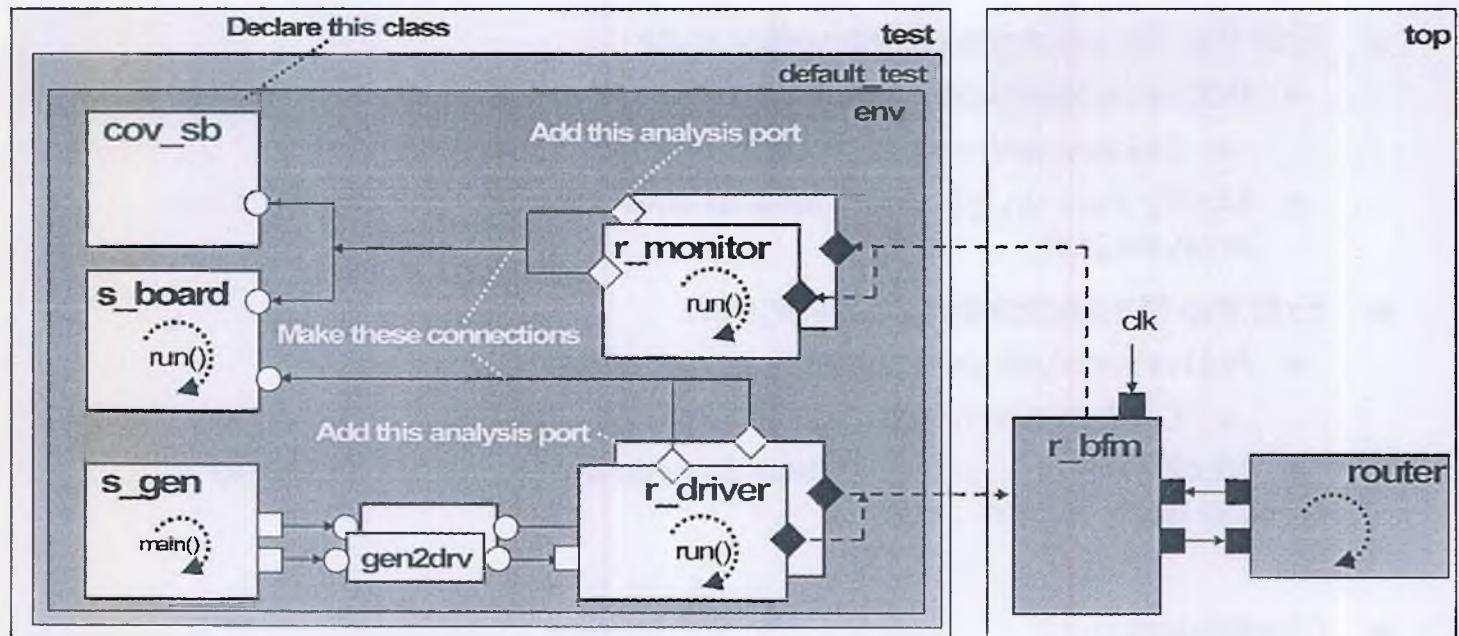
code in examples/wb_mem_bind

176

Notes:

Lab - Analysis: Overview

- Add Analysis ports to the monitors
- Connect up the analysis ports on monitors and drivers



Notes:

Lab - Analysis: Instructions - 1

- Working directory: **analysis**
- Edit the file **analysis/coverage_sb.svh**
 - Declare the `coverage_sb` class and inherit from `uvm_subscriber` of type `Packet`
- Edit the file **xactors/rtr_monitor.svh**:
 - Add an analysis port called `out_ap` of type `Packet`
 - ◆ Declare and create
 - Modify the `run_phase()` task to write the `mon_txn` object to the analysis port
- Edit the file **xactors/rtr_driver.svh**:
 - Add an analysis port called `drv_ap` of type `Packet`
 - ◆ Declare and create
 - Modify the `run_phase()` task to write the `stim_txn` object to the analysis port
- Continued...

Notes:

Lab - Analysis: Instructions - 2

■ Edit the file `env/router_env.svh`:

- In `connect_phase()`, Register (i.e. connect) all the subscribers to the analysis ports
 - ◆ Analysis export (`analysis_export`) of the coverage scoreboard (`cov_sb`) to the analysis port (`out_ap`) of all the monitors (`r_monitor[i]`)
 - ◆ Analysis export (`after_axp`) of the `comp_sb` class (`s_board`) to the analysis port (`out_ap`) of all the monitors (`r_monitor[i]`)
 - ◆ Analysis export (`before_axp`) of the `comp_sb` class (`s_board`) to the analysis port (`drv_ap`) of all of the drivers (`r_driver[i]`)

■ Compile & run

■ Optional

- Create your own scoreboard by inheriting from the `whdl_ooo_comparator` class
- Name of the class: `comp_sb`
- File to edit: `analysis/comp_sb.svh`
 - ◆ Remove existing and write your own

Notes:

Lab - Analysis: Sample Output

```
# UVM_INFO @ 0: reporter [RNTST] Running test default_test...
# Router size = 8x8
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] -----
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage = 100.000000%
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] 100% Coverage met with 266
transactions
# UVM_INFO @ 1972300: uvm_test_top.env.cov_sb [COVERAGE] -----
# 
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Matches: 991
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
# UVM_INFO @ 1972300: uvm_test_top.env.s_board [Packet_Comparator] Missing: 1
#
#
# -- UVM Report Summary --
#
# ** Report counts by severity
# UVM_INFO: 8
# UVM_WARNING: 0
# UVM_ERROR: 0
# UVM_FATAL: 0
# ** Report counts by id
# [COVERAGE] 4
# [Packet_Comparator] 3
# [RNTST] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 1972300 ns Iteration: 100 Instance: /test
```

Sol



uvm_intro_3.5

© Willamette HDL Inc.

180

Notes:

UVM Hierarchical Testbenches

In this section



Components & Hierarchy
Hierarchical connectivity
Agents

Notes:

Testbenches & Hierarchy

- Almost all the examples we have shown so far do not have hierarchy
- Complex testbenches may require or are best constructed hierarchically
 - Encapsulation of a subset or portion of the testbench
 - ◆ Agents
 - For reuse purposes
 - Polymorphic substitution for different configurations
 - Flat structure becomes unwieldy with too many components
 - Natural subsets of testbench structure
- UVM facilitates hierarchical testbench structure by using components

Notes:

UVM_Component Class

- An uvm_component derived object
 - Manages the hierarchy of the testbench
 - ◆ Has a hierarchical name
 - ◆ Has a parent
 - ◆ Has a list of children
 - Could be an empty list
 - ◆ Enables traversing hierarchy for starting process execution, reporting etc.
 - Has virtual methods for overriding in derived classes
 - ◆ Connectivity, configuring, and reporting
- Since all objects in the test environment are components
 - Provides for uniform way of handling hierarchy in test bench components

Note: Use of child/parent here implies instantiation relationship not inheritance



Notes:

UVM Component Constructor

UVM

```
function new(string name , uvm_component p = null );
```

- **string name**
 - ◆ The local instance name of the component
 - Used in its hierarchical path name
 - » Assembled automatically
 - Must be unique at the level it exists
- **uvm_component p = null**
 - ◆ The component's parent
 - If a component is instantiated inside another component it has a parent
 - » Set argument value to **this** to obtain the reference to the parent
 - ◆ If the parent handle is null, the component is added as a child to the singleton global component **uvm_top**.

WILLAMETTE
HDL

uvm_intro_3.5

©Willamette HDL Inc.

184

Notes:

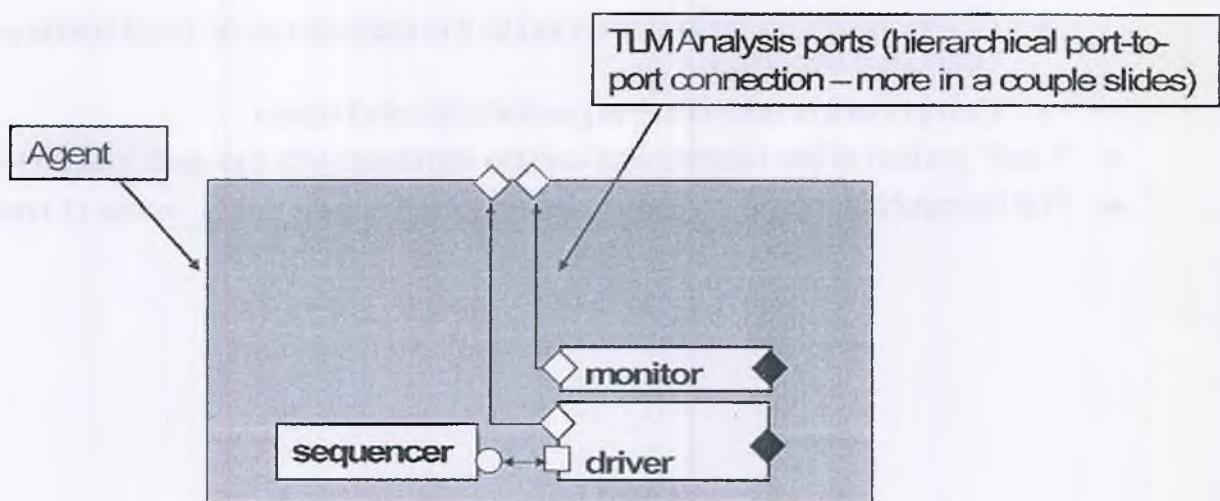
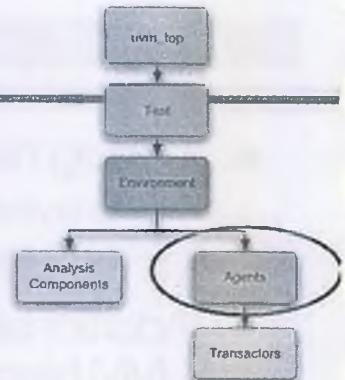
UVM Component Hierarchical Name

- Verilog hierarchical name
 - Everything has a Verilog hierarchical name
 - ◆ May be retrieved with `$display ("%m")` ;
- Additionally each component in an UVM testbench has a unique UVM hierarchical name
 - Different from the Verilog hierarchical name
 - When a component is constructed, you pass in a name and a handle to the parent object
 - ◆ Component's name is appended to the parents hierarchical name with a ":" used as a separator
 - ◆ Component is added to the parent's list of children
 - "Leaf" portion of the hierarchical can be retrieved with the `get_name()` method
 - Full hierarchical name can be retrieved with the `get_full_name()` method

Notes:

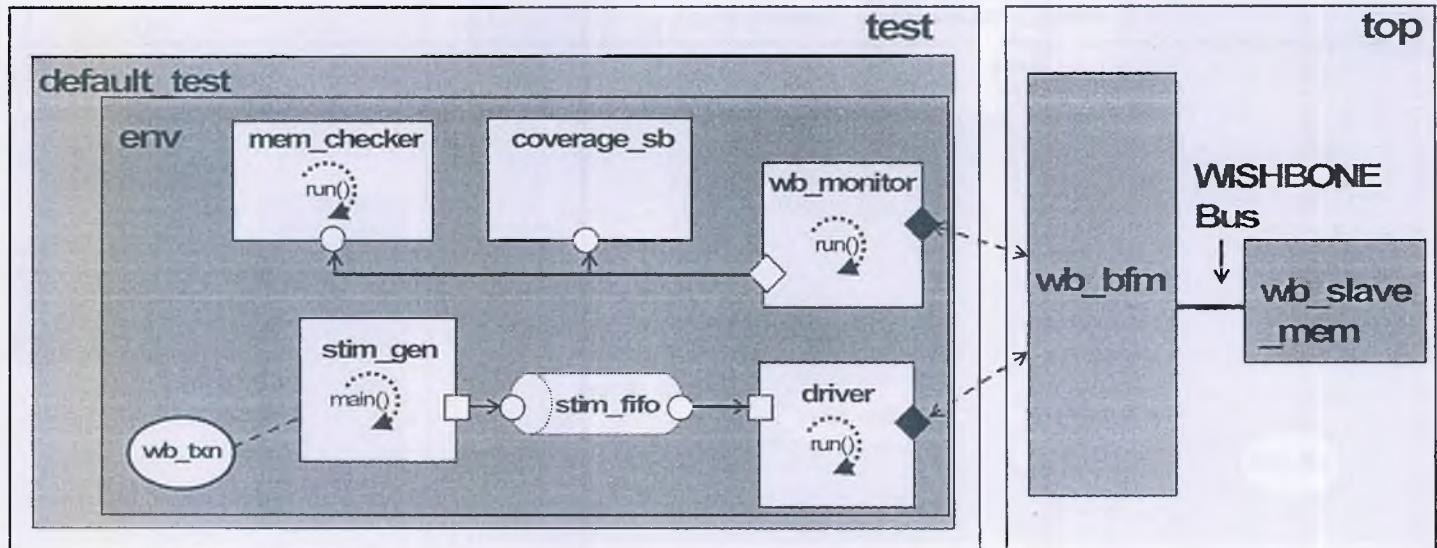
Agent

- Term used for a specific type of hierarchical container class
 - Encapsulation of elements related to an interface of the DUT
 - ◆ Drivers, monitors, sequencers etc.
- For each interface in the DUT, there should be a corresponding agent in the testbench environment



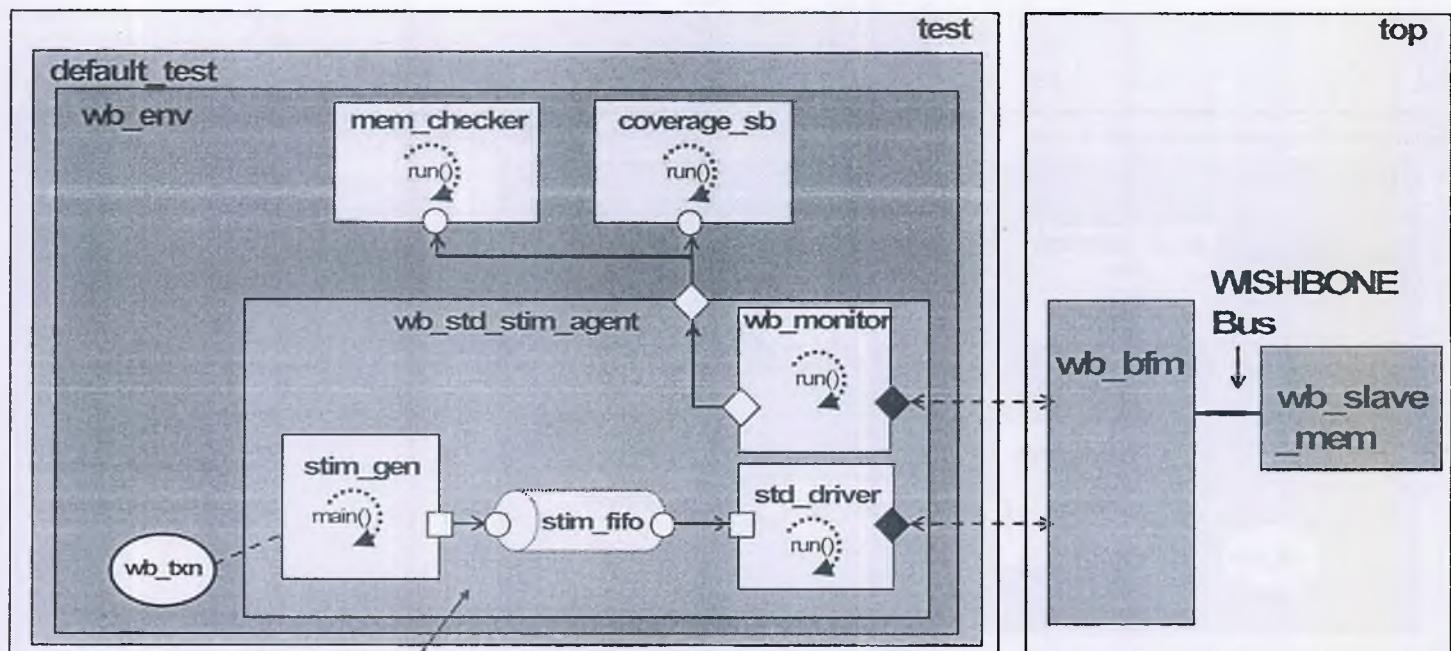
Notes:

Wishbone Memory Testbench *Without* an Agent



Notes:

Wishbone Memory Testbench With an Agent

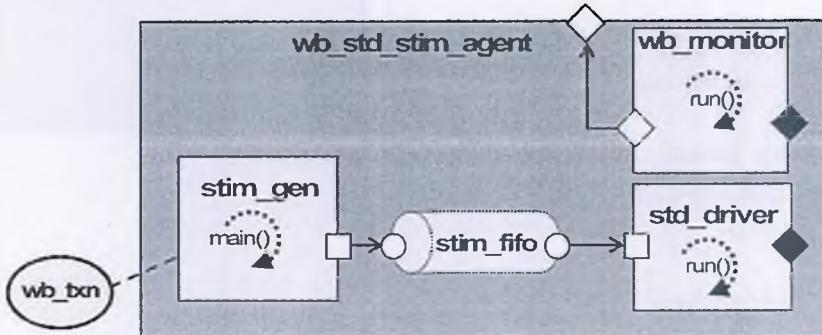


Later we will substitute this WB agent (`wb_std_stim_agent`) with a different WB agent (`wb_agent`) that uses sequences to generate stimulus to the Wishbone bus. Both of these agents are derived from a base class agent (`wb_agent_base`) to enable polymorphic substitution.

Notes:

Hierarchical Port to Port Connections

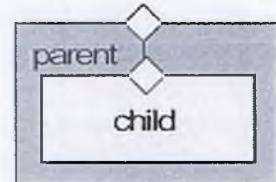
- An agent typically has a monitor with an analysis port inside of it



- Results in hierarchical *port to port* TLM Connection

`child.childPort.connect(this.parentPort)`

- Connection is done in the context of the parent's `connect_phase()`



- NOTE: A general rule for all TLM connections is:

"Whatever is closest to the provider goes inside the `connect()` parenthesis"

Notes:

Example Port to Port Connections: wb_agent_base

```
class wb_agent_base extends uvm_agent;
`uvm_component_utils(wb_agent_base)

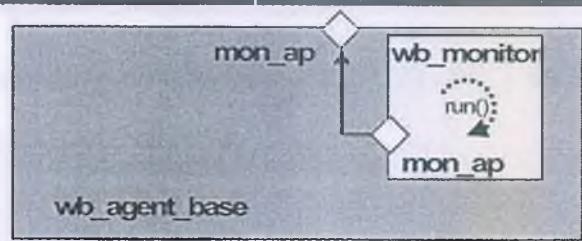
  uvm_analysis_port #(wb_txn) mon_ap;
  wb_monitor mon;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_ap = new("mon_ap", this);
    mon = wb_monitor::type_id::create("mon", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    mon.mon_ap.connect(this.mon_ap);
  endfunction

endclass
```



Down: Connect agent's analysis port to the monitor's analysis port (make available down)

Notes:

Example Connections: mem_checker

```
class mem_checker extends uvm_scoreboard;
`uvm_component_utils(mem_checker)

  uvm_analysis_export #(wb_txn) mon_exp;
  uvm_tlm_analysis_fifo #(wb_txn) buff;

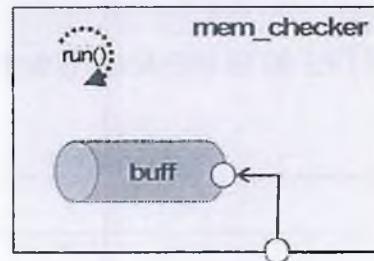
  bit[31:0] shadow_mem[bit[31:0]];
  int txn_count;
  int error_count;

  // constructor not shown

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_exp = new("mon_exp", this);
    buff    = new("buff", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    mon_exp.connect(buff.analysis_export); //connect export & buffer
  endfunction

  ...
endclass
```

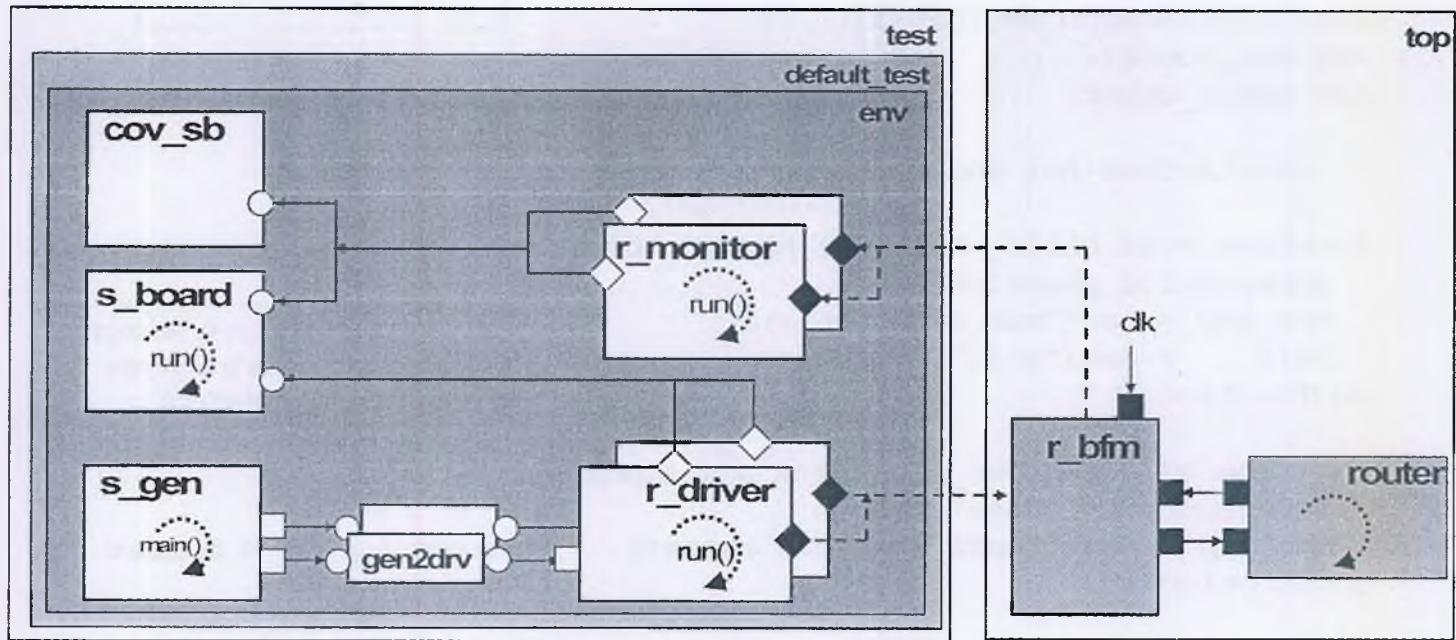


Up: Connect fifo's analysis export to the mem_checker's analysis export (make available up)

Notes:

Lab - Hierarchical Testbench: Before Hierarchy

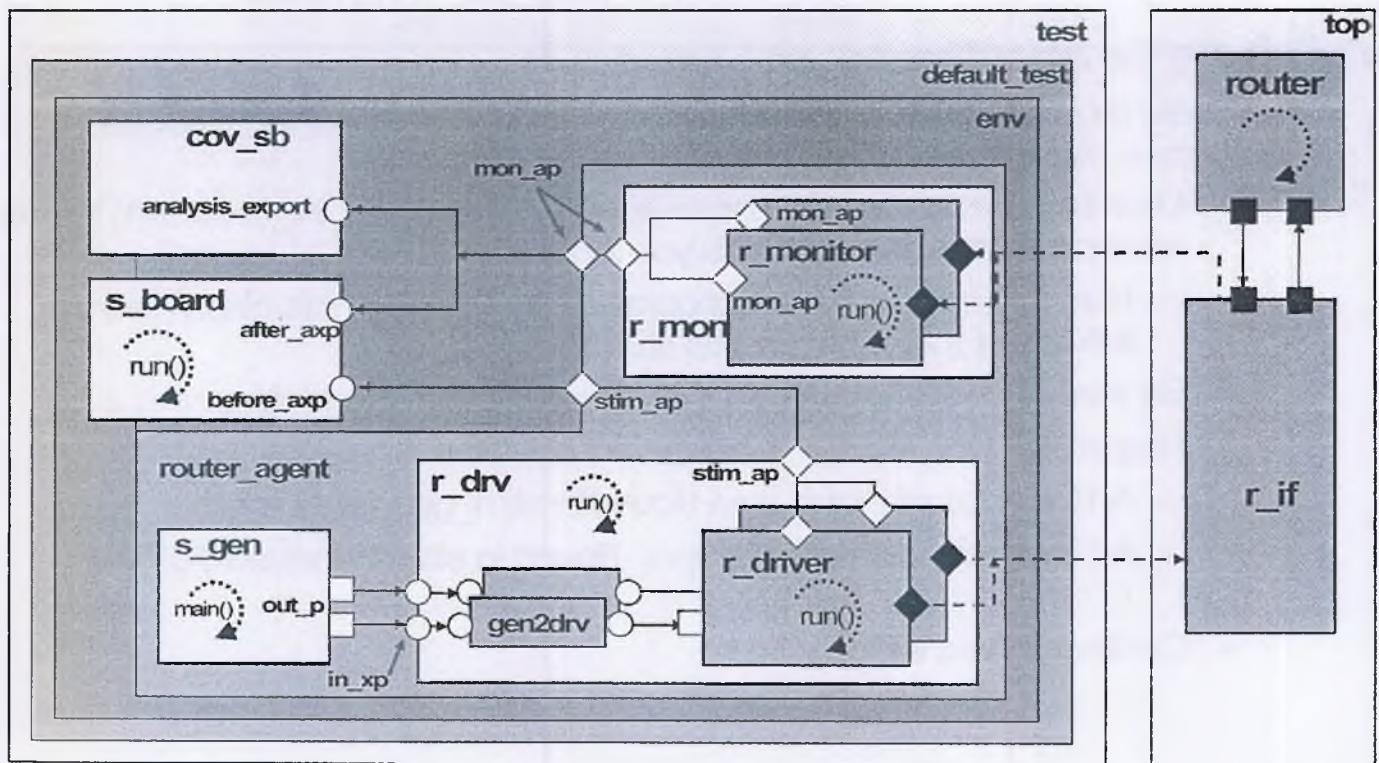
- This is a view of the router testbench before hierarchy added



Notes:

Lab - Hierarchical Testbench: Overview

- Working directory: *hierarchy*
- Change to have hierarchy in the testbench



Notes:

Lab - Hierarchical Testbench: Instructions - 1 of 2

- Working directory: *hierarchy*
- NOTE: For this lab the test used is *test_std_overrides*
- Edit the file **agent/router_std_agent.svh**
 - Create a container class called `router_std_agent` which extends `router_agent_base`
 - ◆ Look at `router_agent_base` (found in `router_agent_base.svh`) to see what properties and methods you inherit!
 - Note: There is some configuration stuff done in this class - we will talk about the configuration stuff later
 - ◆ Be sure and register `router_std_agent` with the factory
 - ◆ Properties
 - An instance of `stim_gen` (found in `stim_gen.svh`) called `s_gen`
 - An instance of `std_drivers` (found in `std_drivers.svh`) called `r_drv`
 - ◆ Continued next slide...

Notes:

Lab - Hierarchical Testbench: Instructions 2 of 2

(Still creating the class `router_std_agent`)

- ◆ Methods
 - `new()`
 - `build_phase()`
 - » Create `s_gen`, `r_drv` instances
 - `connect_phase()`
 - » Connect `r_drv` port & exports
 - » Connect `s_gen` ports
 - » NOTE: for the size of the router use
`m_config.m_router_size`. This is configuration stuff that we will talk about later

■ Compile & Run

- No errors: `make`
- Errors: `make bad`

Notes:

Lab - Hierarchical Testbench: Sample Output

```
# UVM_INFO@1451000: uvm_test_top.env.cov_sb [COVERAGE]
# UVM_INFO@1451000: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage = 100.000000%
# UVM_INFO@1451000: uvm_test_top.env.cov_sb [COVERAGE] 100% Coverage met with 12 transactions
# UVM_INFO@1451000: uvm_test_top.env.cov_sb [COVERAGE]
#
# UVM_INFO@1451000: uvm_test_top.env.s_board [Packet_Comparator] Matches: 992
# UVM_INFO@1451000: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
# UVM_INFO@1451000: uvm_test_top.env.s_board [Packet_Comparator] Missing: 8

#
#— UVM Report Summary —
#
# ** Report counts by severity
# UVM_INFO: 9
# UVM_WARNING: 0
# UVM_ERROR: 0
# UVM_FATAL: 0
# ** Report counts by id
# [COVERAGE] 4
# [Packet_Comparator] 3
# [RNTST] 1
# [UVMTOP] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 1451 us Iteration: 98 Instance: /test
```

Sol



uvm_intro_3.5

©Willamette HDL Inc.

196

Notes:

Test Class

In this section

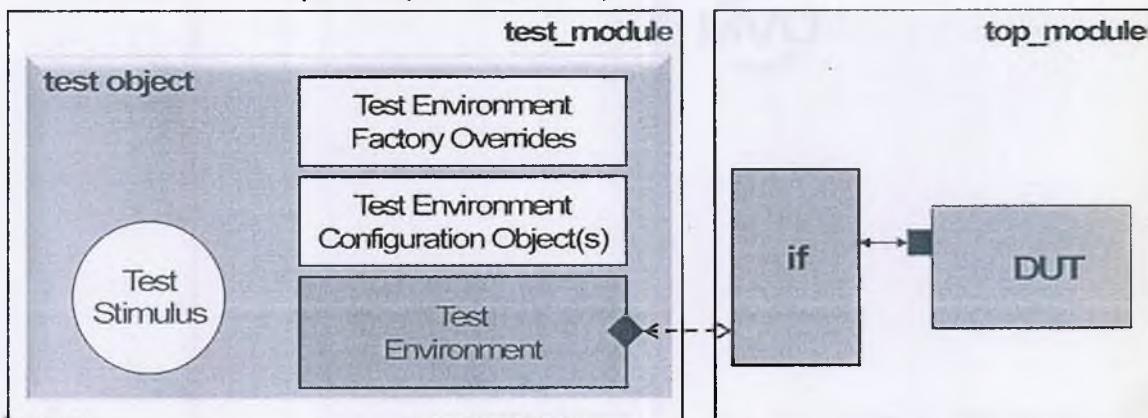
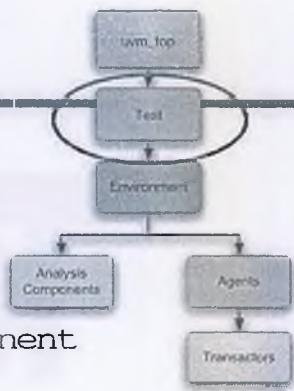


Test class

Notes:

Test as Top-level Class - review

- The top level user-defined class is a "test" class
 - "test" in diagram below
- A test class "should" extend `uvm_test`
 - `uvm_test` is an empty container class that extends `uvm_component`
 - A typical test class contains the following:
 - ◆ Instantiation of the test environment
 - ◆ Factory overrides for the test environment (later in this section)
 - ◆ Setting up configuration objects for configuring the test environment (later section)
 - ◆ A test
 - Stimulus or sequences (later sections)

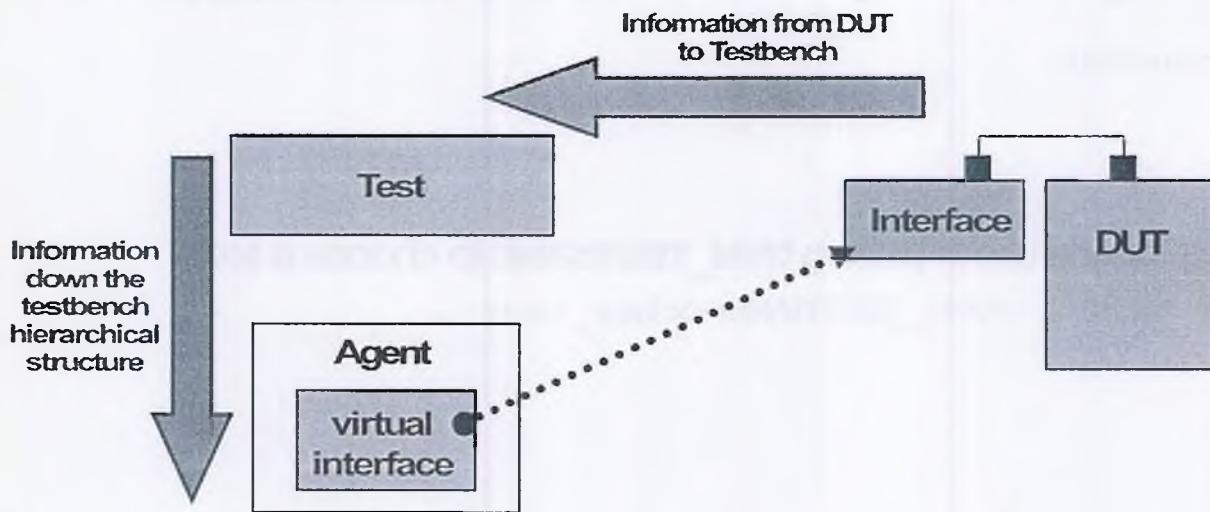


198

Notes:

Test Class

- The test class is the one class that "knows" about the configuration of the entire testbench
 - Central location to make changes when configuration changes occur
 - "Gathers" DUT information
 - Parameters, virtual interfaces etc.
 - Distributes configuration information to the environment
 - Should be the only class that is edited based on new or different configuration information



Notes:

Module test

- `run_test()` determines the "test" to be run
 - Argument is the default "test"
 - "Override" argument with simulator command line plusarg `UVM_TESTNAME`

```
module test;  
  ...  
  initial  
    run_test("default_test"); // create and start running test  
  
endmodule
```

- Example use of plusarg `UVM_TESTNAME` to choose a test

```
vsim ... +UVM_TESTNAME=other_test ...
```

Notes:

Test Example: wb_mem_test_base

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

wb_env env;
virtual wb_reset_if v_wb_reset;

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

function void build_phase(uvm_phase phase);
super.build_phase(phase);
env = wb_env::type_id::create("env", this);
// Get virtual wb reset interface
if(!uvm_config_db #(virtual wb_reset_if)::get(
    this, "", "v_wb_reset", v_wb_reset))
`uvm_fatal("RSRCNF", "uvm_config_db #(virtual wb_reset_if)::get() error")
// configuration stuff in next sections
endfunction

task reset_phase(uvm_phase phase);
phase.raise_objection(this);
v_wb_reset.reset_wb_bus(); // reset Wishbone bus
phase.drop_objection(this);
endtask
endclass
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem

201

Notes:

Factory Overrides

In this section



Factory Overrides
Debugging Overrides

Notes:

Overriding Types in the Factory

- Overriding allows for substitution of one type (new type) for another (original type) in the factory
 - Done at runtime!
- After overriding, when a request is made for the original type, the factory will instead create the new type
 - No change is necessary to the original code that requests the object
 - New type must be type-compatible with overridden type, otherwise assignment will fail
 - ◆ Use class inheritance
- Great for modifications!
 - Allows for flexible configurations
 - ◆ Create base class place holders that are substituted with different derived classes
 - Allows for changing behavior without modifying code
 - ◆ Replace an object with another of which has different behavior

Notes:

Override Mechanics

- Both the new type and the original type must be registered with the factory
- When `create()` is called on the original type:
 - Before creating the object the factory checks to see if an override is in place for the original type
 - ◆ If yes then it creates an object of the new type instead
 - ◆ If no, then it creates an object of the original type
- An override can either "by type" or "by instance"
 - By type means that the new type is substituted for the original type anywhere in the UVM structure
 - By instance means the new type is substituted for the original type only at the specified location(s) in the UVM structure

Notes:

Overriding by Type: `set_type_override()`

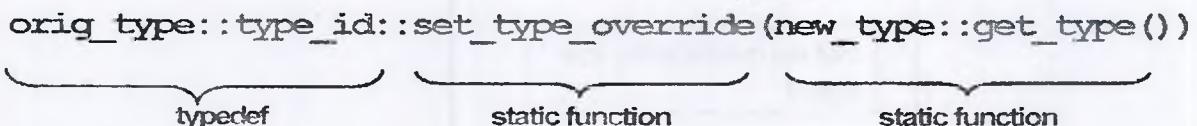
UVM

```
static function void set_type_override (
    uvm_object_wrapper override_type, bit replace=1);
```

- `uvm_object_wrapper override`
 - ◆ Handle to a singleton `uvm_object_wrapper` of the new type
??!! Easy – just call the method `new_type::get_type()` to get the handle to the singleton wrapper object
 - `bit replace=1`
 - ◆ 1: if a type override exists new override will take effect (default value)
 - ◆ 0: if a type override exists it will remain in place
-
- `set_type_override` is a static function in `type_id` - use `class_name::type_id::` prefix:

Follow this pattern

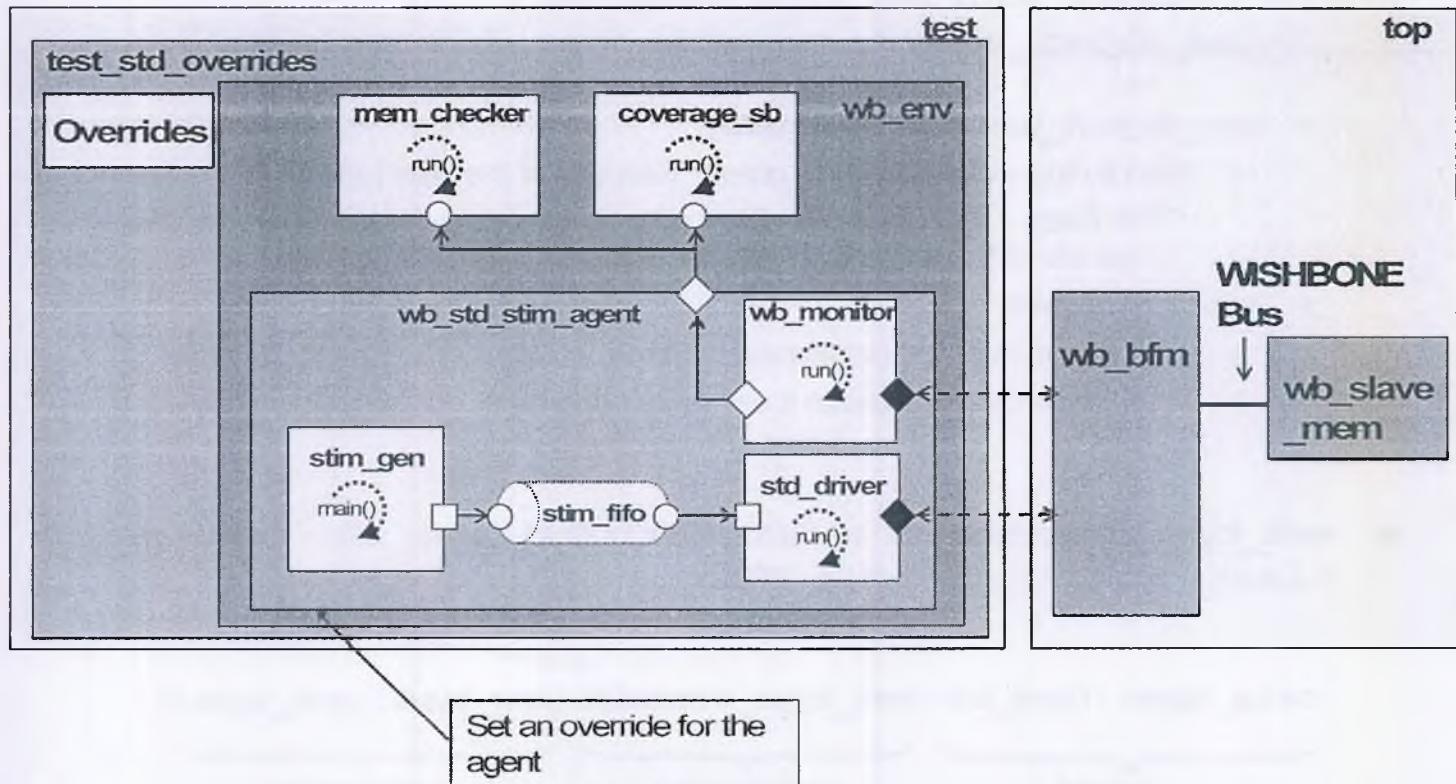
```
orig_type::type_id::set_type_override(new_type::get_type())
```



The code `orig_type::type_id::set_type_override(new_type::get_type())` is annotated with curly braces under the words `orig_type`, `type_id`, and `set_type_override`. An arrow points from the word `type_id` to the box labeled "Follow this pattern". Below each brace is a label: "typedef" under `orig_type::type_id::`, "static function" under `set_type_override`, and "static function" under `(new_type::get_type())`.

Notes:

Example Circuit



Notes:

Example by Type Override

```
class wb_env extends uvm_env;
  ...
  wb_agent_base agent;
  function void build_phase(uvm_phase phase);
    ...
    agent = wb_agent_base::type_id::create("agent", this);
    ...
  endfunction
  ...
endclass

class test_std_overrides extends wb_mem_test_base;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Overrides
    wb_agent_base::type_id::set_type_override(wb_std_stim_agent::get_type());
    ...
  endfunction

```

Overrides done in the build_phase() of the test class. This guarantees overrides set before any factory requests. Note: Careful where super.build_phase() call is done. May need to be after overrides

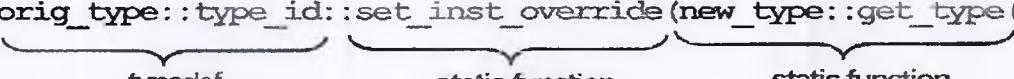
Notes:

Overriding by Instance Path

UVM

- ```
static function void set_inst_override (
 uvm_object_wrapper override_type, string inst_path,
 uvm_component parent = null);
```
- `uvm_object_wrapper override_type`
    - ◆ Wrapper singleton instance provided by the method `class_name::get_type()`
  - `string inst_path`
    - ◆ Instance name to be overridden
    - ◆ May include wildcards (\*) and (?)
  - `uvm_component parent`
    - ◆ If provided, `inst_path` arg is relative to parent path
  - `set_inst_override` is a static function in `type_id` - use  
`class_name::type_id::` prefix:

```
orig_type::type_id::set_inst_override(new_type::get_type(), "orig_inst_path")
```



typedef                    static function                    static function

Notes:

---

---

---

---

## Example by Instance Override

```
class wb_std_stim_agent extends wb_agent_base;
 ...
 wb_driver_base drv; // base class handle
 function void build_phase(uvm_phase phase);
 ...
 //components
 drv = wb_driver_base::type_id::create("drv", this);
 ...
 endfunction
 ...
endclass

class test_std_overrides extends wb_mem_test_base;
 ...
 mac_env env; // environment class
 function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 // Overrides
 wb_driver_base::type_id::set_inst_override(std_driver::get_type(),
 "uvm_test_top.env.agent.drv");
 ...
 endfunction
 ...
endclass
```



Notes:

# Debugging Overrides

- It is disconcerting to think you have put in an override to find you really didn't
  - Common errors include miss spelling names and instance paths
- The Factory is a singleton object with the global handle `factory`
  - Has some debug functions

```
function void debug_create_by_type (uvm_object_wrapper requested_type,
 string parent_inst_path = "",
 string name = "");
```

```
function void debug_create_by_name (string requested_type_name,
 string parent_inst_path = "",
 string name = "");
```

- Performs the same lookup as `create`, but does not actually create an object
- Provides detailed information about what would be created, listing each override applied

```
function void print (int all_types=1);
```

- Prints out all the overrides in the factory
- If argument `all_types` is set to a 1 then all types registered with the factory are printed regardless of overrides

Notes:

---

---

---

# Example Debugging Overrides – test class

```
class test_std_overrides extends wb_mem_test_base;
`uvm_component_utils(test_std_overrides)
...
function void build_phase(uvm_phase phase);
 super.build_phase(phase);
// overrides
//wb_driver_base::type_id::set_type_override(std_driver::get_type());
//wb_driver_base::type_id::set_inst_override(std_driver::get_type(), "*");
//wb_driver_base::type_id::set_inst_override(std_driver::get_type(),
// "uvm_test_top.env.agent.*");
//wb_driver_base::type_id::set_inst_override(std_driver::get_type(),
// "*drv");
wb_driver_base::type_id::set_inst_override(std_driver::get_type(),
 "uvm_test_top.env.agent.drv");

factory.print(); // print out factory overrides and registrations
endfunction
...
endclass
```



UVM\_intro\_3.5

© Williamette HDL Inc.

code in examples/wb\_mem

211

Notes:

---

---

---

# Example Debugging Overrides Output

```
Factory Configuration (*)
Instance Overrides:
#
Requested Type Override Path Override Type

wb_driver_base uvm_test_top.env.agent.drv std_driver
#
No type overrides are registered with this factory
#
All types registered with the factory: 47 total
(types without type names will not be printed)
#
Type Name

coverage_sb
default_test
mem_checker
mem_coverage
wb_monitor
std_driver
stim_gen
test_std_overrides
wb_agent_base
wb_driver_base
wb_env
wb_mem_test_base
wb_std_stim_agent
wb_txn
(*) Types with no associated type name will be printed as <unknown>
```

Notes:

# Testbench Topology Printout

- Print out the topology of the testbench
  - Do after build phase!

```
function void uvm_root::print_topology(uvm_printer printer=null);
 • Method of uvm_top
 • Prints out the testbench topology

class test_std_overrides extends wb_mem_test_base;
 ...

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

Topology printed in  
end\_of\_elaboration\_phase()  
after build is complete

Notes:

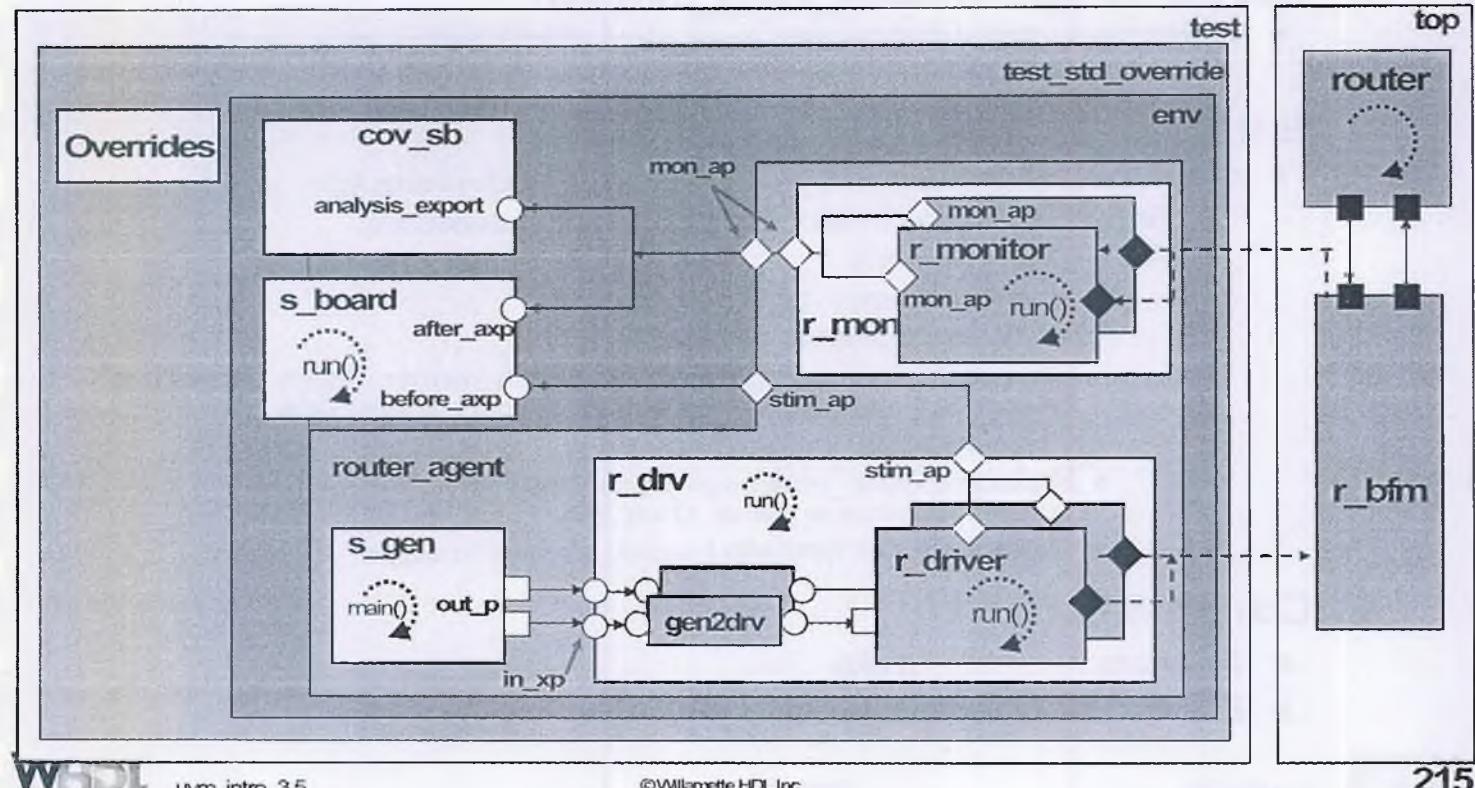
# Example Print Topology Output

| #                 | Type                       | Size | Value |
|-------------------|----------------------------|------|-------|
| # Name            |                            |      |       |
| #                 |                            |      |       |
| # uvm_test_top    | test_std_overrides         | -    | @500  |
| # env             | wb_env                     | -    | @512  |
| # agent           | wb_std_stim_agent          | -    | @547  |
| # drv             | std_driver                 | -    | @583  |
| # rsp_port        | uvm_analysis_port          | -    | @600  |
| # sqr_pull_port   | uvm_seq_item_pull_port     | -    | @591  |
| # stim_p          | uvm_blocking_get_port      | -    | @655  |
| # mon             | wb_monitor                 | -    | @567  |
| # mon_ap          | uvm_analysis_port          | -    | @673  |
| # mon_ap          | uvm_analysis_port          | -    | @558  |
| ...               |                            |      |       |
| # chk             | mem_checker                | -    | @522  |
| # buff            | uvm_tlm_analysis_fifo #(T) | -    | @721  |
| # analysis_export | uvm_analysis_imp           | -    | @765  |
| # get_ap          | uvm_analysis_port          | -    | @756  |
| # get_peek_export | uvm_get_peek_imp           | -    | @738  |
| # put_ap          | uvm_analysis_port          | -    | @747  |
| # put_export      | uvm_put_imp                | -    | @729  |
| # mon_exp         | uvm_analysis_export        | -    | @712  |
| # cov_sb          | coverage_sb                | -    | @530  |
| # analysis_imp    | uvm_analysis_imp           | -    | @538  |
| #                 |                            |      |       |

Notes:

# Lab - Factory Override: Overview

- Working directory: `factory_override`
- Use factory overrides to generate a derived stimulus object and substitute new coverage scoreboard to handle coverage properly



Notes:

# Lab - Factory Override Instructions

NOTE: For this lab the test used is `test_std_overrides`

- Working directory: `factory_override`
- Edit the file `agent/Packet_pass_thru.svh`
  - Create a new class called `Packet_pass_thru` which inherits from `Packet`
    - ◆ Add a constraint to do "pass through" txns (`src_id == dest_id`)
- Edit the file `tests/test_std_overrides.svh`
  - Create a test class called `test_std_overrides` which inherits from `test_base`
    - ◆ Be sure and register it with the factory and add a constructor
    - ◆ Add a `build_phase()`
      - Be sure to call `super.build_phase()`
      - Add an instance override
        - » Replace the `router_agent_base` type in `router_agent` instance in the `router_env` class with the `router_std_agent` type
      - Add type overrides
        - » Replace `router_coverage_base` type with `router_coverage_pass_thru` type
        - » Replace `Packet` type with `Packet_pass_thru` type
  - Compile & Run
    - No errors:                   `make`
    - Errors:                      `make bad`

Notes:

# Lab - Factory Override: Sample Partial Output

```
Factory Configuration (*)
#
Instance Overrides:
#
Requested Type Override Path Override Type

router_agent_base uvm_test_top.env.router_agent router_std_agent
#
Type Overrides:
#
Requested Type Override Type

router_coverage_base router_coverage_pass_thru
Packet Packet_pass_thru
#
UVM_INFO ./analysis/coverage_sb.svh(37)@ 1451000: uvm_test_top.env.cov_sb [COVERAGE] ****
UVM_INFO ./analysis/coverage_sb.svh(38)@ 1451000: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage =
100.000000%
UVM_INFO ./analysis/coverage_sb.svh(41)@ 1451000: uvm_test_top.env.cov_sb [COVERAGE] 100% Coverage met with 12
transactions
UVM_INFO ./analysis/coverage_sb.svh(42)@ 1451000: uvm_test_top.env.cov_sb [COVERAGE] ****
#
UVM_INFO @ 1451000: uvm_test_top.env.s_board [Packet_Comparator] Matches: 992
UVM_INFO @ 1451000: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
UVM_INFO @ 1451000: uvm_test_top.env.s_board [Packet_Comparator] Missing: 8
```

Sol



uvm\_intro\_3.5

© Willamette HDL Inc.

217

Notes:

### UVM

```
function void set_type_override_by_name (
 string original_type_name,
 string override_type_name,
 bit replace=1);
```

- `string original_type_name`
    - ◆ Original type that is being overridden
  - `string override_type_name`
    - ◆ New type that will be created instead of the original type
  - `bit replace=1`
    - ◆ 1: if a type override exists new override will take effect
    - ◆ 0: if a type override exists it will remain in place
- Method of singleton factory
  - Before overriding the new type must be registered with the factory

# Overriding by Name & by Instance Path

Reference Page

UVM

```
function void set_inst_override_by_name (
 string original_type_name,
 string override_type_name,
 string full_inst_path);
```

- `string original_type_name`
    - ◆ Original type that is being overridden
  - `string override_type_name`
    - ◆ New type that will be created instead of the original type
  - `string full_inst_path`
    - ◆ Instance name to be overridden
    - ◆ May include wildcards (\*) and (?)
- Places an instance-specific override in the factory
    - Both the instance & the lookup\_str must match
  - Method of singleton `factory`
  - Before overriding the new type must be registered with the factory



uvm\_intro\_3.5

© Willamette HDL Inc.

219

Notes:

---

---

---

# Example by Type & by Instance Overriding

## Reference Page

```
class test_std_overrides extends alu_test_base;
`uvm_component_utils(test_std_overrides)

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

function void build_phase(uvm_phase phase);

// factory overrides before anything else
factory.set_type_override_by_name("stim_group_base", "std_stim_group");
factory.set_type_override_by_name("analysis_group_base", "std_analysis_group");
// Replace alu_txm
factory.set_type_override_by_name("alu_txm","alu_txm_ADD")
Substitute a
// Replace coverage collector
ADD_alu_txm for
factory.set_inst_override_by_name(),"cov_collector",
any request for a
"ADD_collector"
alu_txm is made
"uvm_test_top.t_env.alu_agent.cov_sb");

super.build_phase(phase); // after overrides
endfunction

endclass
```

When the `uvm_test_top.t_env.alu_agent.cov_sb` instance (and only this instance) is created and a request is for a `cov_collector`, substitute `ADD_collector`.

No other instance requests are affected

Notes:

---

---

---

+uvm\_set\_inst\_override=<req\_type>,<override\_type>,<full\_inst\_path>

- Works like set\_inst\_override\_by\_name()

+uvm\_set\_type\_override=<req\_type>,<override\_type>[,<replace>]

- Works like set\_type\_override\_by\_name()
  - ◆ Optional *replace* argument value is 0 or 1 (default is 1)
    - Specifies whether previous overrides should be replaced

# Configuration Database

In this section



Resources  
`configuration_db`  
Debugging configurations

Notes:

# UVM Resources

- Centralized database to store/retrieve arbitrary configuration data
  - Often referred to as the "resource database", "configuration database" or "config database"
- A way to pass configuration data throughout the hierarchy in a common way
  - Can't use constructor arguments with the factory
  - No need to create lots of non-standard and incompatible / non-reusable global variables
  - Use database access methods instead
- Typical uses are:
  - Configuring structural topology
  - Provide virtual interface information
  - Configuring internal parameters (array sizes, constants, timing)
  - Configuring operational modes (error injection, debug)

Notes:

---

---

---

# Resources

- Configuration data is stored in a container class called a resource
- Resources have a:
  - Name
    - ◆ String used for accessing the resource
  - Type
    - ◆ Type of the data object stored
    - ◆ Can be any SV type or user defined type
  - Scope
    - ◆ Scope defines a context where the resource is visible
  - Methods that define an access API
    - ◆ These are not used directly but rather through a convenience layer API discussed later
    - ◆ See reference slides after the lab in this section

Resource



uvm\_intro\_3.5

©Willamette HDL Inc.

224

Notes:

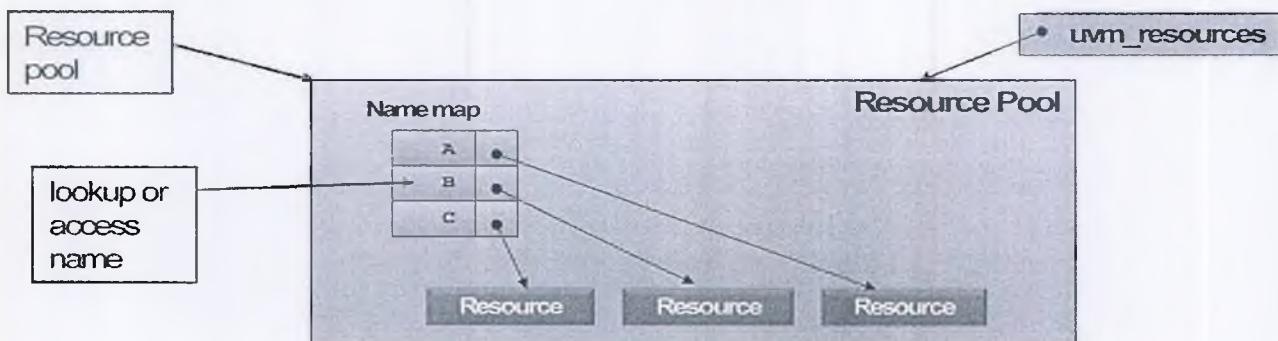
---

---

---

# Resource Database

- Resource database
  - Pool of resources (type `uvm_resource_pool`)
    - ◆ Singleton called `uvm_resources` with a `uvm_pkg` scope
  - Resources are accessed by name
    - ◆ It is possible to access by type but we will ignore that
- Accessible from any type of UVM object (components, sequences, transactions etc.)



Notes:

---

---

---

# Resource Scope

- Scope defines a context where the resource is visible
  - In other words the scope of a resource defines what object(s) may access the resource
- Scope is the UVM hierarchical path of object(s) that may access the resource
  - Path may be expressed fully or may include wild card characters
    - \* 0 or more characters
    - + 1 or more characters
    - ? 1 character exactly

Notes:

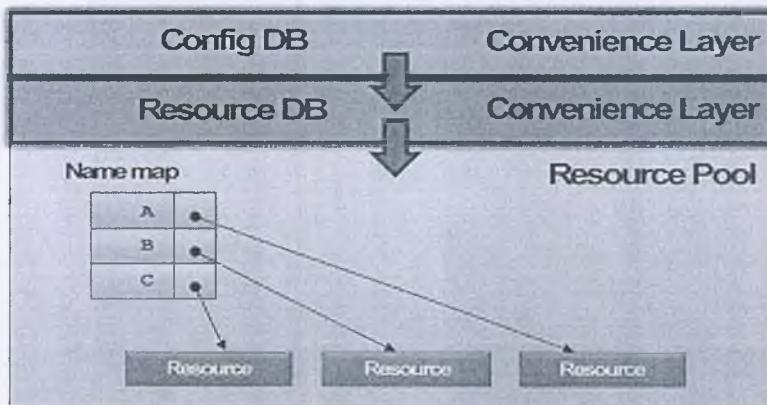
---

---

---

# Convenience Layers

- There are two convenience layers provided for putting resources into the database and accessing resources in the database
- `resource_db`
  - Convenience layer on top of the resource database
    - ◆ Not discussed here - see reference slides after the lab in this section
- `config_db`
  - Convenience layer on top of the `uvm_resource_db`
    - ◆ We will use this layer!



Notes:

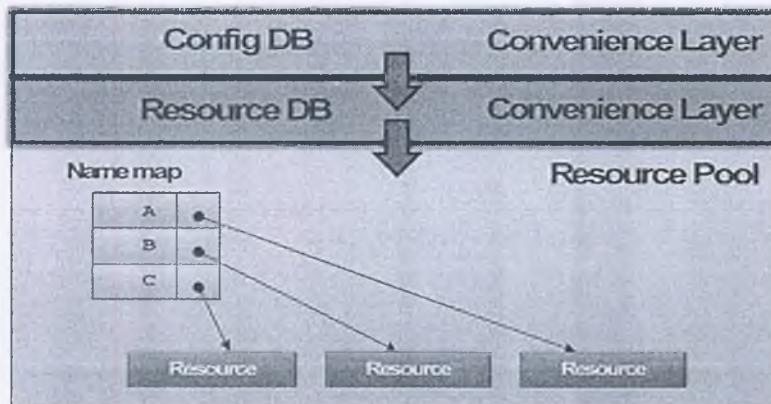
---

---

---

# config\_db Convenience Layer

- Has two sets of API methods
  1. Methods for setting and getting resources
    - ◆ We will focus here
  2. OVM backward compatibility methods (not discussed in this class)
    - ◆ Allows for code converted from OVM to UVM to run with resources without replacing configuration database calls
      - `set_config_xxx()`, & `get_config_xxx()`



Notes:

---

---

---

## uvm\_config\_db#(T)

- Parameterized class
  - Parameter **T** is the type of data stored in the resource in the resource database
    - ◆ Types must match exactly for setting and getting!
- API methods (not including OVM backward compatible methods)
  - `set()`
    - ◆ Place or modify a resource in the resource database
  - `get()`
    - ◆ Fetch the value of an existing resource in the resource database
  - `exists()`
    - ◆ Check to see if a resource is available in the database
  - `wait_modified()`
    - ◆ Block until a resource is set (or updated)

Notes:

---

---

---

## set()

**UVM** static function void **set**(uvm\_component cntxt, string inst\_name,  
string field\_name, T value);

- Create a new or update an existing resource at `field_name` with `value`
  - The full name of the `cntxt` component is concatenated with the `inst_name` to form the scope of resource
    - ◆ If `cntxt` is null then it is replaced with `uvm_top`
- If a resource with the same type and the same context already exists then its value is updated
  - Else a new resource is created
- Processes that are waiting on the creation or an update of this resource are triggered
- Since `set()` is a static method it may be called from anywhere
  - Components, transactions, sequences etc.

Notes:

---

---

---

# Example set()

```
class alu_config extends uvm_object;
`uvm_object_utils(alu_config)

int m_alu_id = 0;
virtual alu_if m_v_alu_if;

endclass

//-----
class test_std_overrides extends alu_test_base;

alu_config m_config; // config object

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

function void build_phase(uvm_phase phase);

// Configuration
m_config = alu_config::type_id::create("m_config", this);
...
uvm_config_db #(alu_config)::set(this,"*", "ALU_CONFIG", m_config); // set config
...
endfunction
...
endclass
```

Resource context is concatenation:  
{this.get\_full\_name(),".",'\*'} = uvm\_test\_top.\*



uvm\_intro\_3.5

© Willamette HDL Inc

code in examples/alu

231

Notes:

# get()

**UVM**

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

- Get the value of an existing resource at `field_name`
  - The value is returned through the `inout argument` `value`
  - `cntxt` defines the search starting point and `inst_name` is relative to `cntxt`
  - An easy way to think of it is the scope of the request must match the scope of the resource to access the resource
    - ◆ The full name of the `cntxt` component is concatenated with the `inst_name` to form the scope of the request
      - Wild card characters may be used
  - A 1 is returned if successful, 0 if not
- Since `get()` is a static method it may be called from anywhere
  - Components, transactions, sequences etc.

Notes:

---

---

---

# Example get()

```
class monitor extends uvm_component;
 `uvm_component_utils(monitor)
 ...
 alu_config m_config;

 function void build_phase(uvm_phase phase);
 // get config object
 if(!uvm_config_db#(alu_config)::get(this, "", "ALU_CONFIG", m_config))
 `uvm_fatal("CONFIG", "Config error in get()")
 ...
 endfunction
 ...
endclass
```

Context for resource match is concatenation:  
{this.get\_full\_name(), ""} =  
uvm\_test\_top.t\_env.alu\_agent.mon  
which matches context from set() of  
uvm\_test\_top.\*



Recommended:

use `get(this, "", "lookup_name", value)` format

Notes:

## exists()

**UVM** static function bit **exists**(uvm\_component cntxt,  
                          string inst\_name, string field\_name,  
                          bit spell\_chk = 0)

- Check to see if a resource exists at `field_name`
- `exists()` behaves the same as `get()` except it does not return a value
- A 1 is returned if the resource exists and is accessible for the given scope, 0 if not
- The `spell_chk` argument if set will cause a warning message to be generated if there is no match

```
if(uvm_config_db #(alu_config)::exists(this,"", "ALU_CONFIG"))
 ...

```

Notes:

---

---

---

---

## wait\_modified()

**UVM** static task **wait\_modified**(uvm\_component cntxt,  
                          string inst\_name, string field\_name);

- Wait for a resource to be set
- The calling process blocks until a new setting is applied that effects the specified resource
- The scope is used in the same manner as `set()`, `get()` to determine what resource is being accessed



Notes:

---

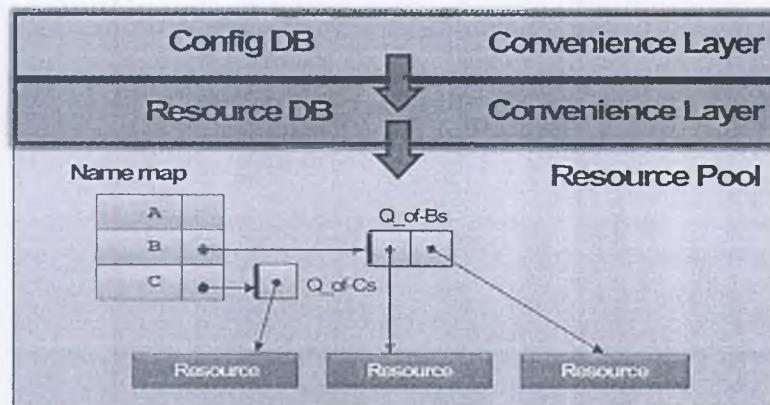
---

---

---

# Precedence with `uvm_config_db#(T)`

- Resources with the same name are stored in queues
  - You can think of the resource pool as an associative array of queues
  - Head of the queue has highest priority
  - New resources are pushed onto the head of the queue regardless of context
    - ◆ Last set wins semantics



Notes:

---

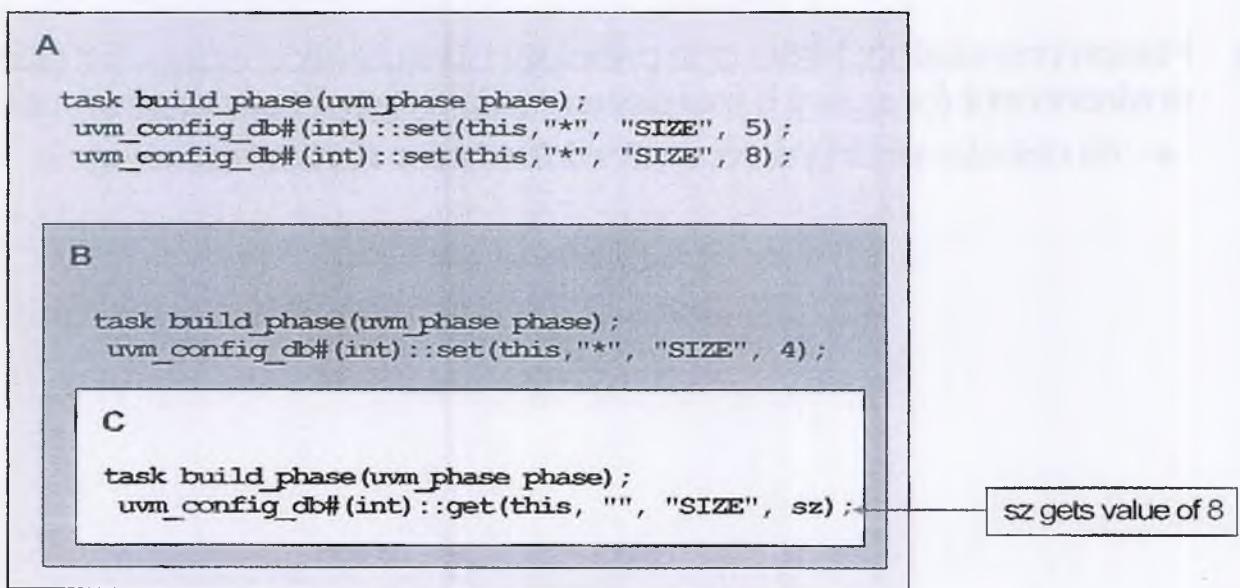
---

---

---

# Build Phase Precedence Exception

- There is an exception to the last set wins precedence in the uvm\_config\_db
  - In the build phase (only) hierarchy is used to determine precedence
    - ◆ Settings from higher hierarchical levels have higher precedence
    - ◆ Settings from the same level of hierarchy have last set wins semantics



Notes:

# Setting Configuration Data

- Remember resource containers may hold any SV type
  - Tempting to have lots of individual containers of string, int etc.
  - Best practice is to limit the number of resources you use
    - ◆ Define custom configuration classes with many properties
  - Limits confusion, better for reuse, more efficient use of memory, etc.
- Recommendation: Have one principal configuration object for each environment (or agent if there are multiple agents per environment)
  - All objects needing configuration information access this resource

Notes:

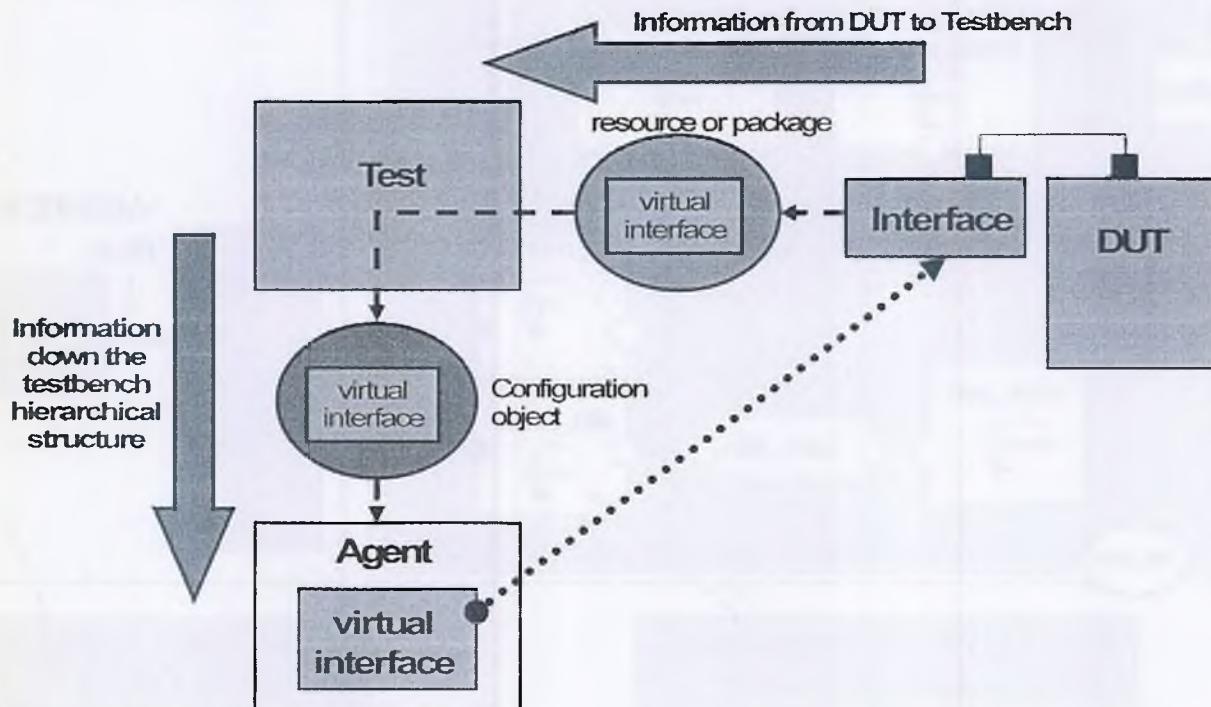
---

---

---

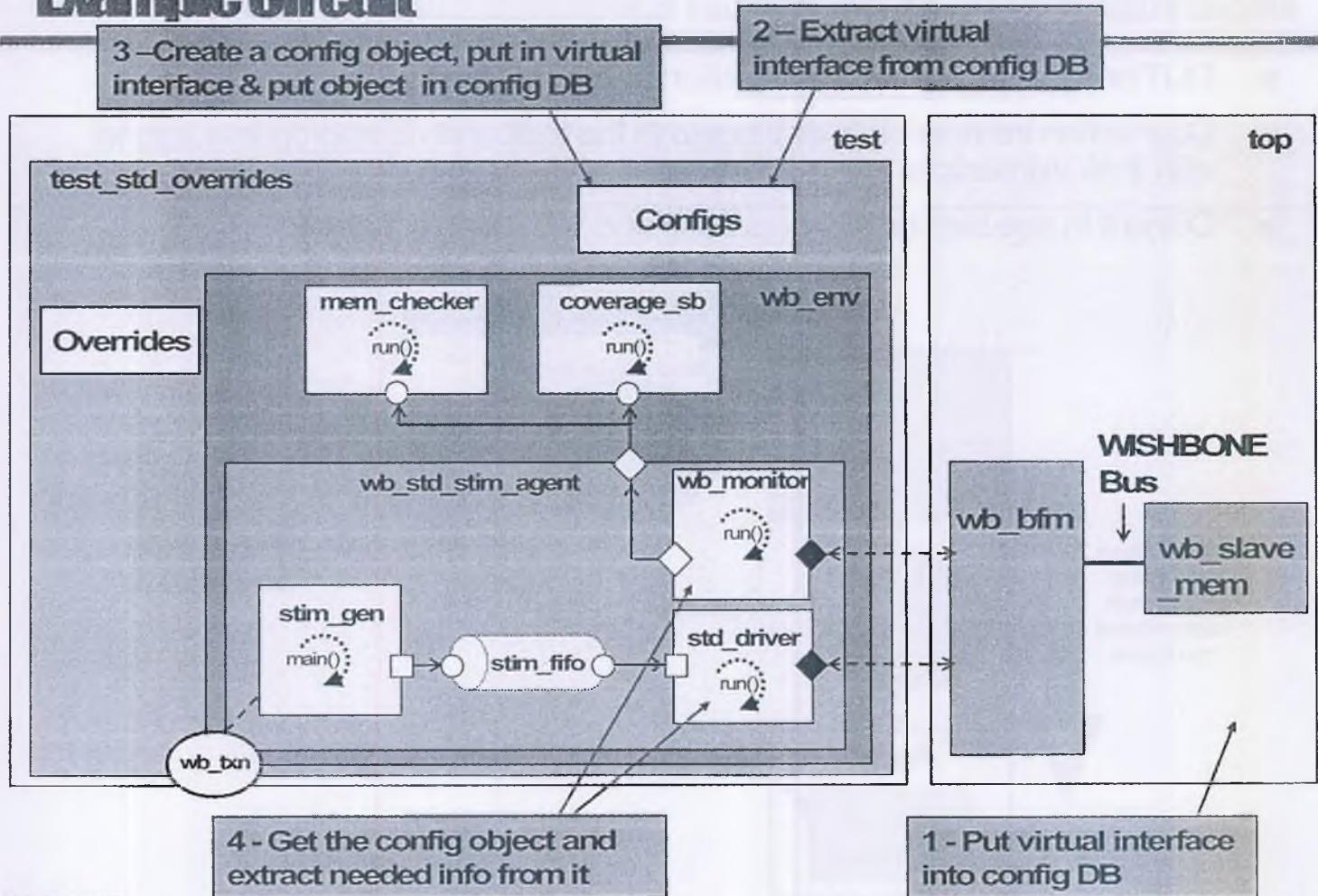
# Distribution of Testbench Information

- DUT provides connection information to the testbench
- Connection information is distributed in the testbench to appropriate agents with their transactors
- Doing it in one step breaks scalability and VIP reuse concerns



Notes:

# Example Circuit



Notes:

---



---



---

## Example: top

1 - Top module places virtual interface into config DB

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

// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wb_syscon_bfm_if wb_bfm();
...

initial begin
 uvm_config_db #(virtual wb_syscon_bfm_if)::set(
 uvm_top, "uvm_test_top", "v_wb_bfm", wb_bfm);
 ...
end
endmodule
```

Place in database with a scope of "uvm\_test\_top" which is the default location of the top level test created by run\_test()

Lookup of "v\_wb\_bfm"

Notes:

# Example Config Object: wb\_config

```
class wb_config extends uvm_object;
 `uvm_object_utils(wb_config)

 virtual wb_syscon_bfm_if v_wb_bfm; // virtual interface
 uvm_sequencer #(wb_txn) wb_seqr; // Wishbone bus agent sequencer

 int m_wb_id; // WISHBONE bus ID (which WISHBONE bus)
 int m_wb_master_id; // WISHBONE ID of WISHBONE agent bus master driver
 // MAC
 int m_mac_id; // WISHBONE ID of MAC master
 int m_mac_s_wb_id; // WISHBONE ID of MAC slave
 int m_dma_m_wb_id; // WISHBONE ID of DMA master
 int m_dma_s_wb_id; // WISHBONE ID of DMA slave
 int unsigned m_mac_wb_base_addr; // Wishbone base address of MAC
 int unsigned m_wb_s_base_addr[16]; // Base addresses of WISHBONE slaves
 int unsigned m_ip_addr; // MAC IP address
 bit [47:0] m_mac_eth_addr; // Ethernet address of MAC
 bit [47:0] m_tb_eth_addr; // Ethernet address of testbench for sends/receives
 // WISHBONE slave memory
 int m_mem_s0_wb_id; // WISHBONE ID of slave memory 0
 int m_mem_s1_wb_id; // WISHBONE ID of slave memory 1
 int m_mem_slave_size[16]; // Size of slave memories in bytes
 function new(string name = "");
 int k;
 foreach(m_wb_s_base_addr[i]) begin
 m_wb_s_base_addr[i] = k;
 k = k + test_params_pkg::SLAVE_ADDR_SPACE_SZ;
 end
 endfunction
 ...

```

Configuration object to hold all configuration information passing from test to environment components



uvm\_intro\_3.5

©Willamette HDL Inc.

code in wishbone/wishbone\_bus

242

Notes:

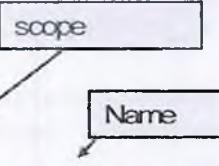
## Example Test: wb\_mem\_test\_base

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

wb_config wb_config_0; // config object for WISHBONE BUS
function new(string name, uvm_component parent);
 super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 ...
 wb_config_0 = new();
 if(!uvm_config_db #(virtual wb_syscon_bfm_if)::get(this, "", "v_wb_bfm",
 wb_config_0.v_wb_bfm))
 `uvm_fatal("RSRCNF", "uvm_config_db #(virtual wb_syscon_bfm_if)::get() error")
 wb_config_0.m_wb_id = 0; // WISHBONE 0
 // note parameters used below are from test_params_pkg
 wb_config_0.m_wb_master_id = WB_AGENT_MASTER_ID; // the ID of the wb agent master
 wb_config_0.m_mem_s0_wb_id = MEM_SLAVE_0_WB_ID; // the ID of the slave memory
 wb_config_0.m_mem_slave_size[0] = MEM_SLAVE_0_SIZE; // size of slave memory
 // set wb_config_0 in config database so env and descendants can access
 uvm_config_db #(wb_config)::set(this, "env*", "wb_config", wb_config_0);
 ...
endclass
```

2 - Extract virtual interface from config DB



Place in database with scope of  
"uvm\_test\_top.env\_0\*" with  
a lookup of "wb\_config"

3 - Create a config object, put in virtual interface & put object in config DB



uvm\_intro\_3.5

© Willamette HDL Inc.

code in examples/wb\_mem

243

Notes:

# Example: uvm\_bus\_monitor

```
class wb_bfm_monitor extends uvm_monitor;
 `uvm_component_utils(wb_bfm_monitor)

 uvm_analysis_port #(wb_txn) mon_ap;
 virtual wb_syscon_bfm if v_wb_bfm;
 wb_config m_config; // Config object
```

4 - Get the config object and extract needed info from it

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

config handle

```
function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 mon_ap = new("mon_ap", this);
 if(!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config))
 `uvm_error("CONFIG", "wb_config not found");
 v_wb_bfm = m_config.v_wb_bfm; // set local virtual interface
endfunction
```

Get config object from database

```
...
endclass
```

Assign local virtual interface from virtual interface in config object

Notes:

---

---

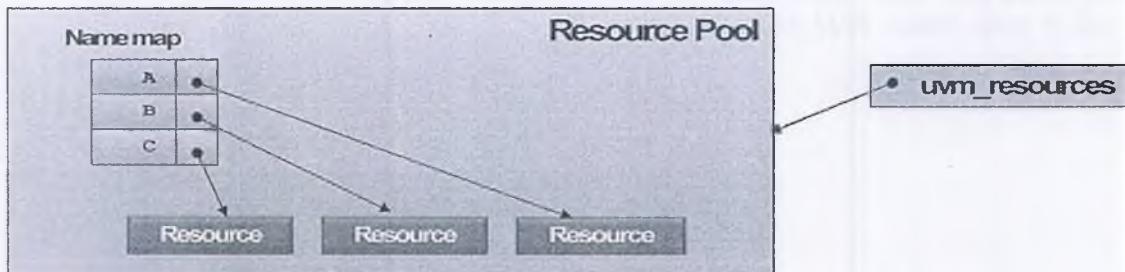
---

---

# Resources not found?

- Common reason is no entry with the name specified in the database
  - Could be due to a typographical error in the name or the scope
- The resource pool has a spell-checker function

```
function bit spell_check(string s)
 ◆ Compares argument s with entries in the name map
 – Returns 1 if a match is found, 0 if not
 ◆ Suggests similar entries to the miss-typed string
```



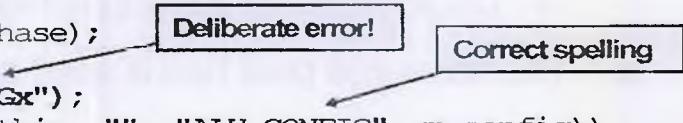
Notes:

# Example Spell Check

```
class monitor extends uvm_component;
...
function void build_phase(uvm_phase phase);
 // get config object
 uvm_resources.spell_check("ALU_CONFIGx");
 if(!uvm_config_db#(alu_config)::get(this, "", "ALU_CONFIG", m_config))
 `uvm_fatal("CONFIG", "Config error in get()")
 ...
endfunction
...
endclass

//-----
// Simulation output

#
ALU_CONFIGx not located
did you mean ALU_CONFIG?
```



Notes:

---

---

---

# Debugging Resources - 1

- Simplest debug tool of all: `dump()` the entire database

```
static function void dump(bit audit = 0)
 ◆ Dumps configuration database to stdout
 ◆ Audit bit == 1 dumps the history of accesses to all resources
```

```
class monitor extends uvm_component;
...
function void build_phase(uvm_phase phase);
 ...
 uvm_resources.dump();
endfunction
...
endclass

//-----
// Simulation output
____ resource pool ____
ALU_CONFIG [/^uvm_test_top\..*$/] : ? Posix regexp notation
-
SEQR_NAME [/^uvm_test_top\.t_env\..*$/] : ?
-
V_ALU_IF [/^uvm_test_top$/] : ?
-
____ end of resource pool ____
```



uvm\_intro\_3.5

© Willamette HDL Inc.

code in examples/alu

247

Notes:

# Debugging Resources - 2

- If the argument audit of dump() is set to a 1 then the accesses to the resources are included in the output of dump()

```
uvm_resources.dump(1);

//-----
// Simulation output
== resource pool ==
ALU_CONFIG [/^uvm_test_top\..*$/] : ?
#

uvm_test_top reads: 0 @ 0 writes: 1 @ 0
uvm_test_top.t_env.alu_agent.drv reads: 1 @ 0 writes: 0 @ 0
uvm_test_top.t_env.alu_agent.mon reads: 1 @ 0 writes: 0 @ 0
#
SEQR_NAME [/^uvm_test_top\.t_env\..*$/] : ?
#

uvm_test_top reads: 0 @ 0 writes: 1 @ 0
uvm_test_top.t_env.alu_agent reads: 1 @ 0 writes: 0 @ 0
#
V_ALU_IF [/^uvm_test_top$/] : ?
#

reads: 0 @ 0 writes: 1 @ 0
uvm_test_top reads: 1 @ 0 writes: 0 @ 0
== end of resource pool ==
```



Notes:

---

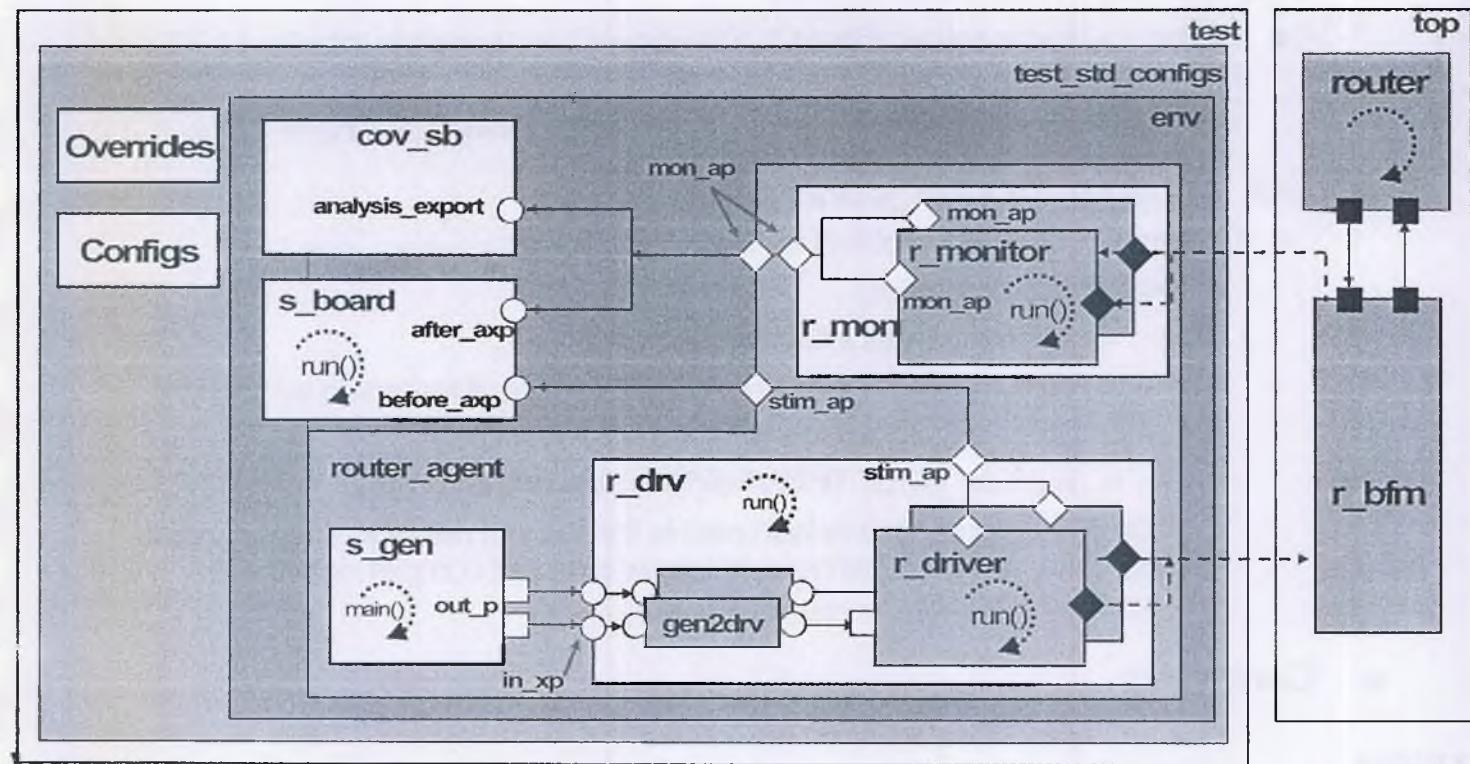
---

---

---

# Lab - Configurations: Overview

- Change to use configurations to configure topology, assign virtual interfaces & set unique property values



WHDL

uvm\_intro\_3.5

© Willamette HDL Inc.

249

Notes:

# Lab - Configuration Instructions – 1

- NOTE: For this lab the test used is `test_std_overrides`
- Working directory: **configuration**
- Edit file: **top\_modules/top.sv**
  - Notice the instantiation of the DUT (`router_rtl`) and the interface (`router_bfm`)
    - ◆ Create a virtual interface entry in the configuration database
- Edit file: **agent/router\_config.svh**
  - Create a configuration object for the router: class `router_config`
    - ◆ Add these properties
      - `v_r_bfm` (virtual interface property)
      - `m_num_routers` (int to hold the number of routers in testbench)
      - `m_router_id` (int to hold the id of the router)
      - `m_router_size` (int to hold the size of the router)
    - ◆ Add this property that is not used in the lab but rather in sequence labs later on - it is needed here, however to avoid compile errors
      - `uvm_sequencer #(Packet) rtr_seqr;`
- Continued...

Notes:

# Lab-Configuration Instructions – 2

## ■ Edit file: *tests/test\_base.svh*

- Declare a handle of type `router_config` named `m_r_config`
- In the `build_phase()` method:
  - ◆ Create `m_r_config`
  - ◆ Initialize `m_r_config`'s 4 properties
    - You will need to fetch the virtual interface entry you made in `top.sv`
    - See comments in code for values to use for the other properties
  - ◆ Save `m_r_config` to the configuration DB so it can be fetched by the testbench components that need it
    - Use "router\_config" as the `field_name` string

## ■ Edit file: *agent/router\_agent\_base.svh*

- Declare a handle of type `router_config` named `m_config`
- In the `build_phase()` method fetch a handle to the `router_config` object from config space and assign it to `m_config`

## ■ Continued...

Notes:

---

---

---

# Lab - Configuration: Instructions – 3

- Edit file: *agent/rtr\_driver.svh*
  - Declare a handle of type `router_config` named `m_config`
  - In the build phase
    - ◆ Fetch the `router_config` object from config DB
    - ◆ Set the virtual interface property `v_r_bfm` from the `router_config` object
- Compile & Run
  - No errors:              `make`
  - Errors:                `make bad`

Notes:

---

---

---

---

# Lab - Configuration : Sample Output

```
UVM_INFO @ 1533300: uvm_test_top.env cov_sb [COVERAGE]
UVM_INFO @ 1533300: uvm_test_top.env cov_sb [COVERAGE] Final Coverage = 100.000000%
UVM_INFO @ 1533300: uvm_test_top.env cov_sb [COVERAGE] 100% Coverage met with 11 transactions
UVM_INFO @ 1533300: uvm_test_top.env cov_sb [COVERAGE] ****
#
UVM_INFO @ 1533300: uvm_test_top.env.s_board [Packet_Comparator] Matches: 991
UVM_INFO @ 1533300: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
UVM_INFO @ 1533300: uvm_test_top.env.s_board [Packet_Comparator] Missing: 9
#
#
#— UVM Report Summary —
#
#** Report counts by severity
UVM_INFO: 9
UVM_WARNING: 0
UVM_ERROR: 0
UVM_FATAL: 0
#** Report counts by id
#[COVERAGE] 4
#[Packet_Comparator] 3
#[RNTST] 1
#[UVMTOP] 1
#** Note: $finish : ../../uvm/src/base/uvm_root.svh(430)
Time: 1533300 ns Iteration: 102 Instance: /test
```

Sol



uvm\_intro\_3.5

©Willamette HDL Inc.

253

Notes:

# uvm\_resource #( T )

## Reference Page

**UVM** class uvm\_resource #(type T = int) extends uvm\_resource\_base

### API Methods

Get a resource from the database:

```
function void write(T t, uvm_object accessor = null)
function T read(uvm_object accessor = null)
Add a new resource to the database
rsrc_t set_default(string scope, string name)
void set(string scope, string name, T val, uvm_object accessor = null)
void set_anonymous(string scope, T val, uvm_object accessor = null)
```

Locate a resource and read it's value

```
bit read_by_name(string scope, string name, ref T val, uvm_object accessor = null)
bit read_by_type(string scope, ref T val, uvm_object accessor = null)
```

Write a value to an existing resource. If resource does not exist, create and write it

```
bit write_by_name (string scope, string name, T val, uvm_object accessor = null)
bit write_by_type (string scope, T val, uvm_object accessor = null)
```

Debug. Dump all resources (the entire DB)

```
void dump()
```

Notes:

---

---

---

---

### UVM

```
class uvm_resource #(type T = int) extends uvm_resource_base
```

#### API Methods

```
function void write(T t, uvm_object accessor = null)
```

- Writes value `t` into the resource
- Issues an error message if the resource is read-only
- `accessor` is the calling object (typically `this`) and is used for auditing

```
function T read(uvm_object accessor = null)
```

- Returns the value of the resource
- `accessor` is the calling object (typically `this`) and is used for auditing

- Note: There are more methods but these are the ones we are interested in

Notes:

---

---

---

# uvm\_resource\_db #( T )

Reference Page

**UVM**

```
class uvm_resource_db #(type T = int)
 typedef uvm_resource #(T) rsrc_t;
```

Static functions:

Get a resource from the database:

```
rsrc_t get_by_type(string scope)
rsrc_t get_by_name(string scope, string name, bit rpterr=1)
```

Add a new resource to the database

```
rsrc_t set_default(string scope, string name)
void set(string scope, string name, T val, uvm_object accessor = null)
void set_anonymous(string scope, T val, uvm_object accessor = null)
```

Locate a resource and read it's value

```
bit read_by_name(string scope, string name, ref T val, uvm_object accessor = null)
bit read_by_type(string scope, ref T val, uvm_object accessor = null)
```

Write a value to an existing resource. If resource does not exist, create and write it

```
bit write_by_name (string scope, string name, T val, uvm_object accessor = null)
bit write_by_type (string scope, T val, uvm_object accessor = null)
```

Create a new resource and place it at the head of the queue

```
void set_override(input string scope, input string name, T val,
 uvm_object accessor = null)
void set_override_type(input string scope, input string name, T val,
 uvm_object accessor = null))
```

Debug. Dump all resources (the entire DB)

```
void dump()
```



uvm\_intro\_3.5

©Willamette HDL Inc.

256

Notes:

---

---

---

---

## **set() and set\_anonymous()**

**Reference Page**

**UVM**

```
static function void set(input string scope,
 input string name, T val,
 input uvm_object accessor = null)
```

- Add a new resource type T to the DB written with val
  - ◆ Regardless if another exists with exactly the same name
- Uses the name and scope as the lookup parameters
- accessor is the calling object (typically this) and is used for auditing

**UVM**

```
static function void set_anonymous(input string scope,
 T val,
 input uvm_object accessor = null)
```

- Add a new resource of type T to the DB written with val
  - ◆ Regardless if another exists with exactly the same type
- Uses scope as the lookup parameters
- accessor is the calling object (typically this) and is used for auditing



uvm\_intro\_3.5

©Willamette HDL Inc.

257

Notes:

---

---

---

---

# Example Setting Configuration data

Reference Page

```
module top;
 import uvm_pkg::*;
 import test_params_pkg::*;
 import tests_pkg::test_mac_simple_duplex;

 // WISHBONE interface instance
 // Supports up to 8 masters and up to 8 slaves
 wishbone_bus_syscon_if wb_bus_if();

 . . .

initial begin
 //set WISHBONE virtual interface to resource DB

 uvm_resource_db#(virtual wishbone_bus_syscon_if)::set("*", "WB_BUS_IF",
 wb_bus_if, null);

 run_test("test_mac_simple_duplex"); // create and start running test end
endmodule
```



uvm\_intro\_3.5

© Willamette HDL Inc.

code in examples/mac\_duplex

258

Notes:

# get\_by\_type() and get\_by\_name()

## Reference Page

### UVM

static function rsrc\_t **get\_by\_type**(string scope)

- **rsrc\_t** is a typedef for **uvm\_resource # (T)**
- Returns a handle to a resource
- Uses the **resource\_db** type parameter **T** and **scope** as the lookup parameters

### UVM

static function rsrc\_t **get\_by\_name**(input string scope,  
                                  input string name, bit rpterr = 1)

- **rsrc\_t** is a typedef for **uvm\_resource # (T)**
- Returns a handle to a resource
- Uses **name** and **scope** as the lookup parameters
- **rpterr** flag indicates whether a warning is reported if no matching resource is found

Notes:

---

---

---

# Example Reading Configuration Data: `get_by_xxx`

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

```
wb_config wb_config_0; // config object for WISHBONE BUS

function void set_wishbone_config_params();
  uvm_resource #(virtual wishbone_bus_syscon_if) tmp;
  ```
 wb_config_0 = new();
  ```
  tmp = uvm_resource_db#(virtual wishbone_bus_syscon_if)::get_by_name("*, "WB_BUS_IF", 1);
  wb_config_0.v_wb_bus_if = tmp.read(this);

```
  ```
  uvm_resource_db#(wb_config)::set("*, "wb_config", wb_config_0, this);
endfunction

endclass
```

Reference Page

`get_by_xxx()` returns the
resource, then `read()` extracts
the value from it



uvm_intro_3.5

©Willamette HDL Inc.

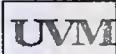
code in examples/mac_duplex

260

Notes:

read_by_name() and read_by_type()

Reference Page



```
static function bit read_by_name(input string scope,  
                      input string name, inout T val,  
                      input uvm_object accessor = null)
```

- Locates resource by name and type (T)
- Value is returned through the reference argument val
- Returns a 1 if successful, 0 if fail

Convenience
methods
(recommended)

```
if( uvm_resource_db #(int)::read_by_name("*", "MY_INT", local_int, this));
```



```
static function bit read_by_type (input string scope,  
                      ref T val, input uvm_object accessor = null)
```

- Locates resource by type (T)
- Value is returned through the reference argument val
- Returns a 1 if successful, 0 if fail

```
if( uvm_resource_db #(my_typ)::read_by_type("*", local_my_typ, this) );
```



uvm_intro_3.5

© Willamette HDL Inc.

261

Notes:

Example Reading Configuration Data: `read_by_xxx`

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

```
wb_config wb_config_0; // config object for WISHBONE BUS

function void set_wishbone_config_params();
```
wb_config_0 = new();

if(! (uvm_resource_db#(virtual wishbone_bus_if)::read_by_name("*", "WB_BUS_IF",
    wb_config_0.v_wb_bus_if, this)))
  `uvm_fatal("RSRCNF", "uvm_resource_db#(virt_if_container)::read_by_name() error")
uvm_resource_db#(wb_config)::set("*", "wb_config", wb_config_0, this);

endfunction

endclass // monitor
```

Reference Page

`read_by_xxx()` conveniently
combines both the
`get_by_xxx()` and the `read()`



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/mac_duplex

262

Notes:

+uvm_set_config_int=<comp>,<field>,<value>

- Works like set_config_int()
 - ◆ `b, 0b, `o, `d, `h, `x, 0x
 - (note: back-tick) as first characters of value specify base
 - ◆ Size specifiers not allowed
 - ◆ Note that set_config_int puts a uvm_bitstream_t type into the configuration database not an int!
 - Means you fetch with uvm_config_db #(uvm_bitstream_t)::get()

+uvm_set_config_string=<comp>,<field>,<value>

- Works like set_config_string()

Stimulus Generation: Sequences

In this section



Traditional Stimulus Generation
UVM Sequence Elements and APIs

Notes:

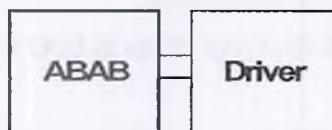
Traditional Stimulus Generation

- Currently, stimulus generation is done by an object that exists in the structural hierarchy
- It generates a sequence of transactions according to some algorithm, pushing them into a queue as they are generated
- Downstream, a driver pulls items out of the queue, then applies any low-level stimulus necessary
- Main problem with this mechanism illustrate its lack of flexibility:
 - The algorithm that generates a sequence of transactions over time is tied to the generator class
 - ◆ To change the algorithm
 - Need to replace the generator object in the hierarchy (e.g. with a polymorphic object)

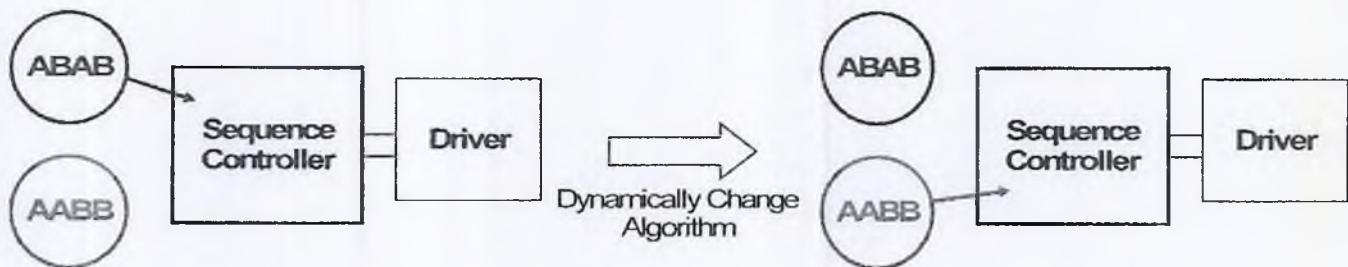
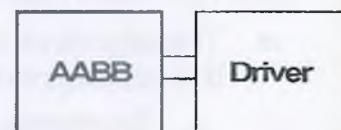
Notes:

Separate Behavior from Structure

- We would like to be able to change the entire transaction sequence generation algorithm dynamically (at run time)
- We want to do this without making changes to the structure
 - Would require changes to the environment
- Need to "de-couple" the algorithm from the structural container (component)

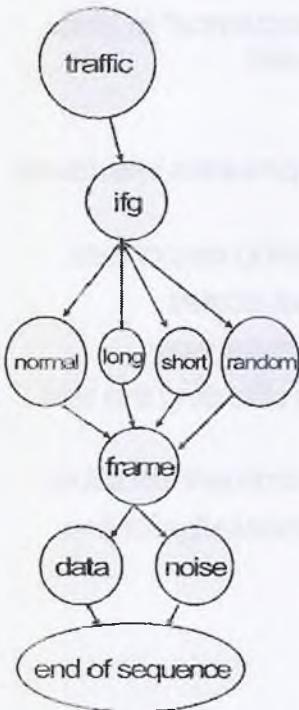


Physically replace stimulus generator



Notes:

Structured Traffic Patterns



- Many modern test environments require complex, structured traffic patterns driven over time
 - Some parts directed, some parts free to randomize within certain constraints
- For reuse, you need a way to build stimulus generation in a modular fashion
- A standard methodology promotes reuse amongst different groups, even external Verification IP vendors

Notes:

Sequence-based Stimulus Generation

- UVM provides several elements, which work together to accomplish dynamic, reusable transaction-based stimulus generation
 - Called "Sequences"
 - ◆ Naming can be confusing at first because the term "sequence" is also used to describe some of the components of sequences!
- Sequences
 - An approach using template (parameterized) classes that provides interfaces similar to the TLM library
 - Bidirectional: Can handle both sending requests and receiving responses
 - Promotes a standard stimulus and control methodology that scales
 - Provides a methodology to organize your stimulus in a modular way
 - ◆ Algorithms can easily be reused and modified *without* affecting the test environment
 - ◆ Stimulus is tied to tests and decoupled from test environment structure
 - ◆ Test writers don't necessarily have to have detailed knowledge of the test environment

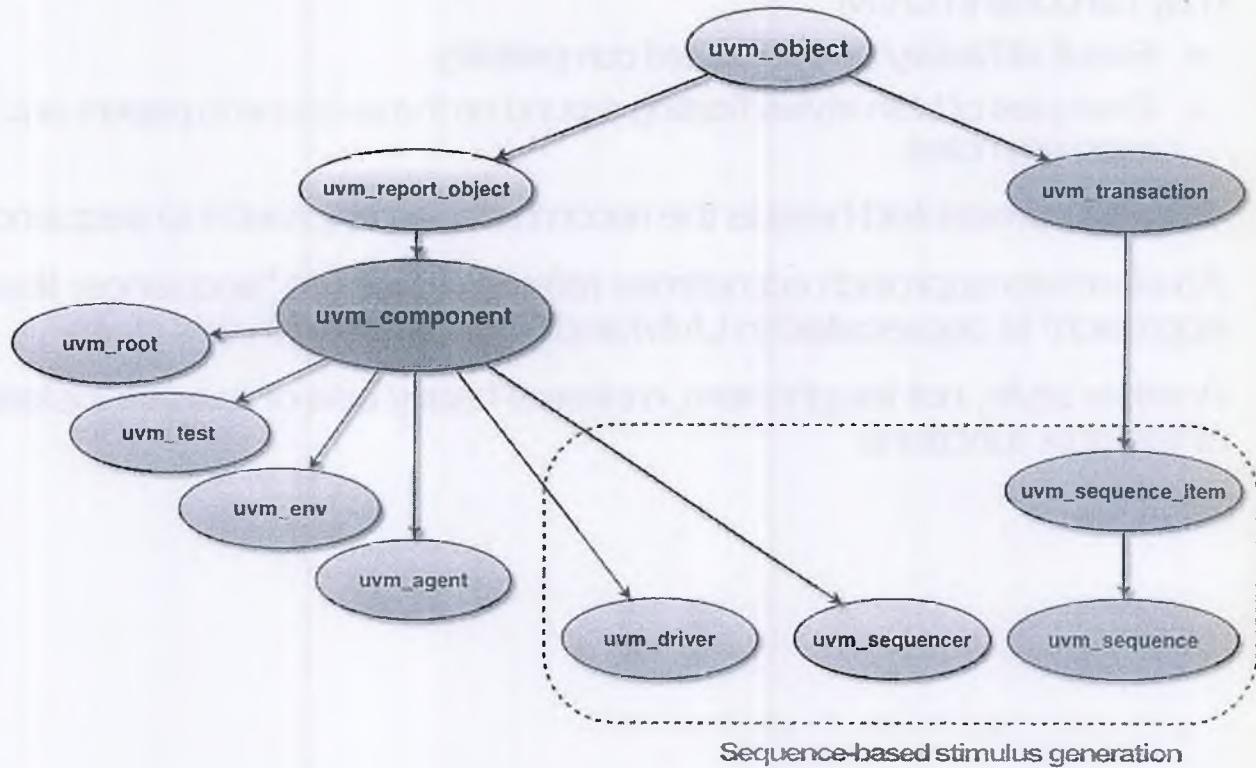
Notes:

Alternate Styles

- Unfortunately there are multiple “styles” of generating sequences that may be done in UVM
 - Result of history and backward compatibility
 - Examples of both styles floating around on the web and in papers and even promoted
- The style presented here is the recommended approach to sequences
- An alternate approach sometimes referred to as the “sequencer library approach” is deprecated in UVM and not discussed in this class
- Another style, not taught here, makes a heavy use of macros instead of tasks or functions

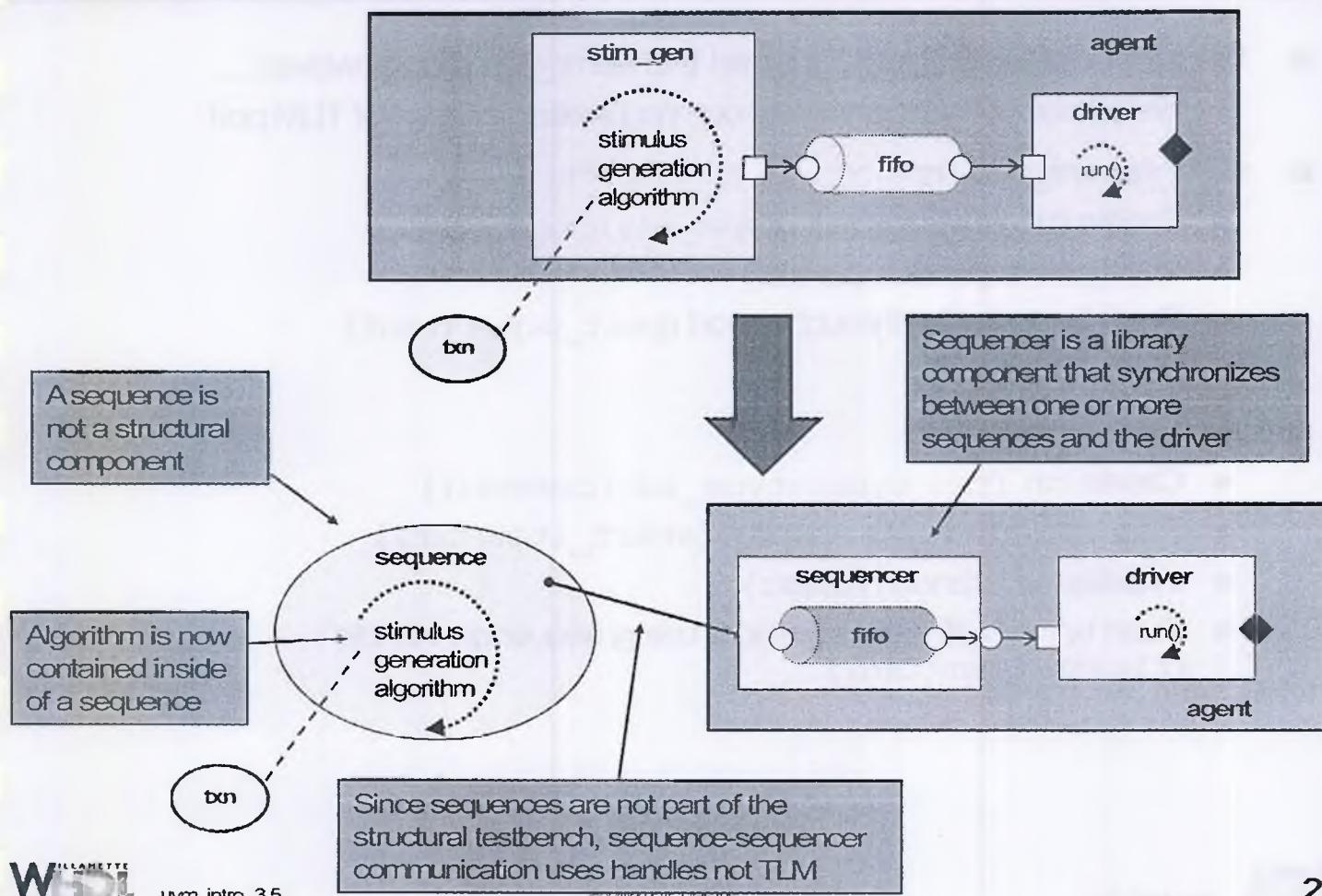
Notes:

UVM Base classes realm



Notes:

Standard to Sequence Stimulus Generation



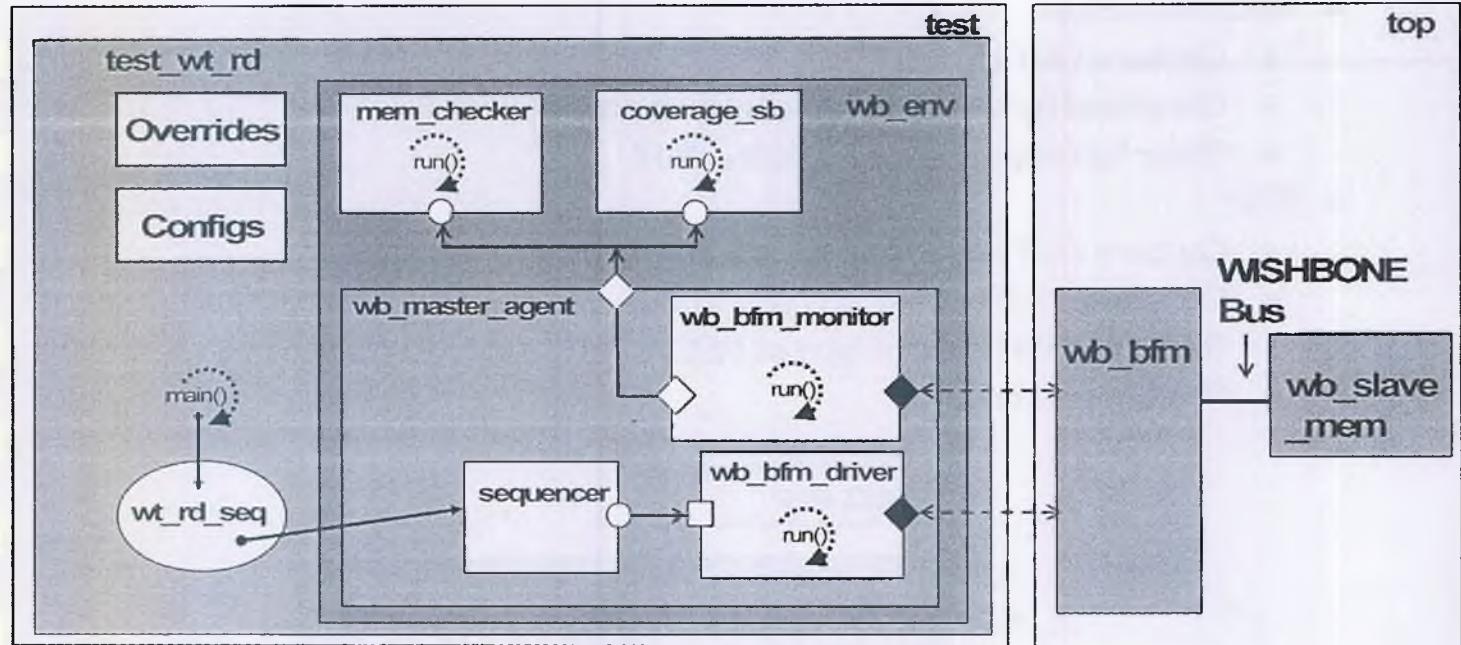
Notes:

Stimulus Generation Algorithm Changes

- Sequence does a "push" just as the `stim_gen` did, however...
 - Sequence uses a handle for communication instead of TLM port
- `stim_gen` algorithm:
 - Create txm (`txn_type::type_id::create()`)
 - Initialize txm (randomize etc.)
 - Push txm to buffer through a port (`port_p.put(txn)`)
- Sequence algorithm
 - Create txm (`txn_type::type_id::create()`)
 - Request sequencer's attention (`start_item(txn)`)
 - Initialize txm (randomize etc.)
 - Push txm to buffer in sequencer using sequencer handle (`finish_item(txn)`)

Notes:

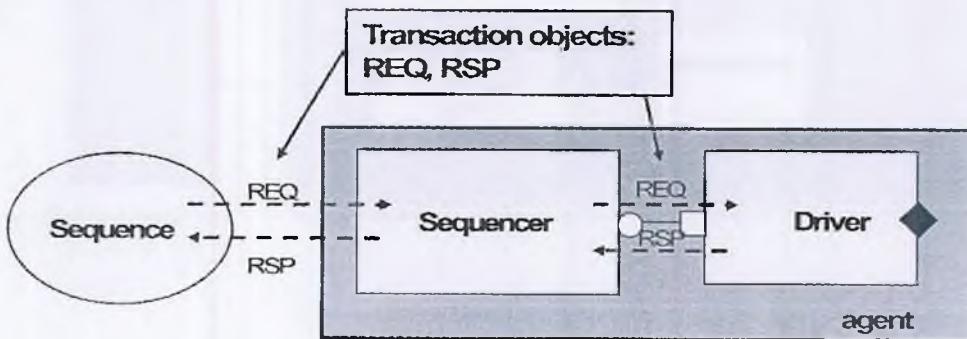
Example Circuit – Wishbone Bus Memory



Notes:

Transaction Objects

- Transaction objects: REQ, RSP
 - Sequence items
 - REQ transaction
 - ◆ Contains DUT stimulus
 - ◆ Generated by sequence and sent to the driver
 - ◆ Driver typically applies REQ to the DUT
 - RSP
 - ◆ Contains DUT response information
 - ◆ Generated by the driver and sent to the sequence
 - ◆ By default RSP is same type as REQ
 - ◆ Rarely if ever used



Notes:

Example Sequence Item—wb_txn

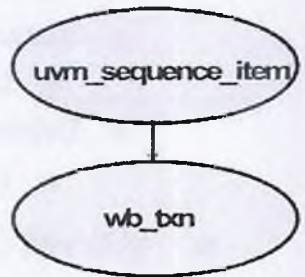
```
class wb_txn extends uvm_sequence_item;
`uvm_object_utils(wb_txn)

  wb_txn_t          txn_type;
  uvm_status_e      t_status;
  rand bit[31:0]    adr;
  rand logic[31:0]  data[];
  rand int unsigned count;
  bit[7:0]          byte_sel;

  function new(string name = "wb_txn");
    super.new(name);
    byte_sel = 4'b1111; // only valid value for byte_sel
  endfunction

  function void init_txn( wb_txn_t op = NONE, bit[31:0] l_addr = 0,
                        logic[31:0] l_data[], cnt = 1);
    txn_type = op;
    adr = l_addr;
    data = l_data;
    count = cnt;
  endfunction

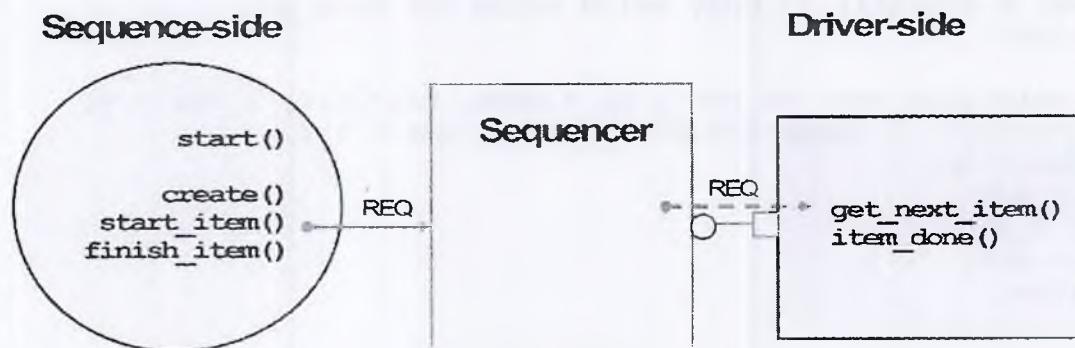
  ...
endclass
```



Notes:

API for Sequences

- The API is divided into two parts - one for sequences, and one for drivers
 - Sequence-side API
 - ◆ Sends transactions (REQ)
 - Driver-side API
 - ◆ Gets transactions (REQ)
- We will talk about responses from driver to sequence later



Notes:

Sequences

UVM

```
uvm_sequence #(type REQ=uvm_sequence_item, RSP=REQ)
```

- You create a sequence by inheriting from the base class `uvm_sequence`
 - Key properties and methods that are inherited:
 - ◆ `m_sequencer`
 - A handle that points to the sequencer the sequence will communicate with
 - ◆ `start()`
 - Method that starts the sequence execution
 - ◆ `body()`
 - Method that contains the stimulus algorithm
 - ◆ Methods for sequencer communication:
 - `start_item()`
 - `finish_item()`



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/wb_mem

277

Notes:

Example Sequence - wt_rd_seq

```
class wt_rd_seq extends uvm_sequence #(wb_txn);
`uvm_object_utils(wt_rd_seq)
int num_ops = 400;

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

task body();
    ...
endtask
endclass
```

Specifies the REQ type &
by default the RSP type

uvm_sequence

wt_rd_seq

Register with factory

Constructor

body() task contains the
stimulus generation
algorithm

Notes:

Starting a Sequence – Call start()

UVM

```
virtual task start (uvm_sequencer_base sequencer,  
                    uvm_sequence_base parent_sequence = null,  
                    integer this_priority = 100,  
                    bit call_pre_post = 1);
```

Normally leave these
arguments at their default
values

- Starts execution of the sequence

- Think of it as the "start me" method of a sequence
- Initializes the sequence
 - ◆ `m_sequencer` property set to `sequencer` argument
 - This will be the sequencer the sequence will communicate with
 - ◆ `m_parent_sequence` property set to `parent_sequence` argument
 - More when we talk about multiple sequences in the next section
 - ◆ `m_priority` property set to `this_priority` argument
 - More on priority later
 - ◆ `sequence_id` property set to -1



uvm_intro_3.5

© Wilamette HDL Inc.

279

Notes:

Example Start of a Sequence: test_wt_rd

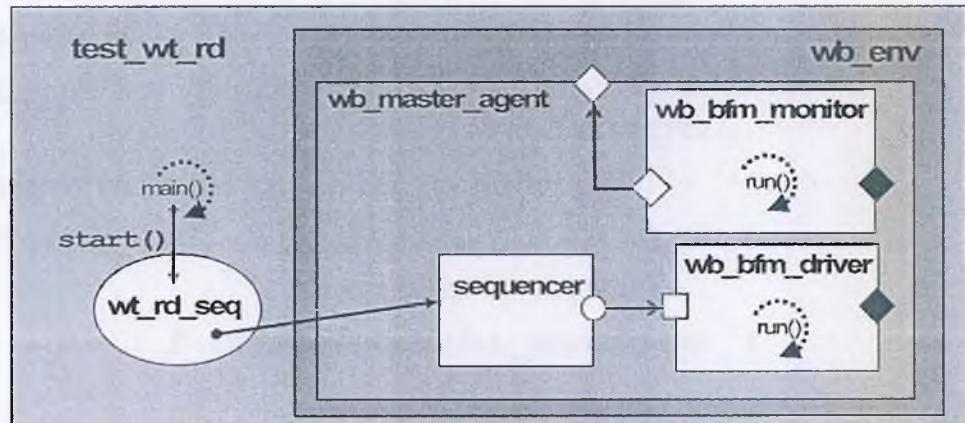
```
class test_wt_rd extends wb_mem_test_base;  
`uvm_component_utils(test_wt_rd)  
  
...  
  
task main_phase(uvm_phase phase);  
  ...  
  // create sequence  
  wt_rd_seq m_seq = wt_rd_seq::type_id::create("m_seq");  
  phase.raise_objection(this, "Start stimulus generation");  
  // start sequenced on sequencer in wb_master_agent  
  m_seq.start(wb_seqr, null);  
  phase.drop_objection(this, "End stimulus generation");  
endtask
```

```
endclass
```

More later on getting the sequencer handle

Create sequence

start sequence



Notes:

What `start()` Does

■ `start()` method is a template method pattern

- Does stuff and calls virtual methods in a fixed order
 - ◆ Pattern of what is done may not be changed (`start()` is non-virtual)
 - However behavior may be modified by overriding virtual methods
 - » Sometimes called "hook" methods
- Pattern:
 - ◆ Initialization of sequence properties using arguments
 - ◆ Calls methods in this order:
 - `virtual task pre_body()`
 - » For setup, optional, does nothing by default
 - `virtual task body()`
 - » Main body of sequence, does nothing by default
 - » Create, randomize, send sequence items here
 - `virtual task post_body()`
 - » For cleanup, optional, does nothing by default
 - ◆ Cleanup of sequence properties
 - ◆ Return



Tip: Override or use `only body()` method. Since `pre-body()` and `post-body()` are optional they may not be called

Notes:

Generating Transaction Items

- Create sequence items by using either
 - The factory with the `create()` method
 - ◆ Advantage can use factory override
 - `new()`
 - ◆ Advantage is performance - creation may be done many, many times

```
class wt_rd_seq extends uvm_sequence #(wb_txn);  
...  
task body();  
    wb_txn wt_txm; // sequence item handle  
    ...  
    // generate stimulus  
    repeat (num_ops) begin  
        wt_txm = wb_txm::type_id::create("wt_txm");  
        ...  
    endtask  
endclass
```

sequence item handle

REQ and RSP types

create sequence item

Notes:

Generating Transaction Items – `start_item()`

UVM virtual task `start_item` (`uvm_sequence_item item,`
`int set_priority = -1);`

- Blocking request to sequencer to provide a sequence item
 - "Pick me! Pick me!"
 - ◆ Sequence is blocked until "picked" by the sequencer

```
class wt_rd_seq extends uvm_sequence #(wb_txn);
  ...
  task body();
    wb_txn wt_txn;
    // generate stimulus
    repeat(num_ops) begin
      wt_txn = wb_txn::type_id::create("wt_txn");
      start_item(wt_txn);
    ...
  endtask
endclass
```

"Pick me! Pick me!"

Notes:

Generating Transaction Items – `finish_item()`

UVM virtual task `finish_item(uvm_sequence_item item,`
 `int set_priority = -1);`

- Blocking send of sequence item to driver through sequencer
 - Sends the REQ item to the sequencer (stored in a fifo)
 - Blocks until driver indicates it is done with the REQ item
- Any initialization of the sequence item should be done after `start_item()` and before `finish_item()`

```
class wt_rd_seq extends uvm_sequence #(wb_txn);  
...  
task body();  
    wb_txn wt_txn;  
    repeat (num_ops) begin  
        wt_txn = wb_txn::type_id::create("wt_txn");  
        start_item(wt_txn);  
        wt_txn.txn_type = WRITE;  
        if(!wt_txn.randomize() with {adr < m_config.m_mem_slave_size[0];  
                                     count == 1; data.size() == count; })  
            `uvm_fatal("SEQ", "Randomization error")  
  
        finish_item(wt_txn);  
    ...  
endtask  
endclass  
uvm_intro_3.5
```

Initialize sequence item

Send to sequencer

© Willamette HDL Inc.

code in examples/wb_mem

284

Notes:

Full Sequence Example: wt_rd_seq

```
class wt_rd_seq extends uvm_sequence #(wb_txn);
  `uvm_object_utils(wt_rd_seq)

  int num_ops = 400;
  function new(string name = "main_seq");
    super.new(name);
  endfunction

  task body();
    wb_config m_config; // Config object
    wb_txn wt_txn;
    wb_txn rd_txn;

    // get config object
    if(!uvm_config_db #(wb_config)::get(m_sequencer, "", "wb_config", m_config))
      `uvm_error("CONFIG", "wb_config not found");
    // generate stimulus
    repeat(num_ops) begin
      wt_txn = wb_txn::type_id::create("wt_txn");
      start_item(wt_txn);
      wt_txn.txn_type = WRITE;
      if(!wt_txn.randomize() with {adr < m_config.m_mem.slave_size[0];
                                  count == 1; data.size() == count; })
        `uvm_fatal("SEQ", "Randomization error")
      finish_item(wt_txn);

      rd_txn = wb_txn::type_id::create("rd_txn");
      start_item(rd_txn);
      rd_txn.init_txn(READ, wt_txn.adr, wt_txn.data);
      finish_item(rd_txn);
      // do nothing with the read data
    end
  endtask
endclass
```



uvm_intro_35

© Willamette HDL Inc.

code in examples/wb_mem

285

Notes:

Sequencecs

UVM

```
uvm_sequencer #(type REQ=uvm_sequence_item, RSP=REQ)
```

- A sequencer is UVM library component

- Instantiate it directly in your testbench
 - ◆ No need to inherit and create your own
- Parameterized
 - ◆ Request item type (REQ)
 - Transaction type sent from a sequence to the driver
 - ◆ Response item type (RSP)
 - Transaction type sent from the driver to the sequence
 - We will never use so ignore and leave as default (RSP=REQ)

```
class wb_master_agent extends wb_agent_base;
  `uvm_component_utils(wb_master_agent)
  uvm_sequencer #(wb_txn) wb_seqr;

  function void build_phase(uvm_phase phase);
    ...
    wb_seqr = new("wb_seqr", this);
  endfunction
```

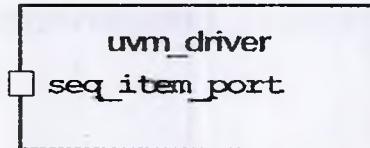
Notes:

Drivers

UVM

```
class uvm_driver #(type REQ=uvm_sequence_item,  
                  type RSP=REQ) extends uvm_component;
```

- The driver transactors are derived from the `uvm_driver` class
 - A number of properties are inherited from this class but of significance is one particular port
 - ◆ `seq_item_port`
 - Port through which the driver communicates with the sequencer



**WILLAMETTE
HDL**

uvm_intro_3.5

©Willamette HDL Inc.

287

Notes:

UVM Driver seq_item_port Methods

Methods:

```
task get_next_item(output REQ req_arg)
```

- ◆ Gets REQ item from sequencer fifo

uvm_driver
seq_item_port

```
function void item_done(input RSP rsp_arg = null)
```

- ◆ Notifies sequencer that driver is done with REQ
 - Unblock the finish_item() call in the providing sequence
- ◆ Note: you must call item_done() between successive calls to get_next_item()



Recommended: Use these methods

Notes:

Other UVM Driver seq_item_port Methods

■ TLM Methods:

```
task get (output REQ req_arg)
```

- ◆ Removes REQ item from fifo
- ◆ Notifies sequencer that driver is done with REQ
 - Unblock the `finish_item()` call in the providing sequence

```
task put (input RSP rsp_arg)
```

- ◆ Provides RSP back to the requesting sequence

```
uvm_driver  
seq_item_port
```



Perfectly OK to use these methods as they are familiar

Sometimes in more advanced settings these are best or even necessary
(whenever multiple REQ outstanding)

Notes:

Driver Example: wb_bfm_driver

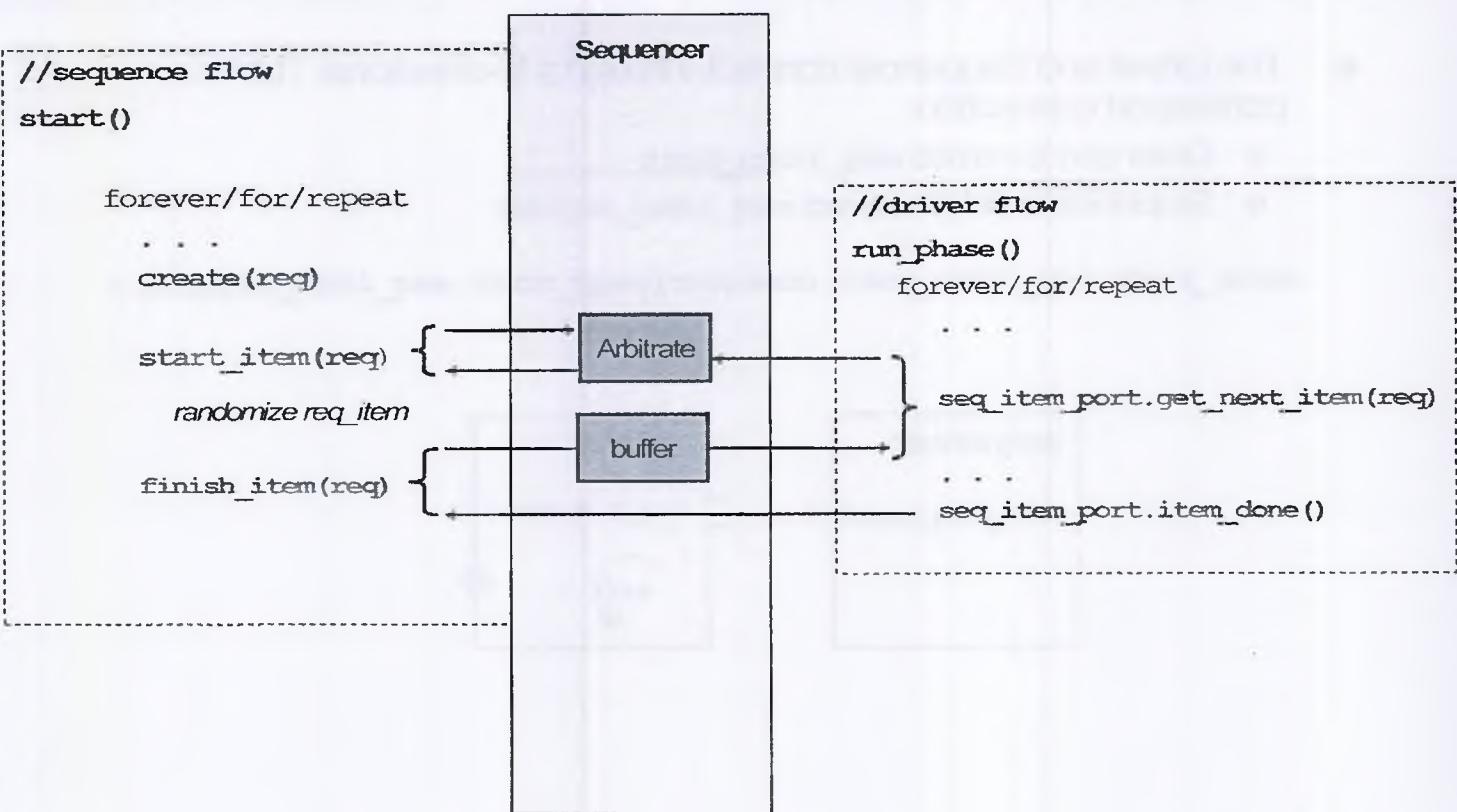
```
class wb_bfm_driver extends wb_driver_base;
`uvm_component_utils(wb_bfm_driver)
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction
task run_phase(uvm_phase phase);
    wb_txn txn;
    forever begin
        seq_item_port.get_next_item(txn);
        case (txn.txn_type)
            WRITE:
            begin
                v_wb_bfm.wb_write_cycle(txn.data, txn.adr, txn.count,
                    .wb_master_id(m_config.m_wb_master_id), .op_stat(txn.t_status));
                seq_item_port.item_done(); // signal done with txn
            end
            READ :
            begin
                v_wb_bfm.wb_read_cycle(txn.data, txn.adr, txn.count,
                    .wb_master_id(m_config.m_wb_master_id), .op_stat(txn.t_status));
                seq_item_port.item_done(); // signal done with txn
            end
            ...
        endcase
    end
endtask
endclass
```



Request transaction item from a sequence

Notes:

Sequence ↔ Driver Synchronization

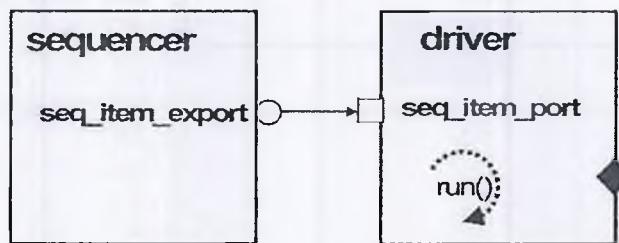


Notes:

Connecting the Driver and Sequencer

- The Driver and Sequencer connect through a bi-directional TLM port/export connection
 - Driver port is named `seq_item_port`
 - Sequencer export is named `seq_item_export`

```
drvrv_inst.seq_item_port.connect(seqr_inst.seq_item_export);
```

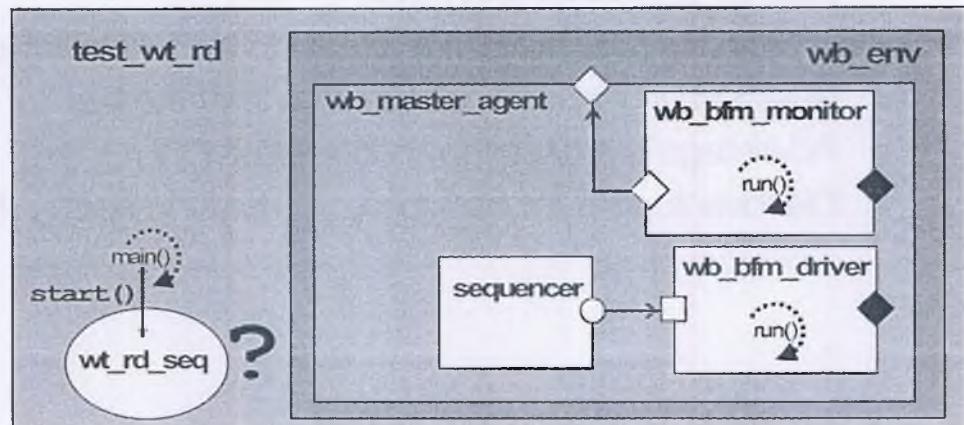


Notes:

Connecting a Sequence to a Sequencer

- Sequences are typically associated with sequencers by an argument to the sequence start () task

```
sequence_handle.start(sequencer_handle);
```
- How does the test know where the sequencer is?
 - Sequencer is "somewhere" in the environment but where?



Notes:

Sequence – Sequencer Connection

- There are multiple ways
 - Each with advantages & disadvantages
- Several options:
 - Simple hierarchical reference
 - ◆ Disadvantage is hard coded path
 - ◆ Advantage is it is simple
 - Agent provides the sequencer handle to the test using config object
 - ◆ Recommended way – shown on next couple slides
 - ◆ Best for scalability and reuse
 - Agent provides the sequencer handle to the test using configuration DB
 - ◆ Illustrated in the reference section after the lab
 - ◆ Advantage is it uses the configuration DB
 - ◆ Disadvantage is for scalability you have to uniquely identify the lookup name

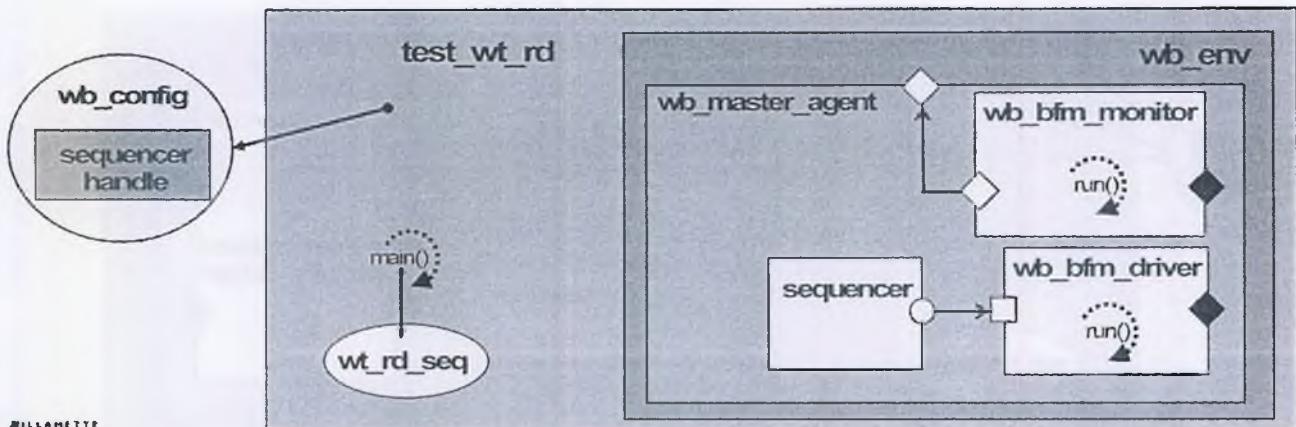
Notes:

Test wb_mem_test_base

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

  function void build_phase(uvm_phase phase);
    ...
    uvm_config_db #(wb_config)::set(this, "env*", "wb_config",
      wb_config_0);
    ...
  endfunction
  ...
endclass
```

Test class places
wb_config object into
config DB

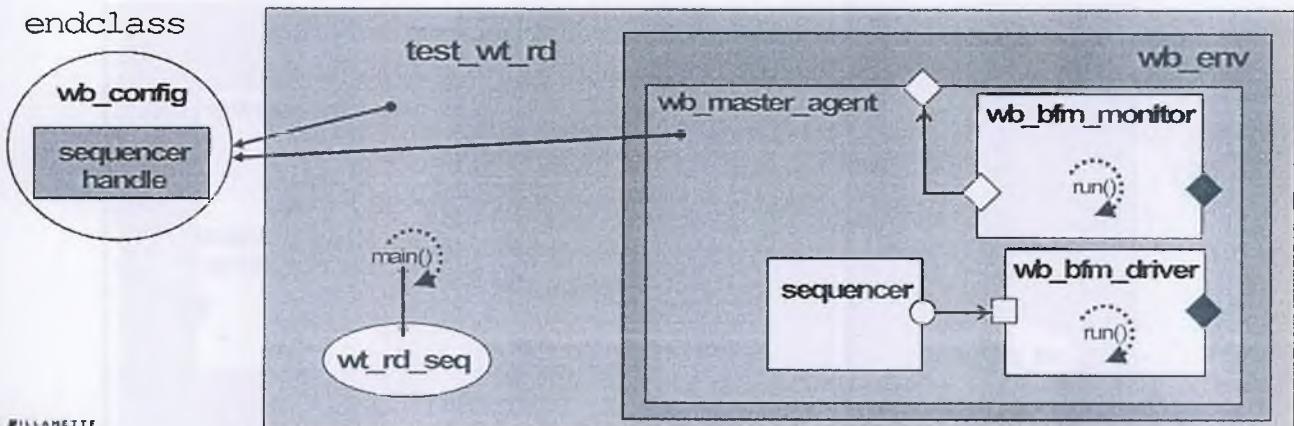


Notes:

Agent wb_agent_base

```
class wb_agent_base extends uvm_agent;
`uvm_component_utils(wb_agent_base)
...
wb_config m_config; // Config object
...
function void build_phase(uvm_phase phase);
super.build_phase(phase);
if(!uvm_config_db #(wb_config)::get(
    this, "", "wb_config", m_config))
    `uvm_fatal("CONFIG", "config error: wb_config not found");
...
endfunction
endclass
```

Get the wishbone configuration object



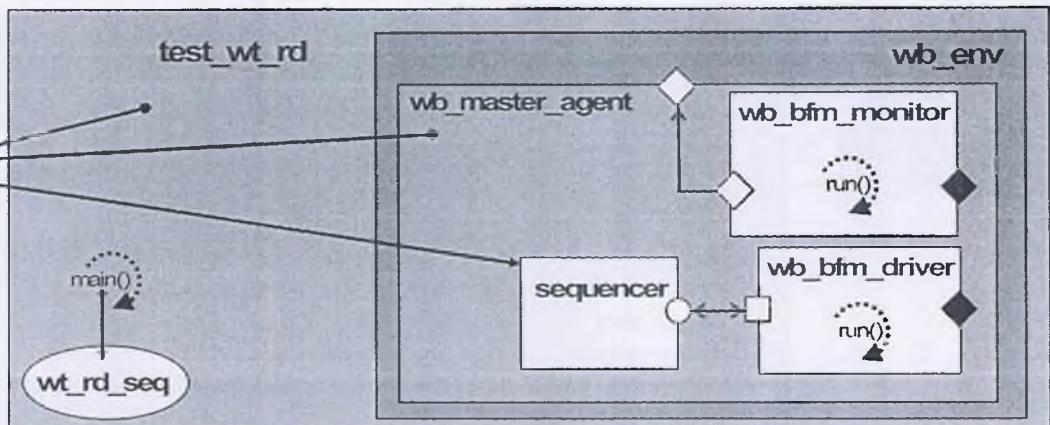
Notes:

Agent: wb_master_agent

```
class wb_master_agent extends wb_agent_base;  
...  
  uvm_sequencer #(wb_txn) wb_seqr;  
...  
function void build_phase(uvm_phase phase);  
...  
  wb_seqr = new("wb_seqr", this);  
  // set the wishbone sequencer for this agent using config object  
  m_config.wb_seqr = wb_seqr;  
endfunction  
endclass
```

Create sequencer

Assign local sequencer to sequencer handle in wb_config object

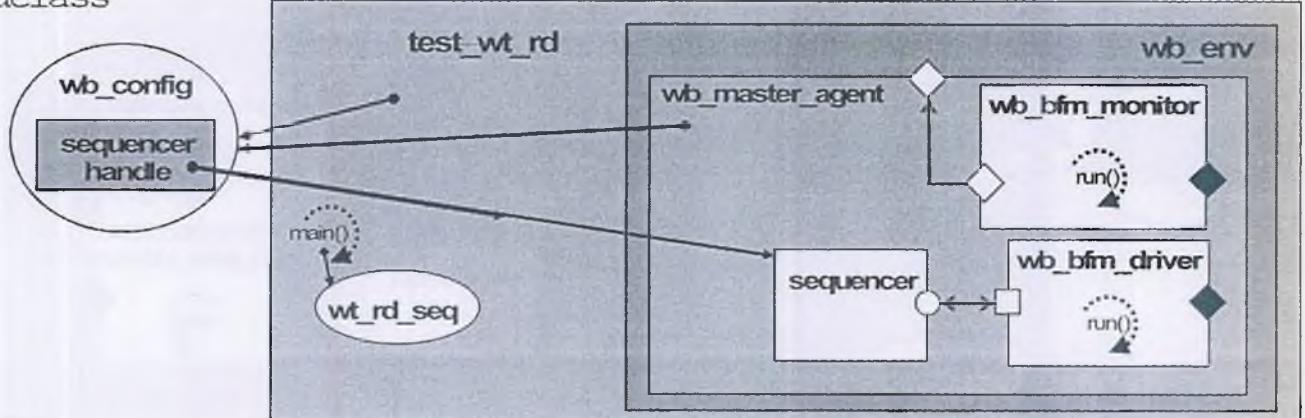


Notes:

Test: test_wt_rd

```
class test_wt_rd extends wb_mem_test_base;
`uvm_component_utils(test_wt_rd)
...
task main_phase(uvm_phase phase);
  // create main sequence
  wt_rd_seq m_seq = wt_rd_seq::type_id::create("m_seq");
  phase.raise_objection(this, "Start stimulus generation");
  // start sequenced on sequencer in wb_master_agent
  m_seq.start(wb_config_0.wb_seqr);
  phase.drop_objection(this, "End stimulus generation");
endtask
...
endclass
```

Start sequence with
sequencer handle
from wb_config
object



Notes:

Example Sequencer, Driver Connection

```
class wb_master_agent extends wb_agent_base;
  `uvm_component_utils(wb_master_agent)

  wb_bfm_driver wb_drv;
  uvm_sequencer #(wb_txn) wb_seqr;
  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // create components
    wb_drv = wb_bfm_driver::type_id::create("wb_drv", this);
    wb_seqr = new("wb_seqr", this);
    // set the wishbone sequencer for this agent using config object
    m_config.wb_seqr = wb_seqr;
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // connect components
    wb_drv.seq_item_port.connect(wb_seqr.seq_item_export);
  endfunction
endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

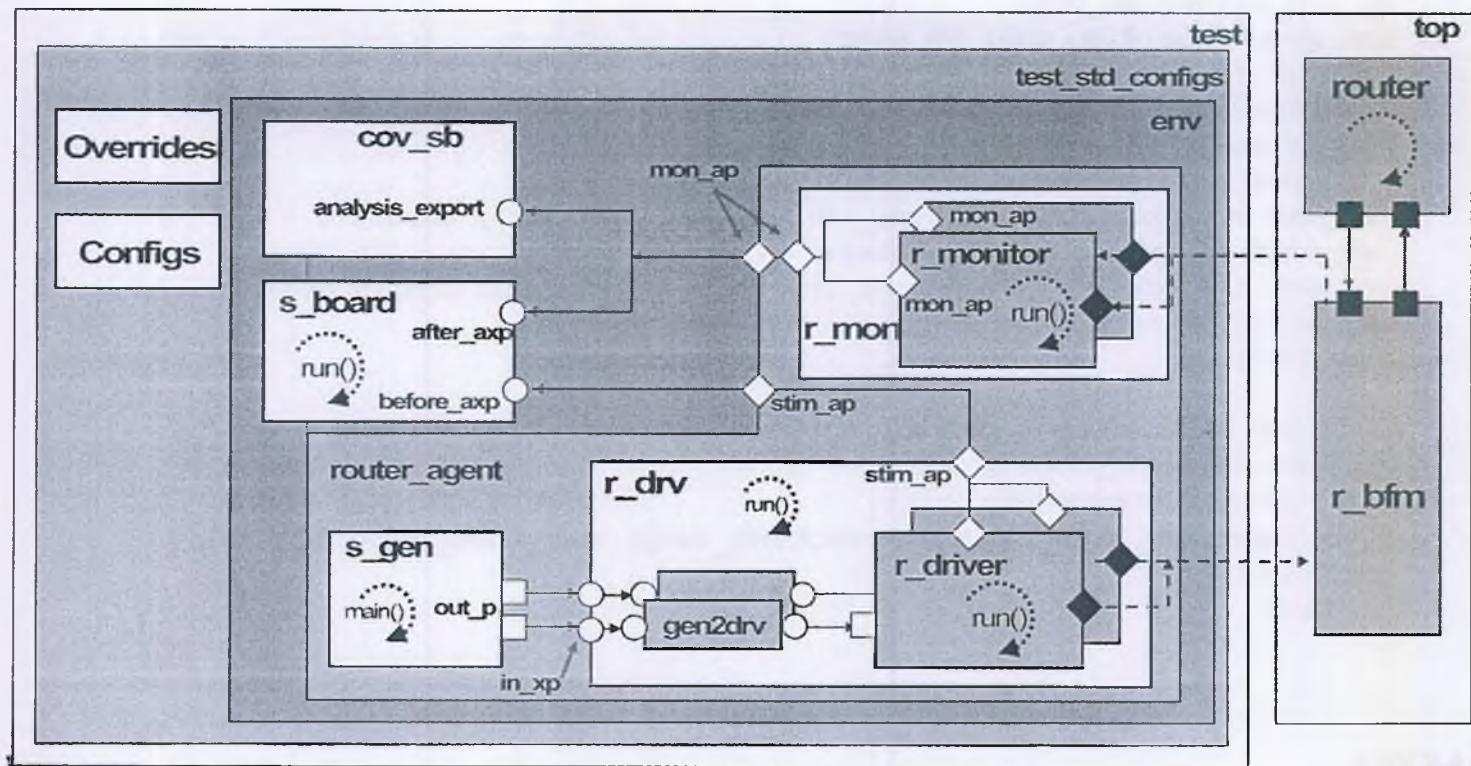
code in examples/wb_mem

299

Notes:

Lab – Basic Sequence: Overview - 1

- Testbench architecture with agent: `router_std_agent`



WHDL

uvm_intro_3.5

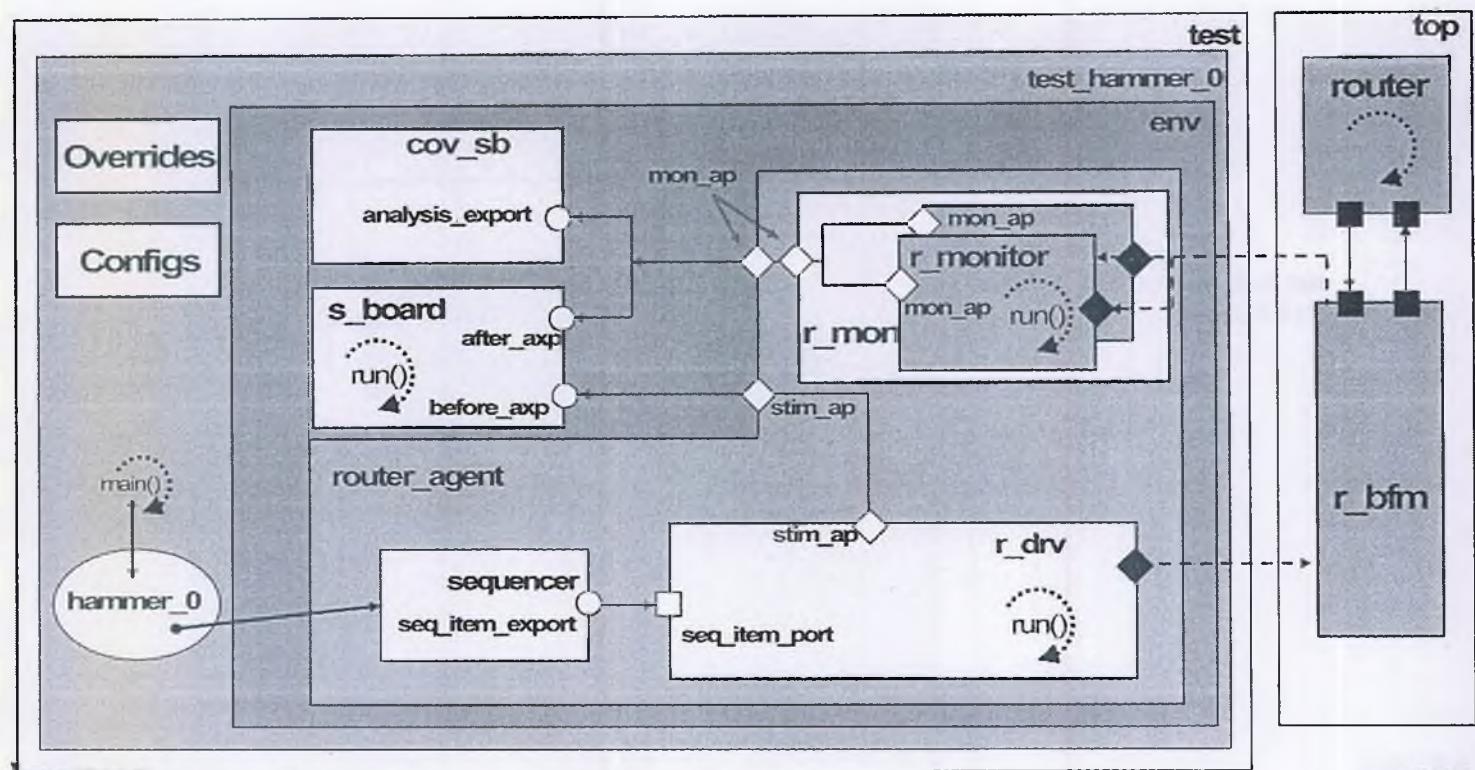
© Willamette HDL Inc.

300

Notes:

Lab – Basic Sequence: Overview - 2

- Replace the `router_std_agent` with `router_seq_agent`
- Create a sequence that "hammers" port 0 of the router



WHDL

uvm_intro_3.5

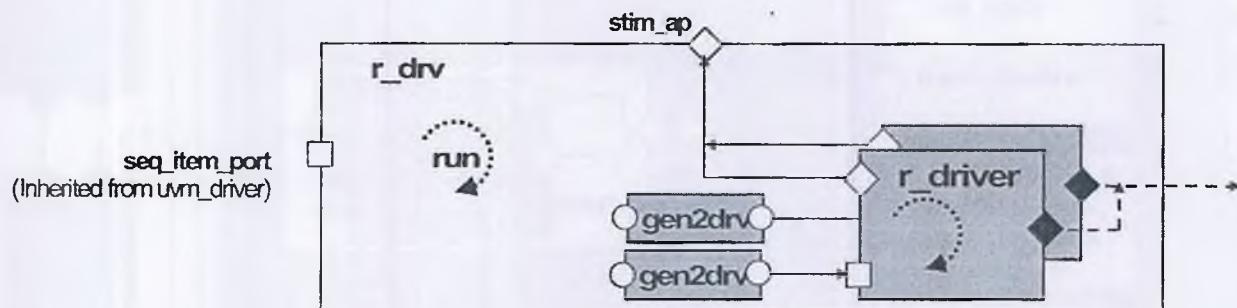
© Willamette HDL Inc.

301

Notes:

Lab – Basic Sequence Overview - 3

- The `r_drv` (type `seq_drivers`) block diagram



Notes:

lab-Sequences: Instructions – 1

- NOTE: For this lab the test used is `test_hammer_0`
- Lab directory: sequences
- Instructions:
 - Edit the file `sequences/hammer_0.svh`
 - ◆ Create a sequence class `hammer_0` parameterized for Packet
 - Write a body task:
 - » Loops 100 times
 - » Create and start a Packet
 - » Randomize with the `dest_id` field set to 0
 - » Set the `pkt_id` field of the Packet to a unique value
 - » Send the Packet to the sequencer
 - Edit the file `agent/router_seq_agent.svh`
 - ◆ In the class `router_seq_agent`
 - Declare a sequencer with the name `router_seqr` of type Packet
 - Declare a driver of type `seq_drivers` named `r_drv`
 - ◆ Continued...

Notes:

Lab—Sequences: Instructions—2

- Still editing the file `agent/router_seq_agent.svh`
 - In the `build_phase()` method
 - Create the sequencer and driver
 - Copy the sequencer handle into `router_config` object's (`m_config`) `rtr_seqr` property (see `router_agent_base`)
 - In the `connect_phase()` method connect the sequencer to the driver
- Edit the file `tests/test_hammer_0.svh`
 - ◆ In `main_phase()`
 - Declare and create a `hammer_0` sequence
 - Raise an objection
 - Start the `hammer_0` sequence on the sequencer in the router agent
 - Use the router configuration object inherited from the base test class
 - Note: you just got done writing the code to set the sequencer handle in the router configuration object in the agent
 - Drop an objection after the sequence has completed
- Compile and run

Notes:

Lab – Sequences: Sample Output

```
# UVM_INFO @ 588700: uvm_test_top.env.cov_sb [COVERAGE] ****
# UVM_INFO @ 588700: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage = 41.666667%
# UVM_INFO @ 588700: uvm_test_top.env.cov_sb [COVERAGE] ****
#
# UVM_INFO @ 588700: uvm_test_top.env.s_board [Packet_Comparator] Matches: 93
# UVM_INFO @ 588700: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0
# UVM_INFO @ 588700: uvm_test_top.env.s_board [Packet_Comparator] Missing: 1
#
#
# ---- UVM Report Summary ----
#
#
# ** Report counts by severity
# UVM_INFO : 8
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [COVERAGE] 3
# [Packet_Comparator] 3
# [RNTST] 1
# [UVMTOP] 1
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 588700 ns Iteration: 115 Instance: /test
```

Sol



uvrn_intro_3.5

©Willamette HDL Inc.

305

Notes:

Alternate Set Sequencer: wb_master_agent

Reference Page

```
class wb_master_agent extends wb_agent_base;  
  ...  
  uvm_sequencer #(wb_txn) wb_seqr;  
  ...  
  function void build_phase(uvm_phase phase);  
    ...  
    wb_seqr = new("wb_seqr", this);  
    uvm_config_db #(uvm_sequencer #(wb_txn))::set(uvm_top, "*",  
                                                 "wb_seqr", wb_seqr);  
  endfunction  
endclass
```

place sequencer handle
in configuration DB



If there is more than one instance of the agent then you have to uniquely
the lookup name

Notes:

Alternate Set Sequencer: test_wt_rd Reference Page

```
class test_wt_wd extends wb_mem_test_base;  
...  
task main_phase(uvm_phase phase);  
    // handle to wb sequencer in wishbone agent  
    uvm_sequencer #(wb_txn) wb_seqr;  
    // create main sequence  
    wt_rd_seq m_seq = wt_rd_seq::type_id::create("m_seq");  
    if(!uvm_config_db #(uvm_sequencer #(wb_txn))::get(this, "",  
        "wb_seqr", wb_seqr))  
        `uvm_fatal("CONFIG", "config error: wb_seqr not found");  
    m_seq.start(wb_seqr, null);  
    phase.raise_objection(this, "Start stimulus generation");  
endtask  
...  
endclass
```

Fetch sequencer handle from config DB



uvm_intro_3.5

©Wilamette HDL Inc.

307

Notes:

More on Sequences

In this section



- Modular Sequences
- Virtual Sequences
- Sequence Priority and Selection
- Handling Responses

Notes:

Modularity – Building Complex Tests

- More complex tests often require that multiple different stimulus be generated in serial or in parallel and often a mixture of both
- Sequences support an approach of building complex tests from smaller, reusable test "building block" sequences
- Hierarchy
 - A sequence may call one or more child sequences, each in turn may call child sequences of its own and so forth
- Sequential
 - Sequences may be called serially by a parent sequence or the parent test bench component
 - ◆ Allows for behavior such as set-up before the application of the stimulus to the DUT and status collection after
- Parallel
 - Sequences may be called in parallel by a parent sequence or the parent test bench component
 - ◆ Allows for mixing different streams of stimulus

Notes:

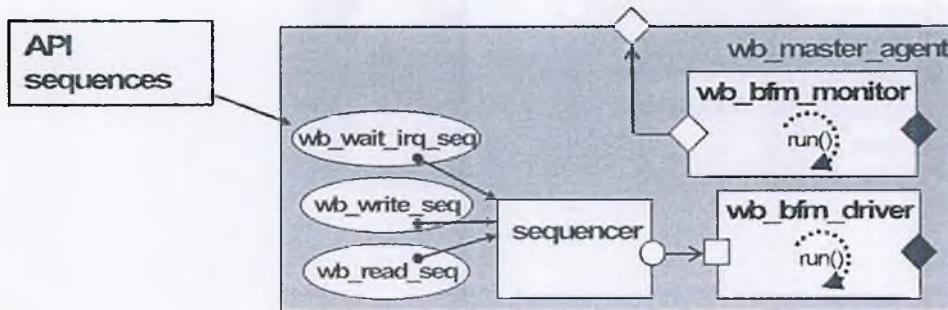
Modularity – Benefits

- Each sequence is easier to write/debug
- Specific functions can be encapsulated within a sequence
 - Provides for greater reuse
- More fine grained control is provided for controlling the mix of stimulus to the DUT
- "Sub-sequences" can essentially be a project- or company-specific API
 - Easier for test-writers to create tests without needing detailed knowledge of test environment infrastructure

Notes:

WISHBONE Agent

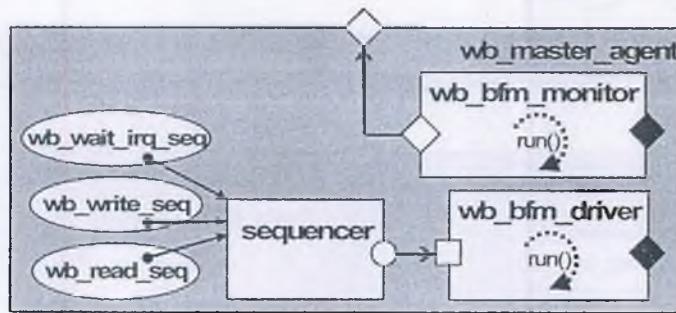
- WISHBONE agent with API sequences
 - Example of a "DUT-Connection" agent
 - API sequences
 - ◆ `wb_write_seq`
 - Generates a WISHBONE write transaction (`wb_txn`)
 - ◆ `wb_read_seq`
 - Generates a WISHBONE read transaction (`wb_txn`)
 - ◆ `wb_wait_irq_seq`
 - Generates wait for interrupt behavior in the `wb_bus_driver` (Advanced class discussion)



Notes:

Basic (Stimulus Generating) Sequences

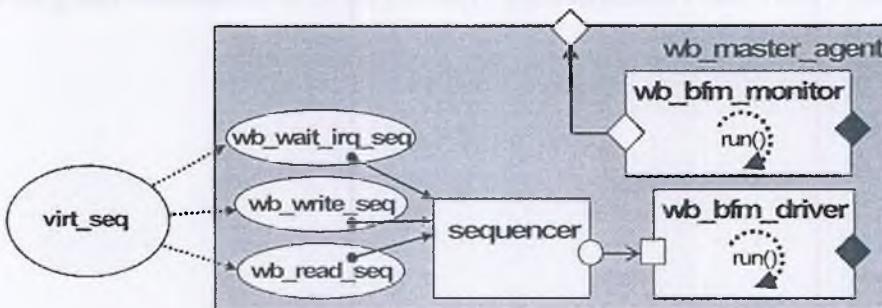
- Usually connected to a driver through a sequencer
 - In the agent below, the sequences are "non-virtual"
 - ◆ They connect to the driver through the sequencer
 - ◆ These sequences forms an API for the WISHBONE bus agent
 - Are "called" as needed – not free running
 - Behave much like task calls



Notes:

"Virtual" Sequence

- "Virtual sequence" is the term used to describe sequences that control or manage other sequences
 - "Control" sequence would be a more descriptive term
 - They do not typically directly generate transactions
 - One way to view sequences is they are like tasks
 - ◆ May be "called" from other sequences or tasks
 - Does not need to know the details of the child sequence
- In the diagram below the virtual sequence `wb_mem_simple_seq` "calls" the non-virtual Wishbone bus agent API sequences
- Virtual sequences may call virtual sequences and so on



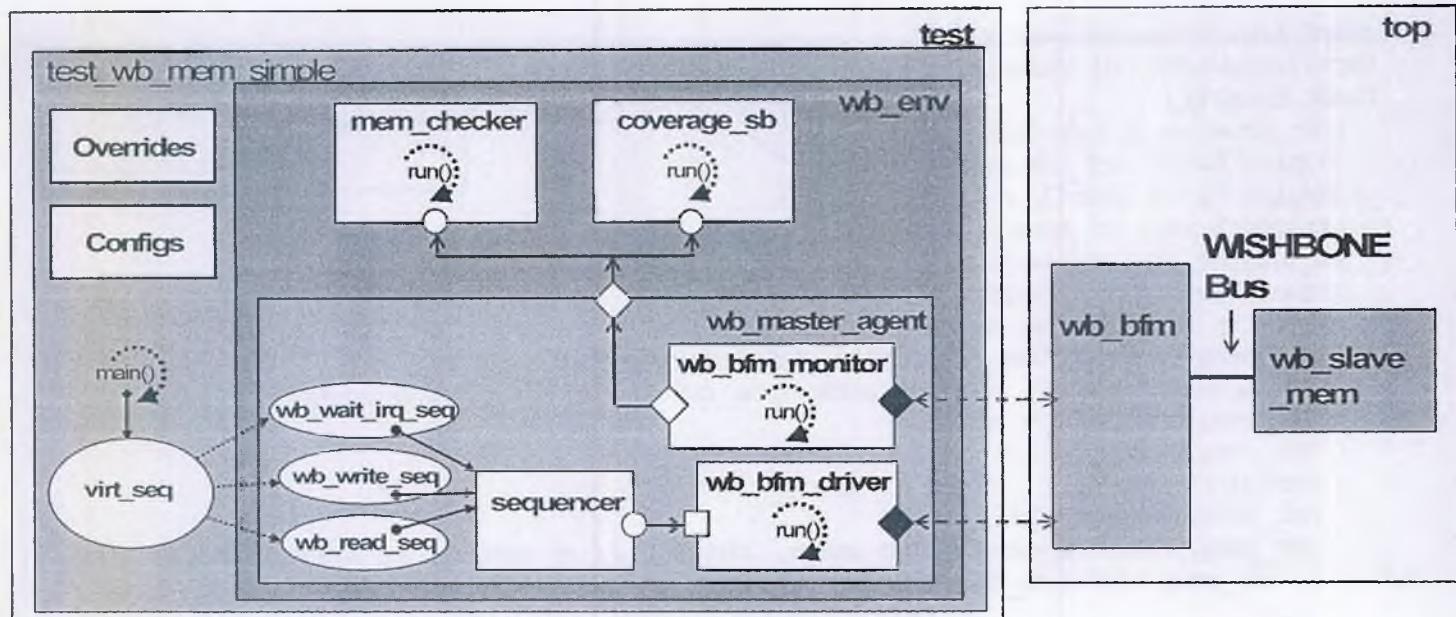
Notes:

Multiple Sequences Relationships

- Virtual sequences may "call" other sequences serially
 - Form a pattern or scenario
 - ◆ For example: Initialize, main stimulus, cleanup
 - ◆ Various patterns of stimulus
 - Can be randomized
- Virtual sequences may "call" other sequences in parallel
 - To apply parallel stimulus to one DUT interface
 - To apply or coordinate parallel stimulus across multiple DUT interfaces
- Virtual sequences may "call" other virtual sequences
 - End up with various hierarchical relationships between sequences

Notes:

Multiple Sequences Example

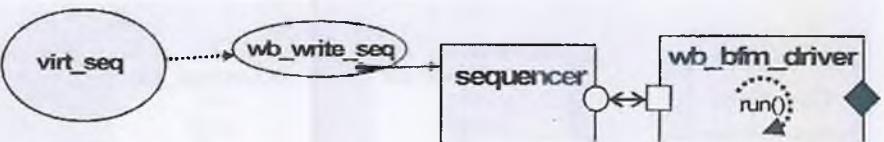


Notes:

Starting a Sequence from another Sequence

- Call the start() method of the sequence

```
class virt_seq extends wb_access_base_seq;  
  ...  
  rand int unsigned addr;  
  rand logic[31:0] data;  
  task body();  
    uvm_status_e txn_status;  
    logic[31:0] rd_data[];  
    logic[31:0] dat[1];  
    wb_read_seq rd_seq;  
    wb_write_seq wr_seq;      super.body();  
    repeat (num_ops) begin  
      // Do a Wishbone write  
      wr_seq = wb_write_seq::type_id::create("wr_seq");  
      this.randomize() with {addr < m_config.m_mem_slave_size[0]; };  
      wr_seq.address = addr;  
      wr_seq.count = 1;  
      dat[0] = data;  
      wr_seq.data = dat;  
      wr_seq.start(m_config.wb_seqr, this); // m_config is inherited  
      //      wr_seq.start(m_sequencer, this); // alternative  
    ...  
  endtask  
endclass
```



property of "this" sequence which was set
when this sequence was started

Notes:

Initialize and Start Method `init_start_seq()`

- Almost always the virtual sequence will need to initialize the sequence that it is "calling" before it starts it
 - A recommended approach is for the sequences to provide a method to do the initialization and starting of that sequence

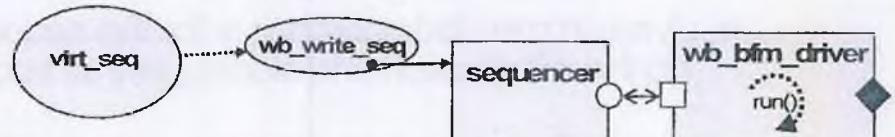
```
class wb_write_seq extends uvm_sequence #(wb_txn);  
    ...  
    // Initialize & start the sequence  
    virtual task init_start_seq(  
        output uvm_status_e txn_status,  
        input uvm_sequencer #(wb_txn) seqr,  
        input uvm_sequence parent_seq,  
        input logic [31:0] addr = 'x,  
        input logic [31:0] dat [],  
        input int cnt = 1);  
        address = addr;  
        data = dat;  
        count = cnt;  
        this.start(seqr, parent_seq);  
        txn_status = txn.t_status;  
    endtask  
    ...  
endclass
```

sequencer and parent sequence are "required" arguments

Notes:

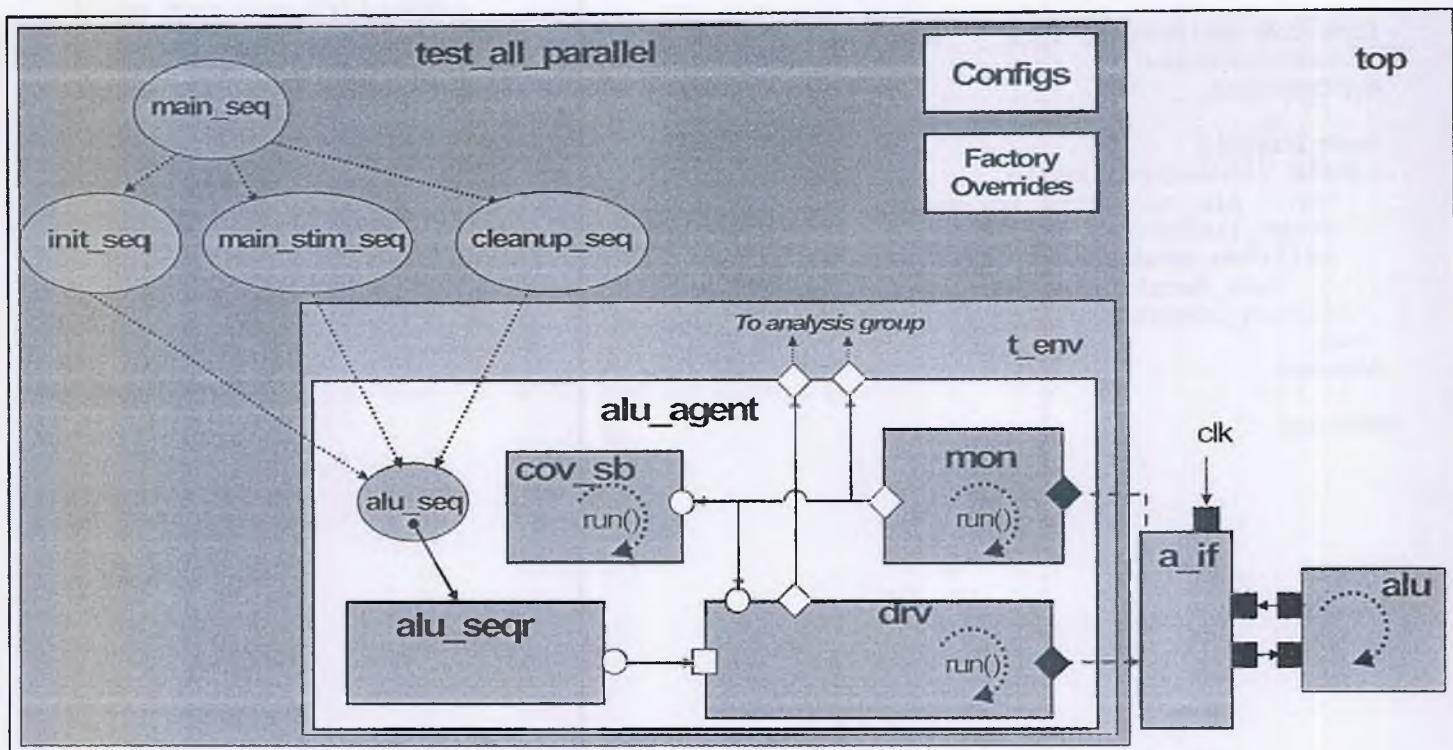
Example call of init_start_seq()

```
class virt_seq extends wb_access_base_seq;
  ...
  rand int unsigned addr;
  rand logic[31:0] data;
  ...
  task body();
    uvm_status_e txn_status;
    logic[31:0] rd_data[];
    logic[31:0] dat[1];
    super.body();
    wr_seq = wb_write_seq::type_id::create("wr_seq");
    // generate stimulus
    repeat(num_ops) begin
      ...
      // Do a Wishbone write with init_start_seq
      this.randomize() with {addr < m_config.m_mem_slave_size[0];};
      dat[0] = data;
      wr_seq.init_start_seq(txn_status, wb_seqr, this,
                            addr, dat);
      ...
    endtask
  ...
endclass
```



Notes:

Another Multiple Sequences Example



WILLAMETTE
HDL

uvm_intro_3.5

©Willamette HDL Inc

code in examples/alu

319

Notes:

Multiple Example: alu_seq

```
class alu_seq extends uvm_sequence #(alu_txn, alu_txn);
`uvm_object_utils(alu_seq)
  alu_txn txn, rsp;
  rand op_type_t op;

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

task body();
  while (!done[op]) begin
    txn = alu_txn::type_id::create("txn");
    start_item(txn);
    if(! (txn.randomize() with {mode == op;}))
      `uvm_fatal("alu_seq", "Randomize error")
    finish_item(txn);
  end
endtask

endclass
```

Sequence that generates one of add, sub, mult, div, operation type chosen by this variable



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu

320

Notes:

Multiple "calls" Example: Parallel

```
class main_stim_seq extends uvm_sequence #(alu_txn, alu_txn);
`uvm_object_utils(main_stim_seq)
alu_seq ADD_s;    alu_seq SUB_s;
alu_seq MUL_s;   alu_seq DIV_s;
task body();
fork
begin
  ADD_s = alu_seq::type_id::create("ADD_s");
  if(! (ADD_s.randomize() with {ADD_s.op == ADD;}))
    `uvm_fatal("Main_stim_seq", "Randomization error")
  ADD_s.start(m_sequencer, this);
end
begin
  SUB_s = alu_seq::type_id::create("SUB_s");
  if(! (SUB_s.randomize() with {SUB_s.op == SUB;}))
    `uvm_fatal("Main_stim_seq", "Randomization error")
  SUB_s.start(m_sequencer, this);
end
// multiply and divide not shown
join
endtask
endclass
```

Customizes each "call" by setting the op property

Uses argument
this.m_sequencer to start child sequences and this to indicate the parent sequence

"calls" 4 sequences in parallel



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu

321

Notes:

Prioritized Sequence Item Selection

- Sequences can be assigned a priority when started:

```
seq.start(seqr, parent, int priority = 100, call_pre_post=1)
```

- Specify a priority to the sequencer
- Good for interrupt sequences

- Sequences can change their priority dynamically

```
sequence.set_priority(pri)
```

- Any item can be assigned its own priority

- Overrides sequence priority

```
start_item(uvm_sequence_item item, int set_priority = -1)
```

- Default priority of -1 tells sequencer to use the current sequence priority

Notes:

Setting Sequencer Arbitration

- Sequencer arbitration is set by:

```
seqr.set_arbitration(mode)
```

Mode	Uses Priority?	Description
SEQ_ARB_FIFO	N	(Default) All requests are granted "first come, first serve". Any priority is <u>ignored</u> .
SEQ_ARB_RANDOM	N	Requests are selected randomly. Priority is ignored.
SEQ_ARB_WEIGHTED	Y	Requests are granted using weighted random selection. Priority value is used as weight.
SEQ_ARB_STRICT_FIFO	Y	Select from only the requests with the highest priority value, in FIFO order.
SEQ_ARB_STRICT_RANDOM	Y	Select from only the requests with the highest priority value, in random order.
SEQ_ARB_USER	Maybe	Calls a user-supplied function to select the request.

Notes:

Priority Example: main_arb_seq

```
class main_arb_seq extends main_stim_seq;
  ...
  task body();
    $display("**** RUNNING MAIN ARB SEQUENCE ****");
    fork
      begin
        ADD_s = alu_seq::type_id::create("ADD_s");
        if(! (ADD_s.randomize() with {ADD_s.op == ADD;}))
          `uvm_fatal("Main_stim_seq", "Randomization error")
        ADD_s.start(m_sequencer, this, 50);
      end
      begin
        SUB_s = alu_seq::type_id::create("SUB_s");
        if(! (SUB_s.randomize() with {SUB_s.op == SUB;}))
          `uvm_fatal("Main_stim_seq", "Randomization error")
        SUB_s.start(m_sequencer, this, 100);
      end
      begin
        MUL_s = alu_seq::type_id::create("MUL_s");
        if(! (MUL_s.randomize() with {MUL_s.op == MUL;}))
          `uvm_fatal("Main_stim_seq", "Randomization error")
        MUL_s.start(m_sequencer, this, 200);
      end
      begin
        DIV_s = alu_seq::type_id::create("DIV_s");
        if(! (DIV_s.randomize() with {DIV_s.op == DIV;}))
          `uvm_fatal("Main_stim_seq", "Randomization error")
        DIV_s.start(m_sequencer, this, 400);
      end
    join
  endtask
endclass
```

Set priority weights



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/alu

324

Notes:

Priority Example: test_arbitration

```
class test_arbitration extends test_all_parallel;
`uvm_component_utils(test_arbitration)

function void build_phase(uvm_phase phase);

main_stim_seq::type_id::set_type_override(main_arb_seq::type_id::get());
super.build_phase(phase);
endfunction

task run_phase(uvm_phase phase);
seqr_handle.set_arbitration(SEQ_ARB_WEIGHTED);
m_seq.start(m_sqr, null);
endtask

endclass
```

Set arbitration on sequencer



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu

325

Notes:

Priority Example: Output

```
# UVM_INFO @ 200: Received correct result for alu_txn 0, mode DIV
# UVM_INFO @ 400: Received correct result for alu_txn 1, mode DIV
# UVM_INFO @ 600: Received correct result for alu_txn 2, mode DIV
# UVM_INFO @ 800: Received correct result for alu_txn 3, mode MUL
# UVM_INFO @ 1000: Received correct result for alu_txn 4, mode MUL
# UVM_INFO @ 1200: Received correct result for alu_txn 5, mode DIV
# UVM_INFO @ 1400: Received correct result for alu_txn 6, mode MUL
# UVM_INFO @ 1600: Received correct result for alu_txn 7, mode ADD
# UVM_INFO @ 1800: Received correct result for alu_txn 8, mode MUL
# UVM_INFO @ 2000: Received correct result for alu_txn 9, mode MUL
# UVM_INFO @ 2200: Received correct result for alu_txn 10, mode ADD
# UVM_INFO @ 2400: Received correct result for alu_txn 11, mode SUB
```

SEQ_ARB_WEIGHTED

```
# UVM_INFO @ 200: Received correct result for alu_txn 0, mode DIV
# UVM_INFO @ 400: Received correct result for alu_txn 1, mode DIV
. . . Lots more DIV packets . .
# 100% coverage of DIV acheived
# UVM_INFO @ 160200: Received correct result for alu_txn 800, mode MUL
# UVM_INFO @ 160400: Received correct result for alu_txn 801, mode MUL
. . . Lots more MUL packets . .
# 100% coverage of MUL acheived
# UVM_INFO @ 221600: Received correct result for alu_txn 1107, mode SUB
# UVM_INFO @ 221800: Received correct result for alu_txn 1108, mode SUB
. . . Lots more SUB packets . .
# 100% coverage of SUB acheived
# UVM_INFO @ 239200: Received correct result for alu_txn 1195, mode ADD
# UVM_INFO @ 239400: Received correct result for alu_txn 1196, mode ADD
. . . Lots more ADD packets . .
# 100% coverage of ADD acheived
```

SEQ_ARB_STRICT_FIFO



uvm_intro_3.5

©Willamette HDL Inc.

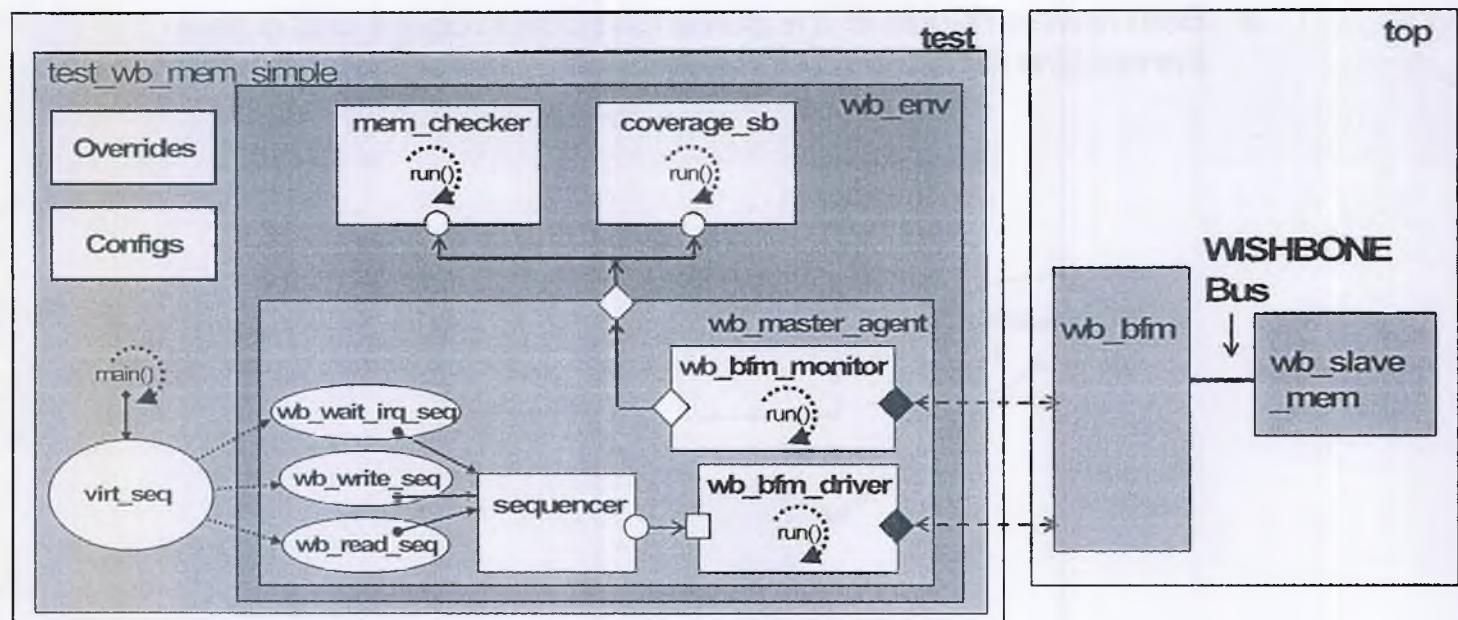
code in examples/alu

326

Notes:

Sequence Example with Response

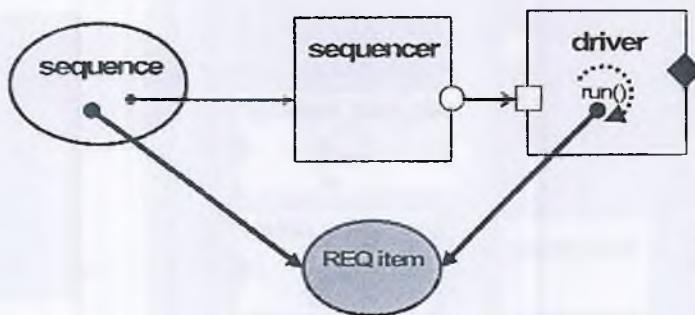
- Sequences can obtain feedback/status (response) from the driver
- Example: Wishbone bus read



Notes:

Shared Transaction

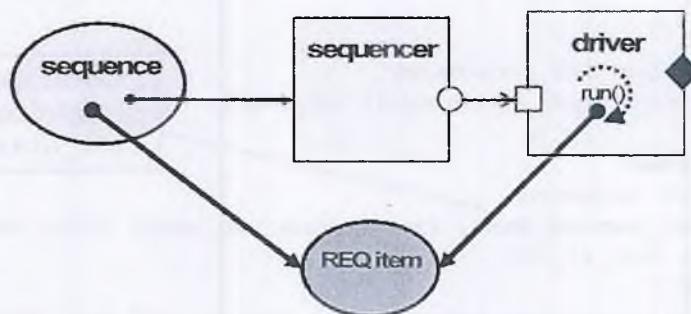
- The sequence and the driver actually share the REQ transaction item
 - Sequence creates the REQ object and passes it to the driver through the sequencer
 - ◆ Both have a handle to the **same** transaction object until a new transaction is created by the sequence



Notes:

Return Response Data – Shared Transaction

- Driver returns response data through the shared REQ transaction item
 - Puts the response data into the data field of the REQ item
 - Calls `item_done()` to indicate that
 - ◆ The response data is in REQ item
 - ◆ The driver is done with REQ item
- When the sequence unblocks from `finish_item()` (when `item_done()` executes in the driver) the data is in the REQ item



Notes:

Responses – Virtual Sequence:

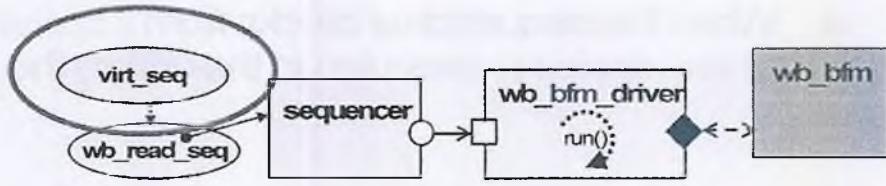
```
class virt_seq extends wb_access_base_seq;
`uvm_object_utils(virt_seq)

int num_ops = 200;
rand int unsigned addr;
rand logic[31:0] data;

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

task body();
uvm_status_e txn_status;
logic[31:0] rd_data[];
logic[31:0] dat[1];
wb_read_seq rd_seq;
```
super.body();
// create Wishbone bus API sequences
rd_seq = wb_read_seq::type_id::create("rd_seq");
```
// Do a Wishbone read
// init and start sequence
rd_seq.init_start_seq(rd_data, txn_status, wb_seqr, this, wr_seq.address, 1 );
if(txn_status != UVM_IS_OK)
`uvm_fatal("SEQ",
$formatf("Wishbone bus transaction status: %s",txn_status.name()))

```



Data returned to virtual sequence through output argument of init_start_seq() task

Notes:

Responses—Sequence:

```

class wb_read_seq extends uvm_sequence #(wb_txn);
  `uvm_object_utils(wb_read_seq)
  rand logic [31:0] address;
  rand logic [31:0] data [];
  rand int count;
  wb_txn txn;
  ...
  task body();
    //create transaction
    txn = wb_txn::type_id::create("txn");
    start_item(txn); // tell sequencer ready to give an transaction
    txn.init_txn(READ, address, data, count); //initialize transaction
    finish_item(txn); // send transaction
  endtask
  virtual task init_start_seq(
    output logic [31:0] dat [],
    output uvm_status_e txn_status,
    input uvm_sequencer #(wb_txn) seqr,
    input uvm_sequence parent_seq,
    input logic [31:0] addr = 'x,
    input int cnt = 1);
    address = addr;
    count = cnt;
    data= new[cnt]; // resize data array
    this.start(seqr, parent_seq);
    dat = txn.data; // return the data
    txn_status = txn.t_status;
  endtask
  ...
endclass

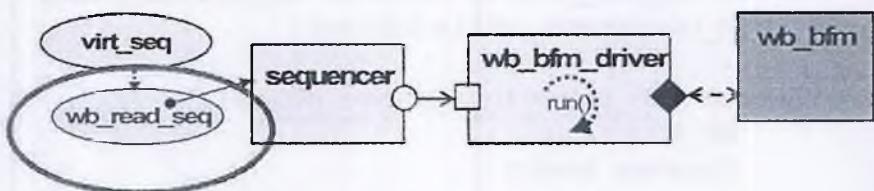
```

uvm_intro_3.5

© Willamette HDL Inc.

code in examples/wb_mem

331



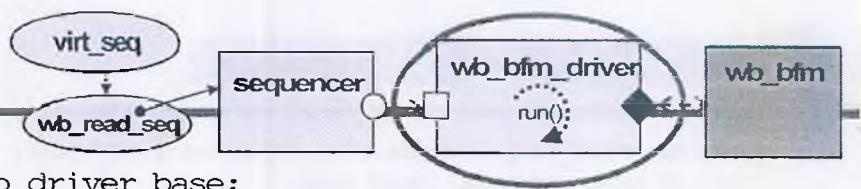
finish_item() sends the txn transaction item to the driver

When finish_item() unblocks the data is in txn transaction item

After body() task finishes then start() returns and the read data and status are returned through task arguments

Notes:

Responses—Driver:



```

class wb_bfm_driver extends wb_driver_base;
  `uvm_component_utils(driver)
  ...
  task run_phase(uvm_phase phase);
    wb_txn txn;
    forever begin
      seq_item_port.get_next_item(txn);
      case (txn.txn_type)
        WRITE:
        ...
        READ :
        begin
          v_wb_bfm.wb_read_cycle(txn.data, txn.adr, txn.count,
            .wb_master_id(m_config.m_wb_master_id), .op_stat(txn.t_status));
          seq_item_port.item_done(); // signal done with txn
        end
        default: begin
          txn.print();
          `uvm_fatal("DRIVER", "Unsupported txn type received\n")
        end
      endcase
    end
  endtask
endclass

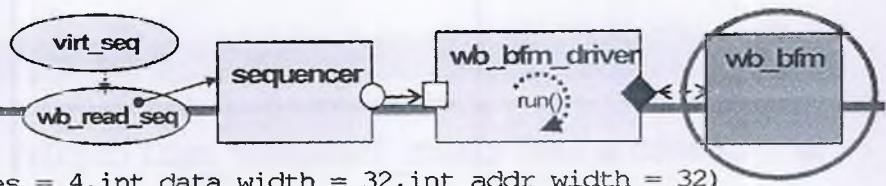
```

The response data is put into the data field of the original transaction

Calls item_done () to indicate data is in the REQ transaction and driver is done with the item

Notes:

Response - BFM



```

interface wb_syscon_bfm_if #(
    int num_masters = 8, int num_slaves = 4, int data_width = 32, int addr_width = 32)
    ( output bit clk, input bit rst);
  
```

// BFM interface

//READ 1 or more cycles

```

task automatic wb_read_cycle(output logic[31:0] data[],
    input bit [31:0] adr, int unsigned count = 1, byte_sel = 4'b1111,
    input wb_master_id = test_params_pkg::wb_agent_master_id,
    output uvm_status_e op_stat);
  
```

logic [31:0] temp_addr = adr;

data = new[count]; //

wait(!rst); // wait for reset to finish

```

@ (posedge clk) #1; // sync to clock edge + 1 time step
for(int i = 0; i < count; i++) begin
  m_addr[wb_master_id] = temp_addr;
  ...
  m_stb [wb_master_id] = 1;
  @ (posedge clk)
  while (! (m_ack[wb_master_id] & gnt[wb_master_id])) @ (posedge clk);
  data[i] = m_rdata; // get data
  
```

the first argument data is an outputs

wb_read_cycle() does a wishbone bus read cycle

The read data is returned through the data argument

```

    temp_addr = temp_addr + 4; // byte address so increment by 4 for word addr
end
  
```

m_cyc[wb_master_id] = 0;

m_stb[wb_master_id] = 0;

```

while (m_ack[wb_master_id]) @ (posedge clk); // Wait for ack to de-assert
endtask
  
```

endinterface



uvm_intro_3.5

©Willamette HDL Inc.

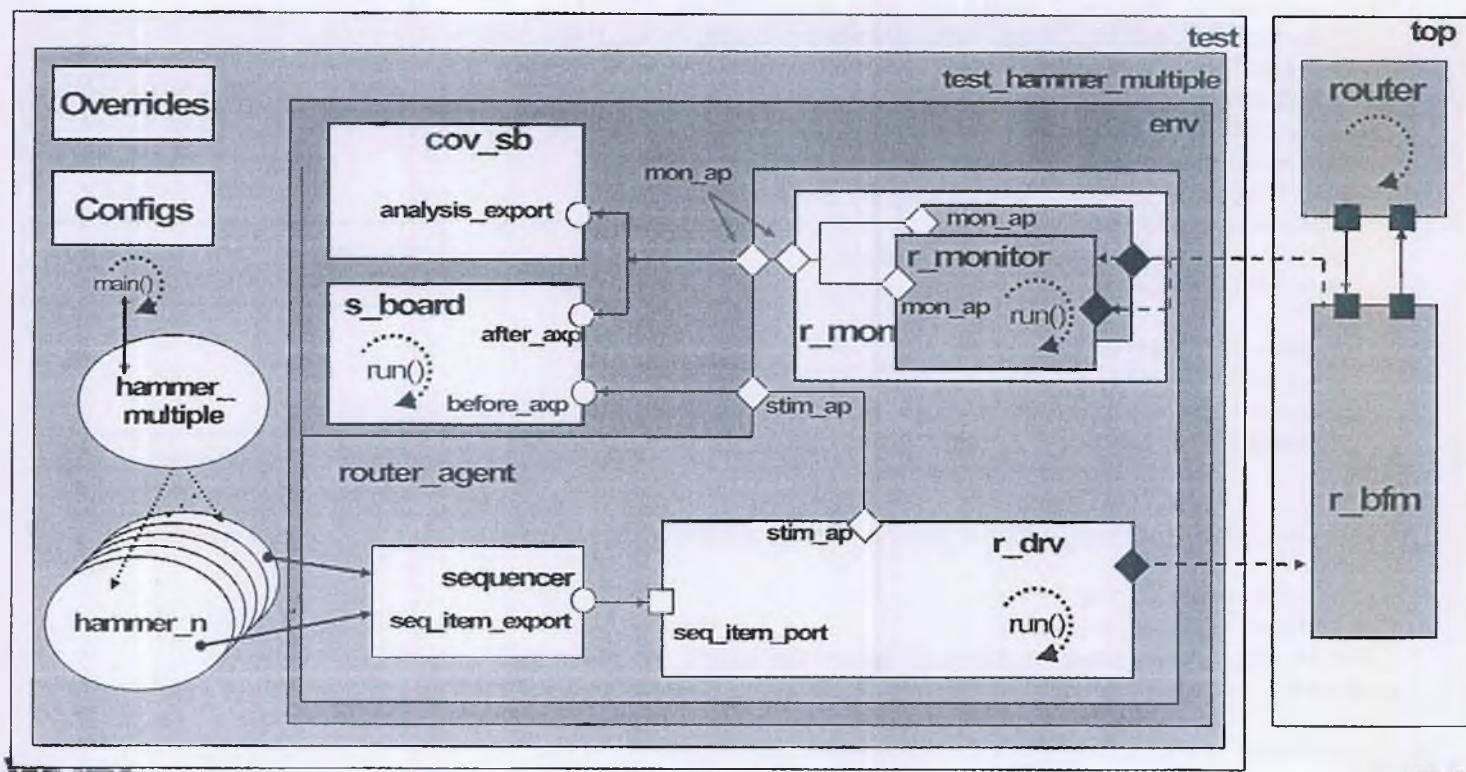
code in examples/wb_mem

333

Notes:

Lab – Multiple Sequences: Overview

- Create a test `test_hammer_multiple`
 - Create sequences to "hammer" all the router output ports



WHDL

uvm_intro_3.5

©Willamette HDL Inc.

334

Notes:

Lab – Multiple Sequences: Instructions

- NOTE: For this lab the test used is `test_hammer_multiple`
- Working directory: `multiple_sequences`
- Instructions:
 - Edit file `sequences/hammer_n.svh`
 - ◆ Create sequence class `hammer_n`
 - ◆ Copy the code from the previous lab's `hammer_0` sequence and modify:
 - Add a variable `hammer_port_num` of type `int`
 - Randomize with the `dest_id` field set to `hammer_port_num`
 - Edit file `sequences/hammer_multiple.svh`
 - ◆ Create a virtual sequence class called `hammer_multiple`
 - ◆ Create an instance of `hammer_n` for each port of the router
 - Provide a unique port number for each to "hammer" by setting the `hammer_port_num` variable of each `hammer_n` to a different port number
 - ◆ In the `body()` method
 - Start all the `hammer_n` sequences in parallel
- Compile and run
 - ◆ No errors: `make`
 - ◆ Errors: `make bad`

Notes:

Lab – Multiple Sequences: Sample Output

```
#  
# UVM_INFO @ 1287100: uvm_test_top.env.cov_sb [COVERAGE] *****  
# UVM_INFO @ 1287100: uvm_test_top.env.cov_sb [COVERAGE] Final Coverage = 100.000000%  
# UVM_INFO @ 1287100: uvm_test_top.env.cov_sb [COVERAGE] 100% Coverage met with 332  
transactions  
# UVM_INFO @ 1287100: uvm_test_top.env.cov_sb [COVERAGE] *****  
#  
# UVM_INFO @ 1287100: uvm_test_top.env.s_board [Packet_Comparator] Matches: 793  
# UVM_INFO @ 1287100: uvm_test_top.env.s_board [Packet_Comparator] Mismatches: 0  
# UVM_INFO @ 1287100: uvm_test_top.env.s_board [Packet_Comparator] Missing: 1  
#  
#  
# --- UVM Report Summary ---  
#  
# ** Report counts by severity  
# UVM_INFO : 9  
# UVM_WARNING : 0  
# UVM_ERROR : 0  
# UVM_FATAL : 0  
# ** Report counts by id  
# [COVERAGE] 4  
# [Packet_Comparator] 3  
# [RNTST] 1  
# [UVMTOP] 1  
# ** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)  
# Time: 1287100 ns Iteration: 112 Instance: /test
```

Sol



uvm_intro_3.5

©Willamette HDL Inc.

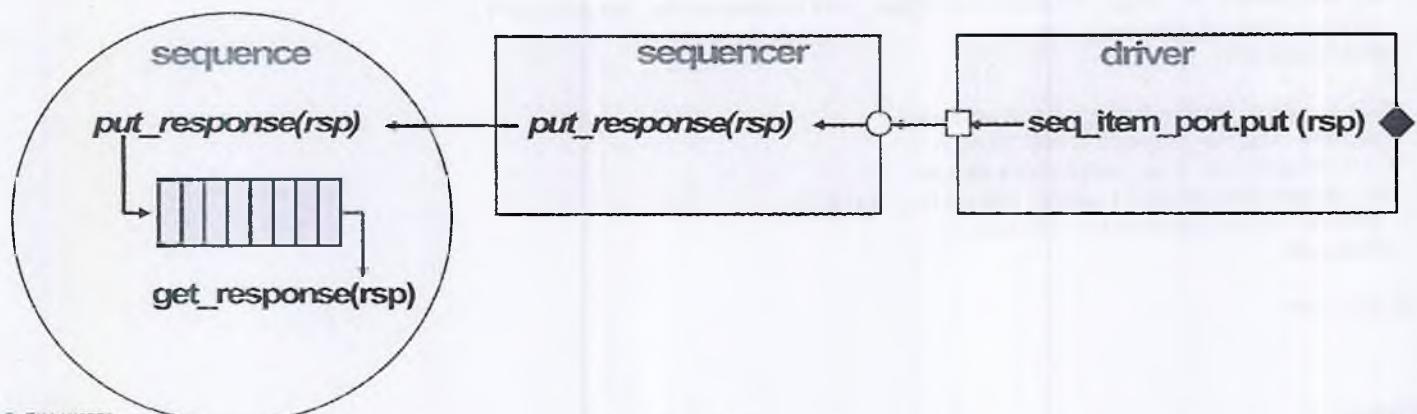
336

Notes:

Response Handling with `get()` / `put()`

Reference Page

- Driver calls `put()` on its `seq_item_port`
- `put()` calls sequencer's `put_response()`
- Sequencer's `put_response()` calls sequence's `put_response()`
- Sequence's `put_response()` puts data in its response FIFO
 - If the FIFO is full
 - ◆ An error is generated
 - ◆ Response is dropped (`put_response()` is a function so can't block)
- Sequence's response FIFO is by default 8 deep
- Sequence gets response out of FIFO with `get_response()`



Notes:

Feedback Example: test_mac

Reference Page

```
class test_mac extends alu_seq_test_base;
`uvm_component_utils(test_mac)

mac_sequence m_sequence; // the main sequence

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

function void build_phase(uvm_phase phase);
    // factory overrides
    analysis_group_base::type_id::set_type_override(mac_analysis_group::type_id::get());
    sc_driver::type_id::set_type_override(sc_driver_rsp::type_id::get());
    factory.print(1);

    // create main sequence
    m_sequence = mac_sequence::type_id::create("m_sequence");
    super.build_phase(phase);
endfunction

task main_phase(uvm_phase phase);
    phase.raise_objection(this);
    // start up the main sequence
    m_sequence.start( seqr_handle , null);
    phase.drop_objection(this);
endtask

endclass
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu

338

Notes:

Feedback Example: mac_sequence

Reference Page

```
class mac_sequence extends uvm_sequence #(alu_txn, alu_txn);
`uvm_object_utils(mac_sequence)
alu_txn txn, rsp;
sequence_data data;
shortint unsigned sum = 0;

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

task pre_body();
data = new();
data.randomize();
endtask

task body();
foreach (data.a[i]) begin
txm = alu_txn::type_id::create("txm");
start_item(txm);
if(!txm.randomize() with
{ txm.mode == MUL; txm.vall == data.a[i]; txm.val2 == data.b[i]; })
`uvm_fatal("mac_seq", "Randomize error")
finish_item(txm);
get_response(rsp);
uvm_report_info($sformatf("MAC_sequence_%s",get_name()),
$sformatf("a=%0d b=%0d ab=%0d",data.a[i],data.b[i],rsp.result));

```

The diagram illustrates the execution flow of the UVM sequence. It starts with the 'Pre-generate data' step, which corresponds to the 'data.randomize()' call. This leads to the 'Send MUL transaction to get product' step, which corresponds to the 'txm.randomize()' call. Both steps are enclosed in boxes.

Notes:

Feedback Example: mac_sequence (cont.)

Reference Page

```
start_item(txn);
if(!txn.randomize() with { mode == ADD; val1 = rsp.result; val2 = sum; })
    `uvm_fatal("mac_seq", "Randomize error")
finish_item(txn);
get_response(rsp);
sum = rsp.result;
uvm_report_info($sformatf("MAC_sequence_%s", get_name()),
                $sformatf("sum = %0d", sum));
end
uvm_report_info($sformatf("MAC_sequence_%s", get_name()),
                $sformatf("Final sum = %0d", sum)); endtask;
endclass
```

```
class sequence_data;
rand byte size;
rand shortint unsigned a[];
rand shortint unsigned b[];

constraint onsize { size inside {[1:10]}; }
constraint asize { a.size() == size; }
constraint bsize { b.size() == size; }
constraint avals {foreach(a[i]) a[i] inside {[0:10]}; }
constraint bvals {foreach(b[i]) b[i] inside {[0:10]}; }
endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/alu

340

Notes:

- Create a sequence item by inheriting from the base class

`uvm_sequence_item`

- Key properties and methods that are inherited:

- ◆ `sequence_id`
 - ID of sequence that generated the sequence item (default value: -1)
 - Set by the sequencer
 - Accessed with methods `set_sequence_id()` and `get_sequence_id()`
 - ◆ `transaction_id`
 - ID of this transaction item (default value: -1)
 - Set by either the sequencer or generating sequence
 - Accessed with methods `set_transaction_id()` and `get_transaction_id()`

`uvm_sequence_item`

Notes:

Feedback Example: Driver with put() Response

Reference Page

```
class sc_driver_rsp extends sc_driver;
`uvm_component_utils(sc_driver_rsp)

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

// run task
// has a response where sc_driver does not
task run_phase(uvm_phase phase);
  alu_txn stim_txn, rsp_txn;
  forever begin
    seq_item_port.get(stim_txn);
    stim_ap.write(stim_txn);
    @ (negedge v_alu_if.clk)
    #5; // since can't do nonblocking in Monitor
    v_alu_if.vall <= stim_txn.vall;
    v_alu_if.val2 <= stim_txn.val2;
    v_alu_if.mode <= stim_txn.mode;
    v_alu_if.txn_id = stim_txn.get_transaction_id();
    v_alu_if.valid_i <= 1; // set valid_i
    @ (posedge v_alu_if.clk)
    v_alu_if.valid_i <= 0; // clear valid_i
    rsp_buf.get(rsp_txn);
    rsp_txn.set_id_info(stim_txn);
    seq_item_port.put(rsp_txn);
  end
endtask
endclass
```

Get the request transaction

Get the response to the request

Need to restore original sequencer_id so that the response is directed back to the correct sequence

Notes:

UVM Registers

In this section



- Creating Register Models
- Integrating Register Models
- Using Register Models
- Other Register "Stuff"

Notes:

Registers

- Most DUTs have registers and memories that need to be controlled, checked and covered
- One may wish to
 - Read and write registers and memories as part of normal operation
 - Analyze register activity
 - Scoreboard - check the DUT registers and memories against a shadow device (with shadow registers)
 - Collect coverage on the registers
 - Initialize DUT registers and memories
 - Randomize the DUT registers and memories



uvm_intro_3.5

©Willamette HDL Inc.

344

Notes:

UVL Registers

- UVM provides a set of classes for developing a register model
 - Provides a way of tracking registers of a DUT
 - Provides a convenience layer for accessing DUT registers & memories



uvl_intro_3.5

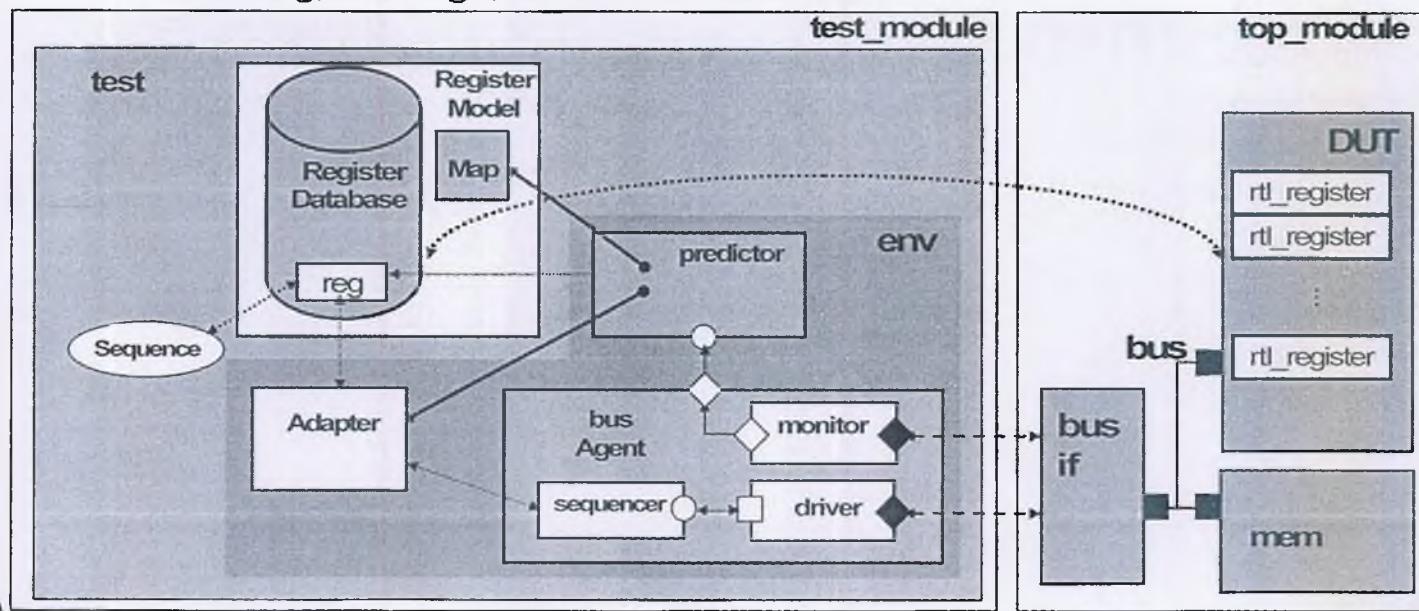
©Willamette HDL Inc.

345

Notes:

The Register Model

- Reflects the hardware-software specification
 - Memories, registers, fields, bits etc.
- Provides for abstraction of stimulus
 - Register name \leftrightarrow register address translation
- Provides for ease of verification
 - Mirroring, coverage, backdoor access etc.



WHDL

uvm_intro_3.5

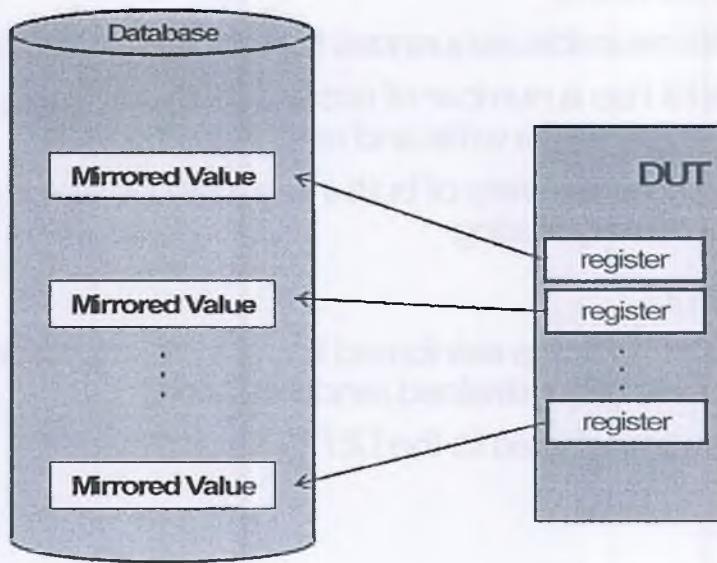
©Willamette HDL Inc.

346

Notes:

Register Model Use - 1

- Means of looking up a mirror of the current DUT hardware state



Notes:

Register Model Use - 2

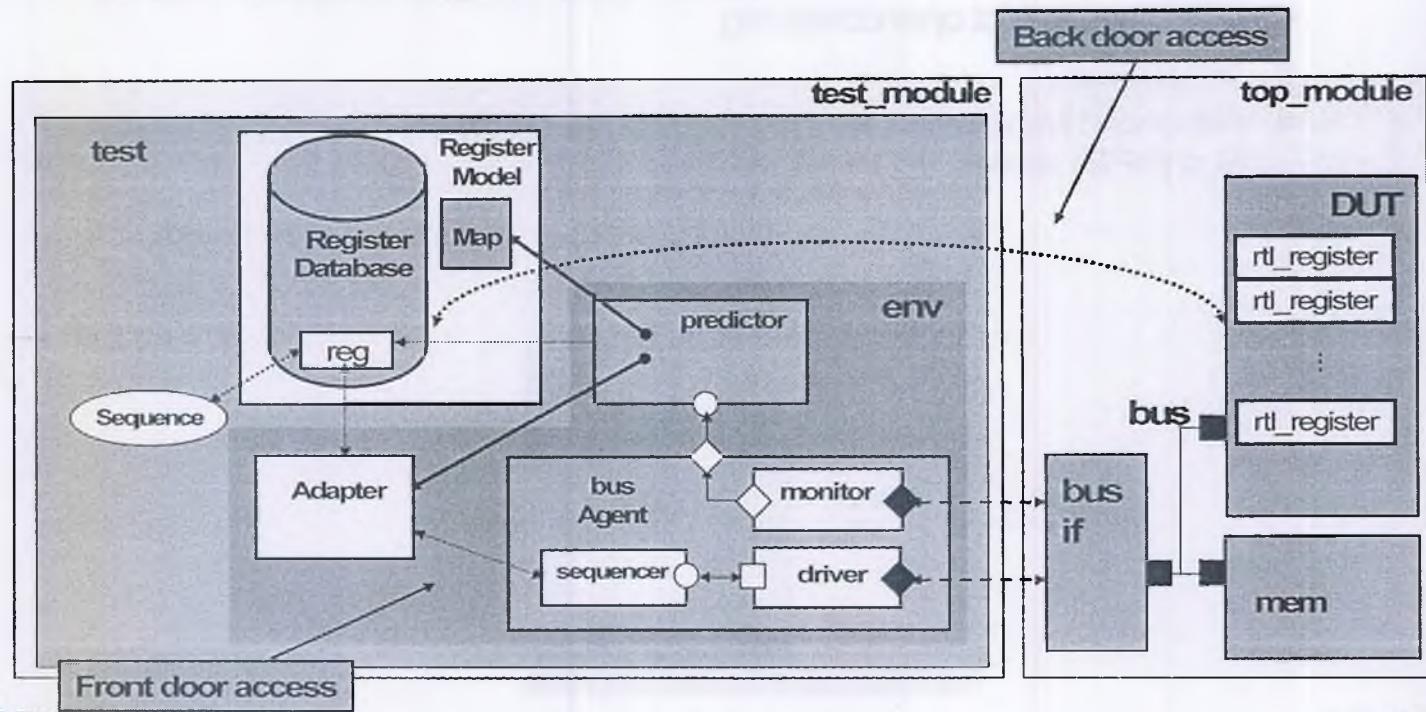
- Means of accessing the DUT
- Means of updating the register model database
- Used to create stimulus
 - Easier to write reusable sequences that access registers and memories
 - Register model has a number of access methods (read, write, peek etc.) which sequences use to write and read registers
 - UVM package has a library of built-in sequences which does basic register and memory testing
- Auto-configuration
 - Register model contents are forced into a state representing device configuration using constrained randomization
 - Contents then transferred to the DUT



Notes:

Register Model Use - 3

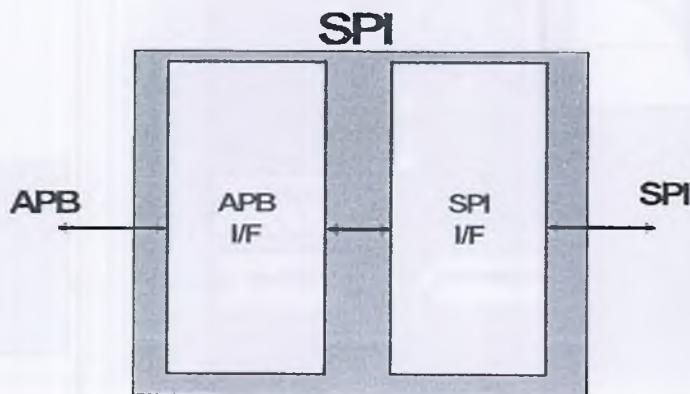
- Front door access uses the bus agent for register accesses using the normal bus protocol
- Back door access uses simulator database access routines to directly force or observe registers in zero time



Notes:

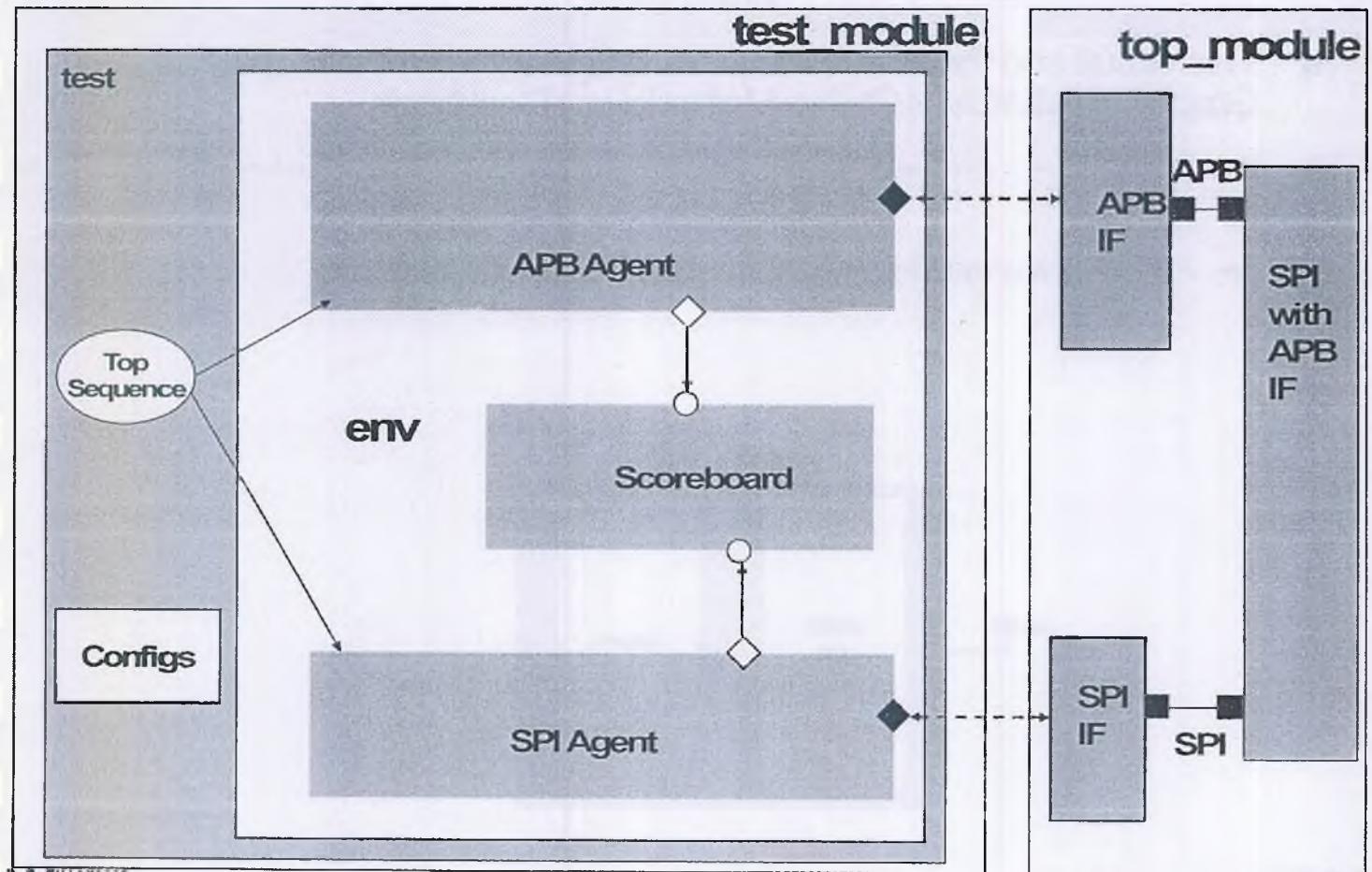
New Example Circuit #1

- This circuit and the associated code originate from the Mentor Graphics UVM/OVM Online Methodology Cookbook
- DUT – Serial Peripheral Interface (SPI) Master Core
 - SPI Core project opencores.org
 - SPI master
 - Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) slave



Notes:

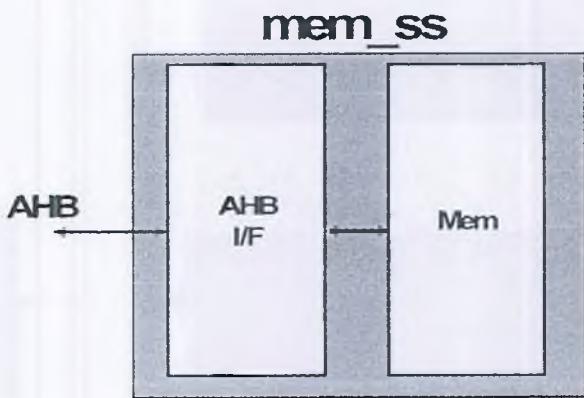
TESTBENCH SPI-APB



Notes:

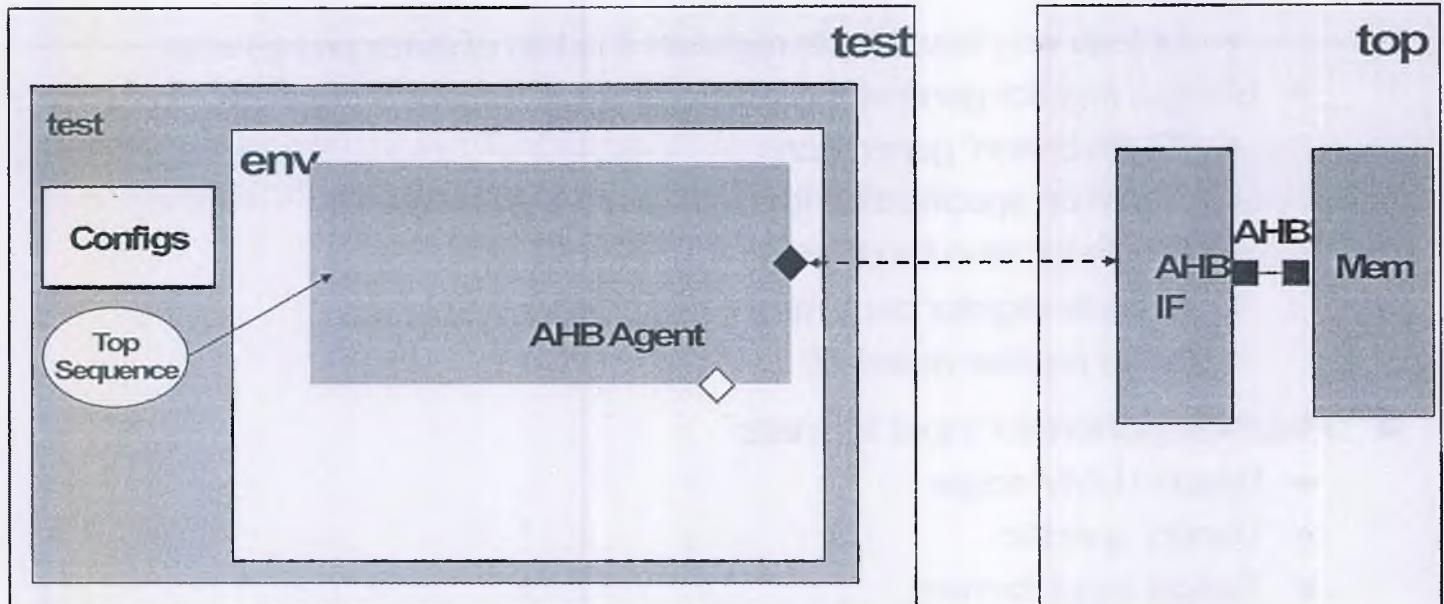
New Example Circuit #2

- This circuit and the associated code originate from the Mentor Graphics UVM/OVM Online Methodology Cookbook
- mem_ss
 - Memory
 - AMBA Advanced High-performance Bus (AHB) interface



Notes:

TESTBENCH mem_SS



Notes:

Generating a Register Model

- A register model may be generated:
 - By hand
 - ◆ Unless very few, simple registers it is lots of (error prone) work
 - Using a register generator (preferred)
 - ◆ "Push button" generation
 - ◆ Common specification for multi-use (HW, SW etc.)
 - ◆ Single location for changes
 - ◆ Multiple register declarations may be easily merged
 - ◆ Certe register assistant
- Register generator input formats
 - Beyond UVM scope
 - Vendor specific
 - Typical input formats:
 - ◆ Spreadsheet
 - ◆ IP-Xact



uvm_intro_35

©Willamette HDL Inc.

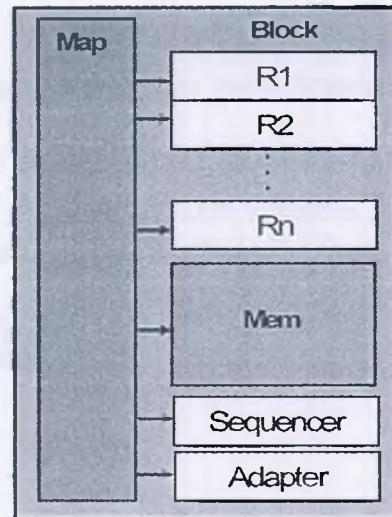
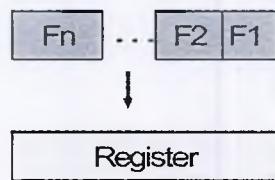
354

Notes:

Register Model Structure

- Implemented with six building blocks

- Register field
- Register
- Register file
- Memory
- Register map
- Register block



Notes:

Register Field

- Modeled by extending from the base class `uvm_reg_field`
- Models a collection of bits of the register
 - Usually associated with a specific function of the register
- Has a width & bit offset
- Has an access mode
 - read/write, read only, write only etc.
- Are contained within the `uvm_reg` class
 - Constructed (`create()`) and then configured (`configure()`) in the `build()` method of the register class



Notes:

Field configure() Method Prototype

UVM

```
function void configure(uvm_reg parent,          // The containing register
                      int unsigned size,        // How many bits wide
                      int unsigned lsb_pos,     // Bit offset within the register
                      string access,           // "RW", "RO", "WO" etc.
                      bit volatile,            // Volatile if bit updated by hardware
                      uvm_reg_data_t reset,    // The reset value
                      bit has_reset,           // Whether the bit is reset
                      bit is_rand,              // Whether the bit can be randomized
                      bit individually_accessible);
                                         // i.e. Totally contained within a byte lane

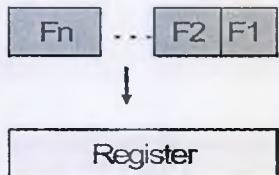
typedef bit unsigned [`UVM_REG_DATA_WIDTH-1:0] uvm_reg_data_t ;
```

- Use of the configure method is shown in the register code example in a couple of slides

Notes:

Register

- Modeled by extending from the base class `uvm_reg`
- Container for one or more fields
- Contained within a register block
- Characteristics are defined in its constructor
- Contains a `build()` method which creates and configures its fields
 - Must be called explicitly!
 - ◆ Not called by the build phase as registers are objects not components



`uvm_reg` constructor

UVM

```
function new (string name="",           // Register name
              int unsigned n_bits,    // Register width in bits
              int has_coverage);    // Coverage model supported by the register
```

WILLAMETTE
HDL

uvm_intro_3.5

©Willamette HDL Inc.

358

Notes:

Register configure() Method Prototype

- Registers are constructed (`create()`) and then configured (`configure()`) in the `build()` method of the register block class
 - Use of the `configure` method is shown in the block code example in a couple of slides

UVM

```
function void configure (uvm_reg_block blk_parent,           // The containing reg block
                      uvm_reg_file regfile_parent = null, // Optional, not used
                      string hdl_path = "");          // Used if HW register can
                                              // be specified in one
                                              // hdl_path string
```



uvm_intro_3.5

© Willamette HDL Inc.

359

Notes:

Register Class Example: ctrl-1

```
class ctrl extends uvm_reg;
  `uvm_object_utils(ctrl)

  uvm_reg_field reserved;
  rand uvm_reg_field ie;
  rand uvm_reg_field lsb;
  rand uvm_reg_field tx_neg;
  rand uvm_reg_field rx_neg;
  rand uvm_reg_field go_bsy;
  uvm_reg_field reserved2;
  rand uvm_reg_field char_len;

  //-
  // new
  //-
  function new(string name = "ctrl");
    super.new(name, 14, UVM_NO_COVERAGE);
  endfunction
```

Code from
spi_reg_pkg

Field declarations

Constructor



Note: This code is typically auto generated!

Notes:

Register Class Example: ctrl-2

```
//  
// build  
//  
virtual function void build();  
    ie = uvm_reg_field::type_id::create("ie");  
    lsb = uvm_reg_field::type_id::create("lsb");  
    tx_neg = uvm_reg_field::type_id::create("tx_neg");  
    rx_neg = uvm_reg_field::type_id::create("rx_neg");  
    go_bsy = uvm_reg_field::type_id::create("go_bsy");  
    reserved2 = uvm_reg_field::type_id::create("reserved2");  
    char_len = uvm_reg_field::type_id::create("char_len");  
  
    ie.configure(this, 1, 12, "RW", 0, 1'b0, 1, 1, 0);  
    lsb.configure(this, 1, 11, "RW", 0, 1'b0, 1, 1, 0);  
    tx_neg.configure(this, 1, 10, "RW", 0, 1'b0, 1, 1, 0);  
    rx_neg.configure(this, 1, 9, "RW", 0, 1'b0, 1, 1, 0);  
    go_bsy.configure(this, 1, 8, "RW", 0, 1'b0, 1, 1, 0);  
    reserved2.configure(this, 1, 7, "RW", 0, 1'b0, 1, 0, 0);  
    char_len.configure(this, 7, 0, "RW", 0, 7'b0000000, 1, 1, 0);  
  
endfunction  
endclass
```

Code from
spi_reg_pkg

Fields creation

Fields
configure



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

361

Notes:

Register File

- A collection of registers and other register files used to create regular repeated structures
 - Optional structure

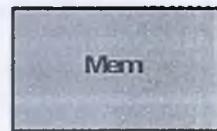
UVM

```
uvm_reg_file constructor
function new (string name="");
```

Notes:

Memory

- Modeled by extending from the base class `uvm_mem`
- Has a specified address range
- Contained within a register block
- Has an offset determined by the register map
- Memory values are not stored because of overhead
- Range and access type specified in the constructor



`uvm_mem constructor`



```
function new (string name,
              longint unsigned size,
              int unsigned n_bits,
              string access = "RW",
              int has_coverage = UVM_NO_COVERAGE);
```

// Name of the memory model
// The address range
// The width of the memory in bits
// Access - one of "RW" or "RO"
// Functional coverage

Notes:

Memory Class Example: mem_1_model

```
// Memory array 1 - Size 32'h2000;
class mem_1_model extends uvm_mem;
`uvm_object_utils(mem_1_model)

function new(string name = "mem_1_model");
    super.new(name, 32'h2000, 32, "RW", UVM_NO_COVERAGE);
endfunction

endclass: mem_1_model
```

Code from
spi_reg_pkg



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/spi_reg

364

Notes:

Register Map

- Modeled by extending from the base class `uvm_map`
- Defines the address space offsets of registers, maps, and sub-blocks
- Contained within a register block
- Multiple register maps may be within a register block
 - Provides different mapping to the same registers from different interfaces
- Specifies sequencer in bus agent is used for register accesses
- Specifies which adapter is used to convert to/from target bus

Map	
Offset	Reg/Mem/Block
0x00	R1
0x04	R2
0x1000	Mem
Agent sequencer handle	
Adapter handle	

Notes:

Map Add Methods Prototypes

- Registers and memories are added to the map with the `add_reg()` and `add_mem()` methods respectively

UVM

```
function void add_reg (uvm_reg rg,
                      uvm_reg_addr_t offset,
                      string rights = "RW",
                      bit unmapped=0,
                      uvm_reg_frontdoor frontdoor=null); // Handle to register
// frontdoor access object
```

UVM

```
function void add_mem (uvm_mem mem,
                      uvm_reg_addr_t offset,
                      string rights = "RW",
                      bit unmapped=0,
                      uvm_reg_frontdoor frontdoor=null); // Handle to memory frontdoor
// access object
```

Notes:

Map create_map() Method Prototype

- Register maps are created within register blocks using the `create_map()` method
 - First map to be created within a register block is assigned the `default_map` member of the block

UVM

```
function uvm_reg_map create_map(string name,  
                                uvm_reg_addr_t base_addr,  
                                int unsigned n_bytes,  
                                uvm_endianness_e endian,  
                                bit byte_addressing=0);  
                                // Name of the map handle  
                                // The maps base address  
                                // Map access width in bytes  
                                // The endianness of the map  
                                // Whether byte_addressing  
                                // is supported
```

Code from
spi_reg_block in
spi_reg_pkg

```
APB_map = create_map("APB_map", 'h0, 4, UVM_LITTLE_ENDIAN);
```



uvm_intro_3.5

©Willamette HDL Inc.

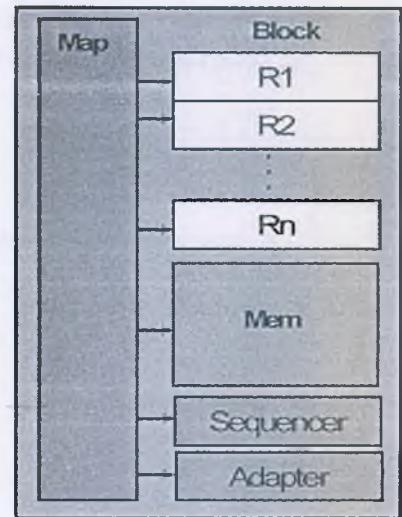
code in examples/spi_reg

367

Notes:

Register Block

- Modeled by extending from the base class `uvm_reg_block`
- Corresponds to hardware block or sub-system
- Contains one or more registers or sub-blocks
- Contains one or more register maps
- May contain memories



Notes:

SPI Register Model

SPI Register Model		
Register Block		
spi_reg_block		
	Map	Base Address
	APB_map	'h0
	Register	Offset Address
	rxb0_reg	32'h00000000
	rxb1_reg	32'h00000004
	rxb2_reg	32'h00000008
	rxb3_reg	32'h0000000c
	ctrl_reg	32'h00000010
	divider_reg	32'h00000014
	ss_reg	32'h00000018

Notes:

Register Block Example: spi_reg_block -1

```
// spi_reg_block
//-
class spi_reg_block extends uvm_reg_block;
  `uvm_object_utils(spi_reg_block)

  rand rxtx0 rxtx0_reg;
  rand rxtx1 rxtx1_reg;
  rand rxtx2 rxtx2_reg; ← Reg declarations
  rand rxtx3 rxtx3_reg;
  rand ctrl ctrl_reg;
  rand divider divider_reg;
  rand ss ss_reg;

  uvm_reg_map APB_map; // Block map ← Map declaration

  // Wrapped APB register access covergroup
  SPI_APB_reg_access_wrapper SPI_APB_access_cg;

//-
// new
//-
function new(string name = "spi_reg_block");
  super.new(name, build_coverage(UVM_CVR_ADDR_MAP));
endfunction
```

Code from
spi_reg_pkg

Constructor

Notes:

Register Block Example: spi_reg_block -2

```
virtual function void build();  
    ...  
    rxtx0_reg = rxtx0::type_id::create("rxtx0");  
    rxtx0_reg.build();  
    rxtx0_reg.configure(this, null, "");  
    ...  
  
    ctrl_reg = ctrl::type_id::create("ctrl");  
    ctrl_reg.build();  
    ctrl_reg.configure(this, null, "");  
    ...  
  
    APB_map = create_map("APB_map", 'h0, 4, UVM_LITTLE_ENDIAN);  
  
    APB_map.add_reg(rxtx0_reg, 32'h00000000, "RW");  
    APB_map.add_reg(rxtx1_reg, 32'h00000004, "RW");  
    APB_map.add_reg(rxtx2_reg, 32'h00000008, "RW");  
    APB_map.add_reg(rxtx3_reg, 32'h0000000c, "RW");  
    APB_map.add_reg(ctrl_reg, 32'h00000010, "RW");  
    APB_map.add_reg(divider_reg, 32'h00000014, "RW");  
    APB_map.add_reg(ss_reg, 32'h00000018, "RW");  
    add_hdl_path("DUT", "RTL");  
  
    lock_model();  
endfunction
```

Code from
spi_reg_pkg

Register creation,
build and configure

Map creation

Add
registers to
the map

lock_model() finalizes the address map and "locks" the
model from alteration by another user



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

371

Notes:

Register Blocks as Sub-Blocks

- A register block may be used as sub-block in a larger design
 - Combined with other blocks in an integration level or "cluster" block
 - Creates a new register block
- Cluster block incorporates each sub-block
 - Adds them to the cluster block address map
- Process may be repeated such that a top level SoC type register map may contain several layers of register blocks

Notes:

Register Block Example: pss_reg_block

Register Block	Map	Base Address
pss_reg_block	AHB_map	'h0
Sub Block	Sub Map	Offset Address
spi	spi_map	32'h00000000
	Registers	Offset Address
	reg_XXX	32'hXXXXXXXXXX

	reg_YYY	32'hYYYYYYYYYY
Sub Block	Sub Map	Offset Address
gpio	gpio_map	32'h00000100
	Registers	Offset Address
	reg_XXX	32'hXXXXXXXXXX

	reg_YYY	32'hYYYYYYYYYY

Notes:

Register Block Example: pss_reg_block

```
class pss_reg_block extends uvm_reg_block;  
  `uvm_object_utils(pss_reg_block)  
  
  function new(string name = "pss_reg_block");  
    super.new(name);  
  endfunction  
  
  rand spi_reg_block spi;  
  rand gpio_reg_block gpio;  
  
  function void build();  
    AHB_map = create_map("AHB_map", 0, 4, UVM_LITTLE_ENDIAN);  
    spi = spi_reg_block::type_id::create("spi");  
    spi.build();  
    spi.configure(this);  
    AHB_map.add_submap(this.spi.default_map, 0);  
  
    gpio = gpio_reg_block::type_id::create("gpio");  
    gpio.build();  
    gpio.configure(this);  
    AHB_map.add_submap(this gpio.default_map, 32'h100);  
  
    lock_model();  
  endfunction:  
endclass: pss_reg_block
```

Code from spi_reg_pkg

Register block declarations

Map declaration

Create, build and configure the sub-block

Add sub map to the map

lock_model() finalizes the address map and "locks" the model from alteration by another user

Notes:

Multiple Access Example: Mem_ss Register Model

- Sometimes more than one bus interface may need to access the same register block
 - Block contains multiple address maps with alternate mapping

Mem_ss Register Model			
Register Block			
mem_ss_req_block			
Map	Base Address	Map	Base Address
AHB_map	'h0	AHB_2_map	'h0
Register	Offset Address	Register	Offset Address
mem_1_offset	32'h00000000	mem_1_offset	32'h80000000
mem_1_range	32'h00000004	mem_1_range	32'h80000004
mem_2_offset	32'h00000008	mem_2_offset	32'h80000008
mem_2_range	32'h0000000c	mem_2_range	32'h8000000c
mem_3_offset	32'h00000010	mem_3_offset	32'h80000010
mem_3_range	32'h00000014	mem_3_range	32'h80000014
mem_status	32'h00000018	mem_status	32'h80000018
Memory	Offset Address	Memory	Offset Address
mem_1	32'hF0000000	mem_1	32'h70000000
mem_2	32'hA0000000	mem_2	32'h20000000
mem_3	32'h00010000	mem_3	32'h00010000

Notes:

Multiple Interface Access to a Register Block

```
// Memory sub-system (mem_ss) register & memory block
//
class mem_ss_reg_block extends uvm_reg_block;
...
// Mem array configuration registers
rand mem_offset_reg mem_1_offset;
rand mem_range_reg mem_1_range;
...
// Memories
rand mem_1_model mem_1;
...
// Map
uvm_reg_map AHB_map;
uvm_reg_map AHB_2_map;

// continued...
```

Declare multiple maps

Notes:

Multiple Access Example: mem_ss_reg_block

```
function void build();
    mem_1_offset = mem_offset_reg::type_id::create("mem_1_offset");
    mem_1_offset.build();
    mem_1_offset.configure(this, null, "");
    mem_1_range = mem_range_reg::type_id::create("mem_1_range");
    mem_1_range.build();
    mem_1_range.configure(this, null, "");
    ...
    // Create the maps
    AHB_map = create_map("AHB_map", 'h0, 4, UVM_LITTLE_ENDIAN);
    AHB_2_map = create_map("AHB_2_map", 'h0, 4, UVM_LITTLE_ENDIAN);

    // Add registers and memories to the AHB_map
    AHB_map.add_reg(mem_1_offset, 32'h00000000, "RW");
    AHB_map.add_reg(mem_1_range, 32'h00000004, "RW");
    AHB_map.add_mem(mem_1, 32'hF000_0000, "RW");
    ...
    // Add registers and memories to the AHB_2_map
    AHB_2_map.add_reg(mem_1_offset, 32'h8000_0000, "RW");
    AHB_2_map.add_reg(mem_1_range, 32'h8000_0004, "RW");
    AHB_2_map.add_mem(mem_1, 32'h7000_0000, "RW");
    ...
    lock_model();
endfunction: build

endclass: mem_ss_reg_block
```

Annotations:

- Boxed text: Create memories (points to the first two lines of code)
- Boxed text: Add memories to first map (points to the three lines of code under AHB_map)
- Boxed text: Add memories to second map (points to the three lines of code under AHB_2_map)



Notes:

Register Model Integration Overview

- Pieces required for integration
 - Register Model
 - Adaption layer
 - ◆ Translates abstract to/from bus specific access
- Register model is constructed
 - Handle is provided to components and sequences that need access to it
 - ◆ Recommended distribution is via resources as part of a configuration object
- Adaption layer is created and integrated
 - Register layering adapter
 - ◆ Adaption of the sequence based layering
 - Register predictor
 - ◆ Analysis based update of the register model

Notes:

Register Model Construction & Distribution

- Register model is constructed in the test
 - Handle is distributed to the rest of the test bench via configuration DB
 - ◆ In a block level environment a handle to the whole model is passed
 - ◆ In a sub-system or SoC environment handles to sub-blocks of the register model passed to different sub-environments
 - Register model build() is not automatically called
 - ◆ It is not a phase method
 - ◆ Register model is not an uvm_component!
 - ◆ Must be called explicitly



uvm_intro_3.5

© Wilamette HDL Inc.

379

Notes:

Model Construction Example: spi_test_base

```
class spi_test_base extends uvm_test;
  ...
  // The environment class
  spi_env m_env;
  // Configuration objects
  spi_env_config m_env_cfg;
  ...
  spi_reg_block spi_rm; // Register block
  ...
  extern function void build_phase(uvm_phase phase);
endclass: spi_test_base

function void spi_test_base::build_phase(uvm_phase phase);
  ...
  // env configuration
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Register model
  // Enable all types of coverage available in the register model
  uvm_reg::include_coverage("*", UVM_CVR_ALL);
  // Create the register model:
  spi_rm = spi_reg_block::type_id::create("spi_rm");
  // Build and configure the register model
  spi_rm.build(); ←
  // Set the handle to the register model in config DB
  uvm_config_db #(spi_reg_block)::set(this, "m_env*", "spi_reg_block", spi_rm);
  ...
end
```

Note explicit call to the
build() method of the
register model



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

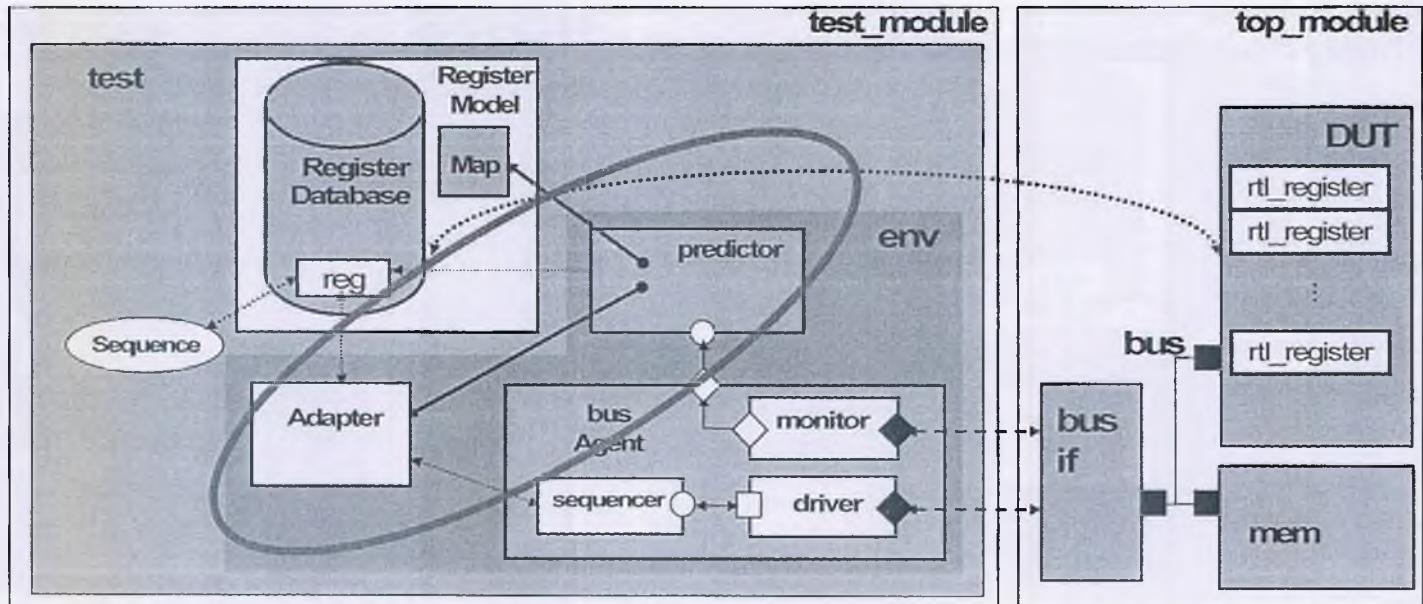
380

Notes:

Adaption Layer Implementation

■ Adaption Layer

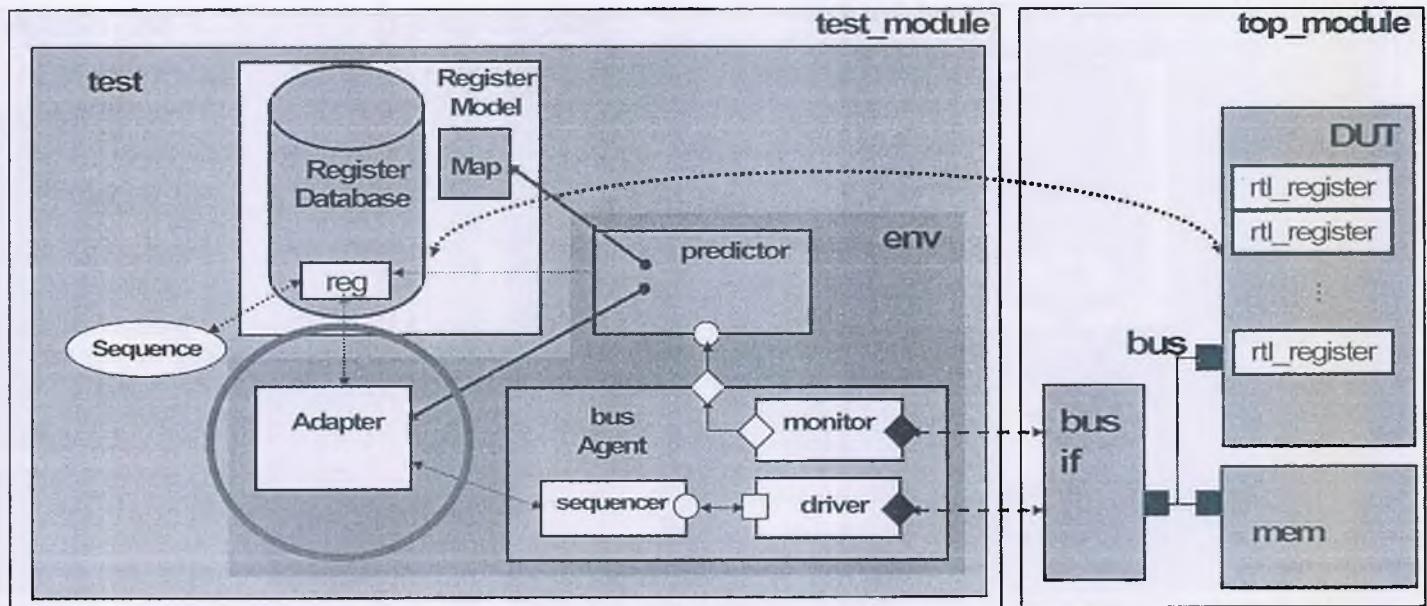
- Register layering adapter implements the sequence based stimulus
- Predictor implements the analysis based update of the register model



Notes:

Register Layering Adapter

- Translates generic register transactions into target bus transactions and vice versa
- Should be considered part of the agent and supplied as part of the agent package



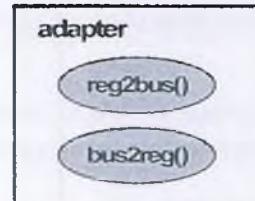
Notes:

Creating an Adapter

- Inherit from the abstract base class `uvm_reg_adapter`
- Override the inherited translation methods
 - `reg2bus()`
 - ◆ Override to convert generic register access items to bus sequence items
 - `bus2reg()`
 - ◆ Override to convert bus sequence items to register model items



```
pure virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);  
pure virtual function void bus2reg(uvm_sequence_item bus_item,  
                                 ref uvm_reg_bus_op rw);
```



uvm_intro_3.5

© Willamette HDL, Inc.

383

Notes:

uvm_reg_bus_op Type

- The register model uses a generic read/write bus operation descriptor or type
 - uvm_reg_bus_op
 - Implemented as a struct, not a class, to minimize memory resource usage
 - Contains 6 fields that need to be mapped to the target bus sequence item

Type	Field	Description
uvm_reg_addr_t	addr	Address field, defaults to 64 bits
uvm_reg_data_t	data	Read or write data, defaults to 64 bits
uvm_access_e	kind	UVM_READ, UVM_WRITE, UVM_BURST_WRITE, UVM_BURST_READ
unsigned int	n_bits	Number of bits being transferred
uvm_reg_byte_en_t	byte_en	Byte enable
uvm_status_e	status	UVM_IS_OK, UVM_IS_X, UVM_NOT_OK

```
typedef bit unsigned [`UVM_REG_DATA_WIDTH-1:0] uvm_reg_data_t ;
typedef bit unsigned [`UVM_REG_ADDR_WIDTH-1:0] uvm_reg_addr_t ;
```

Notes:

Creating an Adapter – Inherited Properties

- Two inherited property bits that need to be set according to the functionality of the target bus
 - `supports_byte_enable`
 - ◆ Set to 1 if the target bus supports byte enables, else 0
 - `provides_responses`
 - ◆ Used by the register model to determine how to communicate with the bus sequencer for responses
 - ◆ Default value is 0 meaning responses from the driver are provided implicitly
 - ◆ Value of 1 means responses are provided explicitly by the driver through the sequencer
 - ◆ The value of this bit MUST match how the driver actually works
 - ◆ Details as to how the RSP is returned will follow later during discussion of the register model API methods like `read()`



Recommended: Use `provides_responses = 0`

Notes:

Example Adapter: reg2apb_adapter

```
class reg2apb_adapter extends uvm_reg_adapter;
  `uvm_object_utils(reg2apb_adapter)

  function new(string name = "reg2apb_adapter");
    super.new(name);
  endfunction

  function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    apb_seq_item apb = apb_seq_item::type_id::create("apb");
    apb.we = (rw.kind == UVM_READ) ? 0 : 1;
    apb.addr = rw.addr;
    apb.data = rw.data;
    return apb;
  endfunction: reg2bus

  function void bus2reg(uvm_sequence_item bus_item,
                        ref uvm_reg_bus_op rw);
    apb_seq_item apb;
    if (!$cast(apb, bus_item)) begin
      `uvm_fatal("NOT_APB_TYPE", "Provided bus_item is not of the correct type")
      return;
    end
    rw.kind = apb.we ? UVM_WRITE : UVM_READ;
    rw.addr = apb.addr;
    rw.data = apb.data;
    rw.status = UVM_IS_OK;
  endfunction: bus2reg

endclass
```

apb_seq_item	
Type	Field
logic[31:0]	addr
logic[31:0]	data
logic	we
int	delay



Notes:

Register Prediction

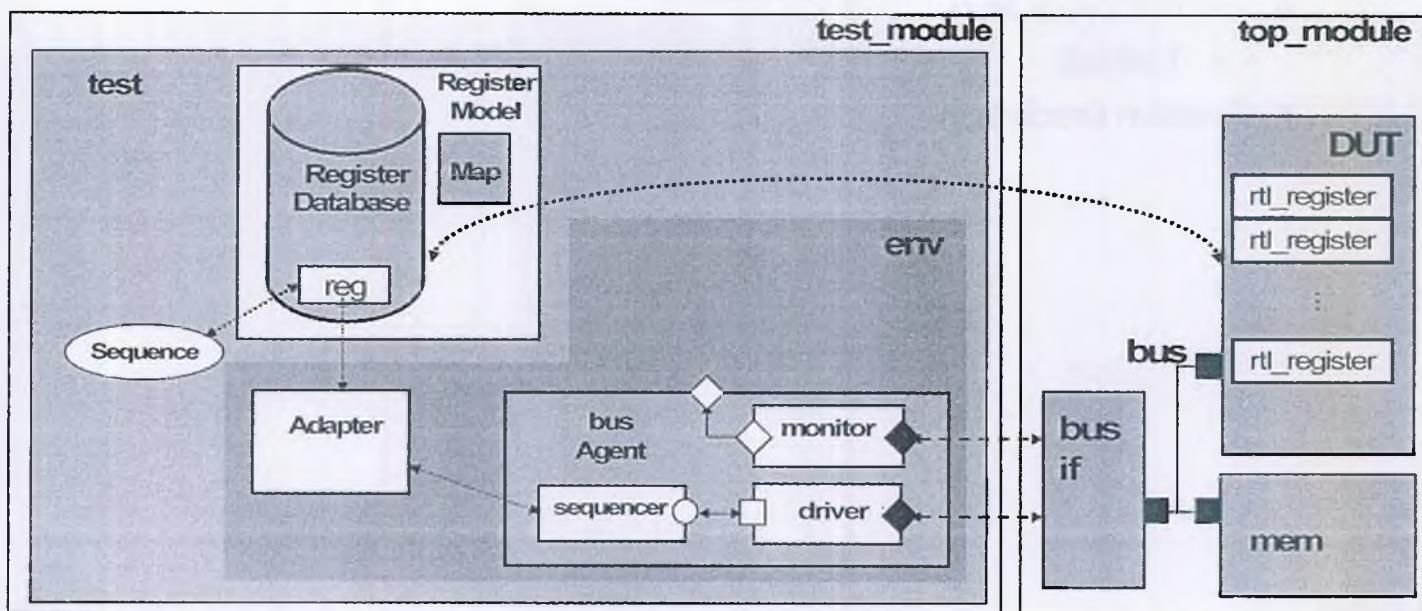
- Every time the register model generates a read or write transaction the register data base needs to be updated to be in sync with the DUT
- There are three different approaches for keeping the register model's mirror values in sync with the DUT
 - Automatic prediction (or implicit prediction)
 - Explicit prediction
 - ◆ Default
 - Passive prediction

Notes:

Automatic (Implicit) Prediction

- Must call method to set to automatic prediction mode:
 - `set_auto_predict(1);`
- Updates to the mirror are predicted automatically at completion of the various access methods

Note no predictor



Notes:

Automatic (Implicit) Prediction Features

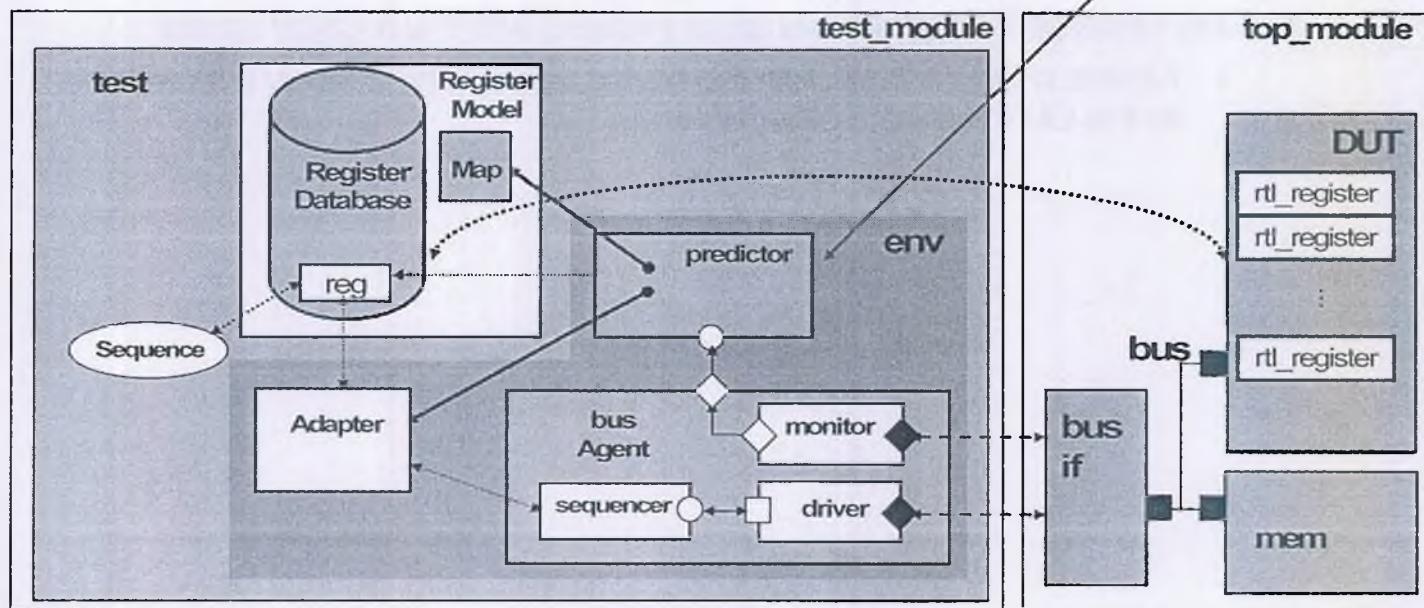
- Access methods call a predict () method
 - predict () uses data that was written to the register or data read back from the register at the end of the bus cycle
- Simplest and quickest to implement
- Drawback:
 - Only updates for register transfers initiated with the register model
 - ◆ Means a failure to update the mirror value appropriately if accesses to the DUT are from other sources

Notes:

Explicit Prediction

- The default mode of operation
 - Can set explicitly with call to method:
 - ◆ `set_auto_predict(0);`
- Recommended approach
- Updates are predicted externally to the register model

Data updated
based on observed
bus transactions

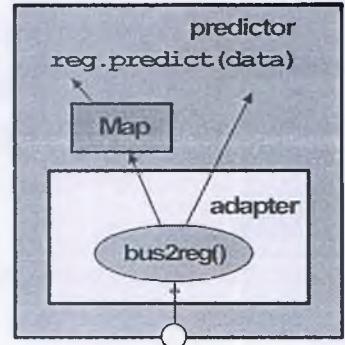


Notes:

Explicit Prediction - 2

- Explicit prediction is made by a combination of components

- Bus monitor
- uvm_reg_predictor library component
 - ◆ Specialized using the bus sequence_item type
 - ◆ Created in the env build_phase () method
 - ◆ Contains:
 - map: handle to a map in the register model
 - Used to convert bus addresses to the corresponding register handle
 - Set this property explicitly after creating the predictor
 - adapter: handle to an adapter which converts bus transactions to register transactions
 - Set this property explicitly after creating the predictor
 - Analysis export (bus_in) to receive broadcast data from the monitor
 - ◆ Calls the register's predict () method to update the mirror



Notes:

Setting Map Properties

- The map has two properties that must be set
 - Agent sequencer handle (`m_sequencer`)
 - Adapter handle (`m_adapter`)
 - Use the `set_sequencer()` method to set these properties

UVM

```
virtual function void set_sequencer (uvm_sequencer_base sequencer,  
                                     uvm_reg_adapter     adapter=null)
```

Map	
Offset	Reg/Mem/Block
0x00	R1
0x04	R2
0x1000	Mem
Agent sequencer handle	
Adapter handle	

**WILLAMETTE
HDL**

uvm_intro_3.5

© Willamette HDL Inc.

392

Notes:

Adaption Layer Creation Example: spi_env -1

```
class spi_env extends uvm_env;
  ...
  spi_reg_block spi_rm;

  // Register layering adapter:
  reg2apb_adapter reg2apb;
  // Register predictor:
  uvm_reg_predictor #(apb_seq_item) apb2reg_predictor;
  ...
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);
endclass:spi_env

function void spi_env::build_phase(uvm_phase phase);
  m_cfg = spi_env_config::get_config(this);
  // get register model handle
  if(!uvm_config_db #(spi_reg_block)::get(this, "", "spi_reg_block", spi_rm))
    `uvm_error("CONFIG", "spi_rm not found");
  if(m_cfg.has_apb_agent) begin
    set_config_object("m_apb_agent*", "apb_agent_config", m_cfg.m_apb_agent_cfg, 0);
    m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);
    // Build the register model predictor
    apb2reg_predictor = uvm_reg_predictor
      #(apb_seq_item)::type_id::create("apb2reg_predictor", this);
  // create the adapter
  reg2apb = reg2apb_adapter::type_id::create("reg2apb");
  ...
endfunction
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/spi_reg

393

Notes:

Adaption Layer Creation Example: spi_env -2

```
function void spi_env::connect_phase(uvm_phase phase);
  if(m_cfg.m_apb_agent_cfg.active == UVM_ACTIVE) begin
    // adapter
    reg2apb.provides_responses = 0; // does not have separate responses

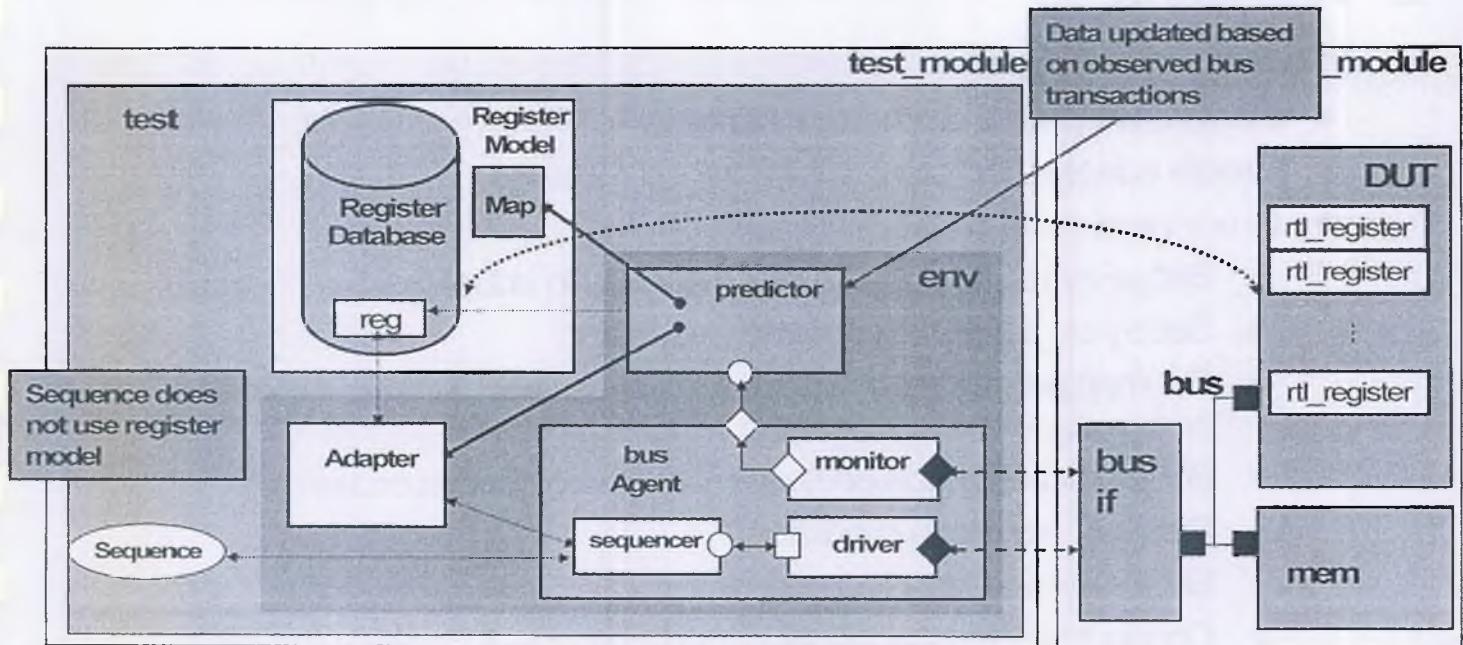
    // set the sequencer
    spi_rm.APB_map.set_sequencer(m_apb_agent.m_sequencer, reg2apb);
    // Disable the register models auto-prediction
    spi_rm.APB_map.set_auto_predict(0);

    // Register prediction part:
    // Replacing implicit register model prediction with explicit prediction
    // Set the predictor map:
    apb2reg_predictor.map = spi_rm.APB_map;
    // Set the predictor adapter:
    apb2reg_predictor.adapter = reg2apb;
    // Connect the predictor to the bus agent monitor analysis port
    m_apb_agent.ap.connect(apb2reg_predictor.bus_in);
    ...
  end
```

Notes:

Passive Prediction

- Register model is not used for DUT access
- Register model is updated by the predictor when bus activity takes place
 - Same update mechanism as explicit predictor model



Notes:

Integration Checklist

1. In the test class:

- Declare register model handle
- Create register model in the `build_phase()`
- Set register model handle in configuration object

2. Write adapter

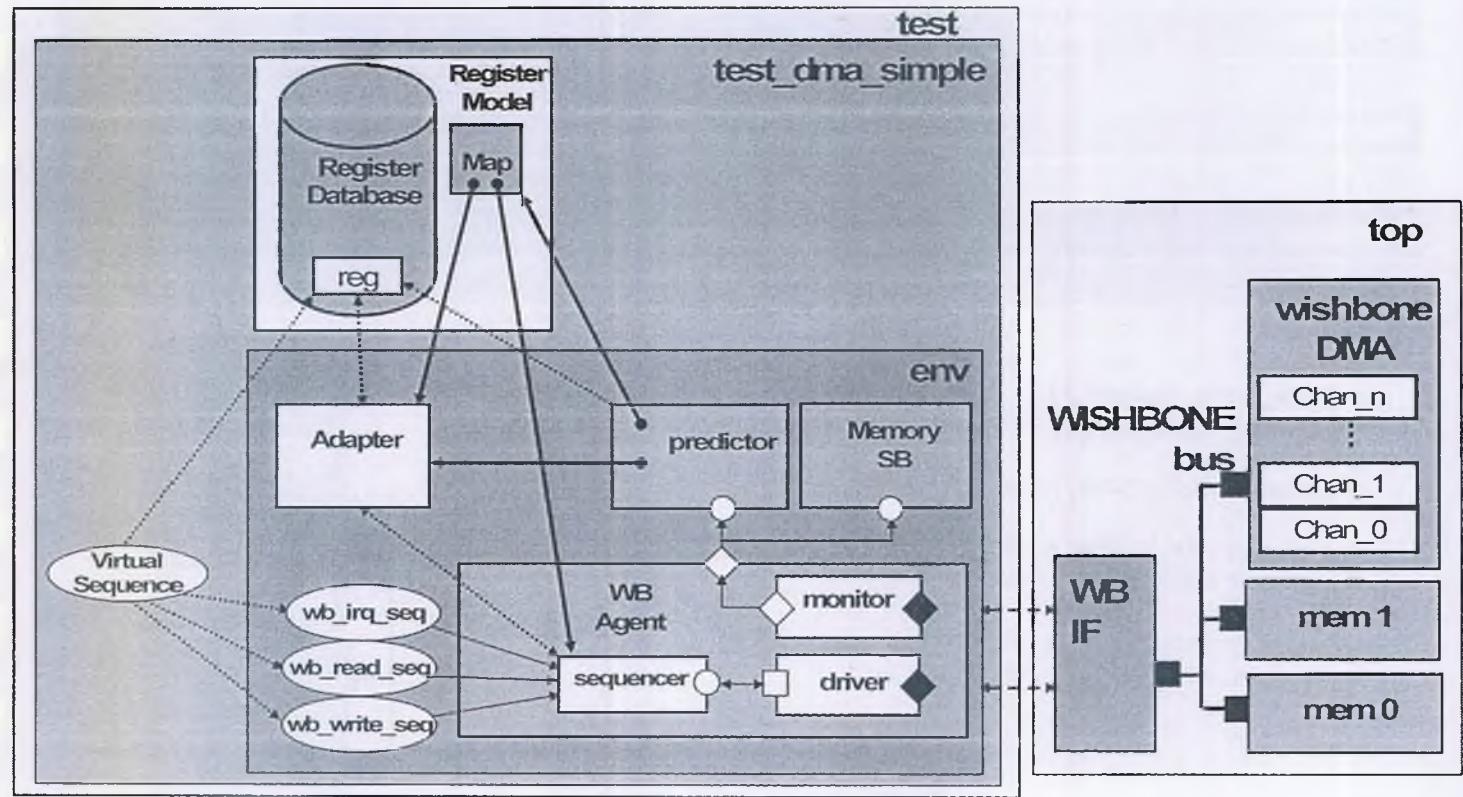
3. In environment class

- Declare adapter and predictor handles
- Create adapter and predictor in `build_phase()`
- In `connect_phase()`:
 - ◆ Set `provides_responses` property in adapter
 - ◆ Set `byte_enable` property in adapter
 - ◆ Set map properties `m_adapter` and `m_sequencer` by calling `set_sequencer()` on map
 - ◆ Set prediction mode on map (`set_auto_prediction()`)
 - ◆ Set map handle in predictor
 - ◆ Set adapter handle in predictor
 - ◆ Connect predictor analysis export `bus_in` to agent's (monitor) analysis port

Notes:

Lab – Integration Overview

- Create and integrate the register model for the WISHBONE DMA



Notes:

Lab – Integrating Registers: WB Register Model

Block	Map	Base Address
dma_system_block	dma_system_map	'h0
Sub Block	Sub Map	Offset Address
dma_top	dma_top_map	DMA_SLAVE_0_WB_ID * SLAVE_ADDR_SPACE_SZ
Memory		Offset Address
mem0		MEM_SLAVE_0_WB_ID * SLAVE_ADDR_SPACE_SZ
Memory		Offset Address
mem1		MEM_SLAVE_1_WB_ID * SLAVE_ADDR_SPACE_SZ

```

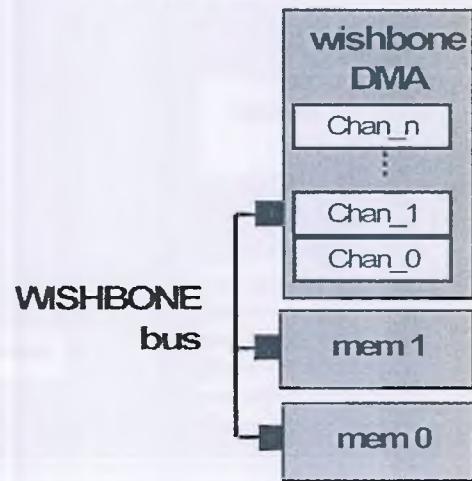
package test_params_pkg;
parameter SLAVE_ADDR_SPACE_SZ = 32'h10000000;

parameter DMA_SLAVE_0_WB_ID = 2;

parameter MEM_SLAVE_0_SIZE = 32'h00100000;
parameter MEM_SLAVE_0_WB_ID = 0;

parameter MEM_SLAVE_1_SIZE = 32'h00100000;
parameter MEM_SLAVE_1_WB_ID = 1;
...
endpackage

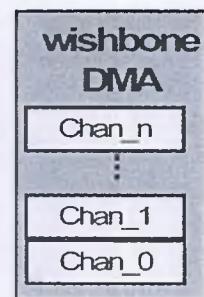
```



Notes:

Lab – Integrating Registers: DMA Register Block

Block	Map	
dma_top	dma_top_map	
Sub Block	Map	Offset Address
dma_main	dma_main_map	'h0
Sub Block	Registers	Offset Address
	CSR_MAIN	32'h00000000
	INT_MSK_A	32'h00000004
	INT_MSK_B	32'h00000008
	INT_SRC_A	32'h0000000c
	INT_SRC_B	32'h00000010
Sub Block	Sub Map	Offset Address
CH_0	dma_channel_map	32'h00000020
Sub Block	Registers	Offset Address
	CHn_CSR_reg	32'h00000000
	CHn_SZ_reg	32'h00000004
	CHn_A0	32'h00000008
	CHn_AM0	32'h0000000c
	CHn_A1	32'h00000010
	CHn_AM1	32'h00000014
	CHn_DESC_reg	32'h00000018
	CHn_SWPTR_reg	32'h0000001c



Notes:

Lab – Integration: Instructions -1

- NOTE: For this lab the test used is `test_dma_simple`
- Working Directory: `register_integration`
- Create the register model in the test class
 - Edit the file `tests/test_dma_base.svh`
 - ◆ Declare a register model handle called `m_dma_system_block` of type `dma_system_block`
 - Diagram 2 slides ago shows the block `dma_system_block`
 - ◆ Build the register model
 - ◆ Set the handle `m_dma_system_block` in the configuration DB with a lookup of "dma_system_block" and a scope that allows only the environment class (declared earlier in the class) to access it
 - ◆ Set the handle to the sub block `dma_top` in the `m_dma_system_block` block in the configuration DB with a lookup of "dma_top_block" and a scope that allows anything below the environment class to access it

Continue on next slide...

Notes:

Lab – Integration Instructions - 2

- Define the adapter
 - Edit the adapter file **wishbone_bus/reg2wb_adapter.svh**
 - ◆ Write the adapter `reg2wb_adapter`
 - Add constructor
 - Override the `reg2bus()` method `bus2reg` methods
 - Translation from `uvm_reg_bus_op` to `wb_txn`
 - Translation from `wb_txn` to `uvm_reg_bus_op`
 - `wb_txn` is found in `wishbone_bus/wb_txn.svh`.
 - NOTE: look at what the constructor does and what methods are available
 - NOTE: Make sure all `wb_txn` properties are set (like `count`)

Continue on next slide...

Notes:

Lab – Integration Instructions - 3

- Instantiate and connect up adapter and predictor
 - Edit the environment file `env/wb_env.svh`
 - ◆ Declare predictor and adapter handles
 - ◆ Declare a handle of type `dma_system_block`
 - ◆ In the `build_phase()` method
 - Get the `dma_system_block` handle from the configuration DB and assign it to the property you just declared
 - Note you set this in the `test_dma_base` class
 - Create predictor
 - Create the adapter

Continue on next slide...

Notes:

Lab – Integration Instructions -4

- ◆ In the `connect_phase()` method
 - Initialize the adapter
 - Set the `provides_responses` bit to 0
 - Initialize the register model
 - Set the `dma_system_block` map's (`dma_system_map`) sequencer handle to the sequencer (`wb_seqr`) in the WB agent (agent) and set the map's adapter handle to the adapter
 - Note: The WB Register Model slide (several ago) shows the structure of the `dma_system_block`
 - Initialize the predictor
 - Set predictor map handle (`map`) to the map (`dma_system_map`) in register model
 - Set predictor adapter handle to the adapter
 - Turn auto-prediction off
 - Connect the predictors export (`bus_in`) to the WB agent's (agent) analysis port (`mon_ap`)

■ Compile and run



uvm_intro_3.5

© Willamette HDL Inc.

403

Notes:

Lab – Integration: Sample Output

```
# UVM_INFO @ 2330: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer complete on channel 0
# UVM_INFO @ 2530: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer data correct
# UVM_INFO @ 4530: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer complete on channel 1
# UVM_INFO @ 4890: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer data correct
# UVM_INFO @ 11930: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer complete on channel 2
# UVM_INFO @ 13970: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer data correct
# UVM_INFO @ 15490: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer complete on channel 3
# UVM_INFO @ 15690: uvm_test_top.env.agent.wb_seqr@em_seq [DMA_SIMPLE_SEQ] DMA transfer data correct
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] ****
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] Final Memory Coverage = 0.000000%
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] *****
```

```
...
# UVM_INFO ./wishbone_bus/wb_mem_scoreboard.svh(115) @ 15690: uvm_test_top.env.wb_mem_sb_0 [MEM_SB_0]
# Number of Wishbone 0 Slave Memory write transactions: 132
# Number of Wishbone 0 Slave Memory read transactions: 132
# Number of Wishbone 0 Non-Slave Memory write cycles: 153
# Number of Wishbone 0 Non-Slave Memory read cycles: 170
# Wishbone 0 Slave Memory read error count: 0
#
# UVM_INFO ./wishbone_bus/wb_mem_scoreboard.svh(115) @ 15690: uvm_test_top.env.wb_mem_sb_1 [MEM_SB_1]
# Number of Wishbone 1 Slave Memory write transactions: 132
# Number of Wishbone 1 Slave Memory read transactions: 132
# Number of Wishbone 1 Non-Slave Memory write cycles: 153
# Number of Wishbone 1 Non-Slave Memory read cycles: 170
# Wishbone 1 Slave Memory read error count: 0
```

Should see all 4 DMA transfers completed with correct transfer data

Exact output may vary by simulator and revision

Should see no errors in the report summary



uvm_intro_3.5

© Willamette HDL Inc

Sol

404

Notes:

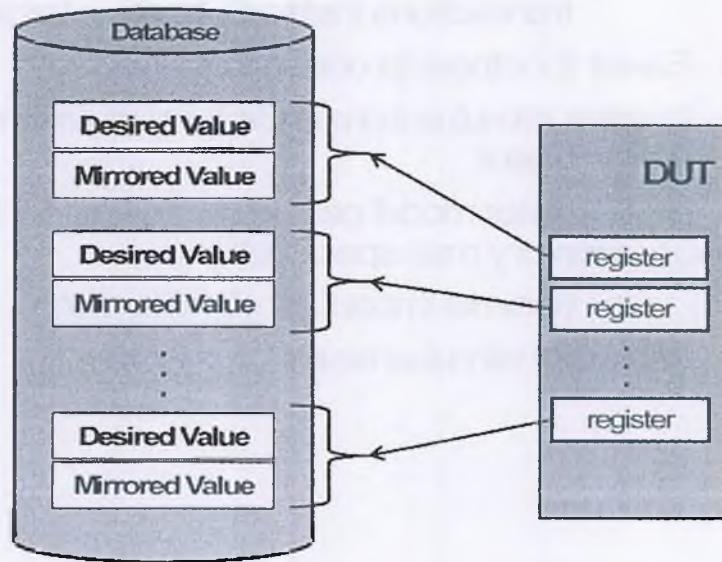
Using a Register Model

- One of the purposes of a register model is to enable the abstraction of stimulus
 - Easier to implement
 - ◆ Register model allows access to registers and fields by name
 - ◆ Register model is integrated with bus agents in testbench
 - Stimulus writer uses register methods to generate bus transactions instead of writing target bus agent sequences items
 - Easier for others to understand
 - Isolates stimulus from underlying register map changes during development
 - ◆ Register model can be regenerated if there is a change in the memory map specification
 - Minimal impact on stimulus code
 - Easier for stimulus reuse

Notes:

Data Value Tracking

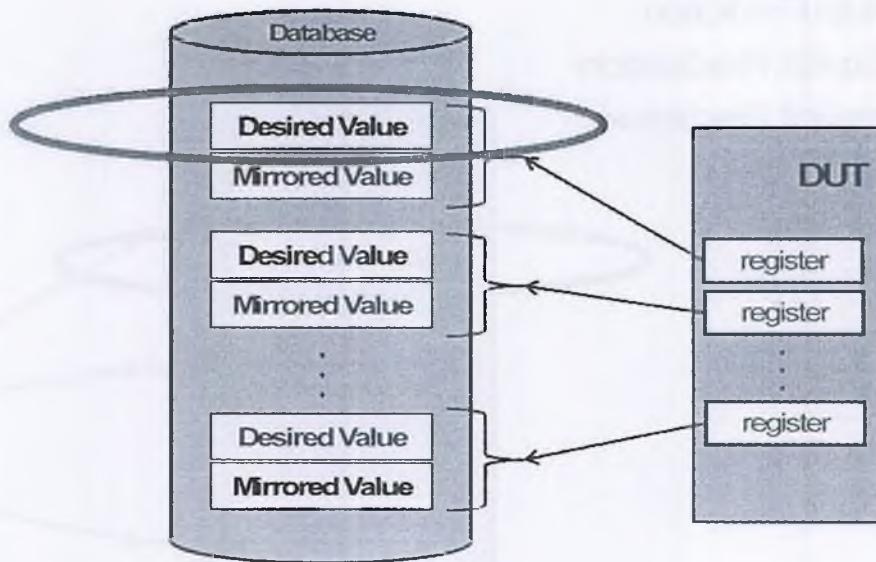
- Register model has its own database
 - Represents the state of the hardware registers
- For each hardware register the database has two values
 - Desired value
 - Mirror value



Notes:

Register Model Desired Value

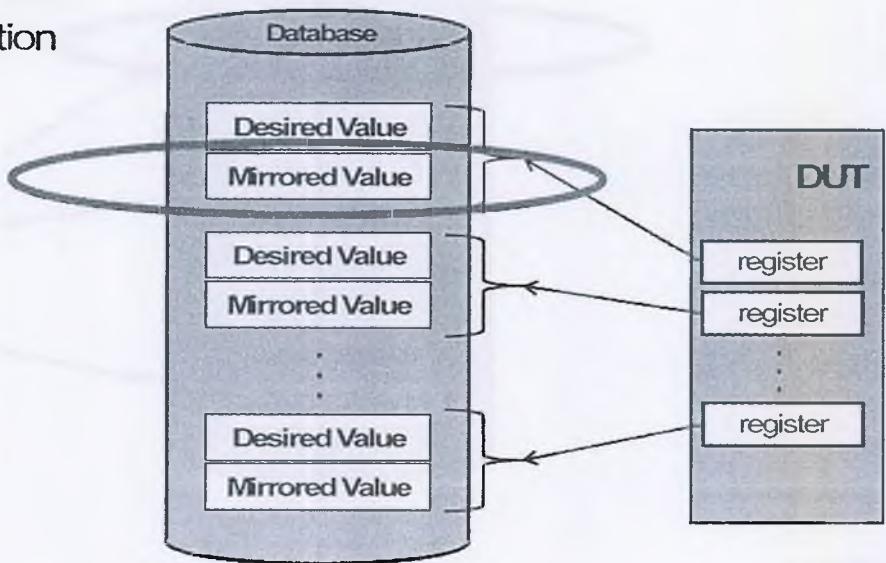
- Represents a state that the register model will use to update the hardware but has not yet done so
 - Allows for setup of register and register fields before doing a write transfer



Notes:

Register Model Mirror Value - 1

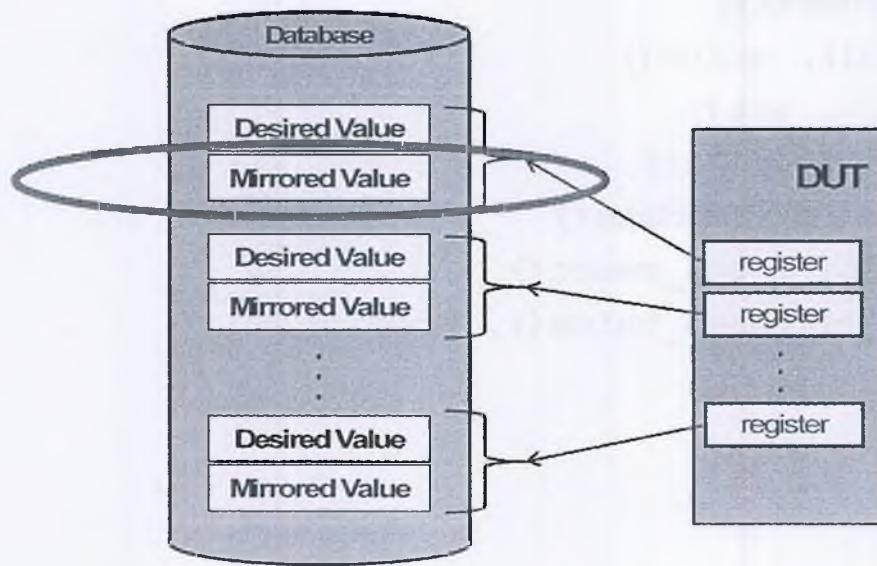
- Represents the current known state of the hardware register
- For front door access
 - Is updated at the end of front end bus read and write cycles
 - ◆ Update mechanism depends upon update mode
 - Auto Prediction
 - Explicit Predication
 - Implicit Prediction



Notes:

Register Model Mirror Value - 2

- For back door access
 - Register model is updated automatically
- Over time mirror value can become dated if bits of a register are volatile
 - Changed based on hardware events as opposed to being programmed



Notes:

Register Access Methods

- A number of access methods are provided for use in reading and writing DUT registers
 - Use the desired and mirrored values to keep in sync with the hardware registers
- Methods are provided for both front door and back door access
- Access methods
 - `read()`, `write()`
 - `get()`, `set()`
 - `peek()`, `poke()`
 - `mirror()`, `update()`
 - `reset()`, `get_reset()`
 - `get_mirrored_value()`, `predict()`

Notes:

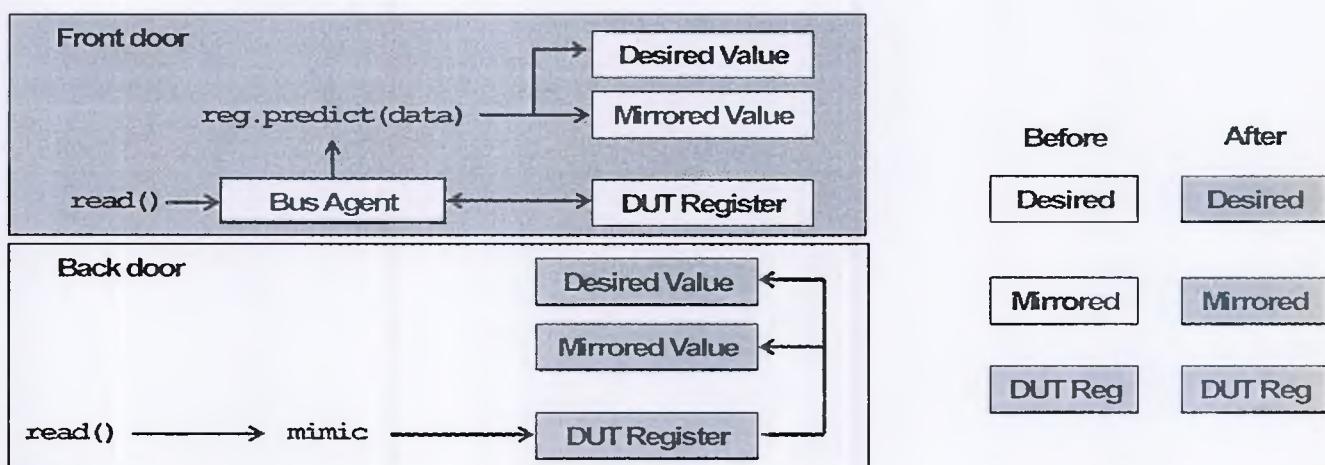
Register Field Level Access

- Only works (well) if the field of a register maps to and fills up an entire byte lane
 - Assumes byte level access on the bus
- Another issue is portability of stimulus if a second target bus does not support byte level access
- So generally not a good idea

Notes:

read() Method

- Returns the value of the hardware register
- Front door read():
 - A bus transfer is executed on the DUT register
 - Desired and mirrored values are updated by the bus predictor (using predict()) on completion of the read cycle
- Back door read():
 - ◆ Mirrored value is updated immediately



Notes:

read() Method Prototype & Example

UVM

```
task read(output uvm_status_e      status,
          output uvm_reg_data_t    value,
          input  uvm_path_e        path = UVM_DEFAULT_PATH,
          input  uvm_reg_map       map = null,
          input  uvm_sequence_base parent = null,
          input  int                prior = -1,
          input  uvm_object         extension = null,
          input  string             fname = "",
          input  int                lineno = 0);
```

```
spi_rm.ctrl.read(status, read_data);
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

413

Notes:

Register Access Method Arguments

Argument	Type	Default Value	Purpose
status	uvm_status_e	None	To return the status of the method call - can be UVM_IS_OK, UVM_NOT_OK, UVM_IS_X
value	uvm_reg_data_t	None	To pass a data value, an output in the read direction, an input in the write direction
path	uvm_path_e	UVM_DEFAULT_PATH	To specify whether a front or back door access is to be used - can be UVM_FRONTDOOR, UVM_BACKDOOR, UVM_PREDICT, UVM_DEFAULT_PATH
map	uvm_reg_map	null	Specify which register model map to use to make the access
parent	uvm_sequence_base	null	Specify parent sequence of the method. Typical to use the default value of null
prior	int	-1	Specify the priority of the sequence item on the target sequencer
extension	uvm_object	null	Allows an object to be passed in order to extend the call
fname	string	""	Used by reporting to tie method call to a file name
lineno	int	0	Used by reporting to tie method call to a line number
kind	string	"HARD"	Used to denote the type of reset



Notes:

parent Argument

- For front-door operations the register model communicates with a sequencer
 - Since the register model is not a sequence it uses a "proxy" sequence for the communication
- The parent argument determines the proxy sequence
 - null
 - ◆ The register model creates its own proxy sequence
 - this
 - ◆ The register model uses the calling sequence (the sequence making the `read()`, `write()` etc. call) as its proxy
 - Recommend using the default value of `null`



Recommended: Use default of `.parent(null)`

Notes:

provides_responses set to 0

- If the `provides_responses` bit is set to 0
 - The register model expects an implicit response for **every** bus transaction
 - Driver provides the response implicitly through the REQ item
 - ◆ Response data and status is put into the REQ item
 - ◆ No RSP item is created or used
 - Recommendation in the driver is to use TLM methods:

```
seq_item_port.get_next_item(REQ) // get transaction  
// indicate transaction complete and that response info is in  
// the REQ item  
seq_item_port.item_done()
```



Recommended: Use `provides_responses = 0`

Notes:

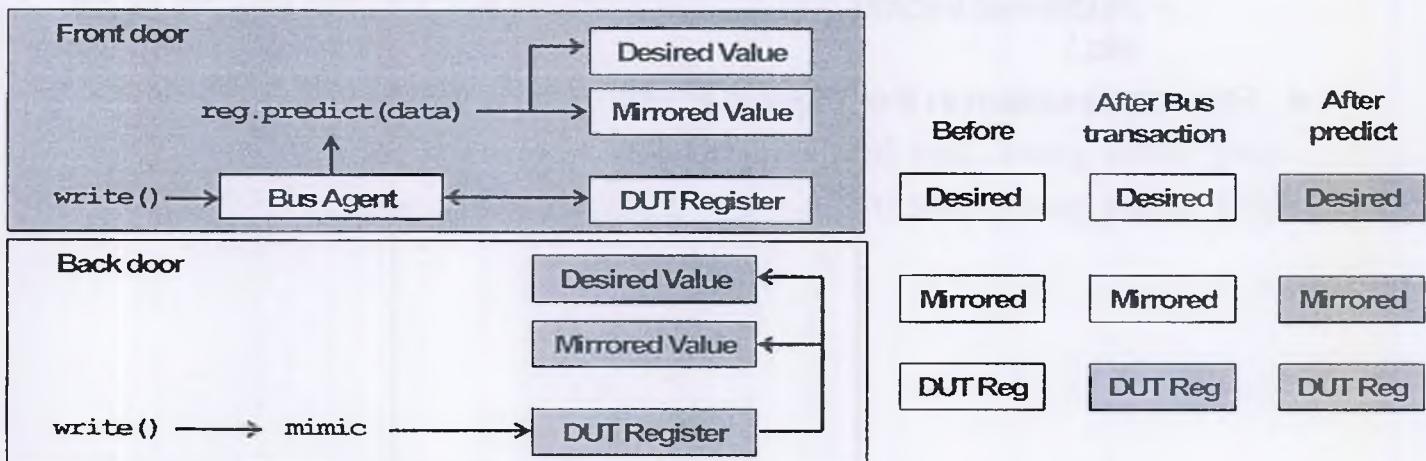
provides_responses set to 1

- If the `provides_responses` bit is set to 1
 - The register model expects the driver to explicitly return a response through the sequencer for every bus transaction
 - Driver must return a response for every bus transaction regardless of the type
 - ◆ If not then the register model will hang waiting for the response
 - ◆ I.e. for every REQ there must be an RSP
 - ◆ The generating sequence will always get status information
 - Additional information depends upon what type the REQ is (read etc.)
 - Recommendation in the driver is to use TLM methods:
`seq_item_port.get(REQ); // get transaction`
`seq_item_port.put(RSP); // return response`

Notes:

write() Method

- Writes a specified value to the hardware register
- Front door write():
 - Front door access:
 - ◆ A bus transfer is executed on the DUT register
 - ◆ Desired and mirrored values are updated by the bus predictor on completion of the write cycle
 - Back door access:
 - ◆ Desired and mirrored values are updated immediately



Notes:

write() Method Prototype & Example

UVM

```
task write(output uvm_status_e      status,
           input  uvm_reg_data_t   value,
           input  uvm_path_e       path = UVM_DEFAULT_PATH,
           input  uvm_reg_map      map = null,
           input  uvm_sequence_base parent = null,
           input  int               prior = -1,
           input  uvm_object        extension = null,
           input  string            fname = "",
           input  int               lineno = 0);  
  
spi_rm.ctrl.write(status, write_data, .parent(null));
```

Before	After Bus transaction	After predict
Desired	Desired	Desired
Mirrored	Mirrored	Mirrored
DUT Reg	DUT Reg	DUT Reg



uvm_intro_3.5

© Willamette HDL Inc.

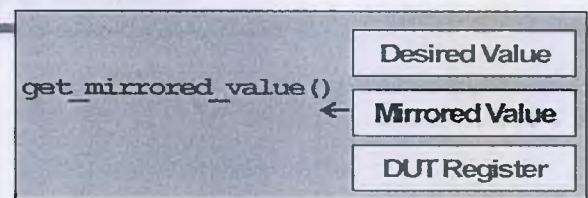
code in examples/spi_reg

419

Notes:

get_mirrored_value() Method

- Operates only on the register data base
 - Returns the mirror value of a register or field
 - Does not result in any bus transfer or access to the hardware



UVM

```
function uvm_reg_data_t uvm_reg::get_mirrored_value(string fname = "",  
                                                 int    lineno = 0);  
  
    uvm_reg_data_t ctrl_value;  
    uvm_reg_data_t char_len_value;  
  
    // Register level get:  
    ctrl_value = spi_rm.ctrl.get_mirrored_value();  
  
    // Field level get (char_len is a field within the ctrl reg):  
    char_len_value = spi_rm.ctrl.char_len.get_mirrored_value();
```



WILLAMETTE
HDL

uvm_intro_3.5

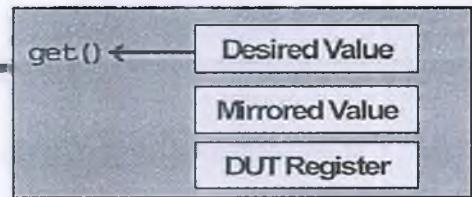
©Willamette HDL Inc

code in examples/spi_reg

420

Notes:

get () Method



- Operates only on the register data base
 - Returns the desired value of a register or field
 - Does not result in any bus transfer or access to the hardware

UVM

```
function uvm_reg_data_t get(string fname = "", int lineno = 0)
```

Before	After
Desired	Desired
Mirrored	Mirrored
DUT Reg	DUT Reg

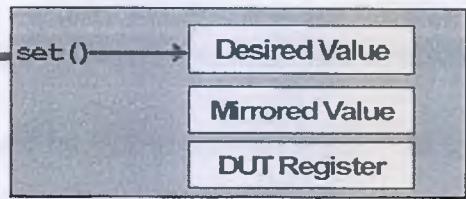
```
uvm_reg_data_t ctrl_value;
uvm_reg_data_t char_len_value;

// Register level get:
ctrl_value = spi_rm.ctrl.get();

// Field level get (char_len is a field within the ctrl reg):
char_len_value = spi_rm.ctrl.char_len.get();
```

Notes:

set () Method



- Operates only on the register data base
 - Sets the desired value of a register or field
 - Does not result in any bus transfer or access to the hardware

UVM

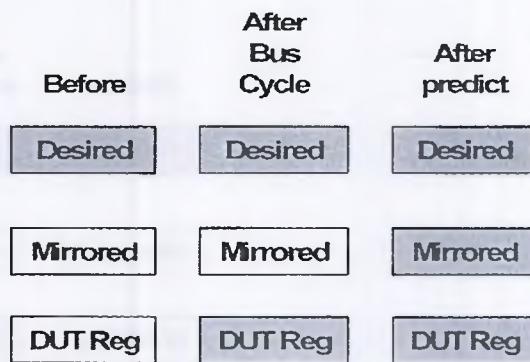
```
function void set(uvm_reg_data_t value, string fname = "", int lineno = 0);  
  
    uvm_reg_data_t ctrl_value;  
    uvm_reg_data_t char_len_value;  
  
    // Register level set:  
    spi_rm.ctrl.set(ctrl_value);  
  
    // Field level set (char_len is a field within the ctrl reg):  
    spi_rm.ctrl.char_len.set(ctrl_value);
```



Notes:

update () Method

- Initiates a write to the hardware
 - Only if there is a difference in value between desired and mirrored values
 - ◆ Perhaps because of set () or randomize ()
 - A use model is to use set () to write values in different fields of the desired value and then use update () to initiate the write to the register
 - Block level call could result in multiple register writes
- Mirrored value is set to the updated value at the completion of the bus write cycle



Notes:

update() Method Prototype & Example

UVM

```
task update(output uvm_status_e           status,
            input  uvm_path_e          path = UVM_DEFAULT_PATH,
            input  uvm_sequence_base   parent = null,
            input  int                 prior = -1,
            input  uvm_object          extension = null,
            input  string              fname = "",
            input  int                 lineno = 0);
```



```
// Block level:  
spi_rm.update(status);  
//  
// Register level:  
spi_rm.ctrl.update(status);
```

Before	After Bus transaction	After predict
Desired	Desired	Desired
Mirrored	Mirrored	Mirrored
DUT Reg	DUT Reg	DUT Reg

Notes:

mirror() Method

- Initiates a front door read or peek access of the hardware
 - Does not return the hardware data value
 - Front door read
 - ◆ Predictor updates the mirrored and desired
 - Peek
 - ◆ Automatically updates the mirrored and desired
 - There is an option to check the value read back from the hardware with the original mirrored value (check argument)
- May be called at the field, register or block level
 - Typically only at the register or block level
 - Block level mirror() call results in read/peek accesses to all the registers within the block



Notes:

mirror() Method Prototype & Example

UVM

```
task mirror(output uvm_status_e           status,
            input   uvm_check_e          check = UVM_NO_CHECK,
            input   uvm_path_e           path  = UVM_DEFAULT_PATH,
            input   uvm_sequence_base    parent = null,
            input   int                  prior = -1,
            input   uvm_object           extension = null,
            input   string               fname = "",
            input   int                  lineno = 0);
```



```
// Check the contents of the ctrl register
spi_rm.ctrl.mirror(status, UVM_CHECK);
```



```
// Mirror the contents of spi_rm block via the backdoor
spi_rm.mirror(status, .path(UVM_BACKDOOR));
```

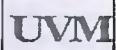
Notes:

predict () Method

- Updates the mirror value for the register (or field)
 - Updates based on the specified observed value on a bus using the specified address map
 - The source of the value argument is determined by the kind argument
 - ◆ UVM_PREDICT_READ: the value was observed in a read transaction
 - ◆ UVM_PREDICT_WRITE: the value was observed in a write transaction
 - ◆ UVM_PREDICT_DIRECT: the value was computed and is updated as-is
 - Used to set the mirror value to a user desired value
- Returns true if the prediction was successful for each field in the register
- Called in the predictor to update the register mirror on reads & writes
- To check a predicted value (generated by a reference model) against the actual value of a register
 - Use predict () to set the mirrored value to the expected value
 - Then call mirror () with check arg set to check for a mismatch

Notes:

predict () Method Prototype & Example



```
function bit predict(      uvm_reg_data_t    value,
                          uvm_reg_byte_en_t be = -1,
                          uvm_predict_e     kind = UVM_PREDICT_DIRECT,
                          uvm_path_e       path = UVM_FRONTDOOR,
                          uvm_reg_map      map = null,
                          string           fname = "",
                          int              lineno = 0);;
```



```
// Set the mirror of the ctrl register
void'(spi_rm.ctrl.predict(5));
```



```
// Check the contents of the ctrl register
spi_rm.ctrl.mirror(status, UVM_CHECK);
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/spi_reg

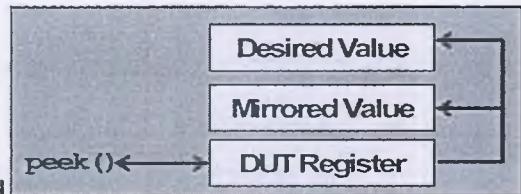
428

Notes:

peek() Method

■ Backdoor access method

- Does a direct read of the hardware
 - ◆ Updates mirrored and desired values
 - ◆ Register value is sampled, not modified
- Register or field access



UVM

```
task peek(output uvm_status_e      status,
          output uvm_reg_data_t   value,
          input  string           kind = "",
          input  uvm_sequence_base parent = null,
          input  uvm_object        extension = null,
          input  string           fname = "",
          input  int              lineno = 0);
```

```
    uvm_reg_data_t ctrl_value;
    uvm_reg_data_t char_len_value;
```

```
// Register level peek:
spi_rm.ctrl.peek(status, ctrl_value );
```

```
// Field level peek (char_len is a field within the ctrl reg):
spi_rm.ctrl.char_len.peek(status, char_len_value);
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

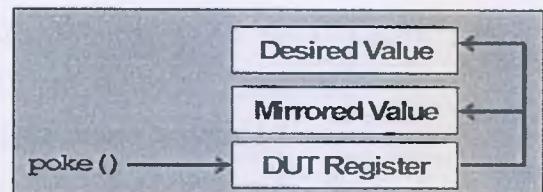
429

Notes:

poke() Method

■ Backdoor access method

- Does a direct write of the hardware
 - ◆ Updates mirrored and desired values
 - ◆ Specified value is deposited in register as-is (no side-effects)
- Register or field access



```
task poke(output uvm_status_e      status,
          input  uvm_reg_data_t   value,
          input  string           kind = "",
          input  uvm_sequence_base parent = null,
          input  uvm_object        extension = null,
          input  string           fname = "",
          input  int              lineno = 0);

  uvm_reg_data_t ctrl_value;
  uvm_reg_data_t char_len_value;

  // Register level poke:
  spi_rm.ctrl.poke(status, ctrl_value);

  // Field level poke:
  spi_rm.ctrl.char_len.poke(status, char_len_value);
```



Notes:

reset () Method

- Sets register model desired and mirrored values to register reset value
- Does not cause bus cycles
- Should be called when a hardware reset is observed

UVM

```
function void reset(string kind = "HARD");  
  
    spi_rm.reset(); // Block level reset  
  
    spi_rm.ctrl.reset(); // Register level reset  
  
    spi_rm.ctrl.char_len.reset(); // Field level reset
```

Before	After
Desired	Desired
Mirrored	Mirrored
DUT Reg	DUT Reg

WILLAMETTE
HDL

uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

431

Notes:

get_reset() Method

- Returns the pre-defined register reset value
- Typical use is together with `read()` or `mirror()` to check that a register was reset properly



```
function uvm_reg_data_t get_reset(string kind = "HARD");
```

Before	After
Desired	Desired

```
uvm_reg_data_t ctrl_value;  
uvm_reg_data_t char_len_value;
```

Mirrored	Mirrored
DUT Reg	DUT Reg

```
ctrl_value = spi_rm.ctrl.get_reset(); // Register level  
char_len_value = spi_rm.ctrl.char_len.get_reset(); // Field level
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/spi_reg

432

Notes:

Register Access Method Summary

Method	Block	Register	Field	Block	Register	Field
	Front Door Access			Back Door Access		
read()	No	Yes	Not Recommended	No	Yes	Yes
write()	No	Yes	Not Recommended	No	Yes	Yes
peek()	No	No	No	No	Yes	Yes
poke()	No	No	No	No	Yes	Yes
update()	Yes	Yes	No	Yes	Yes	No
mirror()	Yes	Yes	No	Yes	Yes	No

- The other access methods access the register database and not the DUT registers
 - Front door, back door have no meaning for these methods
`get()`, `set()`, `reset()`, `get_reset()`, `get_mirrored_value()`

Notes:

Concurrent Accesses

- Register model may be accessed from multiple concurrent execution threads
 - Internally the accesses to the same register are serialized
 - Each register has a semaphore to ensure that it can be read or written only one process at a time
 - ◆ Any other processes attempting access will block
 - Resumes after the current operation completes (and after any other blocked processes ahead of it)
 - ◆ If a process is killed in the middle of executing a register operation it becomes necessary to release the semaphore
 - Do so by calling the register's `reset()` method



uvm_intro_3.5

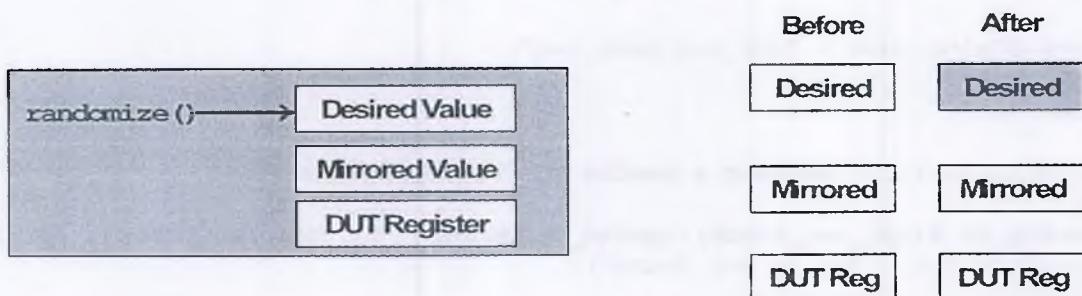
©Willamette HDL Inc

434

Notes:

Randomization of Registers

- `randomize()` may be called at the register model, block, register or field level
 - Fields of the item need to be defined as `rand`
 - ◆ Can be randomized with or without constraints
- The desired value is updated with the randomized value



Notes:

SPI Example – Sequence: spi_bus_base_seq

```
// Base class that used by all the other sequences in the package:  
// Gets the handle to the register model - spi_rm  
// Contains the data and status fields used by most register access methods  
//  
class spi_bus_base_seq extends uvm_sequence #(uvm_sequence_item);  
  `uvm_object_utils(spi_bus_base_seq)  
  
  spi_reg_block spi_rm; // SPI Register model:  
  
  // Properties used by the various register access methods:  
  rand uvm_reg_data_t data; // For passing data  
  uvm_status_e status; // Returning access status  
  
  function new(string name = "spi_bus_base_seq");  
    super.new(name);  
  endfunction  
  
  // Common functionality: Getting a handle to the register model  
  task body;  
    if(!uvm_config_db #(spi_reg_block)::get(m_sequencer, "", "spi_reg_block", spi_rm))  
      `uvm_error("CONFIG", "spi_rm not found");  
  endtask: body  
endclass
```

Notes:

SPI Example - Sequence: ctrl_set_seq

```
// Ctrl set sequence - loads one control params but does not set the go bit

class ctrl_set_seq extends spi_bus_base_seq;
  `uvm_object_utils(ctrl_set_seq)

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

  // Controls whether interrupts are enabled or not
  bit int_enable = 0;

  task body;
    super.body;
    // Constrain to interesting data length values
    assert(spi_rm.ctrl_reg.randomize() with
      {char_len.value inside {0, 1, [31:33], [63:65], [95:97], 126, 127};});
    // Set up interrupt enable
    spi_rm.ctrl_reg.ie.set(int_enable);
    // Don't set the go_bsy bit
    spi_rm.ctrl_reg.go_bsy.set(0);
    // Write the new value to the control register
    spi_rm.ctrl_reg.update(status, .path(UVM_FRONTDOOR));
    // Get a copy of the register value for the SPI agent
    data = spi_rm.ctrl_reg.get();
  endtask: body

endclass: ctrl_set_seq
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

437

Notes:

Memories

- The register model supports memory accesses
- Represented by memory models
 - Configured with a size (range) and an offset in the register map
 - Access type is one of
 - ◆ Read-write (RW)
 - ◆ Read-only (RO – ROM)
 - ◆ Write-only (WO)
- Memory model does not store state – i.e. not mirrored
 - Access layer only
 - Too severe of a simulation penalty from overhead
- Supports front door and back door access
- Six types of access supported
 - read, write
 - peek, poke
 - burst_read, burst_write

Notes:

Memory Read- read ()

- Read from a memory location
 - Address is the offset within the memory

UVM

```
task read(output uvm_status_e      status,          // Outcome of the read cycle
          input  uvm_reg_addr_t    offset,           // Offset address within memory region
          output uvm_reg_data_t    value,            // Read data
          input  uvm_path_e        path = UVM_DEFAULT_PATH, // Front or backdoor access
          input  uvm_reg_map       map = null,         // Which map, memory might be in >1 map
          input  uvm_sequence_base parent = null,       // Parent sequence
          input  int                prior = -1,         // Priority on the target sequencer
          input  uvm_object         extension = null,   // Object allowing extension
          input  string              fname = "",        // Filename for messaging
          input  int                lineno = 0);       // File line number for messaging

// Using default map
mem_ss.mem_1.read(status, 32'h1000, read_data);

// Using alternative map
mem_ss.mem_1.read(status, 32'h2000, read_data, .map(AHB_2_map));
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/mem_example

439

Notes:

MemoryWrite - write()

■ Write to a memory location

- Address is the offset within the memory

UVM

```
task write(output uvm_status_e           status,          // Outcome of the write cycle
          input  uvm_reg_addr_t    offset,         // Offset address within the memory region
          input  uvm_reg_data_t    value,          // Write data
          input  uvm_path_e        path = UVM_DEFAULT_PATH, // Front or backdoor access
          input  uvm_reg_map       map = null,       // Which map, memory might be in >1 map
          input  uvm_sequence_base parent = null,   // Parent sequence
          input  int                prior = -1,      // Priority on the target sequencer
          input  uvm_object         extension = null, // Object allowing method extension
          input  string              fname = "",       // Filename for messaging
          input  int                lineno = 0);     // File line number for messaging

// Using default map
mem_ss.mem_1.write(status, 32'h1000, write_data);

// Using alternative map
mem_ss.mem_1.write(status, 32'h2000, write_data, .map(AHB_2_map));
```

**WILLAMETTE
HDL**

uvm_intro_35

©Willamette HDL Inc.

code in examples/mem_example

440

Notes:

Memory Burst Read - `burst_read()`

- Read an array of data from a memory from consecutive locations
 - Address is the offset within the memory for starting location

UVM

```
task burst_read(output uvm_status_e      status,          // Outcome of the read cycle
               input  uvm_reg_addr_t   offset,           // Offset address within the memory region
               output uvm_reg_data_t   value[],         // Read data array
               input  uvm_path_e       path = UVM_DEFAULT_PATH, // Front or backdoor access
               input  uvm_reg_map      map = null,        // Which map, memory might be in >1 map
               input  uvm_sequence_base parent = null,    // Parent sequence
               input  int               prior = -1,        // Priority on the target sequencer
               input  uvm_object        extension = null, // Object allowing method extension
               input  string            fname = "",        // Filename for messaging
               input  int               lineno = 0);      // File line number for messaging

uvm_reg_data_t read_data[];

// 8 Word transfer:
read_data = new[8]; // Set read_data array to size 8
mem_ss.mem_1.burst_read(status, 32'h1000, read_data); // Using default map

// 4 Word transfer from an alternative map:
read_data = new[4]; // Set read_data array to size 4
mem_ss.mem_1.burst_read(status, 32'h2000, read_data, .map(AHB_2_map));
```



uvm_intro_35

©Willamette HDL Inc.

code in examples/mem_example

441

Notes:

Memory Burst Write—burst_write()

- Write an array of data to a memory at consecutive locations
 - Address is the offset within the memory for starting location

UVM

```
task burst_write(output uvm_status_e      status,    // Outcome of the write cycle
                input  uvm_reg_addr_t   offset,     // Offset address within the memory region
                input  uvm_reg_data_t   value[],    // Write data array
                input  uvm_path_e       path = UVM_DEFAULT_PATH, // Front or backdoor access
                input  uvm_reg_map      map = null,    // Which map, memory might be in >1 map
                input  uvm_sequence_base parent = null, // Parent sequence
                input  int               prior = -1,   // Priority on the target sequencer
                input  uvm_object        extension = null, // Object allowing method extension
                input  string            fname = "",    // Filename for messaging
                input  int               lineno = 0); // File line number for messaging
```

Notes:

Example: burst_write()

```
uvm_reg_data_t write_data[];  
  
// 8 Word transfer Using default map  
write_data = new[8]; // Set write_data array to size 8  
foreach(write_data[i]) begin  
    write_data[i] = i*16;  
end  
mem_ss.mem_1.burst_write(status, 32'h1000, write_data);  
  
// 4 Word transfer from an alternative map:  
write_data = new[4]; // Set read_data array to size 4  
write_data = '{32'h55AA_55AA, 32'hAA55_AA55, 32'h55AA_55AA, 32'hAA55_AA55};  
mem_ss.mem_1.burst_write(status, 32'h2000, write_data, .map(AHB_2_map));
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/mem_example

443

Notes:

Memory Peek - peek()

- Read current value from a memory location using back-door access
 - Address is the offset within the memory
 - Memory location is sampled not modified

```
task peek(output uvm_status_e      status,
          input  uvm_reg_addr_t    offset,
          output uvm_reg_data_t    value,
          input  string            kind = "",
          input  uvm_sequence_base parent = null,
          input  uvm_object         extension = null,
          input  string            fname = "",
          input  int                lineno = 0);
```



Notes:

Memory Poke - poke()

- Write to a memory location using back-door access
 - Address is the offset within the memory
 - Value is deposited in memory location as-is

```
task poke(output uvm_status_e      status,
          input  uvm_reg_addr_t   offset,
          input  uvm_reg_data_t   value,
          input  string           kind = "",  

          input  uvm_sequence_base parent = null,
          input  uvm_object        extension = null,
          input  string           fname = "",  

          input  int              lineno = 0);  
  
// Using default map  
mem_ss.mem_1.poke(status, 32'h1000, write_data);  
  
// Using alternative map  
mem_ss.mem_1.poke(status, 32'h2000, write_data, .map(AHB_2_map));
```



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/mem_example

445

Notes:

Example: mem_ss_test

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

// mem_ss register/memory model
mem_ss_reg_block mem_ss_rm;
// mem_ss env and config object
mem_ss_env_config m_cfg;
mem_ss_env m_env;
// AHB agent config object
ahb_agent_config m_ahb_cfg;

extern function new(string name = "mem_ss_test", uvm_component parent = null);
extern function void build();
extern task run;
endclass: mem_ss_test

task mem_ss_test::run_phase(uvm_phase phase);
mem_1_test_seq mem_1_test = mem_1_test_seq::type_id::create("m_1_test");
phase.raise_objection(this);
mem_1_test.start(m_env.m_ahb_agent.m_sequencer);
phase.raise_objection(this);
endtask
```

Notes:

Example: mem_1_test_seq

```
// Write to 10 random locations within the memory storing the data written
// then read back from the same locations checking against the original data
class mem_1_test_seq extends mem_ss_base_seq;
`uvm_object_utils(mem_1_test_seq)
rand uvm_reg_addr_t addr;

// Buffers for the addresses and the write data
uvm_reg_addr_t addr_array[10];
uvm_reg_data_t data_array[10];
...
task body;
super.body();
// Write loop
for(int i = 0; i < 10; i++) begin
// Constrain address to be within the memory range:
assert(this.randomize() with {addr <= mem_ss_rm.mem_1.get_size()});
mem_ss_rm.mem_1.write(status, addr, data);
addr_array[i] = addr;
data_array[i] = data;
end
// Read loop
for(int i = 0; i < 10; i++) begin
mem_ss_rm.mem_1.read(status, addr_array[i], data);
if(data_array[i][31:0] != data[31:0]) begin
`uvm_error("mem_1_test", $sformatf("Memory access error: expected %0h,
actual %0h", data_array[i][31:0], data[31:0]))
end
end
endtask: body
endclass: mem_1_test_seq
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/mem_example

447

Notes:

Backdoor Access

- Backdoor accesses use the simulator database to directly access registers within the DUT
 - Write operations force registers to a specified value
 - Read operations return the current value
 - Takes zero time since it bypasses normal bus protocol to access the DUT
- Potential benefits include:
 - DUT configuration – putting DUT into a non-reset state before doing front door configuration
 - Adding an extra level of debug when checking data paths
 - ◆ Use a backdoor peek after a front door write
 - Checking a buffer before it is read by a front door access
- Potential issues
 - Simulators may optimize away detailed structure information needed for backdoor accesses
 - ◆ May need to turn off optimization

Notes:

Defining the Backdoor HDL Path

- User must specify hdл path to the signals the register model represents
- Path is specified in "hierarchical sections"
 - Register model top level block specifies path to the top level of the DUT
 - Register model sub-system block specifies path to the DUT subsystem
 - Register model register specifies the DUT path within the sub-system
 - ◆ Also specifies which register bits correspond to what DUT signals
- Done as part of the data entry and code is auto generated

Notes:

Backdoor HDL Path Methods

- HDL paths are specified with the following methods:

```
function void configure ( uvm_reg_block blk_parent,  
                        uvm_reg_file    regfile_parent = null,  
                        string          hdl_path = "" )
```

- Static method of `uvm_reg_block`, `uvm_reg_file`, `uvm_reg`, `uvm_mem`

```
static function void add_hdl_path ( string path, string kind = "RTL" )
```

- Static method of `uvm_reg_block`, `uvm_reg_file`

```
static function void add_hdl_path_slice( string name,  
                                         int      offset,  
                                         int      size,  
                                         bit      first   =      0,  
                                         string   kind    =      "RTL"    )
```

- Static method of `uvm_reg`, `uvm_mem`

Notes:

Example Back Door Path Specifications

```
class spi_reg_block extends uvm_reg_block;
  ...
  virtual function void build();
    ...
    ctrl_reg = ctrl::type_id::create("ctrl");
    ctrl_reg.build();
    ctrl_reg.configure(this, null, "");
    // Add the ctrl hdl_path starting at bit 0, hardware target is 14 bits wide
    ctrl_reg.add_hdl_path_slice("ctrl", 0, 14);
    ...
    APB_map = create_map("APB_map", 'h0, 4, UVM_LITTLE_ENDIAN);
    ...
    APB_map.add_reg(ctrl_reg, 32'h00000010, "RW");
    ...
    // Assign DUT to the hdl path
    add_hdl_path("DUT", "RTL");
    ...
    lock_model();
  endfunction
  ...
endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

code in examples/spi_reg

451

Notes:

Backdoor Access – RTL vs. Gate level

- Problem
 - RTL and gate level netlists will not have the same HDL path
- Solution:
 - Register model supports specification of more than one path
 - The backdoor `hdl_path` can be set for both RTL and gate level



uvm_intro_3.5

©Willamette HDL Inc.

452

Notes:

Quirky Registers

- Same as other registers except . . quirky behavior!
 - Behavior can't be described using register base class
 - ◆ Clear on the 4th read
 - ◆ Read a register that is collected from multiple registers
 - ◆ etc.
- Several options to build a quirky register
 - Some are provided by the UVM library and can be extended
 - ◆ uvm_reg_fifo
 - ◆ uvm_reg_indirect
 - Some provided not as part of the base library but part of the UVM download in the examples directory
 - Build your own
 - ◆ Inherit from base classes and override the behavior of the virtual tasks like set (), get (), etc. or add callbacks.

Notes:

Model Coverage

- Coverage model information is added by the register model generator
- UVM register classes provide the API:
 - For a covergroup to sample data
 - To control covergroup instantiation (yes/no) and to turn on/off measurement



uvm_intro_3.5

©Willamette HDL Inc.

454

Notes:

Built in Sequences

- UVM library contains automatic test sequences
 - Basic tests on registers and memories
 - ◆ Register reset value is correct, validate the read-write paths etc.
 - ◆ Can be used for quick sanity checks
- Registers and memories can opt out
 - Set the DO_NOT_TEST attribute checked by the automatic sequences
 - Set the NO_REG_TEST attribute for registers
 - Set the NO_MEM_TEST attribute for memories
 - Attributes are set using the resource_db
 - ◆ Done as part of the register generation
- To use one of the built in sequences
 - Create the sequence (just like any other sequence)
 - ◆ seq = built_in_seq::type_id::create ("seq")
 - Set its model property to point to the register model
 - Start the sequence (just like any other sequence)
 - ◆ seq.start (sequencer)

Notes:

Built in Sequences

- Built in sequences – see Class Reference for details:

```
uvm_reg_hw_reset_seq  
uvm_reg_single_bit_bash_seq  
uvm_reg_bit_bash_seq  
uvm_reg_single_access_seq  
uvm_reg_access_seq  
uvm_reg_mem_access_seq  
uvm_reg_shared_access_seq  
uvm_mem_shared_access_seq  
uvm_reg_mem_shared_access_seq  
uvm_mem_single_access_seq  
uvm_mem_access_seq  
uvm_mem_single_walk_seq  
uvm_mem_walk_seq  
uvm_reg_mem_hdl_paths_seq
```

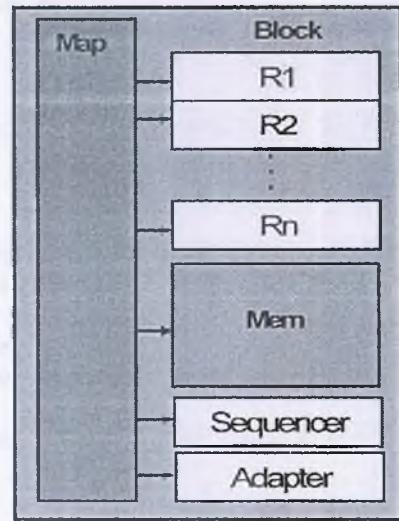
Notes:

Register Block and Register Convenience Methods

- Notice on the previous examples to access a block or register you give a full path name
 - While explicit it can be tedious
- There are a number of methods that can be useful

```
task body;
    super.body;

    ...
    // Set up interrupt enable
    spi_rm.ctrl_reg.ie.set(int_enable);
    // Don't set the go_bsy bit
    spi_rm.ctrl_reg.go_bsy.set(0);
    // Write the new value to the control register
    spi_rm.ctrl_reg.update(status, .path(UVM_FRONTDOOR));
    // Get a copy of the register value for the SPI agent
    data = spi_rm.ctrl_reg.get();
endtask: body
...
```



Notes:

Register Block and Register Convenience Methods



Useful Stuff!

```
virtual function string get_full_name();
virtual function uvm_reg_block get_parent();

static function int find_blocks (input string name,
                                 ref uvm_reg_block blks[$],
                                 input uvm_reg_block root = null,
                                 input uvm_object accessor = null);
static function uvm_reg_block find_block(input string name,
                                         input uvm_reg_block root = null,
                                         input uvm_object accessor = null);

static function void get_root_blocks(ref uvm_reg_block blks[$]);
virtual function void get_blocks (ref uvm_reg_block blks[$],
                                  input uvm_hier_e hier=UVM_HIER);
virtual function void get_maps (ref uvm_reg_map maps[$]);
virtual function void get_registers (ref uvm_reg regs[$],
                                    input uvm_hier_e hier=UVM_HIER);
function void get_fields (ref uvm_reg_field fields[$],
                          input uvm_hier_e hier=UVM_HIER);

virtual function uvm_reg_block get_block_by_name (string name);
virtual function uvm_reg_map get_map_by_name (string name);
virtual function uvm_reg get_reg_by_name (string name);
virtual function uvm_reg_field get_field_by_name (string name);
virtual function uvm_mem get_mem_by_name (string name);

virtual function uvm_reg_addr_t get_address (uvm_reg_map map = null);
virtual function int get_addresses (uvm_reg_map map = null,
                                   ref uvm_reg_addr_t addr[]);
```

Wilamette HDL

uvm_intro_3.5

©Wilamette HDL Inc.

code in examples/spi_reg

458

Notes:

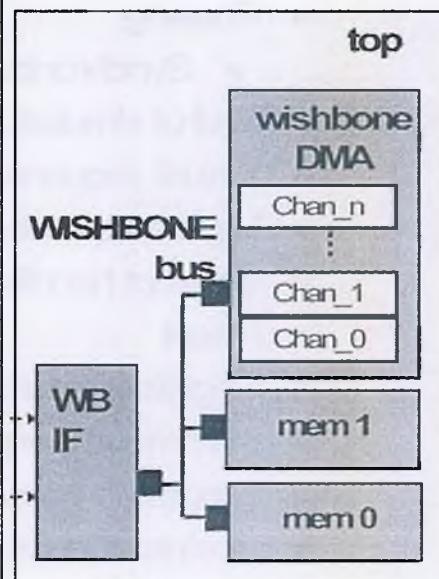
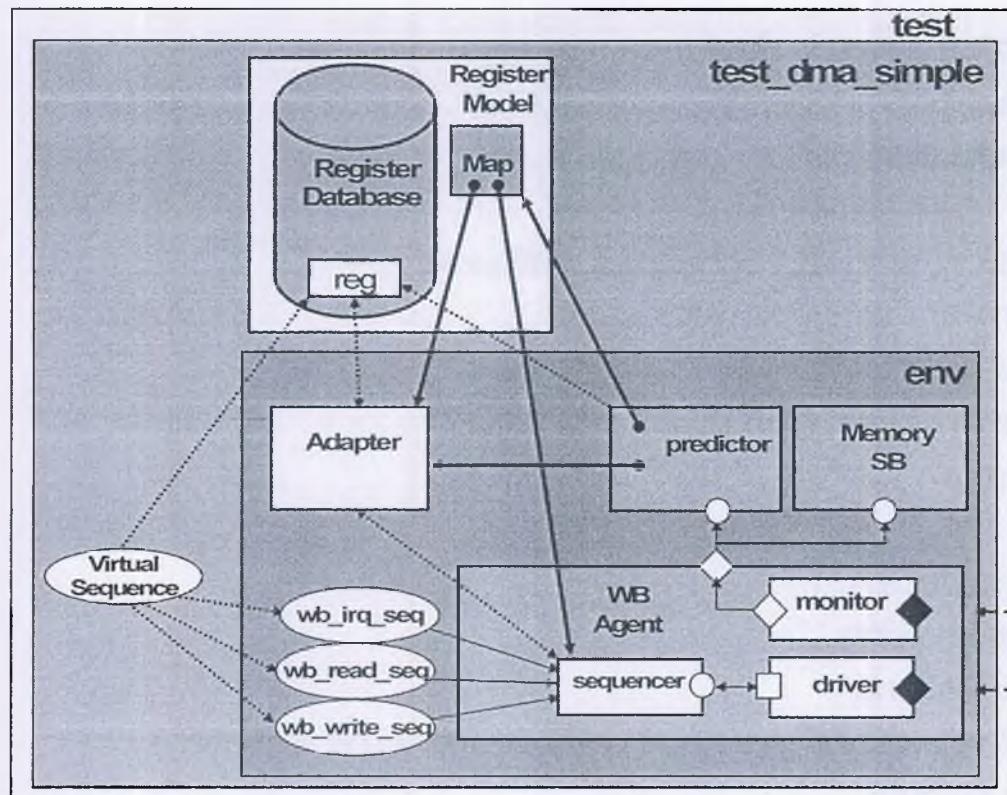
Advanced UVM Class topics

- UVM topics not covered here that are covered in the Advanced UVM class:
 - DUT-TB Connection (more in depth)
 - Container classes
 - Process synchronization
 - Phasing
 - ◆ Synchronization, domains, schedules, callbacks
 - End of simulation
 - Virtual sequences (more in depth)
 - Layered stimulus (much more in depth)
 - Interrupt handling
 - Reset
 - Registers (more in depth)
 - Command line processing
 - Emulation considerations
 - Abstract/Concrete class DUT connection (Two Kingdoms approach)
 - Coverage driven testing

Notes:

Lab – Using Registers: Overview

- Use the register model in a virtual sequence



Notes:

Lab – Using Registers: WB Register Model

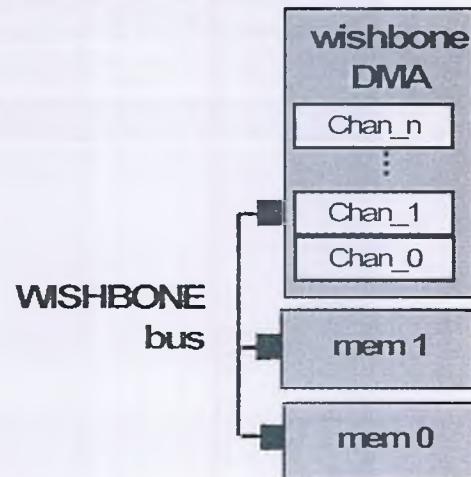
Block	Map	Base Address
dma_system_block	dma_system_map	'h0
Sub Block	Sub Map	Offset Address
dma_top	dma_top_map	DMA_SLAVE_0_WB_ID * SLAVE_ADDR_SPACE_SZ
Memory		Offset Address
mem0		MEM_SLAVE_0_WB_ID * SLAVE_ADDR_SPACE_SZ
Memory		Offset Address
mem1		MEM_SLAVE_1_WB_ID * SLAVE_ADDR_SPACE_SZ

```
package test_params_pkg;
parameter SLAVE_ADDR_SPACE_SZ = 32'h10000000;

parameter DMA_SLAVE_0_WB_ID = 2;

parameter MEM_SLAVE_0_SIZE = 32'h00100000;
parameter MEM_SLAVE_0_WB_ID = 0;

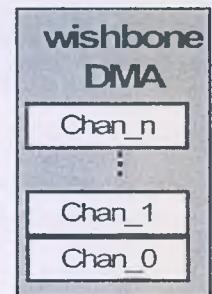
parameter MEM_SLAVE_1_SIZE = 32'h00100000;
parameter MEM_SLAVE_1_WB_ID = 1;
...
endpackage
```



Notes:

Lab – Using Registers: DMA Register Block

Block	Map	
dma_top	dma_top_map	
Sub Block	Map	Offset Address
dma_main	dma_main_map	'h0
	Registers	Offset Address
	CSR_MAIN	32'h00000000
	INT_MSK_A	32'h00000004
	INT_MSK_B	32'h00000008
	INT_SRC_A	32'h0000000c
	INT_SRC_B	32'h00000010
Sub Block	Sub Map	Offset Address
CH_0	dma_channel_map	32'h00000020
	Registers	Offset Address
	CHn_CSR_reg	32'h00000000
	CHn_SZ_reg	32'h00000004
	CHn_A0	32'h00000008
	CHn_AM0	32'h0000000c
	CHn_A1	32'h00000010
	CHn_AM1	32'h00000014
	CHn_DESC_reg	32'h00000018
	CHn_SWPTR_reg	32'h0000001c



Notes:

Lab – Using Registers: Part 1 Instructions

- Part 1: Run a built-in register test
 - NOTE: For Part 1 of this lab the test used is `test_reset_regs`
 - Working Directory: `register_reset`
 - Edit the file `tests/test_reset_regs.svh`
 - ◆ Create the built-in sequence `uvm_reg_hw_reset_seq`
 - Checks that registers contain their reset value
 - ◆ Initialize the `model` property of the `uvm_reg_hw_reset_seq` object to `m_dma_system_block.dma_top`
 - The `model` property tells the sequence which block's registers to test
 - ◆ Start the sequence with the sequencer from the configuration object: `wb_config_0.wb_seqr`
 - Compile and run
 - ◆ Linux: `make`
 - ◆ Windows: `run_reset.do`
 - ◆ Should be no errors – scroll up to view output of test

Notes:

Lab – Using Registers: Part 2 Instructions

■ Part 2: Initialize the DMA chip

- NOTE: For Part 2 of this lab the test used is `test_dma_simple`
- Working Directory: `register_dma_init` (NOTE this is a different directory!!!)
- Edit the file `sequences/dma_init_seq.svh`
 - ◆ This sequence initializes the interrupt source, interrupt mask and channel control registers of the DMA chip
 - ◆ In the task `init_dma()`...
 - ◆ Write the `m_main_block.INT_MSK_A` register with the value `32'hf`
 - Sets interrupt masks (enables) for channels 0,1,2,3 of interface A of the DMA chip
 - ◆ Read the `m_main_block.INT_SRC_A` register
 - Clears the interrupt source register for interface A of the DMA chip
 - ◆ Write the `m_chan_block[i].CHn_CSR_reg` registers with the value 0
 - In a loop where index i's range is from 0 to 3
 - Clears the Configuration Status Register (`CHn_CSR_reg`) in DMA channels 0,1,2,3
- Compile and run

Notes:

Lab – Using Registers: Part 3 Instructions - 1

■ Part 3: Initiate a DMA transfer

- NOTE: For Part 3 of this lab the test used is `test_dma_simple`
- Working Directory: `register_dma_send` (NOTE this is a different directory!!!!)
- Edit the file `sequences/dma_simple_seq.svh`
 - ◆ This sequence initiates 4 DMA transfers – one in each channel
 - ◆ You will edit the task `dma_setup()` which sets up a single DMA transfer
 - Among its arguments are the channel number (`chan_num`), the source (`srce_addr`) and destination (`dest_addr`) addresses and the size (`tot_sz`) of the transfer
 - These arguments are used in the register writes/reads inside this task
 - The variable `op_status` is created to be used in the status fields of the commands you create
 - ◆ Write the channel size register `m_chan_block[chan_num].CHn_SZ_reg` with the variable `wt_data`
 - This write sets the size of the DMA transfer
 - The variable `wt_data` contains the size of the transfer
 - Note that his value is computed from the `tot_sz` and `chk_sz` arguments
 - ◆ Continued...

Notes:

Lab – Using Registers: Instructions - 2

- ◆ Write the channel source address register
`m_chan_block[chan_num].CHn_A0 with srce_addr`
- ◆ Write the channel destination address register
`m_chan_block[chan_num].CHn_A1 with dest_addr`
- ◆ Update the channel configuration status register
`m_chan_block[chan_num].CHn_CSR_reg`
 - Note the value of the register fields are set in the `desired` variable in the lines of code prior to where you do the `update()`
 - Writing to this register starts the DMA and therefore is the last thing done in the task
- Compile and run

Notes:

Lab – Using Registers: Part 1 Sample Output

```
# UVM_INFO @ 3530: uvm_test_top.env.agent.wb_seqr@m_seq [uvm_reg_hw_reset_seq] Verifying reset  
value of register m_dma_system_block.dma_top.CH_1.CHn_AM1 in map  
"m_dma_system_block.dma_top.dma_top_map"..."  
# UVM_INFO @ 3630: uvm_test_top.env.agent.wb_seqr@m_seq [uvm_reg_hw_reset_seq] Verifying reset  
value of register m_dma_system_block.dma_top.CH_1.CHn_DESC_reg in map  
"m_dma_system_block.dma_top.dma_top_map"..."  
# UVM_INFO @ 3730: uvm_test_top.env.agent.wb_seqr@m_seq [uvm_reg_hw_reset_seq] Verifying reset  
value of register m_dma_system_block.dma_top.CH_1.CHn_SWPTR_reg in map  
"m_dma_system_block.dma_top.dma_top_map"  
***  
# UVM_INFO @ 3830: uvm_test_top.env.cov_sb [COVERAGE] ****  
# UVM_INFO @ 3830: uvm_test_top.env.cov_sb [COVERAGE] Final Memory Coverage = 0.000000%  
# UVM_INFO @ 3830: uvm_test_top.env.cov_sb [COVERAGE] ****  
#  
***  
# --- UVM Report Summary ---  
#  
# ** Report counts by severity  
# UVM_INFO : 47  
# UVM_WARNING : 0  
# UVM_ERROR : 0  
# UVM_FATAL : 0  
# ** Report counts by id  
# [COVERAGE] 3  
# [RNIST] 1  
# [STARTING_SEQ] 1  
# [UVMTOP] 1  
# [WB_MEM_SLAVE] 2  
# [Wishbone bus: 0, WB_MEM_SB_0] 1  
# [Wishbone bus: 0, WB_MEM_SB_1] 1  
# [uvm_reg_hw_reset_seq] 37
```

Should see no errors

Exact output may vary by simulator and revision.



uvm_intro_3.5

©Willamette HDL Inc

467

Notes:

Lab – Using Registers: Parts 2 & 3 Sample Output

```
# UVM_INFO @ 2330: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer complete on channel 0
# UVM_INFO @ 2530: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer data correct
# UVM_INFO @ 4530: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer complete on channel 1
# UVM_INFO @ 4890: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer data correct
# UVM_INFO @ 11930: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer complete on channel 2
# UVM_INFO @ 13970: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer data correct
# UVM_INFO @ 15490: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer complete on channel 3
# UVM_INFO @ 15690: uvm_test_top.env.agent.wb_seqr@0m_seq [DMA] DMA transfer data correct
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] ****
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] Final Memory Coverage = 0.00000%
# UVM_INFO @ 15690: uvm_test_top.env.cov_sb [COVERAGE] ****
#
...
--- UVM Report Summary ---
#
** Report counts by severity
# UVM_INFO : 25
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
** Report counts by id
# [COVERAGE] 3
# [DMA] 8
# [RNTST] 1
# [RegModel] 8
# [UVMTOP] 1
# [WB_MEM_SLAVE] 2
# [Wishbone bus: 0, WB_MEM_SB_0] 1
# [Wishbone bus: 0, WB_MEM_SB_1] 1
** Note: $finish : ../../uvm/src/base/uvm_root.svh(408)
# Time: 25930 ns Iteration: 103 Instance: /test
```

Should see no errors

Should see all 4 DMA transfers completed with correct transfer data

Exact output may vary by simulator and revision.

Sol

WHDL

uvm_intro_3.5

©Willamette HDL Inc.

468

Notes:

Example Testbenches

In this section



- Wishbone bus slave memory
- MAC DUT with WISHBONE bus and MII interfaces
- DMA Controller on SoC bus using register package
- Crossbar Router DUT
- SPI DUT with APB interface
- Memory with AHB interface

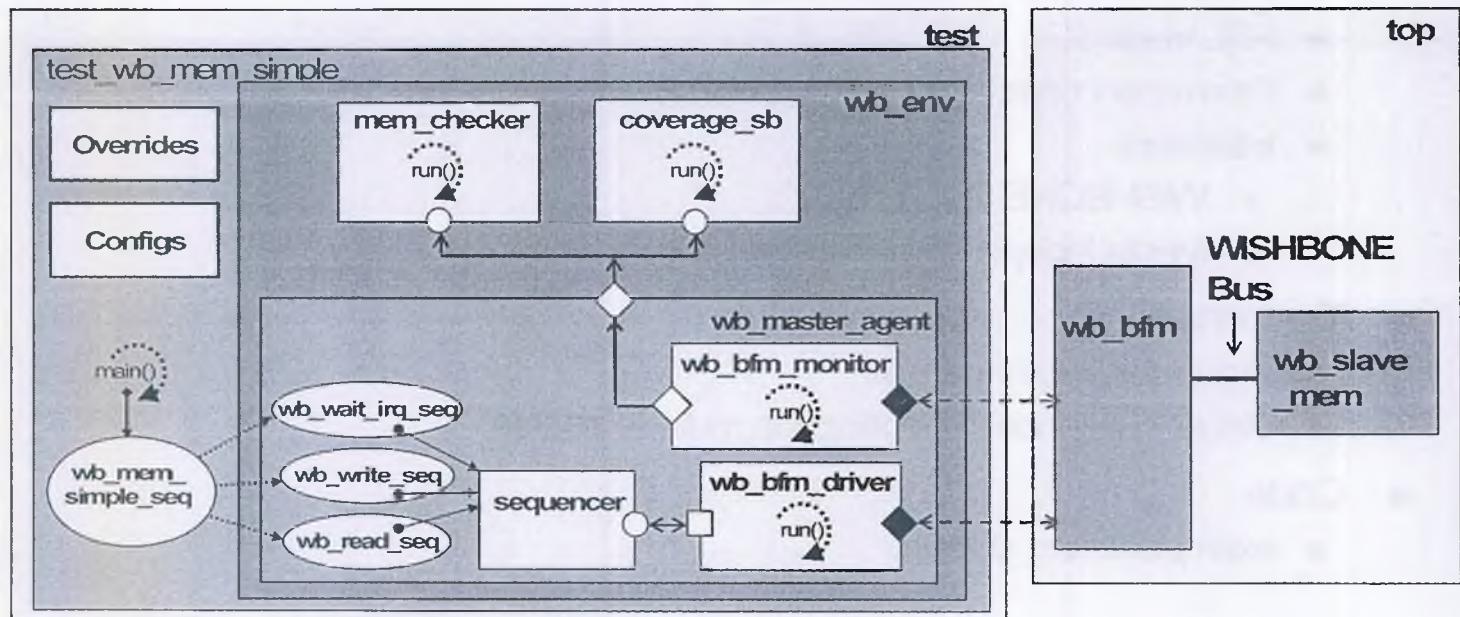
Notes:

Single SoC bus Slave Memory DUT

- DUT
 - Slave Memory
 - RTL Verilog
 - From opencores.org (Ethmac project)
 - Interfaces
 - ◆ WISHBONE SoC bus
- Testbench
 - WISHBONE agent
- Code
 - examples/wb_mem

Notes:

Example Circuit – Wishbone Bus Memory



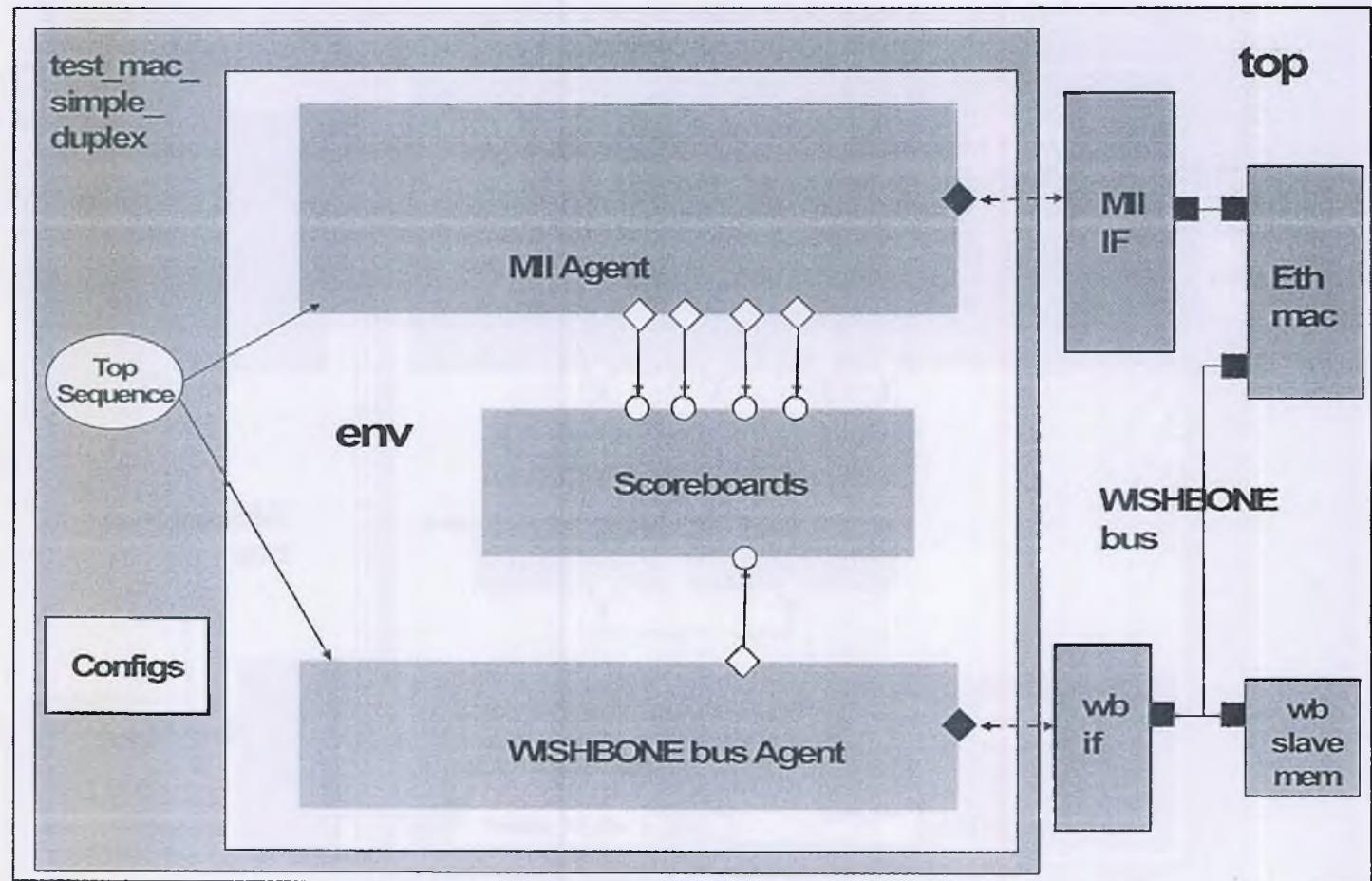
Notes:

Single DUT, Multiple Interfaces

- DUT
 - Ethernet Media Access Controller (MAC)
 - RTL Verilog
 - From opencores.org (Ethmac project)
 - Interfaces
 - ◆ WISHBONE SoC bus
 - ◆ Media Independent Interface (MII)
- Testbench
 - Agent for each interface
 - Virtual sequence controlling the multiple agents
- Code
 - examples/mac_duplex

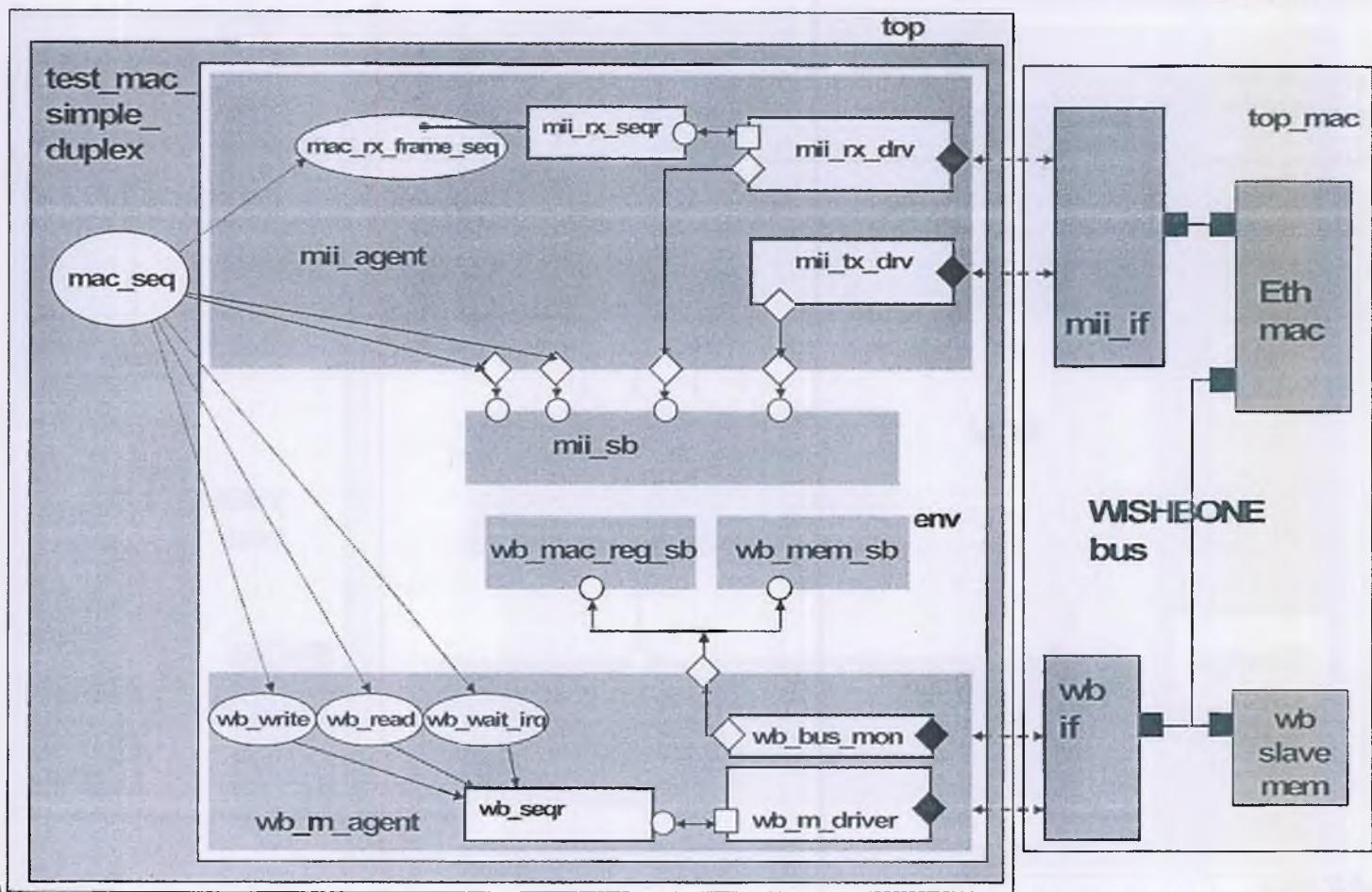
Notes:

Diagram mac_duplex



Notes:

Detailed Diagram mac_duplex



WHDL

uvm_intro_3.5

© Willamette HDL Inc.

code in examples/mac_duplex

474

Notes:

Crossbar Switch Testbench

- DUT
 - 8x8 Crossbar router
 - SystemVerilog
- Testbench
 - Router agent
- Code
 - examples/router_hier



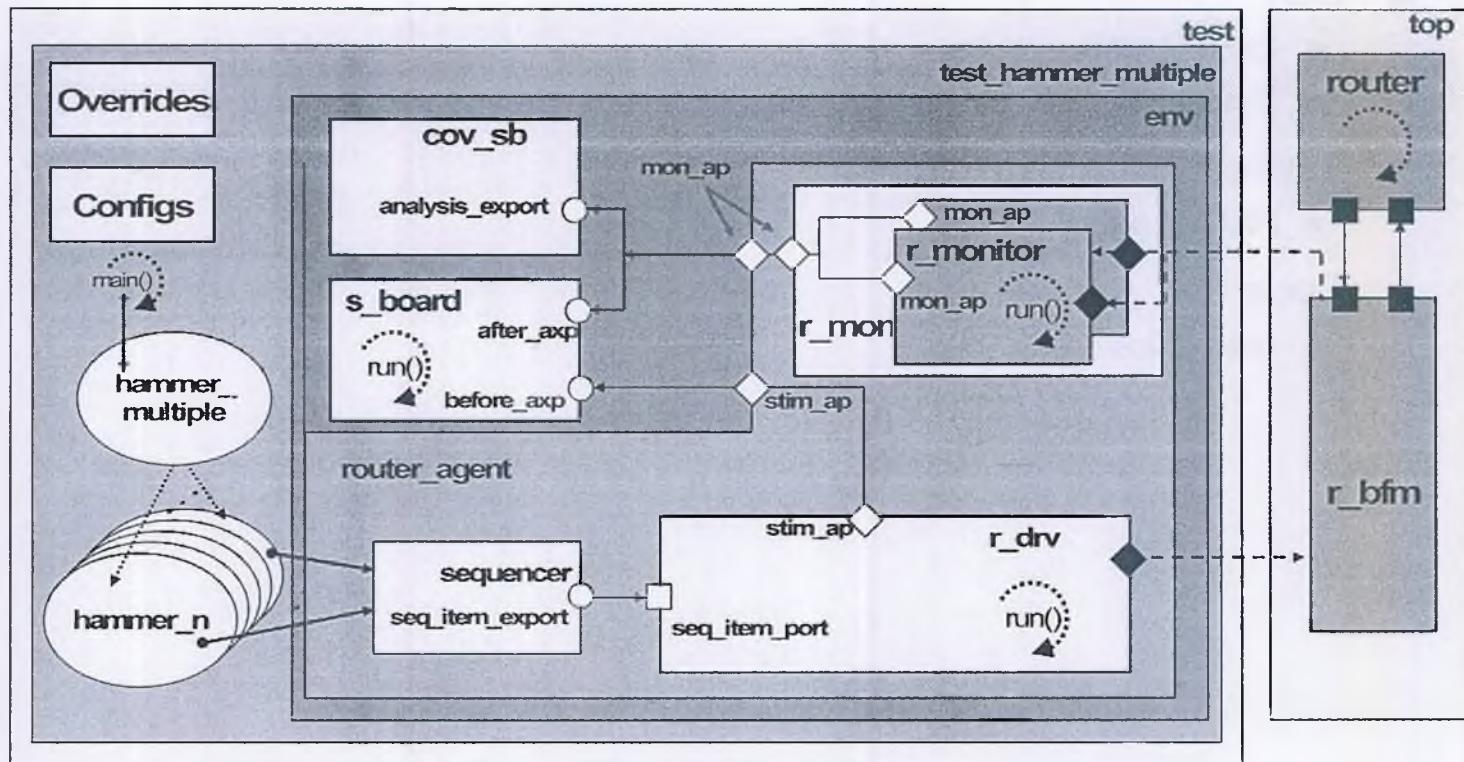
uvm_intro_3.5

© Willamette HDL Inc.

475

Notes:

Diagram: Crossbar Switch



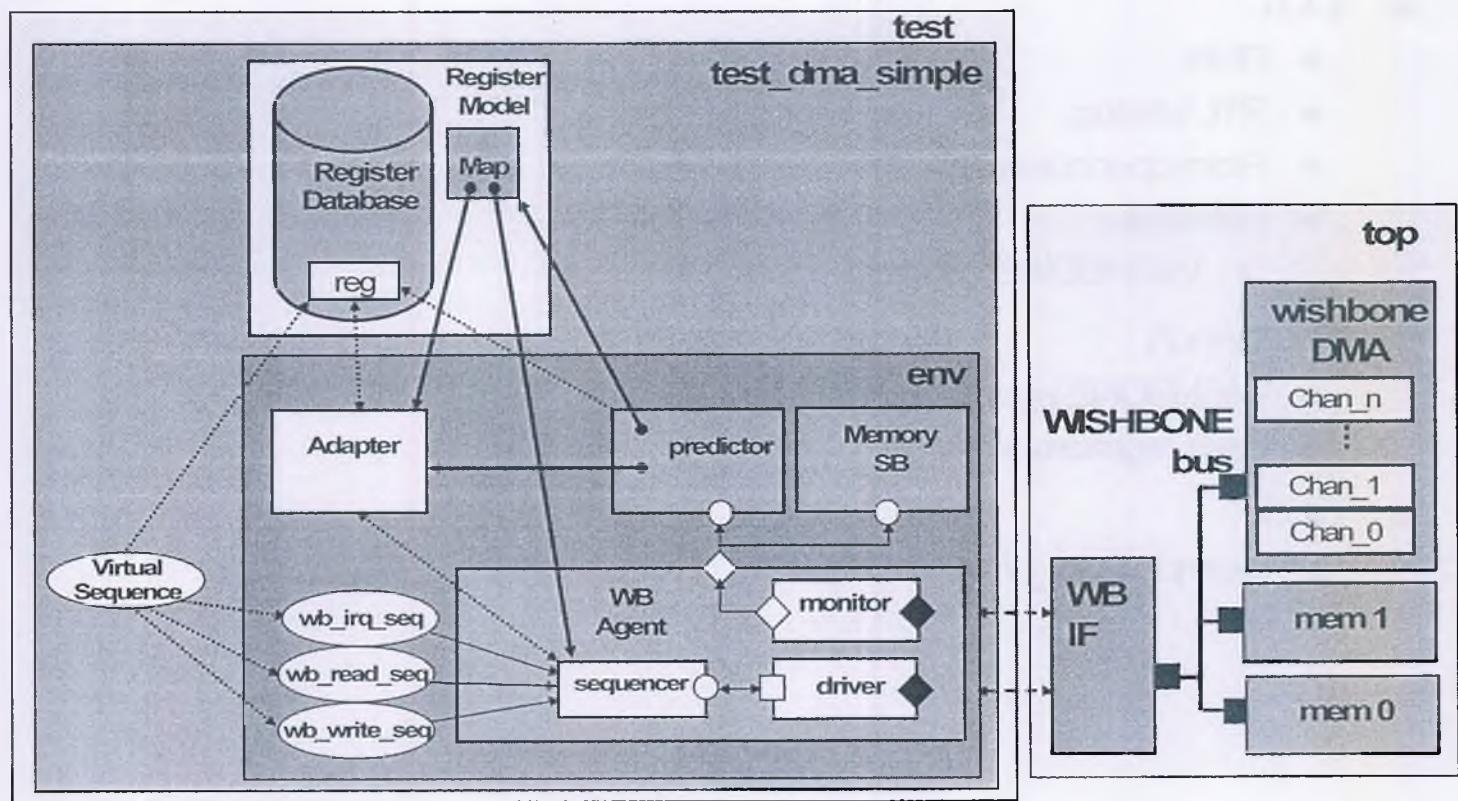
Notes:

Register Package Testbench

- DUT
 - DMA
 - RTL Verilog
 - From opencores.org
 - Interface
 - ◆ WISHBONE SoC bus
- Testbench
 - WISHBONE agent
 - Uses registers package
- Code
 - examples/wb_dma

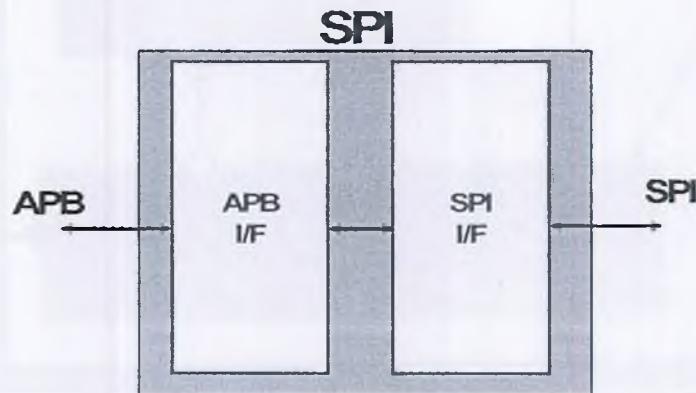
Notes:

Diagram: wishbone DMA



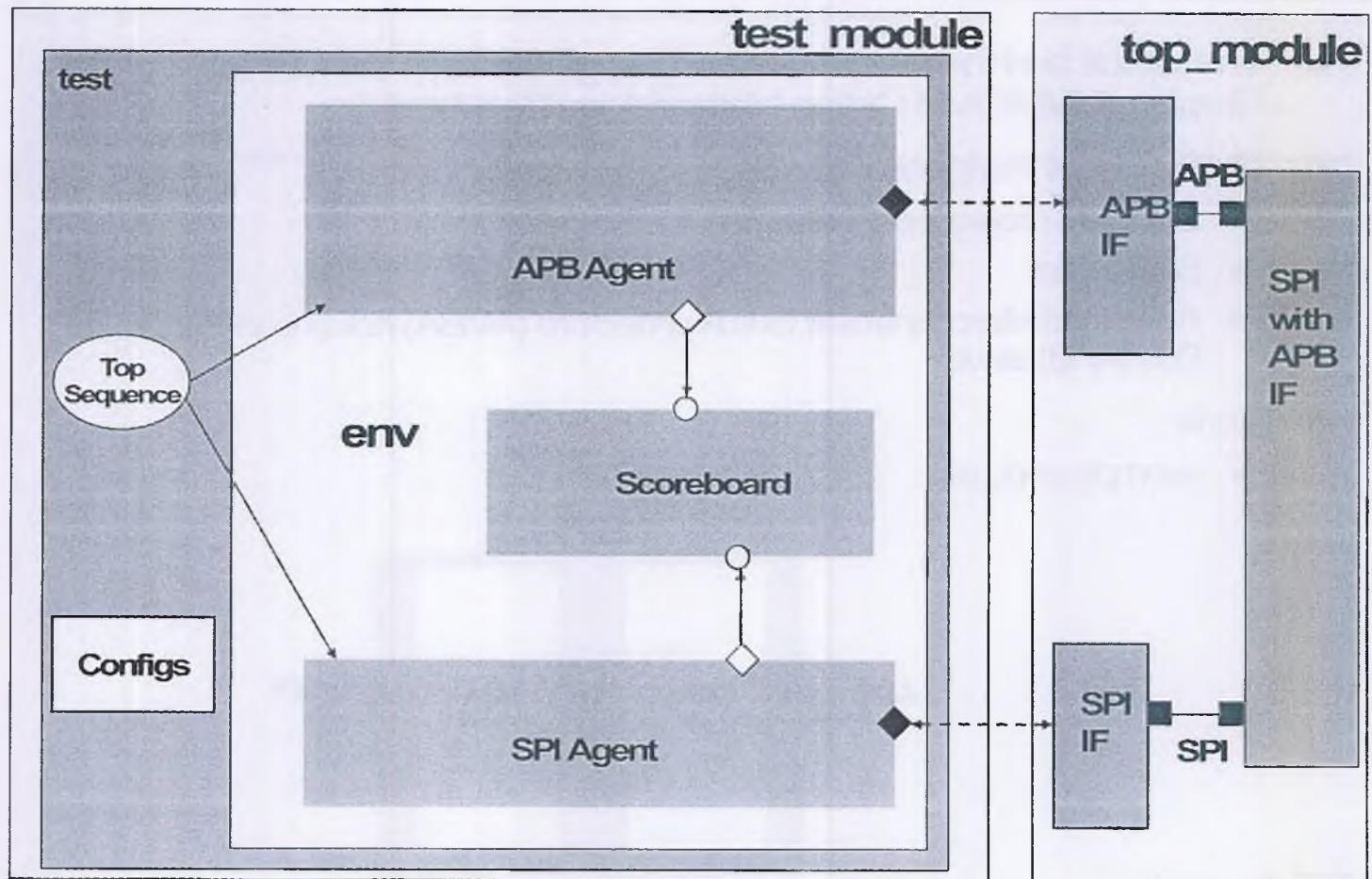
Notes:

- This circuit and the associated code originate from the Mentor Graphics UVM/OVM Online Methodology Cookbook
- DUT – Serial Peripheral Interface (SPI) Master Core
 - SPI Core project opencores.org
 - SPI master
 - Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) slave
- Code
 - examples/spi_reg



Notes:

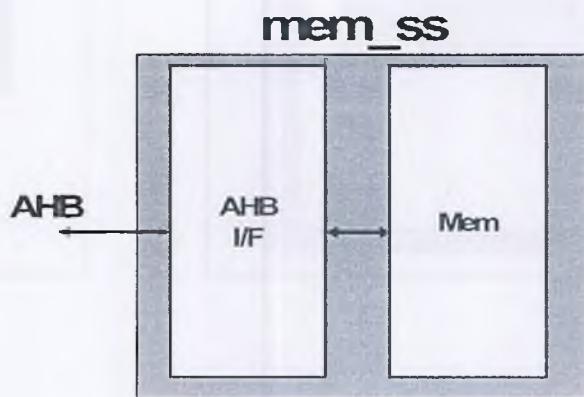
TESTBENCH SPI-APB



Notes:

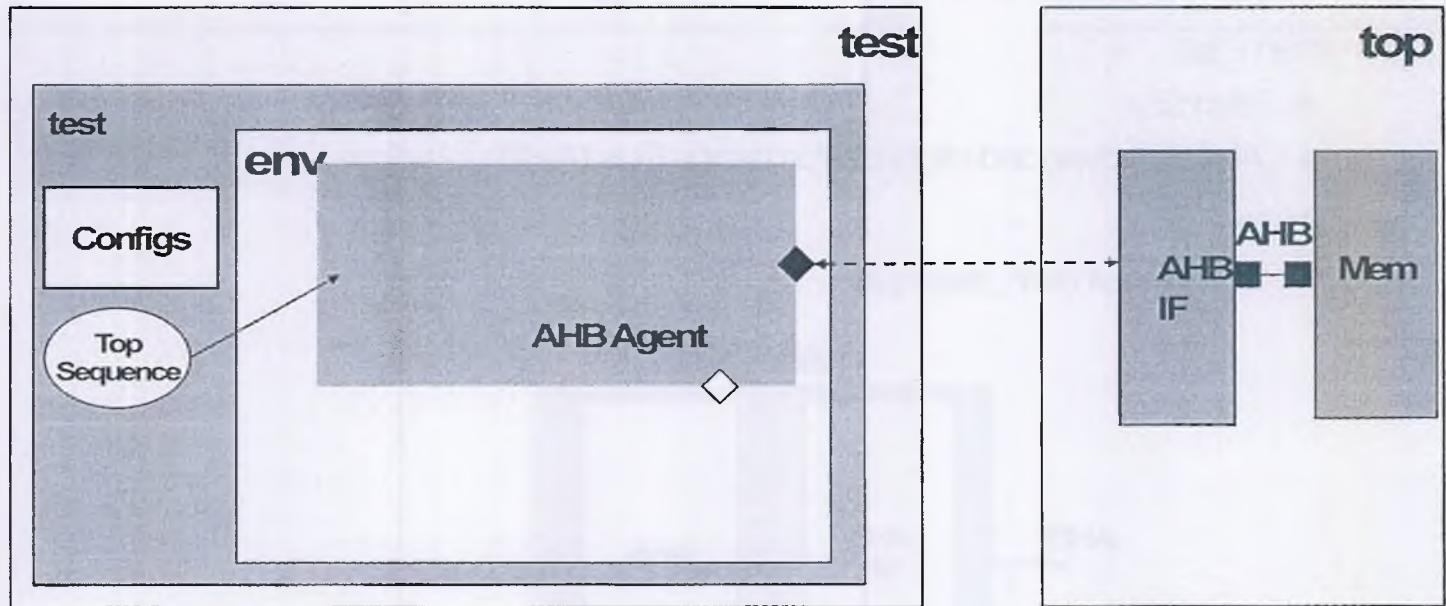
AHB Memory

- This circuit and the associated code originate from the Mentor Graphics UVM/OVM Online Methodology Cookbook
- mem_ss
 - Memory
 - AMBA Advanced High-performance Bus (AHB) interface
- Code
 - examples/mem_example



Notes:

TESTBENCH mem_ss



Notes:

Appendix A

UVM Reporting Facilities

- In this section**



Reporting methods
Reporting levels and actions

Notes:

Reporting API Methods

- A report is issued by calling one of the reporting methods
- Each method has a built in severity level (`UVM_INFO`, `UVM_WARNING`, `UVM_ERROR`, `UVM_FATAL`)
- Each method has five arguments
 - `id`: Type `string`, used as a message category
 - ◆ Used to group or separate messages
 - `mess`: Type `string`, the actual message to print/write
 - `verbosity_level`: Type `int`, the verbosity level of the message
 - `filename`: Type `string`, A filename for the location of the message
 - `line`: Type `int`, the line number of the message

```
protected function void uvm_report_info( input string id,
                                         input string mess, int verbosity = UVM_MEDIUM,
                                         string filename = "", int line = 0 );
protected function void uvm_report_warning( input string id,
                                            input string mess, int verbosity = UVM_MEDIUM,
                                            string filename = "", int line = 0 );
protected function void uvm_report_error( input string id,
                                         input string mess, int verbosity = UVM_LOW,
                                         string filename = "", int line = 0 );
protected function void uvm_report_fatal( input string id,
                                         input string mess, int verbosity = UVM_NONE,
                                         string filename = "", int line = 0 );
```

Notes:

Example Reporting

```
task run_phase(uvm_phase phase);
  alu_txn stim_txm;
  alu_txn result_txm;
  string s1, s2;
  forever begin
    stim_in_p.get(stim_txm);
    result_in_p.get(result_txm);
    txm_cnt++;
    if (stim_txm.result == result_txm.result)
      uvm_report_info("SB",$sformatf(
        "Received correct result for alu_txm %0d, mode %s ",
        result_txm.get_transaction_id(), result_txm.mode.name()));
    else begin
      st1 = $sformatf(" Received incorrect result for alu_txm %0d, \n",
                      stim_txm.get_transaction_id());
      st2 = $sformatf(" Expected alu_txm: %s", stim_txm.convert2string());
      st3 = $sformatf(" Received alu_txm: %s", result_txm.convert2string());
      `uvm_error("SB MISMATCH",{st1, st2, st3})
      error_cnt++;
    end
  end
endtask
```

\$sformatf() is like \$display() but returns a string (without a newline) instead of printing to standard out



uvm_intro_3.5

© Willamette HDL Inc.

code in examples/alu

485

Notes:

Command Line Reporting

`+UVM_VERBOSITY=<verbosity>`

- Sets initial verbosity for all components
 - ◆ Value can be one of: UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL

`+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase>`

`+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>`

- Sets verbosity for specific components at specific phases or time window
 - ◆ *id* argument can be `_ALL_` or specific message id

`+uvm_set_action=<comp>,<id>,<severity>,<action>`

- Sets the message action for a component
 - ◆ *id* and *severity* arguments can be `_ALL_` or specific values to match (no regex)
 - ◆ *action* can be `UVM_NO_ACTION` or a | separated list of other message actions

`+uvm_set_severity=<comp>,<id>,<current_severity>,<new_severity>`

- Sets a severity override for a message
 - ◆ *id* and *current_severity* arguments can be `_ALL_`

Notes:

Report Filtering: Verbosity Level

- Each report handler has a maximum verbosity level
 - If a report is received with a verbosity greater than the maximum verbosity level, it is ignored by the handler
- Maximum verbosity level of a report handler may be set by calling the `set_report_verbosity_level()` method
 - Default value is 10000

```
function void set_report_verbosity_level (int v);  
function void set_report_verbosity_level_hier (int v);
```

```
//ignore reports with verbosity greater than 100  
set_report_verbosity_level(100);
```



Notes:

Actions

- Besides displaying a message a reporting method may invoke actions
 - **UVM_NO_ACTION**
 - ◆ Do nothing
 - **UVM_DISPLAY**
 - ◆ Display report to standard output
 - **UVM_LOG**
 - ◆ Send report to a file
 - **UVM_COUNT**
 - ◆ Count up to a threshold before exiting
 - **UVM_EXIT**
 - ◆ Exit simulation immediately
 - **UVM_CALL_HOOK**
 - ◆ Call hook method

```
UVM
typedef enum uvm_action
{
    UVM_NO_ACTION = 5'b00000,
    UVM_DISPLAY   = 5'b00001,
    UVM_LOG        = 5'b00010,
    UVM_COUNT      = 5'b00100,
    UVM_EXIT       = 5'b01000,
    UVM_CALL_HOOK  = 5'b10000
} uvm_action_type;
```

Notes:

Default Severity Actions

- Each severity level has one or more actions associated with it
 - The default actions for each severity level:
 - ◆ **UVM_INFO**
 - **UVM_DISPLAY**,
 - ◆ **UVM_WARNING**
 - **UVM_DISPLAY**,
 - ◆ **UVM_ERROR**
 - **UVM_DISPLAY, UVM_COUNT**
 - ◆ **UVM_FATAL**
 - **UVM_DISPLAY, UVM_EXIT**
 - The Actions may be combined by using the "|" operator
 - ◆ Example:

```
uvm_action act = UVM_DISPLAY | UVM_EXIT // both actions happen
```

Setting Severity Actions

- The `set_report_severity_action()` method sets the severity action for a report handler
 - Changes the current (perhaps default) action for a particular severity level

Sets only this object's report handler

```
function void set_report_severity_action (input uvm_severity s,  
                                         input uvm_action a);  
function void set_report_severity_action_hier(input uvm_severity s,  
                                             input uvm_action a);
```

Sets this object's report handler and recursively sets all its children's report handlers

```
// set UVM_WARNING severity to have DISPLAY, LOG & COUNT actions  
set_report_severity_action(UVM_WARNING, UVM_DISPLAY | UVM_LOG | UVM_COUNT);
```

Notes:

id Actions

- One or more actions may be associated with a particular `id` (reporting method argument)
- By default, in a report handler, an `id` does not have an action associated with it
 - The action for a particular `id` in a particular report handler may be set by calling the `set_report_id_action()` method
- If a report handler has an action set for a particular `id`
 - This "id action" is done
 - Else the action specified by the severity level is done

```
function void set_report_id_action (input string id,  
                                  input uvm_action a);  
function void set_report_id_action_hier(input string id,  
                                       input uvm_action a);
```



Notes:

severity,id Pair Actions

- One or more actions may also be associated with a particular severity,id pair
- By default, in a report handler, a severity,id pair does not have an action associated with it
 - The action for a particular severity,id in a particular report handler may be set by calling the `set_report_severity_id_action()` method
- If a report handler has an action set for a particular severity,id pair
 - This "severity, id pair action" is done
 - Else the specified by the "id action" or the severity level is done

```
function void set_report_severity_id_action (input uvm_severity s,  
                                         input string id,  
                                         input uvm_action a);  
function void set_report_severity_id_action_hier (input uvm_severity s,  
                                                input string id,  
                                                input uvm_action a);
```

Notes:

Message Hooks

- Methods that provide for user defined actions when reporting methods are called
- Virtual functions
 - Inherited from `uvm_report_object` so may be overridden in components
 - ◆ Do nothing by default
 - Called by the reporting methods of the report handler if the `CALL_HOOK` action is specified
 - ◆ Each hook method is associated with the reporting method of corresponding severity
 - ◆ Receive same arguments as the calling reporting method
 - If the return value of the hook method is 0 then processing of the message is discontinued

```
virtual function bit report_info_hook (input string id, input string mess,  
                                     input verbosity);  
virtual function bit report_warning_hook (input string id, input string mess,  
                                         input verbosity);  
virtual function bit report_error_hook (input string id, input string mess,  
                                         input verbosity);  
virtual function bit report_fatal_hook (input string id, input string mess,  
                                         input verbosity);
```

Notes:

File Output

- If a report has a UVM_LOG action associated with it
 - Report Handler checks to see if there is a valid file handle to log the report
 - ◆ If so then it will write to the file(s) associated with the file handle
 - The handle is an ordinary Verilog handle
 - » Multiple files may be combined in one handle as a way to write to multiple files at once
- Report handle does not open or close files - user does
- Default value of the file handle is zero meaning no file will be written
- To set the default value of the file handle call the `set_report_default_file()` method

```
typedef int UVM_FILE;
function void set_report_default_file (input UVM_FILE f);
function void set_report_default_file_hier(input UVM_FILE f);
```

Notes:

File Handle Association Methods

- Methods to associate file handles with severity and id with report handlers
 - Priority is the same as for other actions

```
function void set_report_severity_file
  (input uvm_severity s,
   input UVM_FILE f);

function void set_report_id_file
  (input string id,
   input UVM_FILE f);

function void set_report_severity_id_file
  (input uvm_severity s,
   input string id,
   input UVM_FILE f);
```

Notes:

Other Reporting API Methods

```
function void report_summarize(UVM_FILE f = 0) ;
```

- Prints statistics to the standard output (optionally to a file)

```
function void dump_report_state() ;
```

- Provides a listing of all the action, file and filter settings of a report handler

```
function void set_report_max_quit_count(int m) ;
```

- Sets the number (UVM_COUNT) of reports before simulation ends
- Default value is 0 (unbounded count)

Notes:

Reporting Structure: `uvm_reporter`

- Class that can be instantiated to provide a report handler
- Another option for reports issued from modules or classes not derived from `uvm_component` classes
- Create an instance of type `uvm_reporter` inside of the module or class
 - You will have to access the methods of the `uvm_reporter` object directly

Notes:

Example Reporting - environment class

```
class alu_rtl_env extends uvm_env;
...
UVM_FILE log_file_id; ← File id
...
function void end_of_elaboration_phase(uvm_phase phase);
    log_file_id = $fopen("log_file.log");
    set_report_default_file_hier(log_file_id); ← Open log file
    set_report_severity_action_hier(UVM_WARNING, UVM_DISPLAY | UVM_LOG);
    set_report_severity_action_hier(UVM_INFO,     UVM_DISPLAY | UVM_LOG);
    set_report_severity_action_hier(UVM_ERROR,    UVM_DISPLAY | UVM_COUNT | UVM_LOG);
    set_report_severity_action_hier(UVM_FATAL,   UVM_DISPLAY | UVM_LOG | UVM_EXIT);
endfunction

function void report_phase(uvm_phase phase); // report
    report_summarize(log_file_id);
endfunction

...
endclass
```

Set all report handlers to point to this file

Open log file

Change actions to include UVM_LOG

Override the report_phase() method

Write summary to log file

Notes:

Example Reporting - scoreboard - 1

```
function void end_of_elaboration_phase(uvm_phase phase);  
    set_report_max_quit_count(1000);  
endfunction
```

Set the COUNT for errors to
1000 before stopping

Override the
report_phase()
method

```
function void report_phase(uvm_phase phase);  
    uvm_report_info("SB",$sformatf(  
        "*****\nALU did %0d operations with %0d errors\n",  
        op_cnt, error_cnt));  
endfunction
```

Notes:

Sample Reporting Output

```
# UVM_INFO ./alu_agent/cov_collector.svh(118) @ 242500: uvm_test_top.t_env.alu_agent.cov_sb [cov_msg] 2424 alu_txns,
#   Current coverage MUL = 66.666667%
# UVM_INFO @ 242600: uvm_test_top.t_env.ag.s_board [SB] Received correct result for alu_txn 2425, mode MUL
# Mode = 1111
# UVM_INFO ./alu_agent/cov_collector.svh(115) @ 242700: uvm_test_top.t_env.alu_agent.cov_sb [cov_msg]
#
# 100% coverage of MUL achieved
# ****
# UVM_INFO @ 242700: uvm_test_top.t_env.ag.s_board [SB] Received correct result for alu_txn 2426, mode MUL
# UVM_INFO ../../uvm/src/base/uvm_objection.svh(1120) @ 242700: reporter [TEST_DONE] 'run' phase is ready to proceed to
the 'extract' phase
# UVM_INFO @ 242700: uvm_test_top.t_env.ag.s_board [SB] ****
# ALU did 2426 operations with 0 errors
#
# UVM_INFO @ 242700: uvm_test_top.t_env [alu_rtl_env] Finished Test
#
#
#— UVM Report Summary —
#
# Quit count : 0 of 1000
# ** Report counts by severity
# UVM_INFO : 4856
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [RNTST] 1
# [SB] 2427
# [TEST_DONE] 1
# [alu_rtl_env] 1
# [cov_msg] 2426
```

Notes:

Sample Solutions



Notes:

Sample Solution transactions: wb_txmsvh-1

```
class wb_txm extends uvm_sequence_item;
`uvm_object_utils(wb_txm)

wb_txn_t txn_type;
bit[31:0] adr;
logic[31:0] data[1];
int unsigned count;
bit[7:0] byte_sel;

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

function void init(wb_txn_t op = NONE, bit[31:0] _adr = 0, logic[31:0] _data = 0 );
  txn_type = op;
  adr = _adr;
  data[0] = _data;
  count = 1;
  byte_sel = 4'b1111;
endfunction

function string convert2string();
  string str;
  str = { "----- Start Wishbone txn -----\\n",
    "WISHBONE txn \\n",
    $sformatf(" type      : %s\\n", txn_type.name()),
    $sformatf(" adr       : 'h%h\\n", adr),
    $sformatf(" data[0]   : 'h%h\\n", data[0]),
    $sformatf(" count     : 'h%d\\n", count),
    $sformatf(" byte_sel : 'b%b\\n", byte_sel),
    "----- End Wishbone txn -----\\n"};
  return str;
endfunction
```

Willamette
UVM
Verilog Function
uvm_intro_3.5

© Willamette HDL Inc.

502

Notes:

Sample Solution transactions: wb_txnsh-2

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    wb_txn rhs_;
    do_compare = ($cast(rhs_, rhs) &&
                  super.do_compare(rhs, comparer) &&
                  txn_type == rhs_.txn_type &&
                  adr == rhs_.adr &&
                  data == rhs_.data &&
                  count == rhs_.count &&
                  byte_sel == rhs_.byte_sel);
endfunction

function void do_copy(uvm_object rhs);
    wb_txn rhs_;
    if (!$cast(rhs_, rhs))
        `uvm_error("TYPE MISMATCH", "Type mismatch in do_copy()")
    super.do_copy(rhs);
    txn_type = rhs_.txn_type;
    adr      = rhs_.adr;
    data     = rhs_.data;
    count    = rhs_.count;
    byte_sel = rhs_.byte_sel;
endfunction

function void do_print(uvm_printer printer);
    super.do_print(printer);
    if (printer.knobs.sprint) //is this a sprint() call?
        printer.m_string = convert2string(); // set string
    else // nope a print() call
        $display(convert2string()); // print out string
endfunction
```

Back

503



uvm_intro_3.5

©Willamette HDL Inc

Notes:

Sample Solution components: stim_gen.svh

```
class stim_gen extends uvm_component;
  `uvm_component_utils(stim_gen)

  uvm_blocking_put_port #(Packet) out_p[`ROUTER_SIZE];

  int num_to_send = 1000;
  int router_size = `ROUTER_SIZE;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    for(int i = 0; i < router_size; i++)
      out_p[i] = new($sformatf("out_p_%0d",i),this);
  endfunction

  ...
endclass
```



Notes:

Sample Solution components: rtr_driver.svh

```
class rtr_driver extends uvm_driver #(Packet);
`uvm_component_utils(rtr_driver)

uvm_blocking_get_port #(Packet) in_p;
uvm_analysis_port #(Packet) drv_ap;

virtual router_bfm #(`ROUTER_SIZE) v_r_bfm;

int m_id;

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

function void build_phase(uvm_phase phase);
super.build_phase(phase);
in_p = new("in_p", this);
drv_ap = new("drv_ap", this);
endfunction

...
endclass
```

Back



uvm_intro_3.5

© Willamette HDL Inc.

505

Notes:

Sample Solution environment: stim_gen.svh

```
class stim_gen extends uvm_component;
  `uvm_component_utils(stim_gen)

  uvm_blocking_put_port #(Packet) out_p[`ROUTER_SIZE];

  int num_to_send = 1000;
  int router_size = `ROUTER_SIZE;
  ...

  task main_phase(uvm_phase phase);
    Packet txn_to_randomize;
    Packet txn_to_send;

    phase.raise_objection(this);
    txn_to_randomize = Packet::type_id::create("txn_to_randomize");
    for(int i = 1; i <= num_to_send; i++) begin
      assert (txn_to_randomize.randomize());
      txn_to_randomize.pkt_id = i; // set packet id
      $cast(txn_to_send, txn_to_randomize.clone()); // make copy to send
      out_p[txn_to_send.src_id].put(txn_to_send);
    end
    phase.drop_objection(this);
  endtask
endclass
```



Notes:

Sample Solution environment: comp_shsvh

```
class comp_sb extends whdl_ooo_comparator #(Packet, int);
`uvm_component_utils(comp_sb)

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

function void report_phase(uvm_phase phase);
    uvm_report_info("Packet_Comparator", $sformatf("Matches: %0d",
get_matches()));
    uvm_report_info("Packet_Comparator", $sformatf("Mismatches: %0d",
get_mismatches()));
    uvm_report_info("Packet_Comparator", $sformatf("Missing: %0d
\n", get_total_missing()));
endfunction

endclass
```

Notes:

Sample Solution environment: Packetsvh & testsv

```
class Packet extends uvm_sequence_item;
`uvm_object_utils(Packet)
...
endclass

-----

module test;
import uvm_pkg::*;
import test_pkg::*;

initial
    run_test("default_test"); //start test environment

endmodule
```



Notes:

Sample Solution environment: router_env.svh-1

```
class router_env extends uvm_env;
`uvm_component_utils(router_env)

int size_router;

stim_gen s_gen;
uvm_tlm_fifo #(Packet) gen2drv[`ROUTER_SIZE];
rtr_driver r_driver[`ROUTER_SIZE];
rtr_monitor r_monitor[`ROUTER_SIZE];
comp_sb s_board;
coverage_sb cov_sb;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase); //base class build
$display("Router size = %0dx%0d", `ROUTER_SIZE, `ROUTER_SIZE);
s_gen = stim_gen::type_id::create("s_gen", this);
s_board = comp_sb::type_id::create("s_board", this);
cov_sb = coverage_sb::type_id::create("cov_sb", this);
for(int i = 0; i < `ROUTER_SIZE; i++) begin
    r_driver[i] = rtr_driver::type_id::create(
        $sformatf("r_driver[%0d]", i), this);
    r_monitor[i] = rtr_monitor::type_id::create(
        $sformatf("r_monitor[%0d]", i), this);
    gen2drv[i] = new($sformatf("gen2drv[%0d]", i), this);
end
endfunction
```

W

-09

Notes:

Sample Solution environment: router_envish_2

```
function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
for(int i = 0; i < `ROUTER_SIZE; i++) begin
    // connect up drivers to stimulus generator buffers
    r_driver[i].in_p.connect(gen2drv[i].blocking_get_export);

    // connect up s_gen to buffers
    s_gen.out_p[i].connect(gen2drv[i].blocking_put_export);

    r_monitor[i].out_ap.connect(cov_sb.analysis_export);
    r_monitor[i].out_ap.connect(s_board.after_axp);

    r_driver[i].drv_ap.connect(s_board.before_axp);
end endfunction
...
endclass
```

Back



uvm_intro_3.5

© Willamette HDL Inc.

510

Notes:

Sample Solution analysis: coverage_sb

```
class coverage_sb extends uvm_subscriber #(Packet);
`uvm_component_utils(coverage_sb)

router_coverage r_cov;
int txn_cnt;
real current_coverage;
int percentage_100_cnt;
bit percentage_100_met;
.
.
endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

511

Notes:

Sample Solution analysis: rtr_monitor.svh

```
class rtr_monitor extends uvm_component;
  `uvm_component_utils(rtr_monitor)
  uvm_analysis_port #(Packet) out_ap;

  int m_id;
  virtual router_bfm #(`ROUTER_SIZE) v_r_bfm;
  . . .

  function void build_phase(uvm_phase phase);
    super.build_phase(phase); //base class build
    out_ap = new("out_ap",this);
  endfunction

  . . .
  task run_phase(uvm_phase phase);
    Packet mon_txn;
    forever begin
      mon_txn = Packet::type_id::create("mon_txn");
      v_r_bfm.read_router_port(m_id, mon_txn.src_id,
                                mon_txn.dest_id, mon_txn.payload,
                                mon_txn.pkt_id);
      out_ap.write(mon_txn);
    end
  endtask
endclass
```



Notes:

Sample Solution analysis: rtr_driver.svh

```
class rtr_driver extends uvm_driver #(Packet);
  `uvm_component_utils(rtr_driver)

  uvm_blocking_get_port #(Packet) in_p;
  uvm_analysis_port #(Packet) drv_ap;

  virtual router_bfm #(`ROUTER_SIZE) v_r_bfm;
  int m_id;

  . . .

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    in_p = new("in_p", this);
   drv_ap = new("drv_ap", this);
  endfunction

  . . .

  virtual task run_phase(uvm_phase phase);
    Packet stim_txn;
    forever begin
      in_p.get(stim_txn);
      v_r_bfm.write_router(stim_txn.src_id, stim_txn.dest_id,
                           stim_txn.payload, stim_txn.pkt_id );
      drv_ap.write(stim_txn); // broadcast txn
    end
  endtask
endclass
```

Notes:

Sample Solution analysis: router_env.svh

```
class router_env extends uvm_env;
  `uvm_component_utils(router_env);

  int size_router;
  stim_gen s_gen;
  uvm_tlm_fifo #(Packet) gen2drv[`ROUTER_SIZE];
  rtr_driver r_driver[`ROUTER_SIZE];
  rtr_monitor r_monitor[`ROUTER_SIZE];
  ...

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    for(int i = 0; i < `ROUTER_SIZE; i++) begin
      // connect up drivers to stimulus generator buffers
      r_driver[i].in_p.connect(gen2drv[i].blocking_get_export);

      // connect up s_gen to buffers
      s_gen.out_p[i].connect(gen2drv[i].blocking_put_export);

      r_monitor[i].out_ap.connect(cov_sb.analysis_export);
      r_monitor[i].out_ap.connect(s_board.after_axp);

      r_driver[i].drv_ap.connect(s_board.before_axp);
    end
  endfunction

  ...
endclass
```

Notes:

Sample Solution hierarchy: router_std_agent

```
class router_std_agent extends router_agent_base;
`uvm_component_utils(router_std_agent)

  // component handles
  std_drivers      r_drv;
  stim_gen         s_gen;

  // constructor not shown

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    r_drv = std_drivers::type_id::create("r_drv", this);
    s_gen = stim_gen::type_id::create("s_gen", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    for(int i = 0; i < m_config.m_router_size; i++)
      //connect up stimulus generator ports to rtr_std_drivers ports
      s_gen.out_p[i].connect(r_drv.in_xp[i]);

    // connect up analysis ports
    r_drv.stim_ap.connect(stim_ap);
  endfunction
endclass
```

Back

515



uvm_intro_3.5

© Willamette HDL Inc.

Notes:

factory_override: Packet_pass_thru & test_std

```
class Packet_pass_thru extends Packet;
  `uvm_object_utils(Packet_pass_thru)
  constraint pass_thru {dest_id == src_id;} . . .
endclass

class test_std_overrides extends test_base;
  `uvm_component_utils(test_std_overrides)

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

function void build_phase(uvm_phase phase);
  // factory overrides
  // router_agent_base::type_id::set_inst_override(router_std_agent::get_type(),"*");
  // router_agent_base::type_id::set_inst_override(router_std_agent::get_type(),
  // "*router_agent");
  router_agent_base::type_id::set_inst_override(router_std_agent::get_type(),
    "uvm_test_top.env.router_agent");
  // Replace router coverage
  router_coverage_base::type_id::set_type_override(router_coverage_pass_thru::get_type());

  // Replace Packet with Packet_pass_thru
  Packet::type_id::set_type_override(Packet_pass_thru::get_type());
  factory.print(1);
  super.build_phase(phase); //after overrides & topology configs
endfunction

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

Back



uvm_intro_3.5

©Willamette HDL Inc.

516

Notes:

Sample Solution configuration: top.sv

```
module top;
    ...
    router_bfm #(ROUTER_SIZE) r_bfm (clk);
    router_rtl #(ROUTER_SIZE) router( . . . ) ;

initial
    uvm_config_db #(virtual router_bfm #(ROUTER_SIZE))::set(
        null,"*","r_bfm",r_bfm); //set virtual interface

endmodule
```

```
// configuration container class

class router_config extends uvm_object;
    `uvm_object_utils( router_config );
    virtual router_bfm #(ROUTER_SIZE) v_r_bfm;

    int m_num_routers;    // number of routers in configuration
    int m_router_id;      // ID of router DUT
    int m_router_size = ROUTER_SIZE; // size of the router
    uvm_sequencer #(Packet) rtr_seqr; // router agent sequencer
    function new( string name = "" );
        super.new( name );
    endfunction
endclass
```



Notes:

Sample Solution configuration test_base.svh

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

router_env env; // environment
router_config m_r_config; // Config object

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

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

    // init router config object
    m_r_config = new("m_r_config"); // create config object
    if(!uvm_config_db #(virtual router_bfm #(ROUTER_SIZE))::get(
        this, "", "r_bfm", m_r_config.v_r_bfm)) // get virtual interface container
        `uvm_fatal("CONFIG", "Config get error")

    m_r_config.m_router_size = ROUTER_SIZE; // redundant since this is the default
    m_r_config.m_router_id = 0; // ID of single router
    m_r_config.m_num_routers = 1; // only one router

    // write config object into config_db
    uvm_config_db #(router_config)::set(this, "*", "router_config", m_r_config);
    env = router_env::type_id::create("env", this);
endfunction
endclass
```



Notes:

Sample Solution configuration: rtr_driver.svh

```
class rtr_driver extends uvm_driver #(Packet);
  `uvm_component_utils(rtr_driver)

  uvm_blocking_get_port #(Packet) in_p;
  uvm_analysis_port #(Packet) drv_ap;

  router_config m_config; // Config object
  virtual router_bfm #(ROUTER_SIZE) v_r_bfm;
  int m_id;
  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // get config object
    if(! (uvm_config_db #(router_config)::get(this,"","router_config",m_config)))
      `uvm_fatal("CONFIG", "get() error");
    v_r_bfm = m_config.v_r_bfm; // assign local virtual interface
    // create ports
    in_p = new("in_p", this);
    drv_ap = new("drv_ap", this);
  endfunction

  ...
endclass
```



Notes:

Sample Solution configuration: router_agent_base.svh

```
class router_agent_base extends uvm_component;
`uvm_component_utils(router_agent_base)

uvm_analysis_port #(Packet) stim_ap;
uvm_analysis_port #(Packet) mon_ap;

monitors r_mon; // router monitors
router_config m_config; // Config object

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

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

// get config object
if(! (uvm_config_db #(router_config)::get(this,"","router_config",m_config)))
`uvm_fatal("CONFIG", "get() error");

// create components
stim_ap = new("stim_ap", this);
mon_ap = new("mon_ap", this);
r_mon = monitors::type_id::create("r_mon", this);
endfunction

...
endclass
```

Back

Notes:

Sample Solution sequences:

```
class hammer_0 extends uvm_sequence #(Packet, Packet);
`uvm_object_utils(hammer_0)

Packet txn;
static int next_pkt_id;

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

task body();
for (int i = 0; i < 100; i++) begin
  txn = Packet::type_id::create("txn");
  start_item(txn);
  if(!(txn.randomize() with {dest_id == 0;}))
    `uvm_fatal("hammer_0", "Randomization Error")
  txn(pkt_id = next_pkt_id++);
  finish_item(txn);
end
endtask
endclass
```

```
class router_seq_agent extends router_agent_base;
`uvm_component_utils(router_seq_agent)

// component handles
seq_drivers r_drv;
uvm_sequencer #(Packet, Packet) router_seqr;

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

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
// create components
r_drv = seq_drivers::type_id::create("r_drv", this);
router_seqr = new("router_seqr",this);
m_config.rtr_seqr = router_seqr; // assign sequencer
endfunction

function void connect_phase(uvm_phase phase);
super.connect_phase(phase);

// connect up monitor analysis ports
r_mon.mon_ap.connect(r_drv.rsp_axp); // driver response

// connect sequencer and driver
r_drv.seq_item_port.connect(router_seqr.seq_item_export);
// connect stim analysis ports
r_drv.stim_ap.connect(stim_ap);

endfunction
endclass
```

Notes:

Sample Solution sequences: test_hammer_0

```
class test_hammer_0 extends test_base;
`uvm_component_utils(test_hammer_0)

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

task main_phase(uvm_phase phase);
  // create the main sequence
  hammer_0 main_seq = hammer_0::type_id::create("main_seq");
  // start up the main sequence
  phase.raise_objection(this, "Begin stimulus generation");
  main_seq.start( m_r_config.rtr_seqr , null);
  phase.drop_objection(this, "End stimulus generation");
endtask

endclass
```

Notes:

Sample Solution multiple_sequences: hammer_n

```
class hammer_n extends uvm_sequence #(Packet, Packet);
`uvm_object_utils(hammer_n)
Packet txn;
int hammer_port_num;
static int next_pkt_id;

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

task body();
repeat(100) begin
  txn = Packet::type_id::create("txn");
  start_item(txn);
  if(! (txn.randomize() with {txn.dest_id == hammer_port_num; }))
    `uvm_fatal("hammer_n", "Randomization Error")
  txn.pkt_id = next_pkt_id++;
  finish_item(txn);
end
endtask
endclass
```



Notes:

Sample Solution multiple_sequences:

```
class hammer_multiple extends uvm_sequence #(Packet, Packet);
`uvm_object_utils(hammer_multiple)
hammer_n hn[];
router_config m_config; // Config object
function new(string name = "");
super.new(name);
endfunction
function void config_hammer();
// Get router size
if(! (uvm_config_db #(router_config)::get(m_sequencer, "", "router_config", m_config)))
`uvm_fatal("RSRCNF", $sformatf("resource id: router_config not found"));
hn = new[m_config.m_router_size];
endfunction
task body();
config_hammer();
for (int i = 0; i < m_config.m_router_size; i++) begin
int j = i; // Need a more "local" variable because i's scope forces late evaluation
of expressions in the fork
fork
begin
// Create the sub-sequence
hn[j] = hammer_n::type_id::create($sformatf("h[%0d]", j));
hn[j].hammer_port_num = j; // Set the port number to hammer
hn[j].start(m_sequencer, this); // Start the sub-sequence
end
join_none
end
wait fork;
endtask
endclass
```

WILLAMETTE
HDL uvm_intro_3.5

© Willamette HDL Inc.

Back

524

Notes:

Sample Solution Integration: test_dma_base

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

...
wb_env env; // WISHBONE bus environment
wb_config wb_config_0; // config object for WISHBONE BUS
virtual wb_reset_if v_wb_reset; // virtual reset interface
dma_system_block m_dma_system_block; // WISHBONE bus DMA register model
wb_mem mem0, mem1;

...
function void build_phase(uvm_phase phase);
super.build_phase(phase);
// set configuration
wb_config_0 = new();

...
// Create the register model
m_dma_system_block = dma_system_block::type_id::create("m_dma_top_block");
m_dma_system_block.build(); // Build the register model
uvm_config_db #(dma_system_block)::set(this, "env", "dma_system_block",
    m_dma_system_block);
uvm_config_db #(dma_top_block)::set(this, "env.*", "dma_top_block",
    m_dma_system_block.dma_top);
...
endclass
```

Notes:

Sample Solution Integration: reg2wb_adapter - 1

```
class reg2wb_adapter extends uvm_reg_adapter;
  `uvm_object_utils(reg2wb_adapter)

  function new(string name = "reg2wb_adapter");
    super.new(name);
  endfunction

  // conversion from uvm_reg_bus_op to wb_txn
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op
                                             rw);
    logic [31:0] data [1];
    wb_txn txn = wb_txn::type_id::create("txn");
    data[0] = rw.data;
    if(rw.kind == UVM_READ)
      txn.init_txn(READ, rw.addr, data);
    else begin
      txn.init_txn(WRITE, rw.addr, data);
    end
    return txn;
  endfunction
```



Notes:

Sample Solution Integration: reg2wb_adapter - 2

```
// conversion from wb_txn type to uvm_reg_bus_op
virtual function void bus2reg(uvm_sequence_item bus_item,
                               ref uvm_reg_bus_op rw);
    wb_txn txn; // txn handle

    if (!$cast(txn, bus_item)) begin
        `uvm_fatal("NOT_WISHBONE_TXN_TYPE", "Provided bus_item is not of the
correct type")
        return;
    end
    if(txn.txn_type == READ)
        rw.kind = UVM_READ;
    else if (txn.txn_type == WRITE)
        rw.kind = UVM_WRITE;
    else `uvm_fatal("NOT_READ_WRITE_TXN", "Provided bus_item is not a read or
write transaction")
    rw.addr = txn.adr;
    rw.data = txn.data[0];
    rw.status = txn.t_status;
endfunction

endclass
```



Notes:

Sample Solution Integration: wb_env-1

```
class wb_env extends wb_env_base;
`uvm_component_utils(wb_env)

wb_scoreboard_base wb_mem_sb_0; // scoreboard for wb slave 0
wb_scoreboard_base wb_mem_sb_1; // scoreboard for wb slave 1
reg2wb_adapter m_reg2wb; // Register layer adapter
uvm_reg_predictor #(wb_txn) wb2reg_predictor; // Register predictor
dma_system_block m_dma_system_block; // WISHEBONE bus DMA system register model
wb_config m_config;
// constructor
function new( string name, uvm_component parent);
super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
string s_name;
super.build_phase(phase);
// create wishbone stuff
if(!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config))
`uvm_fatal("CONFIG", "config error: wb_config not found");
// get register model handle
if(!uvm_config_db #(dma_system_block)::get(this, "", "dma_system_block",
m_dma_system_block))
`uvm_error("CONFIG", "dma_system_block not found");
wb_mem_sb_0 = wb_mem_scoreboard #(0)::type_id::create("wb_mem_sb_0", this);
wb_mem_sb_1 = wb_mem_scoreboard #(1)::type_id::create("wb_mem_sb_1", this);
wb2reg_predictor = uvm_reg_predictor #(wb_txn)::type_id::create
("wb2reg_predictor", this);
m_reg2wb = reg2wb_adapter::type_id::create("m_reg2wb", this);
endfunction
```



Notes:

Sample Solution Integration wb_env-2

```
function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //Adapter configuration
    m_reg2wb.provides_responses = 0; // set provides_responses to explicit
    // Register model map configuration
    // set the map sequencer to point to wishbone agent sequencer and
    // map adapter to the reg2wb adapter
    m_dma_system_block.dma_system_map.set_sequencer(
        agent.wb_seqr, // set map sequencer
        m_reg2wb); // set map adapter
    // Register prediction configuration
    wb2req_predictor.map = m_dma_system_block.dma_system_map; // set predictor map
    wb2req_predictor.adapter = m_reg2wb; // set predictor adapter
    m_dma_system_block.dma_system_map.set_auto_predict(0); // turn off auto-prediction
    // connect ports, exports
    agent.mon_ap.connect(wb2req_predictor.bus_in); // connect predictor
    agent.mon_ap.connect(wb_mem_sb_0.wb_txn_axp); // connect scoreboard 0
    agent.mon_ap.connect(wb_mem_sb_1.wb_txn_axp); // connect scoreboard 1
endfunction
endclass
```



Notes:

Sample Solution Use: test_reset_regs

```
class test_reset_regs extends test_dma_base;
  `uvm_component_utils(test_reset_regs)

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

  task main_phase(uvm_phase phase);
    // create main sequence
    uvm_req_hw_reset_seq m_seq = uvm_req_hw_reset_seq::type_id::create("m_seq");
    phase.raise_objection(this);
    //init the sequence
    m_seq.model = m_dma_system_block.dma_top;
    // start up the main sequence
    m_seq.start(wb_config_0.wb_seqr);
    phase.drop_objection(this);
  endtask

endclass
```



uvm_intro_3.5

©Willamette HDL Inc.

530

Notes:

Sample Solution Use: dma_init_seq

```
class dma_init_seq extends wb_mem_map_access_base_seq;
  logic [31:0] srce_data[], dest_data[];
  task body;
    wb_txn wb_rd_results;
    super.body(); // there is stuff in super.body() that needs to be run
    init_dma(); // init the dma's registers
  endtask
  virtual task init_dma(); // init dma core
    wb_txn wb_rd_results;
    uvm_status_e op_status;
    logic [31:0] wt_data, rd_results, get_results;
    INT_MSK_n int_msk_a;
    // set up interrupt masks
    $cast(int_msk_a, m_main_block.get_reg_by_name("INT_MSK_A"));
    int_msk_a.INT_E_CH0.set(1); // Enable DMA Channel 0 interrupts
    int_msk_a.INT_E_CH1.set(1); // Enable DMA Channel 1 interrupts
    int_msk_a.INT_E_CH2.set(1); // Enable DMA Channel 2 interrupts
    int_msk_a.INT_E_CH3.set(1); // Enable DMA Channel 3 interrupts
    // write desired value to dma with an update()
    m_main_block.INT_MSK_A.update(op_status, .path(UVM_FRONTDOOR));
    m_main_block.INT_MSK_A.read(op_status, rd_results);
    `uvm_info("DMA_SETUP_INFO",
      $sformatf("main_block.INT_MSK_A read() result: %h", rd_results), UVM_HIGH);
    // clear CSR Registers by writing 0
    foreach(m Chan_block[i])
      m Chan_block[i].Chn_CSR_reg.write(op_status, 0);
    // clear interrupt source by reading it
    m_main_block.INT_SRC_A.read(op_status, rd_results);
    `uvm_info("DMA_SETUP_INFO",
      $sformatf("main_block.INT_SRC_A read() result: %h", rd_results), UVM_HIGH);
  endtask
endclass
```

Notes:

Sample Solution Use: dma_simple_seq-1

```
class dma_simple_seq extends wb_mem_map_access_base_seq;
  `````` set up a dma transfer in the wishbone dma chip
 virtual task dma_setup(
 int chan_num,
 int sroe_addr, int dest_addr,
 int tot_sz = 1,
 bit[2:0] ch_priority = 0,
 int chk_sz = 0
);
 string chan = $sformatf("CH_%0d.", chan_num);
 uvm_status_e op_status;
 logic [31:0] wt_data, rd_results, get_results;
 CHn_CSR chn_csr_reg;
 wt_data = 0; // clear all bits
 wt_data[11:0] = tot_sz; // total words to transfer
 wt_data[24:16] = chk_sz; // chunk size
 m_chan_block[chan_num].CHn_SZ_reg.write(op_status, wt_data);
 m_chan_block[chan_num].CHn_SZ_reg.read(op_status, rd_results);

  `````` set up Channel address Registers
  `````` source address
 m_chan_block[chan_num].CHn_A0.write(op_status, sroe_addr);
 m_chan_block[chan_num].CHn_A0.read(op_status, rd_results);
 m_chan_block[chan_num].CHn_A1.write(op_status, dest_addr);
 m_chan_block[chan_num].CHn_A1.read(op_status, rd_results);
  `````` set up Channel Configuration Status Register
  $cast(chn_csr_reg, m_chan_block[chan_num].get_reg_by_name("CHn_CSR_reg"));
  chn_csr_reg.CH_EN.set(1); // Channel Enabled
  chn_csr_reg.SRC_SEL.set(0); // Interface 0(A) is source
  chn_csr_reg.DST_SEL.set(0); // Interface 0(A) is destination
  chn_csr_reg.INC_DST.set(1); // Increment destination address
```

Notes:

Sample Solution Use dma_simple_seq-2

```
chn_csr_reg.INC_SRC.set(1); // Increment source address
chn_csr_reg.MODE.set(0); // Normal mode
chn_csr_reg.ARS.set(0); // off - enable auto restart
chn_csr_reg.USE_ED.set(0); // off - use external descriptor

chn_csr_reg.SZ_WB.set(0); // off - used in conjunction with USE ED
chn_csr_reg.STOP.set(0); // off - stop current transfer
// BUSY, DONE, ERR are Read only (bits 12:10)

chn_csr_reg.CH_PRIORITY.set(ch_priority); // Channel priority

chn_csr_reg.REST_EN.set(0); // off - Enable Hardware restart
chn_csr_reg.INE_ERR.set(1); // Enable interrupt on errors
chn_csr_reg.INE_DONE.set(1); // Enable interrupt when channel is done
chn_csr_reg.INE_CHK_DONE.set(0); // off - Enable interrupt after each chunk

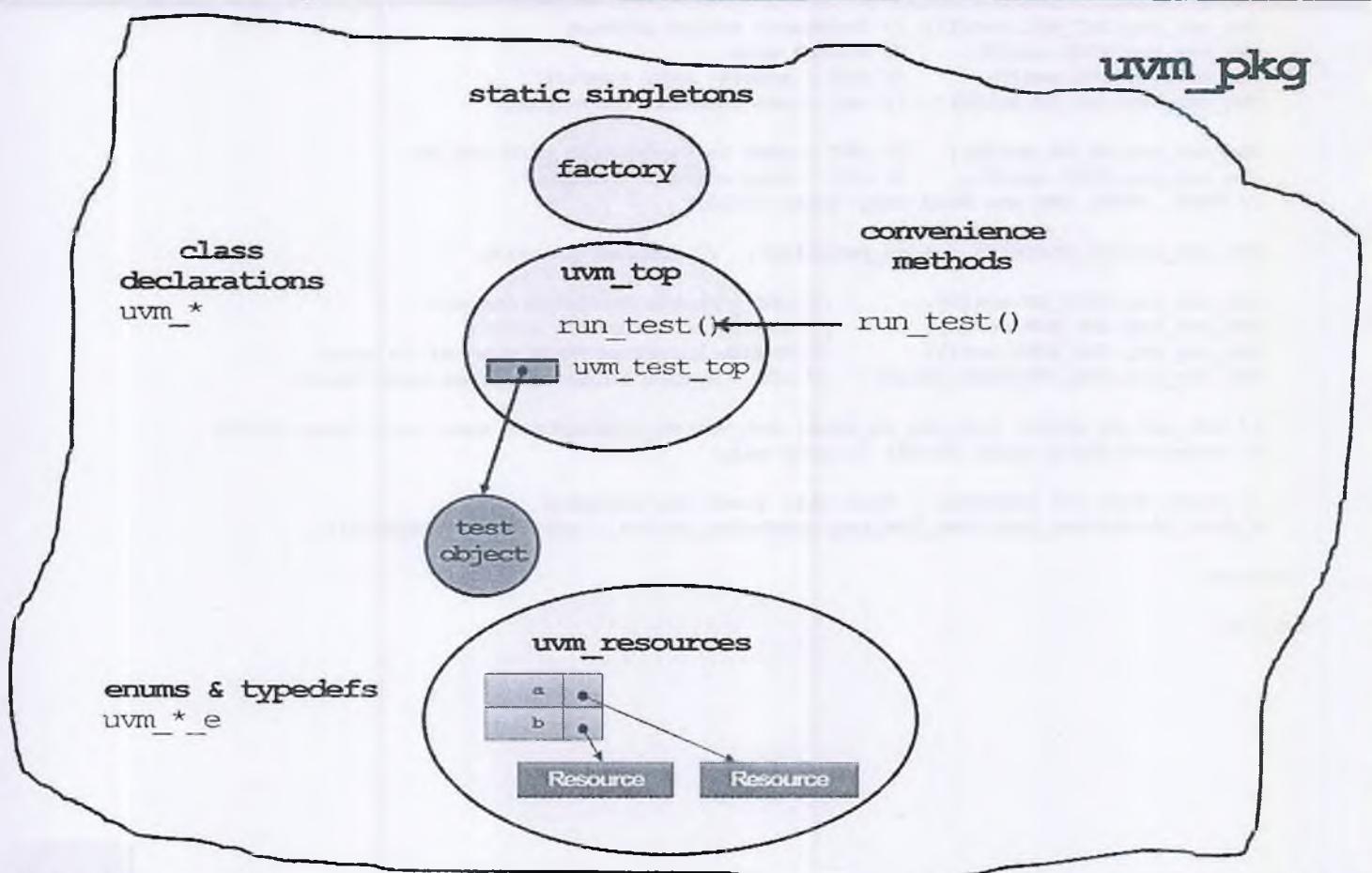
// int_src_ch_error, int_src_ch_done, int_src_ch_xferred are read only (bits 22:20)
// Reserved field (bits 31:23) is read only

// write chan csr register - this will start the transfer
m_chan_block[chan_num].CHn_CSR_reg.update(op_status, .path(UVM_FRONTDOOR));

endtask
...
endclass
```

Notes:

uvm_pkg Contents



Notes:

SystemVerilog Simulation Steps

Compilation

- Parse and analyze code
 - Syntax checking, type declarations etc.

Elaboration

1 - "HDL" elaboration

- Allocation of module & interface instances, ports, always & initial blocks, net connections etc.
 - Customization via parameters
- Hierarchy set

2 - "class statics" elaboration

- Allocation of
 - Class static methods (function int myfunc ();)
 - Class static properties (int a;)
 - optional: static initialization (a = myfunc ();)

Run-time

- Processes execute, simulation time kept
- "Stuff" created during elaboration may not be created nor destroyed in run-time
 - Module & interface instances, ports, always & initial blocks etc.
 - Class statics
- Class objects may be created/destroyed
 - Customization via constructors

Back

Notes:

Symbol Key

- TLM port
- TLM export or Analysis export
- Analysis Port
- Virtual interface
- RTL/Gate pin-level port

→ Direction of data flow



process

Notes:
