OSVVM: Advanced Verification for VHDL

Jim Lewis

VHDL Training Expert SynthWorks - IEEE VHDL WG Chair - OSVVM Architect

accellera SYSTEMS INITIATIVE™

2014 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

OSVVM: Advanced Verification for VHDL

- OSVVM consists of packages + methodology for:
  - Constrained Random (CR)
  - Functional Coverage (FC)
  - Intelligent Coverage - Random test generation using FC holes
- Key Benefits
  - Works in any VHDL testbench
  - Mixes well with other approaches (directed, algorithmic, file)
  - Recommended to be use with transaction based testbenches
  - Readable by All (in particular RTL engineers)
- Low cost solution to leading edge verification
  - Packages are FREE
  - Works with VHDL simulators that support minimal VHDL2008

Basic 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

Basic Randomization

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

```
variable RV : RandomPType ;
. . .
StimGen : while TestActive loop
  case RV.DistInt( (7, 2, 1) ) is   -- Select sequence
    when 0 =>  -- Normal Handling   -- Selected 70%
       . . .
    when 1 =>  -- Error Case 1      -- Selected 20%
       . . .
    when 2 =>  -- Error Case 2      -- Selected 10%
       . . .
```

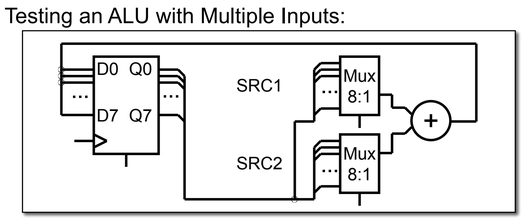
Use of Transactions simplifies mixing with other test approaches

What is Functional Coverage?

- Code that observes execution of your test plan
  - Tracks requirements, features, and boundary conditions
- Item Coverage (aka Point Coverage)
  - Track relationships within a single object
  - Bins of 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?
- Why not just Code Coverage?
  - Does not correlate independent items: Bins & Register Pairs
- Key Benefits:
  - Required for randomization
  - Recommended for any complex design
  - Test Done = 100 % Functional Coverage + 100 % Code Coverage

Writing Functional Coverage

- Testing an ALU with Multiple Inputs:



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

SRC2

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

Writing Functional Coverage

```
architecture Test3 of tb is
  shared variable ACov : CovPType ;          -- Coverage Object
begin
 CollectCov : process
   variable RV : RandomPType ; -- randomization object
   variable Src1, Src2 : integer ;
 begin
   ACov.AddCross( GenBin(0,7), GenBin(0,7) );   -- Create Cross Coverage Bins
   while not ACov.IsCovered loop                -- Covered = Done
     Src1 := RV.RandInt(0, 7) ;                 -- Uniform Randomization
     Src2 := RV.RandInt(0, 7) ;
     DoAluOp(TRec, Src1, Src2) ;                -- Do Transaction
     ACov.ICover( ( Src1, Src2 ) ) ;            -- Collect Coverage at Transaction
   end loop ;
   ACov.WriteBin ;
EndStatus(. . . ) ;
end process ;
```

Functional Coverage with OSVVM is as concise as language syntax.

Constrained Random is Too Slow!

- Constrained random (CR) tests do uniform randomization (VHDL & SV).
  - 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

SRC1	R0	R1	R2	R3	R4	R5	R6	R7
R0	6	6	9	1	4	6	6	5
R1	3	4	3	6	9	5	5	4
R2	4	1	5	3	2	3	4	6
R3	5	5	6	3	3	4	4	6
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

Other studies show that CR test can be much slower

Intelligent Coverage

- Goal: Generate N Unique Test Cases in N Randomizations
  - Benefit: generates each test case only one time = Intelligent Testbench

SRC2

SRC1	R0	R1	R2	R3	R4	R5	R6	R7
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

- Random walk across functional coverage holes
  - Randomization only selects holes in the Functional Coverage

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 ;
  EndStatus(. . . ) ;
end process ;
```

Runs 64 iterations @ 5X faster

Intelligent Coverage Randomization

Intelligent Coverage Methodology
- Write FC
- Randomize using FC
- Refine

Refinement of Intelligent Coverage

- Refinement is done with code directly in the testbench
- 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 ;
```

Weighted Intelligent Coverage

- One of a condition, transaction, or sequence may not be enough
  - A coverage goal specifies number of occurrences for a bin to be covered
  - Each coverage bin can have a different coverage goal
- 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 this exact distribution

OSVVM is More Capable

- Functional Coverage is a data structure
  - Incremental additions supported
  - Use any sequential construct (loop, if, case, …)
  - 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 ;
...
```

Coverage Closure

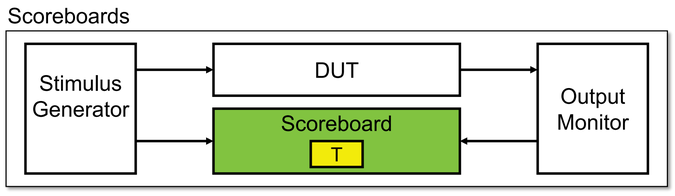
- Closure = Cover all legal bins in the coverage model
- Intelligent Coverage
  - Write FC.
  - Only selects bins that are not covered in the FC
  - Closure depends on running test long enough.
  - Tests partitioned based on what coverage we want in this test.
- Constrained Random
  - Write CR. Write FC.
  - All CR controls are open-loop – No FC information is available during sim
  - After simulation, merge FC from all tests
  - Analyze tests and prune out ones that are not increasing FC
  - Tests partitioned based on modified controls, constraint sets, and seeds
- Intelligent Coverage is less work than Constrained Random

Additional Pieces of VHDL 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
- Other packages support above + Synchronization and Reporting

OSVVM Summary

- Simple, Powerful, Concise Methodology = Intelligent Coverage (IT)
  - Define Functional Coverage
  - Randomize across coverage holes
  - Refine with directed, algorithmic, file, CR, or IT based methods
- Better than SystemVerilog / 'e' Capabilities
  - Functional Coverage – Incremental, Conditional
  - Intelligent Testbenches built-in
  - FC, CR, and IT that can be refined with code
  - Extensible, just add to the packages
  - Works in any VHDL environment – including TLM
  - Supports mixed approaches (directed, algorithmic, file, CR, IT)
  - Simple and Readable by All (Verification and RTL Engineers)
  - Faster Test Construction - Focus on FC
  - Faster Simulations - No redundant stimulus (log N) and no solver
    - Faster Coverage Closure