VHDL Intelligent Coverage Using Open Source VHDL Verification Methodology (OSVVM)

by

Jim Lewis

VHDL Training Expert at SynthWorks

IEEE 1076 Working Group Chair

OSVVM Chief Architect

Jim@SynthWorks.com

SynthWorks

SynthWorks

VHDL Intelligent Coverage Using OSVVM

In particular,without express written permission of SynthWorks Design Inc,

You may not alter, transform, or build upon this work,

You may not use any material from this guide in a group presentation, tutorial, training, or classroom

You must include this page in any printed copy of this document.

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

VHDL Intelligent Coverage Using OSVVM

- What, Why, and How of OSVVM's Randomization and Functional Coverage

Topics

- What and Why OSVVM, Functional Coverage, Randomization?
- Writing Item (Point) Coverage
- Writing Cross Coverage
- Constrained Random is 5X or More Slower
- Intelligent Coverage
- OSVVM is More Capable
- Additional Randomization in OSVVM
- Weighted Intelligent Coverage
- Coverage Closure
- Additional Pieces of Verification
- Objections to VHDL
- OSVVM Summary

What is OS-VVM?

- Open Source VHDL Verification Methodology

- Packages + Methodology for:
  - Constrained Random (CR)
  - Functional Coverage (FC)
  - Intelligent Coverage - Test generation using FC holes

- Leading edge verification for your VHDL team
  - Works in any VHDL testbench
  - Mixes well with other approaches (directed, algorithmic, file, random)
  - Recommended to be use with transaction based testbenches
  - Readable by All (in particular RTL engineers)

- Low cost solution to leading edge verification
  - Works with regular VHDL simulators
  - Packages are FREE

What is Functional Coverage?

- Code that observes execution of your test plan
  - Tracks requirements, features, and boundary conditions
  - Model interface and design requirements
  - Required for randomized tests.
- Item Coverage (aka Point Coverage)
  - Track relationships within a single object
  - Bins of values, such as transfer sizes:
    - 1, 2, 3, 4-127, 128-252, 253, 254, 255
- Cross Coverage
  - Track relationships between multiple objects
  - Has the each pair of registers been used with the ALU?
- Test Done =
  - 100 % Functional Coverage + 100 % Code Coverage

Why Functional Coverage?

- "I have written a directed test for each item in the test plan, I am done right?"
  - For a small design maybe
- As complexity grows and the design evolves, are you sure?
  - When the FIFO size quadruples, does the test still fill it?
  - Have you covered all possible use modes and orderings?
  - Did you add all required features?
- To avoid missing items, use functional coverage for all tests.
  - Rather than assume, functional coverage observes that the test plan points actually get exercised.

Randomization Methodologies

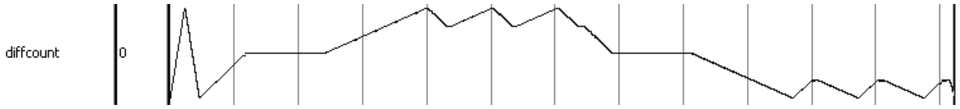
- Constrained Random (CR)
  - Generate stimulus using randomization constraints
  - Constraints can be equations (SystemVerilog) or code (VHDL)
  - SystemVerilog uses a solver to balance the randomization
  - Requires functional coverage to determine what was done

- Intelligent Coverage
  - Generate stimulus by randomizing across holes in the FC model
  - Requires functional coverage
  - No top-level randomization constraints

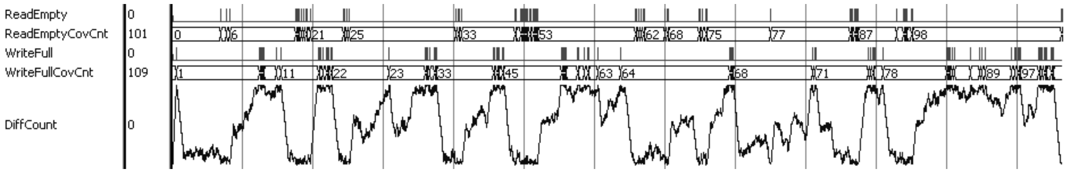
- Intelligent Coverage is less work (2X?) than Constrained Random

Why Randomize?

- Directed test of a FIFO (tracking words in FIFO):



- Constrained Random test of a FIFO:



- With randomization,
  - We can generate more realistic stimulus
  - Ideal for different modes, instructions, … network packets.
  - Sequences with different orders

Item Coverage

- Relationships within a single object = Bins of values.

Transfer Sizes	Count
1	
2	
3	
4 to 127	
128 to 252	
253	
254	
255	

- Boundary conditions occur at smaller and larger transfer sizes
- Methods:
  - Manual
  - Using CoveragePkg

Item Coverage: Manual

```
signal Bin : integer_vector(1 to 8) ;
. . .
process
begin
  wait until rising_edge(Clk) and DValid = '1' ;
  case to_integer(ActualData) is
    when   1 =>          Bin(1) <= Bin(1) + 1 ;
    when   2 =>          Bin(2) <= Bin(2) + 1 ;
    when   3 =>          Bin(3) <= Bin(3) + 1 ;
    when   4 to 127 =>   Bin(4) <= Bin(4) + 1 ;
    when 128 to 252 =>   Bin(5) <= Bin(5) + 1 ;
    when 253 =>          Bin(6) <= Bin(6) + 1 ;
    when 254 =>          Bin(7) <= Bin(7) + 1 ;
    when 255 =>          Bin(8) <= Bin(8) + 1 ;
    when others =>       null ;
  end case ;
end process ;
```

Declare
8 Bins

Sample at
Clock

Define Bins
+
Collect
Coverage

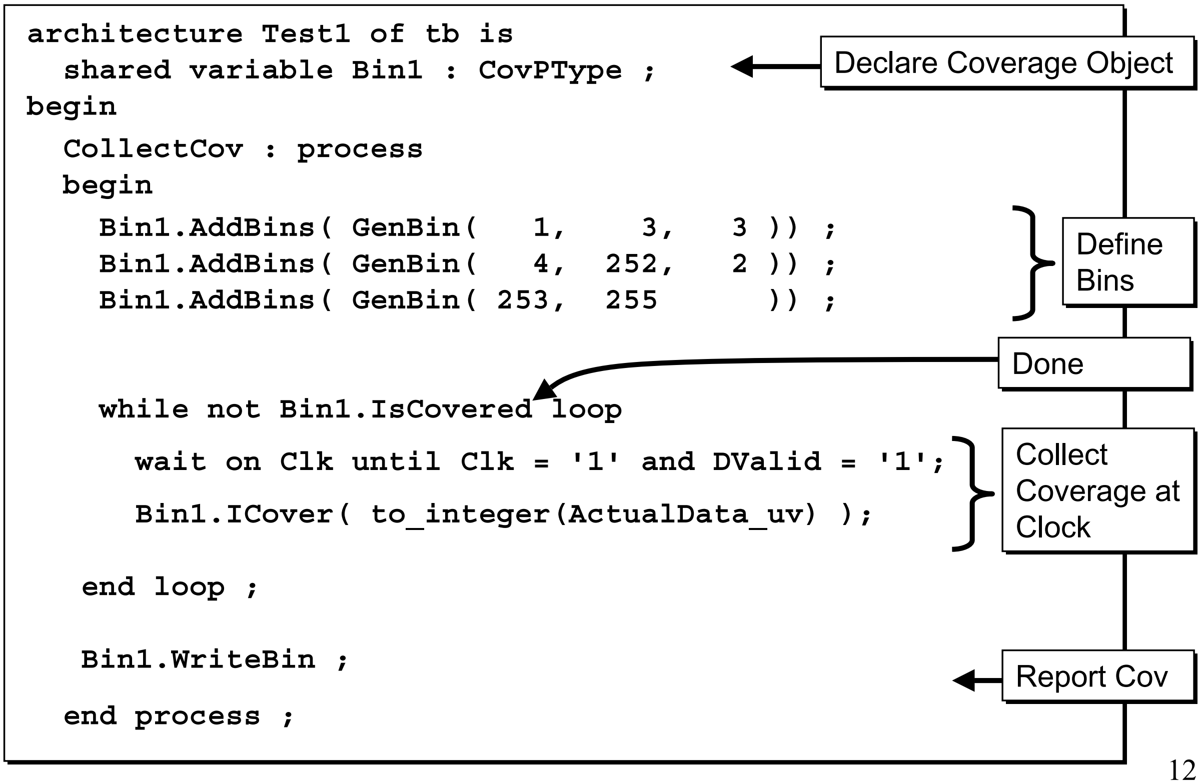
Not Rocket Science, but:
Too much work, Too specific to a problem,
No reuse, No built-in reporting

CoveragePkg

- CoveragePkg simplifies coverage definition, collection, and reporting
  - Protected Type: CovPType
  - Implements a data structure and configuration parameters (via variables)
  - Methods to implement all coverage features

```
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 ;
```

Item Coverage w/ CoveragePkg



```
architecture Test1 of tb is
  shared variable Bin1 : CovPType ;
begin
  CollectCov : process
  begin
    Bin1.AddBins( GenBin(   1,    3,   3 )) ;
    Bin1.AddBins( GenBin(   4,  252,   2 )) ;
    Bin1.AddBins( GenBin( 253,  255      )) ;

    while not Bin1.IsCovered loop
      wait on Clk until Clk = '1' and DValid = '1';
      Bin1.ICover( to_integer(ActualData_uv) );
    end loop ;

    Bin1.WriteBin ;
  end process ;
```

Declare Coverage Object

Define Bins

Done

Collect Coverage at Clock

Report Cov

Define Bins: AddBins + GenBin

- Method AddBins: Add item coverage bin(s) to internal data structure.
- Function GenBin: Create array of bin inputs to AddBins
- Create 3 bins with ranges: 1 to 1, 2 to 2, and 3 to 3 .

```
--                     min, max, #bins
Bin1.AddBins( GenBin(  1,   3,   3   ));
```

- Additional calls to AddBins creates additional bins

```
--                     min, max, #bins
Bin1.AddBins( GenBin(  4,   252,   2 ) );
```

  - Create 2 additional bins with ranges: 4 to 127, 128 to 252.

- GenBin without NumBins creates one bin per value

```
--                      min, max
Bin1.AddBins( GenBin( 253, 255 ) );
-- Bin1.AddBins( GenBin( 253, 255, 3 ) ); -- equivalent
```

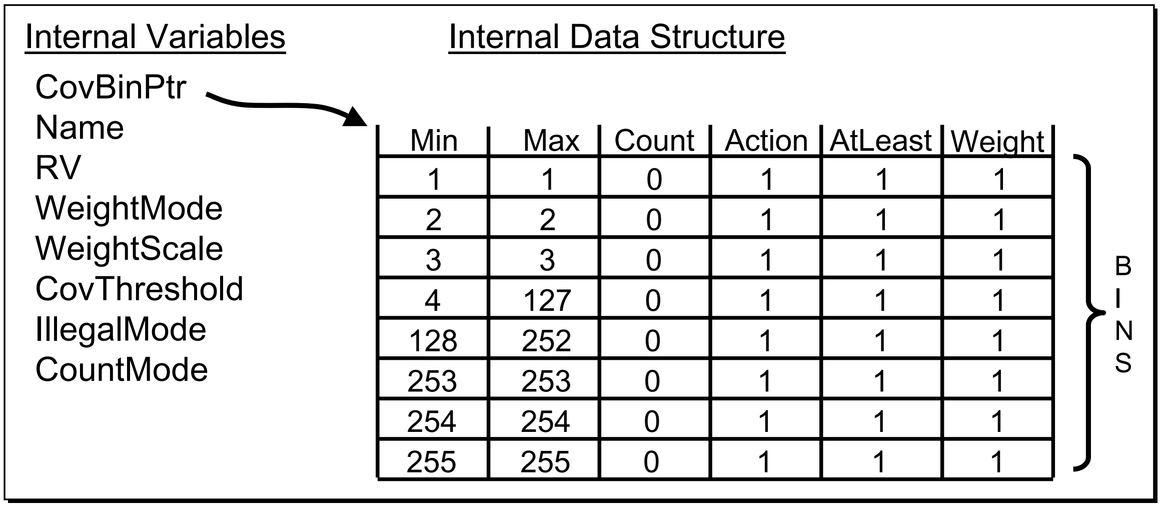
  - 3 additional bins with ranges: 253 to 253, 254 to 254, and 255 to 255.

Coverage Model Data Structure

- Use a shared variable

```
shared variable Bin1 : CovPType ;
```

- Data structure and related settings are stored in the shared variable



Internal Variables

CovBinPtr
Name
RV
WeightMode
WeightScale
CovThreshold
IllegalMode
CountMode

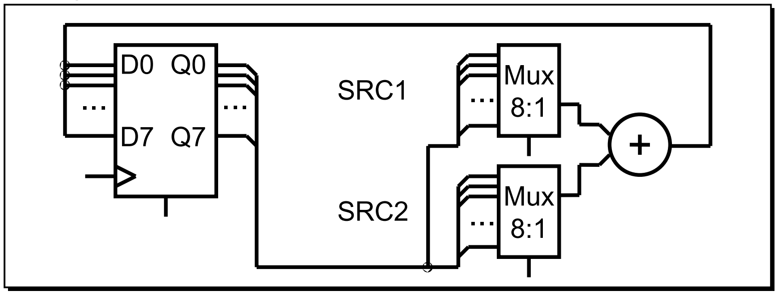
Internal Data Structure

Min	Max	Count	Action	AtLeast	Weight
1	1	0	1	1	1
2	2	0	1	1	1
3	3	0	1	1	1
4	127	0	1	1	1
128	252	0	1	1	1
253	253	0	1	1	1
254	254	0	1	1	1
255	255	0	1	1	1

- Each row in the data structure is a separate bin

Cross Coverage

- Testing an ALU with Multiple Inputs:



- Need to test every register in SRC1 with every register in SRC2

		SRC2							
		R0	R1	R2	R3	R4	R5	R6	R7
SRC1	R0								
	R1								
	R2								
	R3								
	R4								
	R5								
	R6								
	R7								

- Result: Matrix of conditions that must be covered

Cross Coverage

```
architecture Test3 of tb is
  shared variable ACov : CovPType ;
begin
 CollectCov : process
   variable RV : RandomPType ; -- randomization object
   variable Src1, Src2 : integer ;
 begin
   ACov.AddCross( GenBin(0,7), GenBin(0,7) );

   while not ACov.IsCovered loop

    Src1 := RV.RandInt(0, 7) ;
    Src2 := RV.RandInt(0, 7) ;

    DoAluOp(TRec, Src1, Src2) ;
    ACov.ICover( ( Src1, Src2 ) ) ;

  end loop ;
  ACov.WriteBin ;
  EndStatus(. . . ) ;
end process ;
```

Coverage Object

Create Cross Coverage Bins

Covered = Done

Uniform Randomization

Do Transaction

Collect Coverage at Transaction

Functional Coverage with OSVVM is as concise as language syntax.

Cross Coverage: Define Bins

- Method AddCross: Add cross coverage bin(s) to internal data structure.

```
ACov.AddCross( GenBin(0,7), GenBin(0,7) );
```

  - One parameter per cross item. Up to 20 parameters supported.
  - GenBin used to construct parameter values.

- Data structure now has one range (min, max) pair per cross item:

Internal Data Structure

	Src1		Src2		Count	Action	AtLeast	Weight
	Min	Max	Min	Max				
	0	0	0	0	0	1	1	1
	0	0	1	1	0	1	1	1
B	0	0	2	2	0	1	1	1
I	0	0	3	3	0	1	1	1
N	0	0	4	4	0	1	1	1
S	...	...	...	...	...	...	...	...
	7	7	6	6	0	1	1	1
	7	7	7	7	0	1	1	1

Constrained Random is 5X or More Slower

- With a good solver, constrained random (CR) does uniform randomization.
  - Uniform distributions repeat before generating all cases
  - In general, to generate N cases, it takes O(N*log N) randomizations
- The uniform randomization in ALU test requires 315 test iterations.
  - 315 is approximately 5X too many iterations (64 test cases)
  - The "log N" factor significantly slows down constrained random tests.

		SRC2							
		R0	R1	R2	R3	R4	R5	R6	R7
	R0	6	6	9	1	4	6	6	5
S	R1	3	4	3	6	9	5	5	4
R	R2	4	1	5	3	2	3	4	6
C	R3	5	5	6	3	3	4	4	6
1	R4	4	5	5	10	9	10	7	7
	R5	4	6	3	6	3	5	3	8
	R6	3	6	3	4	7	1	4	6
	R7	7	3	4	6	6	5	4	5

- "From Volume to Velocity" shows CR tests that are 10X to 100X too slow

Intelligent Coverage

- Goal: Generate N Unique Test Cases in N Randomizations
  - Same goal of Intelligent Testbenches

		SRC2							
		R0	R1	R2	R3	R4	R5	R6	R7
SRC1	R0	1	1	1	1	1	1	1	1
	R1	1	1	1	1	1	1	1	1
	R2	1	1	1	1	1	1	1	1
	R3	1	1	1	1	1	1	1	1
	R4	1	1	1	1	1	1	1	1
	R5	1	1	1	1	1	1	1	1
	R6	1	1	1	1	1	1	1	1
	R7	1	1	1	1	1	1	1	1

- Randomly select holes in Functional Coverage Model
  - "Coverage driven randomization" - but term is misused by others

Intelligent Coverage

Same test using Intelligent Coverage

```
architecture Test3 of tb is
  shared variable ACov : CovPType ; -- Cov Object
begin
 CollectCov : process
   variable Src1, Src2 : integer ;
 begin
  ACov.AddCross( GenBin(0,7), GenBin(0,7) );

  while not ACov.IsCovered loop

    (Src1, Src2) := ACov.RandCovPoint ;

    DoAluOp(TRec, Src1, Src2) ;

    ACov.ICover( (Src1, Src2) ) ;

  end loop ;
  ACov.WriteBin ; -- Report Coverage
  EndStatus(. . . ) ;
end process ;
```

Intelligent Coverage Randomization

Runs 64 iterations @ 5X faster

Refinement of Intelligent Coverage

- Refinement can be as simple or complex as needed
- Use either directed, algorithmic, file-based or randomization methods.

```
while not ACov.IsCovered loop

  (Reg1, Reg2) := ACov.RandCovPoint ;

  if Reg1 /= Reg2 then
    DoAluOp(TRec, Reg1, Reg2) ;
    ACov.ICover( (Reg1, Reg2) ) ;

  else
    -- Do previous and following diagional
    DoAluOp(TRec, (Reg1-1) mod 8, (Reg2-1) mod 8) ;
    DoAluOp(TRec,  Reg1, Reg2 ) ;
    DoAluOp(TRec, (Reg1+1) mod 8, (Reg2+1) mod 8) ;

    -- Can either record all or select items
    ACov.ICover( (Reg1, Reg2) ) ;
  end if ;

end loop ;
```

OSVVM is More Capable

- Functional Coverage is a data structure
  - Modeled using any sequential construct (loop, if, case, …)
  - Incremental additions supported
  - Use generics to make coverage conditional on test parameters

```
TestProc : process
begin
  for i in 0 to 7 loop
    for j in 0 to 7 loop
      if i /= j then
        -- non-diagonal
        ACov.AddCross(2, GenBin(i),  GenBin(j));
      else
        -- diagonal
        ACov.AddCross(4, GenBin(i),  GenBin(j));
      end if ;
    ...
```

Additional Randomization in OSVVM

- Implemented in RandomPkg
- 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: Value + Weight

```
. . .                -- ((val1, wt1), (val2, wt2), ...)
Data5 := RV.DistValInt( ((1,7), (3,2), (5, 1)) );
```

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

```
Data6 := RV.DistInt ( (7, 2, 1) ) ;
```

Additional Randomization in OSVVM

- Code patterns create constraints for CR tests,
  - Example: Weighted selection of test sequences (CR)

```
StimGen : while TestActive loop

  case  RV.DistInt( (7, 2, 1) ) is   -- Select sequence

    when 0 =>  -- Normal Handling   -- 70%
       . . .

    when 1 =>  -- Error Case 1      -- 20%
       . . .

    when 2 =>  -- Error Case 2      -- 10%
       . . .
```

- In OSVVM, Intelligent Coverage is the primary randomization,
  - Code patterns are used primarily for refinement.
  - Usage of CR alone is O(logN) slower

Weighted Intelligent Coverage

- Each coverage bin can have a different coverage goal
  - Goal = Number of times of value must occur to be covered

- Weighted selection of test sequences (Intelligent Coverage):

```
Bin1.AddBins( 70, GenBin(0) ) ; -- Normal Handling, 70%
Bin1.AddBins( 20, GenBin(1) ) ; -- Error Case 1,    20%
Bin1.AddBins( 10, GenBin(2) ) ; -- Error Case 2,    10%

StimGen : while not Bin1.IsCovered loop
  iSequence := Bin1.RandCovPoint ;
  case iSequence is
    when 0 =>  -- Normal Handling   -- 70%
       . . .
    when 1 =>  -- Error Case 1      -- 20%
       . . .
    when 2 =>  -- Error Case 2      -- 10%
       . . .
```

Set Coverage Goals

Select sequence

Generates the exact distribution

Coverage Closure

- Closure = Cover all legal bins in the coverage model

- Intelligent coverage
  - Focus on FC. Only selects bins that are not covered
  - Just need a mapping from selected coverage to an input sequence
  - In complex cases may require more than one transaction
  - Tests partitioned based on what coverage we want in this test.

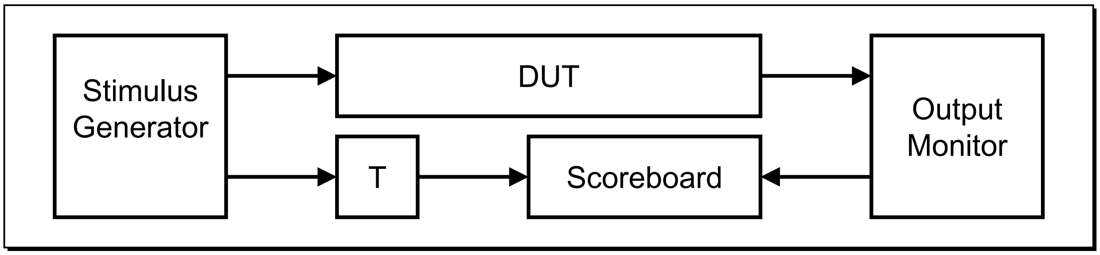
- Constrained Random
  - Requires CR to accurately drive the inputs to the FC
  - Closure is more challenging
  - After simulation, analyze FC
  - Prune out tests that are not increasing FC
  - Tests partitioned based on modified constraint sets and seeds
  - Must merge FC database for all tests

Additional Pieces of Verification

- TLM = Abstract Initiation + Transaction Models (entity/architecture)

```
CpuWrite( CpuRec, ADDR0, X"A5A5" );
CpuRead ( CpuRec, ADDR0, DataO );
```

- Scoreboards



- Memory Modeling
  - Large memories need space saving algorithm + Easy access
- Packages for above +
  - Synchronization - synchronize concurrent processes
  - Reporting

Objections to VHDL

- No OO
  - Functional Coverage requires data structures not OO
  - TLM / BFMs are easier to implement using an entity + architecture
- No Factory Class
  - Factory classes allow swapping of implementations in OO programming
  - Architectures give the same capability for concurrent programming
- No Solver
  - Intelligent coverage is balanced and O(logN) faster than the best solver
- No Fork & Join
  - Fork & Join are for sequential programming - writing threads.
  - VHDL is already a concurrent language
    - Use entity + architecture for bundling
    - Use separate processes for independent handling of sequences
    - Use handshaking (like hardware) to coordinate separate activities
    - Just like RTL

SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
A design and verification engineer's introduction to VHDL syntax, RTL coding, and testbenches. Students get VHDL hardware experience with our FPGA based lab board.

VHDL Testbenches and Verification 5 days - OSVVM bootcamp
http://www.synthworks.com/vhdl_testbench_verification.htm
Learn the latest VHDL verification techniques including transaction-based testing, bus functional modeling, self-checking, data structures (linked-lists, scoreboards, memories), directed, algorithmic, constrained random and coverage driven random testing, and functional coverage.

VHDL Coding for Synthesis 4 Days
http://www.synthworks.com/vhdl_rtl_synthesis.htm
Learn VHDL RTL (FPGA and ASIC) coding styles, methodologies, design techniques, problem solving techniques, and advanced language constructs to produce better, faster, and smaller logic.

SynthWorks offers on-site, public venue, and on-line classes. See:
http://www.synthworks.com/public_vhdl_courses.htm

OSVVM Summary

- Intelligent Coverage = Simple, Powerful, Concise Methodology
  - Define Functional Coverage
  - Randomize across coverage holes
  - Refine with directed, algorithmic, file-based or CR methods
- Faster
  - Test construction: Focus on FC, hence, less work (approx 1/2)
  - Simulations: No redundant stimulus (LogN faster) and No solver
- OSVVM
  - Goes beyond other verification languages (SV and 'e')
  - Is language accessible. Add code to refine.
  - Works in any VHDL environment – including TLM
  - Readable by All (Verification and RTL engineers)
- SystemVerilog?
  - Less powerful, alienates RTL engineers, requires a specialist

Going Further / References

- Jim's OSVVM Blog: www.synthworks.com/blog/osvvm
- OSVVM Website: www.osvvm.org
- Coverage Package Users Guide and Random Package Users Guide
- "From Volume to Velocity" by Walden Rhines of Mentor Graphics, Keynote speech for DVCon 2011.
  - See http://www.mentor.com/company/industry_keynotes/
- Getting the packages:
  - Maybe already installed in your simulator's osvvm library
  - http://www.osvvm.org/downloads
  - http://www.synthworks.com/downloads

This page is intentionally left blank