VHDL Testbench Techniques that Leapfrog SystemVerilog

by
Jim Lewis
VHDL Training Expert at SynthWorks
IEEE 1076 Working Group Chair
OSVVM Chief Architect
Jim@SynthWorks.com

SynthWorks

VHDL Testbench Techniques

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 Testbench Techniques

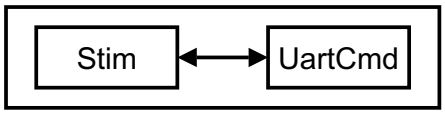
- Goals: Thorough, Timely, and Readable Testing

- Agenda
  - Testbench Architecture
  - Transactions
  - Writing Tests
  - Randomization
  - Functional Coverage
  - Constrained Random is Too Slow!
  - Intelligent Coverage is More Capable
  - Coverage Closure is Faster with Intelligent Coverage
  - Self-Checking & Scoreboards
  - Scoreboards
  - Dispelling FUD

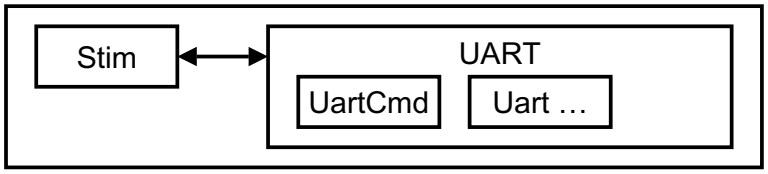
Testbench Architecture

- Historically a separate testbench is written for different levels of testing

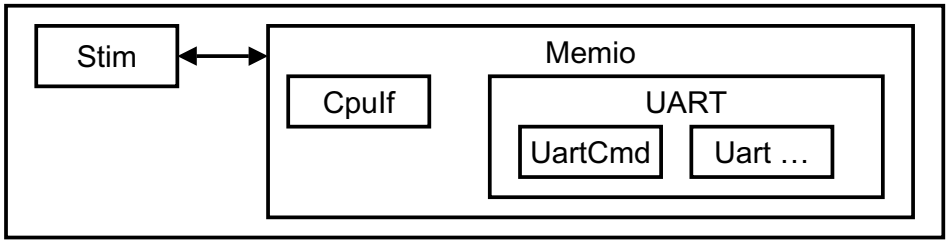
RTL



Core



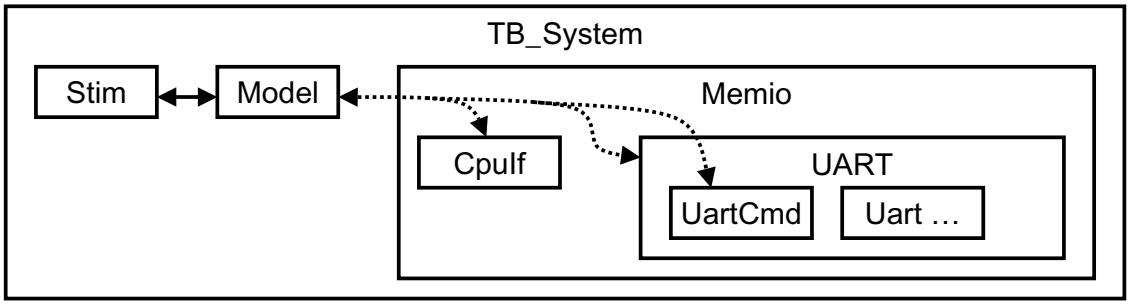
System



- Each testbench slightly larger than the previous.

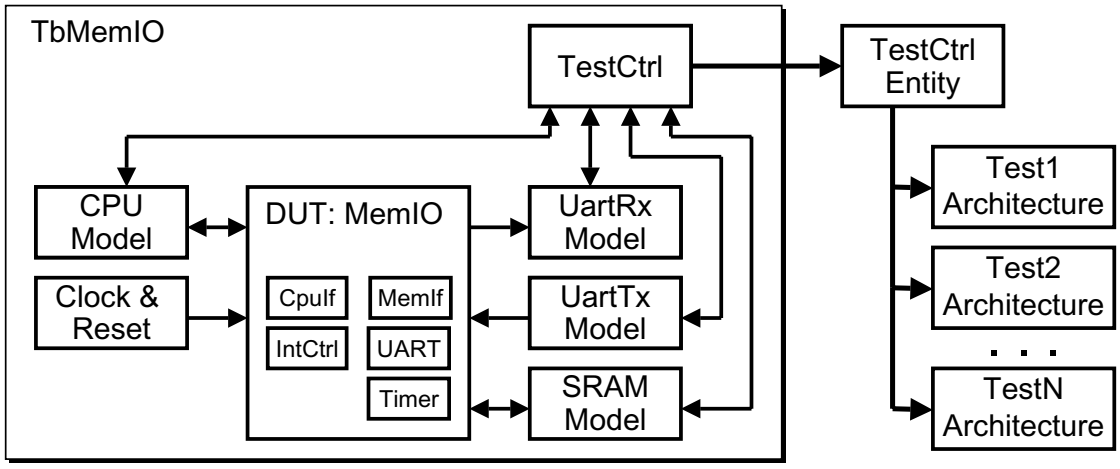
Testbench Architecture

- Many interfaces transport the same information



- Change from RTL to Core to System by:
  - Separating stimulus from interface signaling (signal wiggling)
  - Changing the Model (changes signal wiggling)
  - Changing the connections (via VHDL-2008 external names)
  - Leave items not used in a test unbound or use a dummy architecture
- Reduction in number of testbenches can minimize wasted effort.
  - But we need to plan from the System level

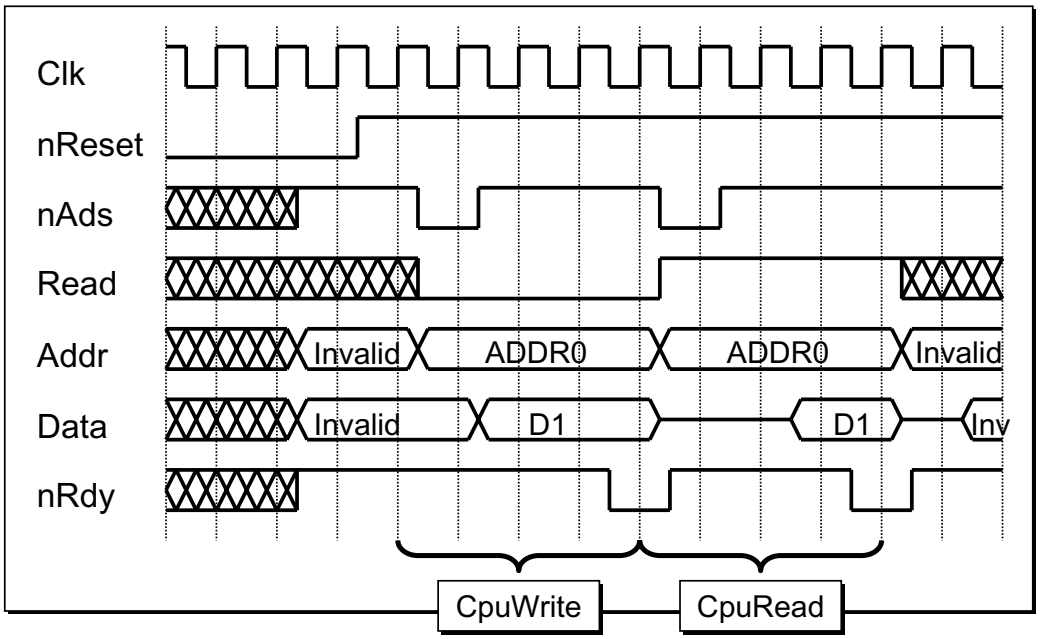
Testbench Architecture



- System / Chip Level Testbench
  - TLM (transaction level model) implements signaling to DUT
  - TestCtrl to sequence and/or synchronize models
  - Each test is a separate architecture of TestCtrl
  - Plan to pre-use system level models for Core and RTL tests

Transactions

- Interface Operations, ie: CpuWrite, CpuRead, …
- Essential for reuse and pre-use
- 2 Aspects: Initiation (stimulus) and Implementation (model)



Transaction Initiation

- Writing tests by wiggling signals is slow, error prone, and inhibits reuse

```
-- CPU Write
nAds <= '0' after tpd, '1' after tperiod + tpd ;
Addr <= ADDR0 after tpd ;
Data <= X"A5A5" after tperiod + tpd ;
Wr_nRd <= '0' after tpd ;
wait until nRdy = '0' and rising_edge(Clk) ;
```

- Transaction initiation using procedures:

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

  - Simplifies writing tests.
  - Increases readability and maintainability

Transaction Implementation

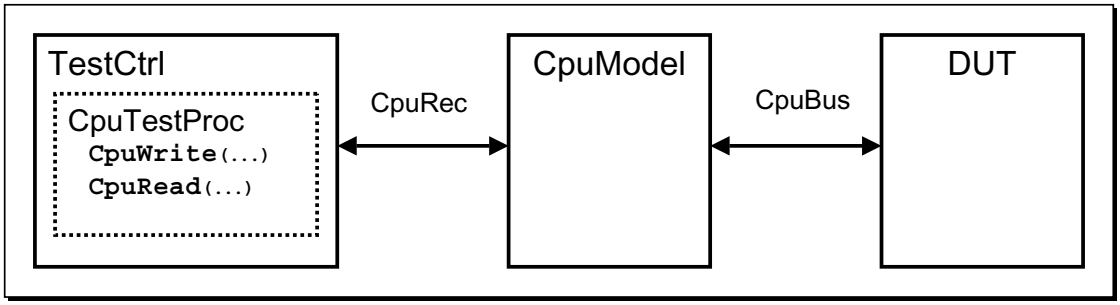
- Implement transactions using either Subprograms or Entities
- Subprograms do both
  - Transaction initiation by calling the subprogram
  - Transaction implementation (signal wiggling) internal to subprogram

```
procedure CpuWrite (
  signal   CpuRec : InOut   CpuRecType ;
  constant AddrI  : In   std_logic_vector;
  constant DataI  : In   std_logic_vector
) is
begin
  CpuRec.nAds   <= '0' after tpd, '1' after tperiod + tpd ;
  CpuRec.Addr   <= AddrI after tpd ;
  CpuRec.Data   <= DataI after tperiod + tpd ;
  CpuRec.Wr_nRd <= '1' after tpd ;
  wait until CpuRec.nRdy = '0' and rising_edge(CpuRec.Clk);
end procedure ;
```

- Use a package to separate the stimulus from implementation (model)

Transaction Implementation

- Entities +
  - Transaction initiation by calling the subprogram
  - Transaction implementation (signal wiggling) done by entity



- Advantages of Entity Approach
  - Model is concurrent
  - Implement it with either behavioral or RTL code
  - Keeps functionality, protocol checkers, timing checkers, result loggers in the same model
- Ok for some interfaces to use Entity and others to use Subprograms

Transaction Interface

- Records simplify interface between subprogram and DUT or model
  - Simplify updates
- Can use 1 record with resolved types

```
type Model1RecType is record
    CmdRdy              : std_logic ;
    CmdAck              : std_logic ;

    UnsignedToModel     : unsigned(15 downto 0) ;
    UnsignedFromModel   : unsigned(15 downto 0) ;

    IntegerToModel      : resolved integer ;
    IntegerFromModel    : resolved integer ;
    TimeToModel         : resolved time ;
    RealToModel         : resolved real ;
end record ;
```

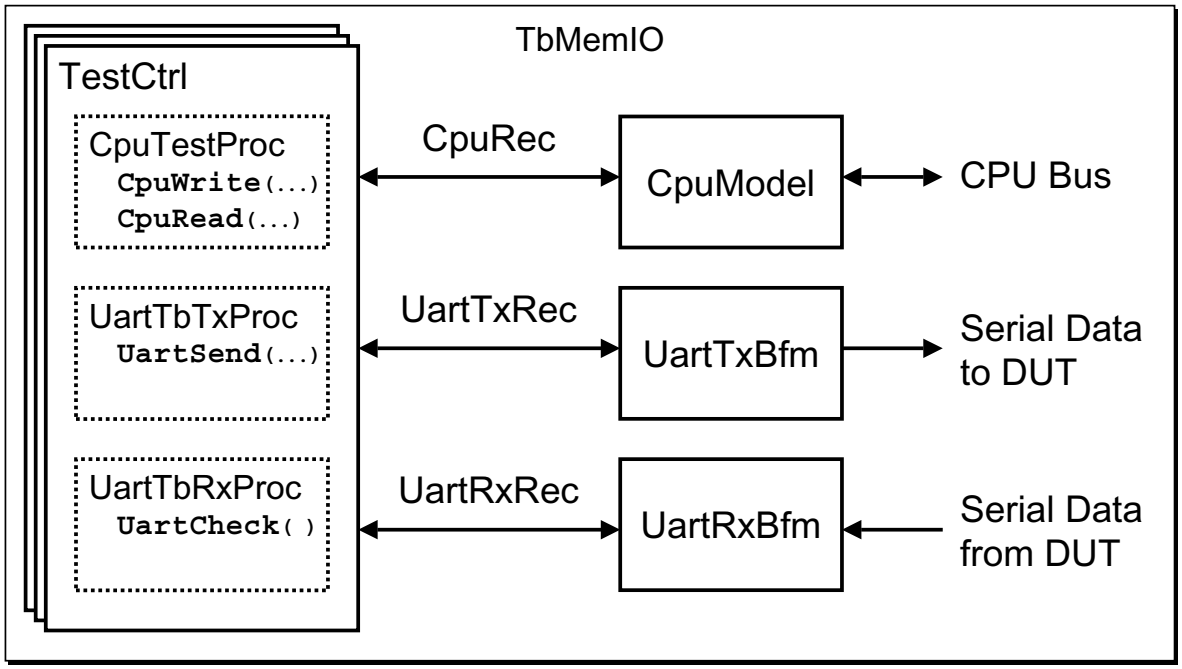
- Can use 2 records
  - 1 to the DUT from the model and 1 to the model from the DUT
  - Increases number of signals on subprogram interface

Writing Tests

- Tests can be either:
  - Directed - particularly good for testing registers
  - Algorithmic - particularly good for math
  - File Based - large or existing data sets (Images, Matlab, …)
  - Constrained Random
  - Intelligent Coverage
- Not one approach works for everything.
- Use of a transaction based framework simplifies mixing of test types.

Writing Tests

- TestCtrl specifies transactions for each model
  - Controls, coordinates, and synchronizes test activities



Writing Tests

```
architecture  Test1  of TestCtrl is
begin
  CpuTestProc : process
  begin
    wait until nReset = '1' ;
    CpuWrite(. . .) ;
    CpuRead(. . .) ;
    . . .
  end process ;

  UartTbTxProc : process
  begin
    WayPointBlock(SyncPoint) ;
    UartSend(. . .) ;
    . . .
  end process ;

  UartTbRxProc : process
  begin
    UartCheck(. . .) ;
    . . .
  end process ;
end Test1 ;
```

Tests are a separate architecture of TestCtrl (TestCtrl_UartRx1.vhd, TestCtrl_UartRx2.vhd, …)

One or more processes for each independent source of stimulus

Interface Stimulus is generated with one or more procedure calls

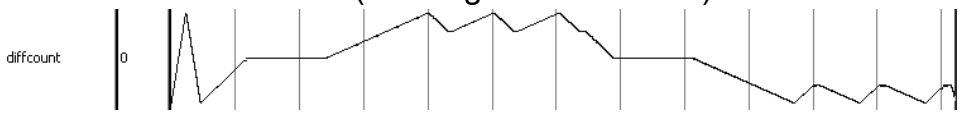
Synchronize processes using handshaking

OSVVM & Writing Tests

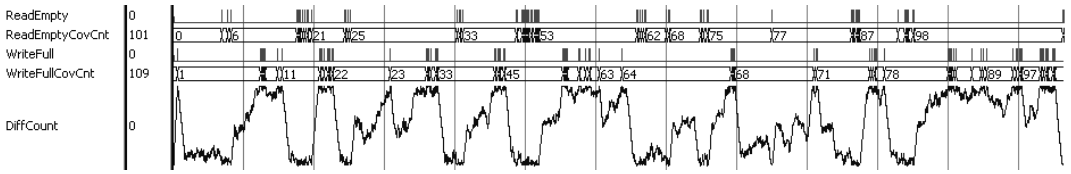
- Open Source VHDL Verification Methodology
- Packages + Methodology for:
  - Constrained Random (CR)
  - Functional Coverage (FC)
  - Intelligent Coverage - Random test generation using FC holes
- Key Benefits
  - Works in any VHDL testbench
  - Mixes well with other approaches (directed, algorithmic, file, random)
  - 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 regular VHDL simulators

Why Randomize?

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



- Constrained Random test of a FIFO:



- Randomization
  - Is ideal for large variety of similar items
    - Modes, processor instructions, … network packets.
  - Generates realistic stimulus in a timely fashion (to write)
  - Is more thorough since stimulus is not ordered (not looping)

Randomization in OSVVM

- 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.

Randomization in OSVVM

- 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%
       . . .
```

- Still uses transactions, so mixes readily with other test approaches

What is Functional Coverage?

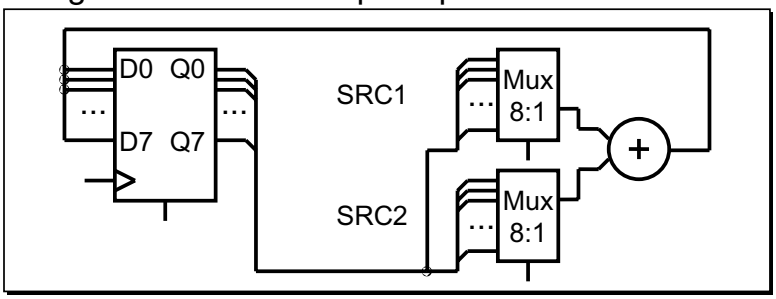
- Functional Coverage (FC)
  - Code that correlates and/or bins items
  - Observes conditions from test plan happening in simulation
  - In VHDL / OSVVM, implemented using a package
- Item Coverage - FC relationships within a single object
  - Bin transfer sizes: 1, 2, 3, 4-127, 128-252, 253, 254, 255
- Cross Coverage - FC relationships between multiple objects
  - Has the each pair of registers been used with the ALU?
- Why not just use Code Coverage?
  - Tells us a line was executed, but does not correlate independent items
  - Not necessarily accurate in combinational logic
- Test Done =
  - 100 % Functional Coverage + 100 % Code Coverage

Why Functional Coverage?

- Randomization requires functional coverage
  - Otherwise what did the test do?
- "I have written a directed test for each item in the test plan, I am done right?"
  - For a small design maybe
  - However, this assumes coverage, but does not verify it
- 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 to observe all tests.

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 ;
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.

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.

SRC1 \ SRC2	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

- "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 (IT)

SRC1 \ SRC2	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

- Randomly select holes in the Functional Coverage
  - Random walk across functional coverage holes
  - "Coverage driven randomization" - but term is misused by others

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 ;
```

Same test using Intelligent Coverage

Runs 64 iterations @ 5X faster

Intelligent Coverage Randomization

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

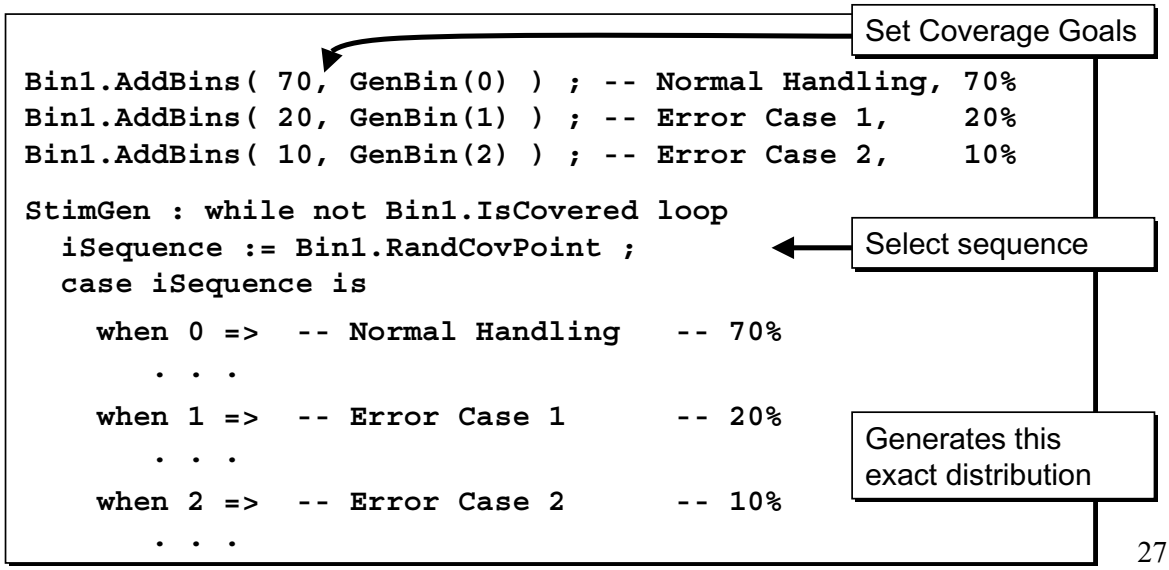
Refinement of Intelligent Coverage

- 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 ;
```

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):



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

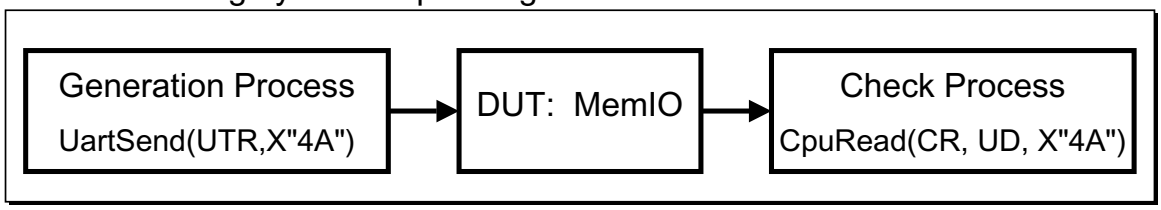
```
TestProc : process
begin
  for i in 0 to 7 loop
    for j in 0 to 7 loop
      if i /= j then
        -- non-diagonal
        ACov.AddCross(2, GenBin(i),  GenBin(j));
      else
        -- diagonal
        ACov.AddCross(4, GenBin(i),  GenBin(j));
      end if ;
    ...
```

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.
  - Closure depends on CR driving inputs to FC.
  - After simulation, analyze FC
  - Prune out tests that are not increasing FC
  - Tests partitioned based on modified controls, constraint sets, and seeds
  - Must merge FC database for all tests
- Intelligent Coverage is less work than Constrained Random

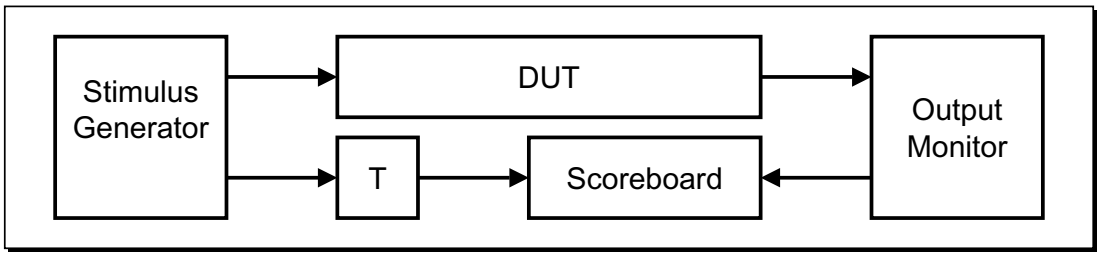
Self-Checking

- Self-Checking = compare results vs. known good results
- Self-Checking methods:
  - Embedding Data
  - Reference Model
  - Compare against saved "Golden" results (text file or waveform)
- Self-Checking by two cooperating interfaces



Scoreboards & Self-Checking

- A Scoreboard is a data structure to facilitate self checking.
  - FIFO of Expected Values
  - Methods to compared a received value with expected value
  - Method to track error count



- Essential for transport of data (networks) with little transformation
- A Scoreboard may have:
  - Support to transform expected value to received value
  - Support for out of order execution
  - Support for dropped packets
  - Generics to facilitate parameterization

Additional Pieces of Verification

- Memory Modeling
  - Large memories need space saving algorithm
  - Data structure needs to be easy to use and help readabity
- Synchronization Utilities
  - Used to synchronize independent processes (threads) of code
- Reporting Utilities

Dispelling FUD

- FC and CR require language syntax and OO
  - FC and CR only require data structures
  - Packages + protected types work as well as or better
- TLM / BFMs Require OO + Factory Class
  - TLM / BFMs easier to implement concurrently - just like RTL code.
  - Architectures are the Factory Class of concurrent programming
- Randomization Requires a Solver
  - Intelligent coverage is O(logN) faster than a solver = more balanced
- Verification Requires Fork & Join
  - Use the concurrency built into VHDL.
  - Use entity + architecture for bundling
  - Use separate processes for independent handling of sequences
  - Use handshaking to synchronize independent processes
  - 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 intelligent coverage, 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

VHDL Testbench Summary

- VHDL support all important testbench features
  - TLM, CR, FC, IT, Reuse, Interfaces, Concurrency and Synchronization, Scoreboards, Memory Models
- Better than SystemVerilog / 'e' Capabilities
  - Functional Coverage - Sequential, Incremental, Conditional
  - Intelligent Testbenches built-in
  - FC, CR, and IT that can be refined with code
  - Extensible, just add to the packages
  - Mixed environments (directed, algorithmic, file, CR, IT)
  - Simple and Readable by All (Verification and RTL Engineers)
  - Faster Coverage Closure
  - Faster Simulations - No redundant stimulus (log N) and no solver
- SystemVerilog?
  - Less capable, slower, requires a specialist , alienates RTL engineers

---

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:
  - May be already installed in your simulator's osvvm library
  - http://www.osvvm.org/downloads
  - http://www.synthworks.com/downloads