Creating High Fidelity Cross Coverage Models

by
Jim Lewis

SynthWorks VHDL Training
Jim@SynthWorks.com

Copyright © 2011, SynthWorks Design Inc.
Rights to this work are licensed under a Creative Commons Attribution-NonCommercial-No Derivatives 3.0 United States License.
http://creativecommons.org/licenses/by-nc-nd/3.0/us/

You are free to share, copy, distribute, and transmit this work under the following conditions:
You must attribute the work with this page.
You may not alter, transform, or build upon this work.
You may not use this work for commercial purposes.

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/downloads

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

www.SynthWorks.com
High Fidelity Cross Coverage

Goal
- Understand why we need functional coverage
- Develop High Fidelity Cross Coverage Models
- Essential to tell us we have tested all requirements / features

Topics
- What about Code coverage?
- What is Functional Coverage?
- Writing Coverage
  - Tracking Transfer Sizes
  - Tracking UART Status Conditions
  - Tracking ALU Input Register Pairs
  - Tracking CPU Coverage
- Randomizing Stimulus Using Coverage
  - ALU Stimulus, Equal Weights
  - UART Stimulus, Different Weights

What about Code Coverage?
- Code coverage is automatically collected by the simulator, however, ...

```
PrioritySel : process (SelA, SelB, SelC, A, B, C)
begin
  Y <= "00000000" ;
  if (SelC = '1') then
    Y <= C ;
  end if ;
  if (SelB = '1') then
    Y <= B ;
  end if ;
  if (SelA = '1') then
    Y <= A ;
  end if ;
end process ;
```

Issues:
- SelA = SelB = SelC = 1, Cov = 100%
- Delta cycle glitches are recorded as Cov

What we do know:
- < 100% coverage, need more tests
- = 100%, ???

Other Issues:
- Cannot detect missing features
- Cannot detect things not in code
  - Independent ports both at worst case
What is Functional Coverage?

- Code that measures requirements and features (design and test)
  - Boundary conditions of different IP cores in worst case operation
- What to Measure?
  - Transfer size: 1, 2, 3, 4-127, 128-252, 253, 254, 255 (test requirement)
  - UART: All normal & error conditions observed in status register
  - ALU: Every source input pair has occurred
  - CPU: Mixture of sources and immediate values
- Classification
  - Point/Item coverage = covering a single value
  - Cross coverage = covering multiple different values
- When to Collect Coverage
  - At an event (rising_edge(Clk))
  - When a transaction completes
- 100 % Functional Coverage + 100 % Code Coverage = Done

Tracking Transfer Sizes

- Worst case conditions occur when transfer sizes are smaller or larger

<table>
<thead>
<tr>
<th>Transfer Sizes</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4 to 127</td>
<td></td>
</tr>
<tr>
<td>128 to 252</td>
<td></td>
</tr>
<tr>
<td>253</td>
<td></td>
</tr>
<tr>
<td>254</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

- Methods:
  - Manual
  - Using CoveragePkg
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 =>       increment( Bin(2) ) ;
    when   3 =>       increment( Bin(3) ) ;
    when 4 to 127 =>     increment( Bin(4) ) ;
    when 128 to 252 =>   increment( Bin(5) ) ;
    when 253 =>     increment( Bin(6) ) ;
    when 254 =>     increment( Bin(7) ) ;
    when 255 =>     increment( Bin(8) ) ;
    when others => null ;
  end case ;
end process ;

---

CoveragePkg

- Simplifies modeling and collection of coverage.
- Contains functions, procedures and a protected type.
  - The protected type encapsulates the data structure and configuration

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

- Usage of the methods and functions replace the need for language syntax
Tracking Transfer Sizes: 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 ));
    loop
      wait until rising_edge(Clk) and DValid = '1';
      Bin1.ICover(to_integer(ActualData));
    end loop;
  end process;
  ReportCov : process
  begin
    wait until rising_edge(Clk) and Bin1.IsCovered;
    Bin1.WriteBin;
  end process;
end;

Bin Construction: AddBins + GenBin

- Method AddBins: Creates internal data structure in CovPType.
- GenBin: Creates an array of bins, the input to AddBins

```
CovBin1.AddBins( GenBin( 1, 3, 3 ));
```

- Create 3 bins with ranges: 1 to 1, 2 to 2, and 3 to 3.

- Consecutive calls to AddBins creates additional bins

```
CovBin1.AddBins( GenBin( 4, 252, 2 ));
```

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

- GenBin without NumBins creates one bin per value

```
CovBin1.AddBins( GenBin( 253, 255 ));
```

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

- GenBin a single parameter creates one bin: 1 to 1

```
CovBin1.AddBins( GenBin( 1 ));
```

SynthWorks
Tracking UART Status Conditions

- Testing interfaces, here a UART:

```
Testbench

UartTxModel  DUT: UART  CpuModel
```

- For the UART, we track the following items

<table>
<thead>
<tr>
<th>Condition</th>
<th>Break Error</th>
<th>Stop Error</th>
<th>Parity Error</th>
<th>Done Flag</th>
<th>Integer Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Transfer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Parity Error</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Stop Error</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Parity &amp; Stop Error</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Break Error</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>9-15</td>
</tr>
</tbody>
</table>

- Result: List of conditions that must be covered

```
architecture Test2 of tb is
  shared variable UCov : CovPType ;  -- Cov Object
begin
  CollectCov : process
  begin
    -- Create Coverage Bins, Binary
    UCov.AddCross(GenBin(0),GenBin(0),GenBin(0),GenBin(1));  -- Normal
    UCov.AddCross(GenBin(0),GenBin(0),GenBin(1),GenBin(1));  -- Parity
    UCov.AddCross(GenBin(0),GenBin(1),GenBin(0),GenBin(1));  -- Stop
    UCov.AddCross(GenBin(0),GenBin(1),GenBin(1),GenBin(1));  -- SE+PE
    UCov.AddCross(GenBin(1), ALL_BIN, ALL_BIN,GenBin(1));  -- Break
    while not TestDone loop
      CpuUartGet(RxRec, Data, Status) ;  -- Rx Transaction
      UCov.ICover(to_integer_vector(status)) ;  -- Collect Cov
    end loop ;
    UCov.WriteBin ;  -- Report Coverage
    EndStatus(. . .) ;
  end process ;
end process ;
```

Coverage sampling is transaction based.
Tracking ALU Input Register Pairs

- Testing an ALU with Multiple Inputs:
  
  ![Diagram of ALU inputs](image)

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

  ![Matrix of register pairs](image)

  - Result: Matrix of conditions that must be covered

---

Tracking ALU Input Register Pairs

```vhdl
architecture Test3 of tb is
  shared variable ACov : CovPType ;  -- Cov Object
begin
  CollectCov : process
  begin
    -- Create Coverage Bins
    ACov.AddCross( GenBin(0,7), GenBin(0,7) );

    while not Done loop
      -- Code to generate RegIn1 & RegIn2 - more later
      DoAluOp(TRec, RegIn1, RegIn2) ;
      ACov.ICover( (RegIn1, RegIn2) ) ;
    end loop ;
    ACov.WriteBin ;  -- Report Coverage
    EndStatus( . . . ) ;
  end process ;
end architecture Test3 of tb ;
```

Matrix coverage creates many bins in a quick simple format.
CPU Coverage

Testing an CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Word</td>
<td>LW</td>
<td>Register Target, Register Base, 16-bit offset</td>
</tr>
<tr>
<td>Store Word</td>
<td>SW</td>
<td>Register Target, Register Base, 16-bit offset</td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>Register Target, Register Src1, Register Src2</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>Register Target, Register Src1, Register Src2</td>
</tr>
<tr>
<td>Branch Equal</td>
<td>BEQ</td>
<td>Register Src1, Register Src2, 16-bit offset</td>
</tr>
<tr>
<td>Branch Not Equal</td>
<td>BNE</td>
<td>Register Src1, Register Src2, 16-bit offset</td>
</tr>
</tbody>
</table>

Coverage

- LW/SW: test two register variations with a set of offset values
- ADD/SUB: test three register variations
- BEQ/BNE: test two register variations with a set of offset values

Result: Hybrid coverage = Mixture of lists and matrices

- Not all instructions use all of the fields.

CPU Coverage

- Constants to simplify coverage capture

```plaintext
-- Individual bins for each instruction
constant LW_INST  : CovBinType := GenBin(0) ;
class constant SW_INST  : CovBinType := GenBin(1) ;
class constant ADD_INST : CovBinType := GenBin(2) ;
class constant SUB_INST : CovBinType := GenBin(3) ;
class constant BEQ_INST : CovBinType := GenBin(4) ;
class constant BNE_INST : CovBinType := GenBin(5) ;

-- 1 bin per register
constant REG_BIN  : CovBinType := GenBin(0,7) ;

-- Offset values in bins with
-- individual small & large values
constant SIGN_OFF : CovBinType :=
    GenBin(-4,4,9) & -- small
    GenBin(5,2**15-5,4) & -- medium positive
    GenBin(-5,-2**15-4,4) & -- medium negative
    GenBin(2**15-4,2**15-1,4) & -- large positive
    GenBin(-2**15-3,-2**15,4) ; -- large negative
```
CPU Coverage

architecture Test3 of tb is
  shared variable CCov : CovPType ; -- Cov Object
  . . .
begin
  CollectCov : process
  begin
  --             Inst     Reg1     Reg2     Reg3     Offset
  C Cov . AddCross ( LW_INST, REG_BIN, REG_BIN, ALL_BIN, SIGN_OFF );
  C Cov . AddCross ( SW_INST, REG_BIN, REG_BIN, ALL_BIN, SIGN_OFF );
  C Cov . AddCross ( SUB_INST, REG_BIN, REG_BIN, REG_BIN, ALL_BIN );
  C Cov . AddCross ( BEQ_INST, REG_BIN, REG_BIN, ALL_BIN, SIGN_OFF );
  C Cov . AddCross ( BNE_INST, REG_BIN, REG_BIN, ALL_BIN, SIGN_OFF );

  while not Done loop
    GetCpuInst(MonitorRec, Inst, Reg1, Reg2, Reg3, Offset) ;
    CCov . ICover ( ( Inst, Reg1, Reg2, Reg3, Offset ) ) ; --> Collect Cov
  end loop ;
  CCov . WriteBin ; -- Report Coverage
  . . .

Randomizing Stimulus Using Coverage

- Constrained random tests generate some conditions many times
  - To cover n conditions requires O(n*logn) in time
  - With additional EDA tools ($$$$$), it can approach O(n)

- To approach O(n) with CoveragePkg, we add goals and randomization

  type CovPType is protected
  . . .
  procedure AddBins ( AtLeast : integer ;
    CovBin : CovBinType
  ) ;
  procedure AddCover ( AtLeast : integer ;
    Bin1, Bin2, ... : CovBinType
  ) ;
  impure function IsCovered ( ... ) return boolean ;
  impure function RandCovPoint ( ... ) return integer_vector ;
  . . .
end protected CovPType ;
ALU Stimulus, Equal Weights

- Testing an ALU with Multiple Inputs:

![Diagram of ALU with multiple inputs]

- Goal: only cover each test case one time or same number of times

<table>
<thead>
<tr>
<th>SRC1</th>
<th>SRC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
</tr>
</tbody>
</table>

architecture Test3 of tb is

shared variable ACov : CovPType ; -- Cov Object

begin

CollectCov : process
variable Reg1, Reg2 : integer ;
begin

ACov.AddCross(2, GenBin(0,7), GenBin(0,7) );

while not ACov.IsCovered loop

(Reg1, Reg2) := ACov.RandCovPoint ;

DoAluOp(TRec, Reg1, Reg2) ;

ACov.ICover( (Reg1, Reg2) ) ;
end loop ;

ACov.WriteBin ; -- Report Coverage

EndStatus( . . . ) ;
end process ;
UART Stimulus, Different Weights

- For the UART, we want more normal transactions than others:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Frequency</th>
<th>Data Value</th>
<th>Idle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Transfer</td>
<td>63</td>
<td>0 to 255</td>
<td>0</td>
</tr>
<tr>
<td>Normal Transfer</td>
<td>10</td>
<td>0 to 255</td>
<td>1 to 15</td>
</tr>
<tr>
<td>Parity Error</td>
<td>10</td>
<td>0 to 255</td>
<td>2 to 15</td>
</tr>
<tr>
<td>Stop Error</td>
<td>10</td>
<td>1 to 255</td>
<td>2 to 15</td>
</tr>
<tr>
<td>Parity &amp; Stop Error</td>
<td>5</td>
<td>1 to 255</td>
<td>2 to 15</td>
</tr>
<tr>
<td>Break Error</td>
<td>2</td>
<td>11 to 30*</td>
<td>2 to 15</td>
</tr>
</tbody>
</table>

* note for a break error, the value we will receive is 0

- Alternate approach, represent status/conditions as integers

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMAL</td>
<td>GenBin(1) ; -- 0001</td>
</tr>
<tr>
<td>PARITY</td>
<td>GenBin(3) ; -- 0011</td>
</tr>
<tr>
<td>STOP</td>
<td>GenBin(5) ; -- 0101</td>
</tr>
<tr>
<td>PE_SE</td>
<td>GenBin(7) ; -- 0111</td>
</tr>
<tr>
<td>BREAK</td>
<td>GenBin(9,15,1) ; -- 1--1</td>
</tr>
</tbody>
</table>

architecture Test2 of tb is

shared variable UCov : CovPType ; -- Cov Object

begin

CollectCov : process

begin

UCov.AddCross(63, NORMAL, GenBin(0,255,1), GenBin(0));
UCov.AddCross(10, NORMAL, GenBin(0,255,1), GenBin(1,15,1));
UCov.AddCross(10, PARITY, GenBin(0,255,1), GenBin(2,15,1));
UCov.AddCross(10, STOP, GenBin(1,255,1), GenBin(2,15,1));
UCov.AddCross( 5, PE_SE, GenBin(1,255,1), GenBin(2,15,1));
UCov.AddCross( 2, BREAK, GenBin(0), GenBin(2,15,1));

while not UCov.IsCovered loop

CpuUartGet(RxRec, Data, Status) ;

UartSboard.CheckActualData((Data, Status)) ;

UCov.ICover((Status, Data, IdleTime)) ;

end loop ;

end process ;
**UART Stimulus, Different Weights**

```
-- continued architecture Test2 of tb is

GenStim : process
  variable Status, Data, Idle : integer ;
  . . .
begin
  while not UCovered loop
    (Status, Data, Idle) := UCov.RandCovPoint;
    if Status = BREAK_ERROR then
      Data := RV.RandInt(11,30) ;
      end if ;
    IdleTime <= Idle ; -- passed to RX side
    UartSboard.PutExpectedData((Data, Status)) ;
    UartSend(TxRec, Data, Idle, Status) ;
  end loop ;
  wait ;
end process ;
```

---

**Summary**

- In VHDL we build functional coverage using data structures.
  - Data structure is hidden within a protected type
  - Incrementally build a high fidelity coverage model
- Basic methods provide similar capability to SystemVerilog and 'e'
- Advanced methods provide better than SystemVerilog and 'e' capability.
  - Randomly select coverage holes, forwarding to stimulus generation
  - Focus on generating missing stimulus
  - Test time approaches O(n) (significant speed-up)
  - Gives us "Intelligent Testbench" like capability using a basic simulator
- CoveragePkg is available as Open Source
  - Presentation is based on version 2.2 of CoveragePkg.
- Since it is open source, the code is reviewable.
- Use with your current VHDL testbench
- No additional licenses required