Transaction-Based Testing with OSVVM and the OSVVM Model Library

Jim Lewis, SynthWorks Design Inc
Patrick Lehmann, PLC2 gmb

Transaction-Based Testing with OSVVM and the OSVVM Model Library

This material is derived from SynthWorks' class, VHDL Testbenches and Verification

This material is updated from time to time and the latest copy of this is available at
http://www.SynthWorks.com/papers

Contact Information
Jim Lewis, President
SynthWorks Design Inc
11898 SW 128th Avenue
Tigard, Oregon 97223
503-590-4787
jim@SynthWorks.com

www.SynthWorks.com

Transaction-Based Testing with OSVVM and the OSVVM Model Library

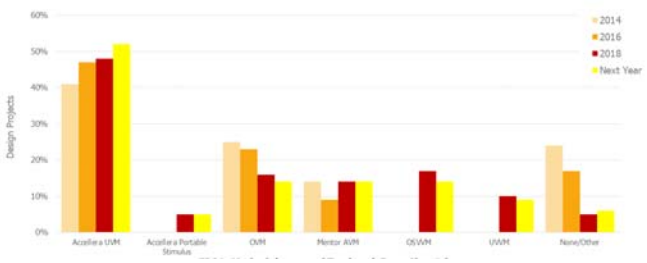
- Agenda
  - Why VHDL? Why OSVVM?
  - What is OSVVM?
  - Transactions
  - OSVVM Testbench Framework
  - A Simple Test
  - Self-Checking
  - Logs
  - Constrained Random
  - Scoreboards
  - Functional Coverage
  - Intelligent Coverage Random

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Why VHDL? Why OSVVM?

- From Wilson Research Group and Mentor, A Siemens Business, 2018 Functional Verification Survey
  - For FPGA design, 62% of all designs world wide use VHDL
  - For FPGA verification, 45% use VHDL, 17% use OSVVM - world wide
    - OSVVM is #1 VHDL FPGA verification methodology world wide



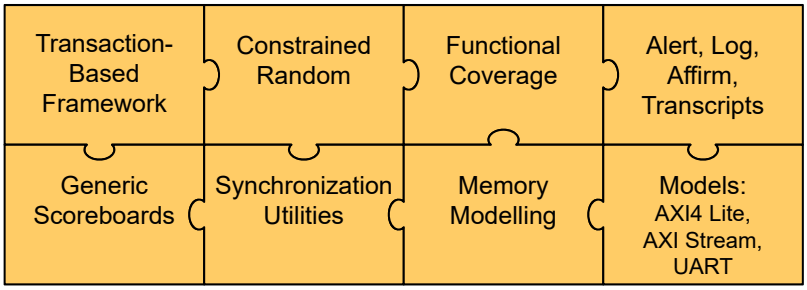
  - For FPGA verification in Europe, 30% use OSVVM, 20% use UVM
    - OSVVM is #1 FPGA verification methodology in Europe

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

What Is OSVVM?

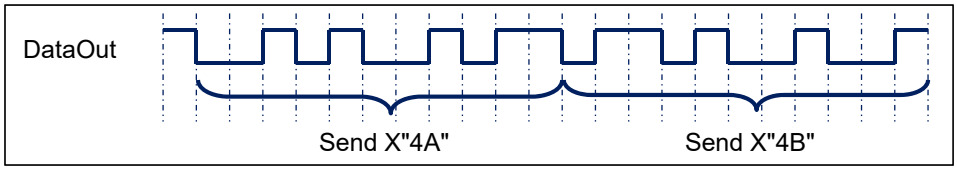
- Open Source VHDL Verification Methodology (OSVVM) =
  - A powerful and concise library and verification methodology



- OSVVM Benefits
  - Tests are Readable and Reviewable by All
  - RTL designers can write Tests and Testbench Models
  - Reuse and/or upgrade your current VHDL testbench and models
  - Supports mixed approaches (directed, algorithmic, file, CR, IT)

OSVVM Model Transactions

- Transaction = Interface Operation (Send) or Directive (NoOp).



- OSVVM Transaction Implementation
  - Transaction Initiation = procedure call
  - Transaction Implementation = either procedure or entity (VIP)

```
UartTbTxProc : process
begin
  WaitForBarrier(StartTest) ;
  Send(UartTxRec, X"4A") ;
  Send(UartTxRec, X"4B") ;
```

- Results: Simple, readable, maintainable, accelerated test construction

OSVVM Model Transactions

- Bus Master Transactions

```
MasterWrite(    AxiMasterTransRec, X"AAAA_AAA0", X"55" ) ;
MasterRead (    AxiMasterTransRec, X"1111_1110", Data) ;
MasterReadCheck(AxiMasterTransRec, X"AAAA_AAA0", X"55");
```

- Network / Bus Transfer Transactions

```
Send (UartTxRec, X"55" ) ;
Get  (UartRxRec, Data) ;
Check(UartRxRec, X"55");
```

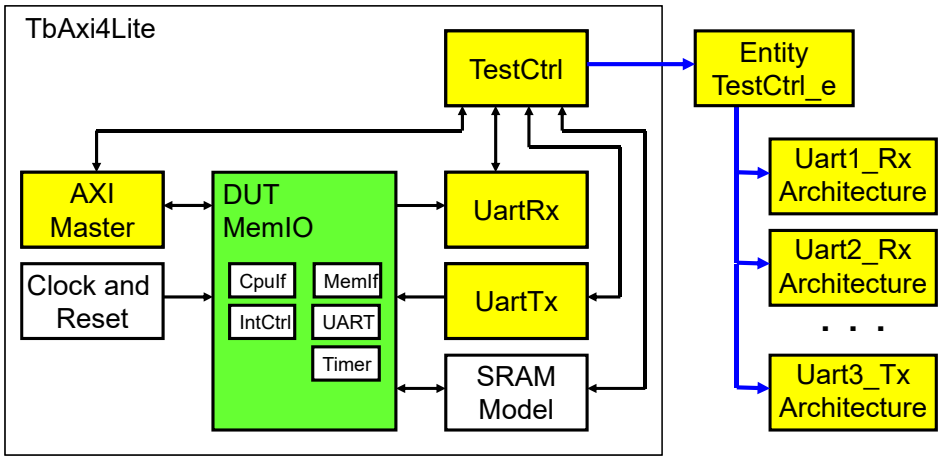
- Common Directive Transactions

```
NoOp(AxiMasterTransRec, 2) ;
```

- By using a set of methodology defined transactions,
  - We can focus on writing model behavior
  - The record for communication and transactions can be reused.

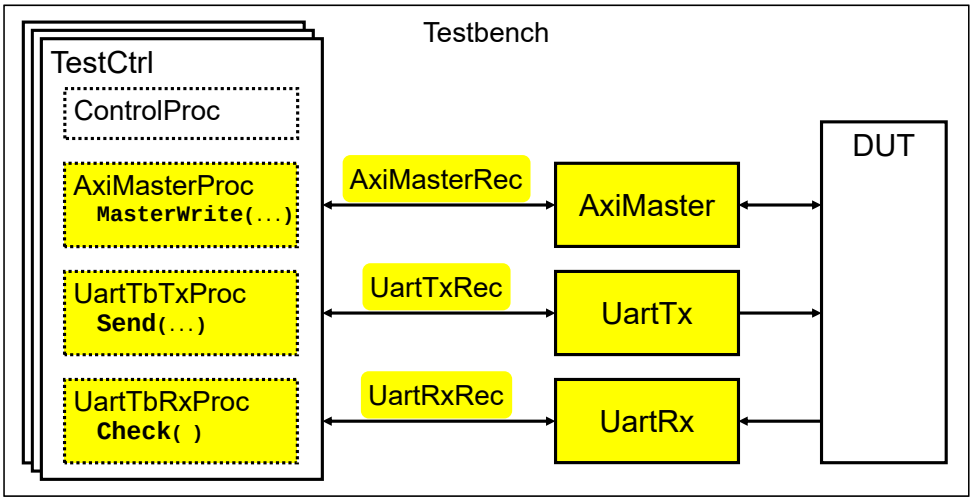
OSVVM Testbench Framework

- Looks identical to a SystemVerilog framework:
  - Verification models implement interface waveforms
  - Test Sequencer = TestCtrl = sequences of transactions



Test Case = TestCtrl Architecture

- Test case = sequence of procedure calls in TestCtrl
- Records transfer transaction information to the models
- Models implement interface waveforms



TestCtrl = Transaction Source

```
entity TestCtrl is
  generic (
    tperiod_Clk        : time := 10 ns
  ) ;
  port (
    UartTxRec    : InOut UartRecType ;
    UartRxRec    : InOut UartRecType ;

    MasterRec    : InOut Axi4LiteMasterTransactionRec ;

    Clk          : In    std_logic ;
    nReset       : In    std_logic
) ;
end TestCtrl ;
```

Recommendation:
Keep TestCtrl entity in a separate file from the architecture(s).
Facilitates using multiple architectures.

```
architecture UartTx1 of TestCtrl is
  . . .
begin
  ControlProc : process
  begin
    . . .
    WaitForBarrier(TestDone, 5 ms) ;
    ReportAlerts ;
    std.env.stop;
  end process ;
  CpuTestProc : process
  begin
    wait until nReset = '1' ;
    MasterWrite(. . .) ;
    WaitForBarrier(CpuRdy);
    . . .
    WaitForBarrier(TestDone) ;
  end process ;
  UartTbTxProc : process
  begin
    WaitForBarrier(CpuRdy);
    Send(. . .) ;
    . . .
    WaitForBarrier(TestDone) ;
  end process ;
  . . .
```

Aspects of a Test Sequencer

- Whole test in one file
- Control Process
  - Initialize & finalize test
- One process per interface
  - Concurrent, just like design
- Tests = calls to transactions
  - Easy to write and read
- Easy to add
  - Directed Tests
  - Functional Coverage
  - Constrained Random
  - Mix of test approaches
- Synchronization
- Error Reporting & Messaging

A Simple Directed Test

- Send transaction on one interface
- Get transaction on another interface and then check with AffirmIf

```
TxProc : process

begin
  Send (TRec, X"10") ;

  Send (TRec, X"11") ;

  . . .
end process TxProc ;
```

```
RxProc : process
  variable RxD : ByteType;
begin
  Get(RRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"10");
  Get(RRec, RxD) ;
  AffirmIfEqual(TBID, RxD, X"11");
  . . .
end process RxProc ;
```

- AffirmIfEqual output in TB uses TB ID ("In TB,")

```
%% Alert ERROR  In TB, Received: 08 /= Expected: 10 at 2150 ns
```

```
%% Log   PASSED In TB, Received: 10 at 2150 ns
```

A Simple Directed Test, Part 2

- Send transaction on one interface
- Check transaction on another interface - moves checking to model

```
TxProc : process

begin
  Send (TRec, X"10");

  Send (TRec, X"11");

  . . .

  WaitForBarrier(TestDone);
end process TxProc ;
```

```
RxProc : process

begin
  Check(RRec, X"10");

  Check(RRec, X"11");

  . . .

  WaitForBarrier(TestDone);
end process RxProc ;
```

- Checkers in the model use the Model ID (UartRx_1):

```
%% Alert ERROR  In UartRx_1, Received: 08 /= Expected: 10 at 2150 ns
```

```
%% Log   PASSED In UartRx_1, Received: 10 at 2150 ns
```


Adding Logs

- Logs are for conditional test printing

```
TxProc : process
begin
  SetLogEnable(DEBUG, TRUE) ;
  SetLogEnable(CpuID, INFO, TRUE) ;

  Log(TbID, "Sequence 1 Starting", ALWAYS) ;
  Send (TRec, X"10");
  Send (TRec, X"11");
  Log(TbID, "Test Last Failed Here", DEBUG) ;
  . . .
```

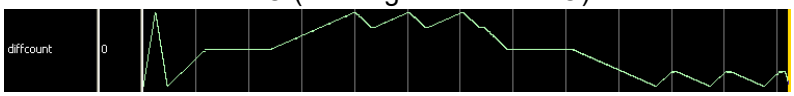
  - Log Levels: ALWAYS (default), DEBUG, INFO, FINAL, PASSED
  - Logs only print when enabled.
  - Logs are disabled by default, except ALWAYS

- Log output for above

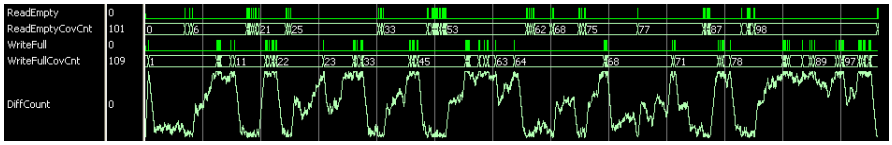
```
%% Log   ALWAYS In TB, Sequence 1 Starting at 2200 ns
%% Log   DEBUG  In TB, Test Last Failed Here at 4800 ns
```

Using Randomization

- What: Randomize values, modes, operations, and sequences
  - Ideal for large variety of similar items
- Directed test of a FIFO (tracking words in FIFO):



- Constrained Random test of a FIFO:



- Benefits: Generates realistic stimulus in a timely fashion (to write)
- If it is not test faster to write or more thorough, then use another method.

Using Randomization

- Randomize a value in an inclusive range, 0 to 15, except 5 & 11

```
Data1 := RV.RandInt(Min => 0, Max => 15) ;
Data2 := RV.RandInt(0, 15, (5,11) ) ;  -- except 5 & 11
```

- Randomize a value within the set (1, 2, 3, 5, 7, 11), except 5 & 11

```
Data3 := RV.RandInt( (1,2,3,5,7,11) ) ;
Data4 := RV.RandInt( (1,2,3,5,7,11), (5,11) ) ;
```

- Weighted Randomization: Weight, Value = 0 .. N-1

```
Data5 := RV.DistInt ( (7, 2, 1) ) ;
```

- Weighted Randomization: Value + Weight

```
. . .                -- ((val1, wt1), (val2, wt2), ...)
Data6 := RV.DistValInt( ((1,7), (3,2), (5, 1)) ) ;
```

- By itself, this is not constrained random.

Constrained Random

- CR = Randomization + Code + Transaction Calls

```
TxProc : process
  variable RV : RandomPType ;
  . . .
for I in 1 to 10000 loop
  case RV.DistInt( (70, 10, 10, 5, 5) ) is
    when 0  =>    -- Nominal case   70%
      Operation := UARTTB_NO_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;

    when 1  =>    -- Parity Error   10%
      Operation := UARTTB_PARITY_ERROR ;
      TxD:= RV.RandSlv(0, 255, Data'length) ;

    when  . . .  -- (2, 3, and 4)
  end case ;

  Send(UartTxRec, TxD, Operation) ;
end loop ;
 . . .
```

Randomize Operation

Nominal 70%

Parity Error 10%

Do Transaction

Constrained Random and Checking?

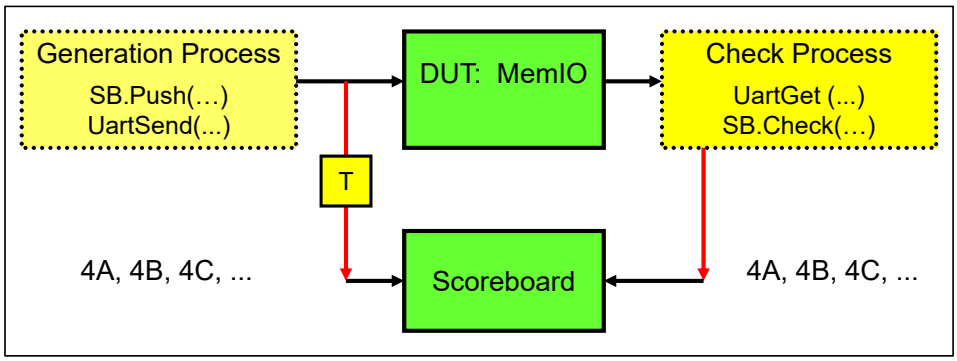
To check, RxProc could repeat the randomization in TxProc, however, this is tedious and potentially error prone.

```
TxProc : process
 variable TxD : ByteType;
 variable RV  : RandomPType;
begin
  for I in 1 to 10000 loop
    case RV.DistInt((. . .)) is
      . . .
    end case ;

    Send(TRec, TxD, Op);
  end loop ;

  . . .

  WaitForBarrier(TestDone);
end process TxProc ;
```

```
RxProc : process
  variable ExpD : ByteType;
  variable RV   : RandomPType;
begin
  for I in 1 to 10000 loop
    case RV.DistInt((. . .)) is
      . . .
    end case ;

    Check(TRec, ExpD, Op);
  end loop ;

  . . .

  WaitForBarrier(TestDone);
end process RxProc ;
```

Scoreboards

- Simplify self-checking when data is minimally transformed



- OSVVM scoreboards use package generics and have:
  - FIFO + Methods - can also be used as a FIFO
  - Handles small data transformations
  - Handles out of order execution
  - Handles dropped values

Scoreboards: Generics

```
package ScoreBoardGenericPkg is

generic(
  type ExpectedType ;
  type ActualType ;
  function Match( . . . ) return boolean ;
  function expected_to_string( . . . ) return string ;
  function actual_to_string  ( . . . ) return string
) ;

  type ScoreBoardPType is protected
    procedure Push (…) ;
    procedure Check (…) ;
    procedure Pop (…) ;
    procedure SetAlertLogId (…) ;
    . . .
  end protected ScoreBoardPType ;
end ScoreBoardGenericPkg ;
```

Scoreboards: Generic Instance

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;

package ScoreBoardPkg_slv is new osvvm.ScoreBoardGenericPkg
 generic map (
   ExpectedType        =>  std_logic_vector,
   ActualType          =>  std_logic_vector,
   match               =>  std_match,
   expected_to_string  =>  to_string,
   actual_to_string    =>  to_string
 ) ;
```

```
package ScoreBoardPkg_int is new osvvm.ScoreBoardGenericPkg
  generic map (
    ExpectedType        =>  integer,
    ActualType          =>  integer,
    match               =>  "=",
    expected_to_string  =>  to_string,
    actual_to_string    =>  to_string
  ) ;
```

Both in OSVVM Library

OSVVM's Scoreboard

```
use work.ScoreboardPkg_uart.all ;
shared variable SB : osvvm.Scoreboardpkg_uart.ScoreboardPType;
```

```
TxProc : process
  . . .
begin
  for I in 1 to 10000 loop
    case RV.DistInt((. . .)) is
      . . .
    end case ;
    SB.Push((TxD, Op));
    Send(TRec, TxD, Op);
  end loop ;
  . . .
```

```
RxProc : process
  variable RxD  : ByteType;
  variable RV   : RandomPType;
begin
  for I in 1 to 10000 loop

    Get(RRec, RxD, RxOp);
    SB.Check((RxD, RxOp));
  end loop ;
  . . .
```

Key Points:

- In TxProc, Always Push then Send. Ensures Pop has data
- In RxProc, Always Get then Pop. Also ensures Pop has data
- Simplified: RxProc does not need to know the expected data

Adding Functional Coverage

- What: Code that tracks that items in the test plan occur
  - Tracks requirements, features, and boundary conditions
- Why?
  - With Randomization, how do you know what the test did?
  - Test Done = Functional Coverage and Code Coverage @ 100 %
- Item Coverage (aka Point Coverage)
  - Track relationships within a single object
  - Bin transfer sizes into: 1, 2, 3, 4-127, 128-252, 253, 254, 255
- Cross Coverage
  - Track relationships between multiple objects
  - Has each set of registers been used with each input of an ALU?
- Why not just use code coverage?
  - Code coverage tracks code execution
  - Misses anything not in code (features, bins, uncorrelated items)

CoveragePkg

- OSVVM's CoveragePkg facilitates implementation of Functional Coverage
  - CoveragePkg simplifies coverage definition, collection, and reporting
  - Implemented as a data structure created within a shared variable.
  - The shared variable also facilitates supporting settings.

```
function GenBin ( . . . ) return CovBinType ;
type CovPType is protected
  procedure AddBins ( CovBin : CovBinType ) ;
  procedure AddCross( Bin1, Bin2, ... : CovBinType ) ;
  procedure ICover  ( val : integer ) ;
  procedure ICover  ( val : integer_vector ) ;
  impure function IsCovered return boolean ;
  procedure WriteBin ;
  procedure WriteCovHoles ;
  procedure ReadCovDb     ( FileName : string ) ;
  procedure WriteCovDb    ( FileName : string; ... ) ;
  . . .
end protected CovPType ;
```

Adding Functional Coverage

- For the UART, we track the following items

Condition	Status Register Values: Break Error	Stop Error	Parity Error	Done Flag	Integer Value(s)
Normal Transfer	0	0	0	1	1
Parity Error	0	0	1	1	3
Stop Error	0	1	0	1	5
Parity & Stop Error	0	1	1	1	7
Break Error	1	-	-	1	9-15

Adding Functional Coverage

```
architecture CR_1 of TestCtrl is
  shared variable RxCov : CovPType ;
  . . .
```

Coverage Object

```
RxProc : process
  . . .
begin
  RxCov.AddBins( GenBin(1) ) ;          -- Normal
  RxCov.AddBins( GenBin(3) ) ;          -- Parity Error
  RxCov.AddBins( GenBin(5) ) ;          -- Stop Error
  RxCov.AddBins( GenBin(7) ) ;          -- Parity + Stop
  RxCov.AddBins( GenBin(9, 15, 1) ) ; -- Break

  for I in 1 to 10000 loop
    Get(RRec, RxD, RxOp);
    SB.Check((RxD, RxOp));

    RxCov.ICover( RxOp ) ;
  end loop ;
  . . .
  RxCov.WriteBin ;
```

Incrementally define coverage model

Collect Coverage

Functional Coverage with OSVVM is as concise as language syntax.

Adding Intelligent Coverage Randomization

```
architecture CR_1 of TestCtrl is
  shared variable StimCov : CovPType ;
  . . .
```

Coverage Object

```
TxProc : process
   . . .
begin
  StimCov.InitSeed( StimCov'instance_name ) ;
  . . .

  StimCov.AddBins( 7000,  NORMAL ) ;
  StimCov.AddBins( 1100,  PARITY ) ;
  StimCov.AddBins( 1100,  STOP ) ;
  StimCov.AddBins(  600,  PARITY_STOP ) ;
  StimCov.AddBins(  200,  BREAK )  ;
  StimCov.SetAlertLogID("UART_RX_STIM_COV") ;

  for I in 1 to 10000 loop
    . . .
```

Add Coverage Goals to the Coverage Model

Adding Intelligent Coverage Randomization

- Coverage goals become randomization weights with RandCovPoint

```
TxProc : process
   variable iOperation : integer ;
   . . .
   for I in 0 to 10000 loop
    iOperation := StimCov.RandCovPoint ;
    case iOperation is

      when 1  =>    . . .  -- Nominal
      when 3  =>    . . .  -- Parity
      . . .
    end case ;
    if TxCount = 100 then   . . .
    UartSB.Push( (Data, Operation) ) ;
    StimCov.ICover( iOperation ) ;
    Send(UartTxRec, Data, Operation) ;
    wait for Idle * UART_BAUD_PERIOD_115200 ;
  end loop ;
  WaitForBarrier(TestDone) ;
```

Randomize

Target values change
Actions do not.

Collect Coverage

Adding Protocol and Parameter Checkers

- Protocol Checks (in a Memory Model)

```
SimultaneousAccessCheck: process
begin
  wait on iCE, iWE, iOE ;
  AlertIf(SramAlertID, (iCE and iWE and iOE) = '1',
      "nCE, nWE, and nOE are all active") ;
end process SimultaneousAccessCheck ;
```

- Parameter Checks (in OSVVM packages)

```
AlertIf (OSVVMID, Max < Min, "RandInt: Max < Min") ;
```

- Alerts can be enabled (default) or disabled

```
SetAlertEnable(ModelID, WARNING, FALSE) ; -- for a model
```

- Simulation can stop after receiving a predetermined number of Alerts

```
SetAlertStopCount(ERROR, 20) ; -- for the entire test
```


Test Initialization and Finalization

```
ControlProc : process
begin
  SetAlertLogName("Test_UartRx_1");
  TBID <= GetAlertLogID("TB");
  RxID <= GetAlertLogID("UartRx_1");
  SB.SetAlertLogId("UART_SB") ;

  SetLogEnable(PASSED, TRUE) ;
  SetLogEnable(RxID, INFO, TRUE) ;

  WaitForBarrier(TestDone, 5 ms) ;

  AlertIf(TBID, NOW >= 5 ms, "Test timed out") ;
  AlertIf(TBID, not SB.Empty, "Scoreboard not empty") ;
  AlertIf(TBID, GetAffirmCount < 1, "Checked < 1 items") ;

  ReportAlerts ;
  wait ;
```

Set Test Name

Set AlertLogIDs

Enable Logs

Run Test and Timeout after 5 ms

Report Errors

- Initialization in UartRx Model (setting ID string to instance name)

```
ModelID <= GetAlertLogID(PathTail(UartRx'INSTANCE_NAME));
```

Test Wide Reporting

- Simple Mode Error Reporting

```
ReportAlerts ;
std.env.stop ;
```

```
%% DONE  PASSED Test_UartRx_1 at 100000 ns
```

```
%% DONE  FAILED Test_UartRx_1 Total Error(s) = 10
Failures: 0  Errors: 1  Warnings: 1  at 100000 ns
```

Test Wide Reporting

- In Advanced testbenches, each model has it own AlertLog ID.

```
ModelID <= GetAlertLogID("UartRx_1") ;
```

- When Report Alerts fails, a detailed report is produced:

```
%% DONE  FAILED   Test_UartRx_1  Total Error(s) = 7  Failures: 0
Errors: 7  Warnings: 0  at 100100100 ns
%%    Default                 Failures: 0  Errors: 2  Warnings: 0
%%    OSVVM                   Failures: 0  Errors: 0  Warnings: 0
%%    TB                      Failures: 0  Errors: 0  Warnings: 0
%%    UART_SB                 Failures: 0  Errors: 0  Warnings: 0
%%    Cpu_1                   Failures: 0  Errors: 5  Warnings: 0
%%      Cpu_1 Data Error      Failures: 0  Errors: 4  Warnings: 0
%%      Cpu_1 Protocol Error  Failures: 0  Errors: 1  Warnings: 0
%%    UartRx_1                Failures: 0  Errors: 0  Warnings: 0
```

- With IDs, Alerts and Logs can be controlled on a model by model basis
  - Turn off info messages in the CPU, but keep them on for the UART
  - Important when one model is slower (UART) than others (CPU)

Synchronization: WaitForBarrier

- Purpose: Stop process until all processes have reached the barrier

```
signal TestDone : integer_barrier := 1;
```

```
ControlProc : process
begin
  SetAlertLogName("Test1") ;
  . . .
  WaitForBarrier(TestDone,5 ms);
  . .
  ReportAlerts ;
  std.env.stop ;
end process ControlProc ;
```

```
TestProc1 : process
 . . .
 WaitForBarrier(TestDone);
 wait ;
```

```
TestProc2 : process
 . . .
 WaitForBarrier(TestDone);
 wait ;
```

- With "TestDone", simulator scripts do not need to know run length
- Barriers can be used to synchronize 2 or more processes
- Simple use model: One signal for each synchronization point
  - Defined for types std_logic and integer_barrier

Including the OSVVM Library

- OSVVM Includes numerous packages.
  - To simplify this, OSVVM library provides context declarations

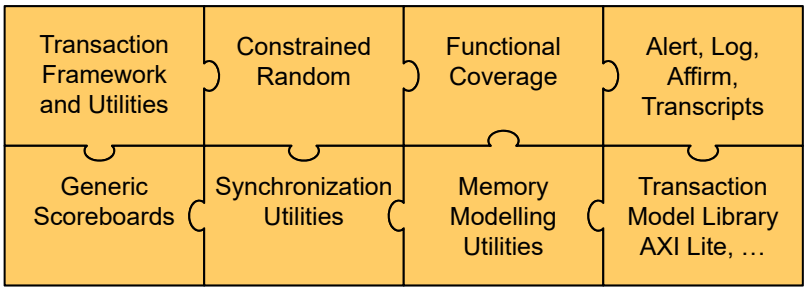
```
-- OSVVM Utility Library, Random, Coverage, …
library osvvm ;
  context osvvm.OsvvmContext ;          -- All OSVVM packages

-- AXI4 Model Library
library osvvm_axi4 ;
  context osvvm_axi4.Axi4LiteContext ;  -- AXI4 Lite Models
  context osvvm_axi4.AxiStreamContext ; -- AXI Stream Models

-- UART Model Library
Library osvvm_uart ;
  context osvvm_uart.UartContext ;      -- UART Models
```

Compiling the OSVVM Library

- OSVVM IP provides generalized tcl compilation scripts
  - Currently available for Mentor and Aldec.
  - Cadence and Synopsys updates are in process

```
cd VerificationIP/Scripts

# load procedures and settings to simplify scripting
do startup.tcl

# build the OSVVM library
build ../osvvm

# build AXI4 Lite models and run testbenches
build ../VerificationIP/AXI4/Axi4Lite/RunTests.pro

# build AXI4 Stream models and run testbenches
build ../VerificationIP/AXI4/AxiStream/RunTests.pro

# build UART models and run testbenches
build ../VerificationIP/UART/RunTests.pro
```


All you need is … OSVVM



- Benefits
  - Powerful and concise - rivals other verification languages
  - Simple and Readable by All
  - Supports mixed approaches (directed, algorithmic, file, CR, IT)
  - Re-use and/or upgrade your current VHDL testbench and models
  - Your RTL Designers can do verification tests and models
  - Simple enough for FPGAs, Powerful enough for ASICs

accellera SYSTEMS INITIATIVE
2019 DVCON EUROPE

Next Steps with OSVVM

- OSVVM on GitHub
  - https://github.com/OSVVM
  - OSVVM library: https://github.com/OSVVM/OSVVM
  - VerificationIP: https://github.com/OSVVM/VerificationIP
- OSVVM community at osvvm.org
- OSVVM has been accepted as an IEEE Open Source project

Questions

Finalize slide set with questions slide