2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION TAIWAN

Better Late Than Never

Collecting Coverage From Zeroes and Ones

Rich Edelman, Siemens EDA, Fremont, CA US (rich.edelman@siemens.com)

Tsung-Yu Tsai, Siemens Taiwan, HsinChu City, Taiwan (tsungyu.tsai@siemens.com)

***Abstract*—This paper lays out a flow and strategy to read a test vector file of ones and zeroes captured from simulation, emulation, prototyping, tester or elsewhere, and then applies those test vectors to a SystemVerilog coverage model and generates coverage reports. After the fact, this technique can help understand the coverage that is represented by the captured tests – the ones and zeros.**

Keywords—SystemVerilog, Coverage, Verificatoin, Simulation, Emulation, Prototyping

I. Introduction

SystemVerilog [1] coverage is an important metric for verification completion. It helps understand risk. But if the coverage isn't built into the original DUT model then the simulation or emulation results have unknown coverage. And the unknown coverage implies unknown risk. This paper implements a solution to help reduce the risk. Two things must happen. The simulation, emulation or prototyping signal values need to be recorded. The second thing that must happen is a coverage model has to be built. With the log of signal values and a coverage model, a coverage report can be obtained which will help reduce risk. Not all kinds of coverage are possible, but useful coverage results can be achieved.

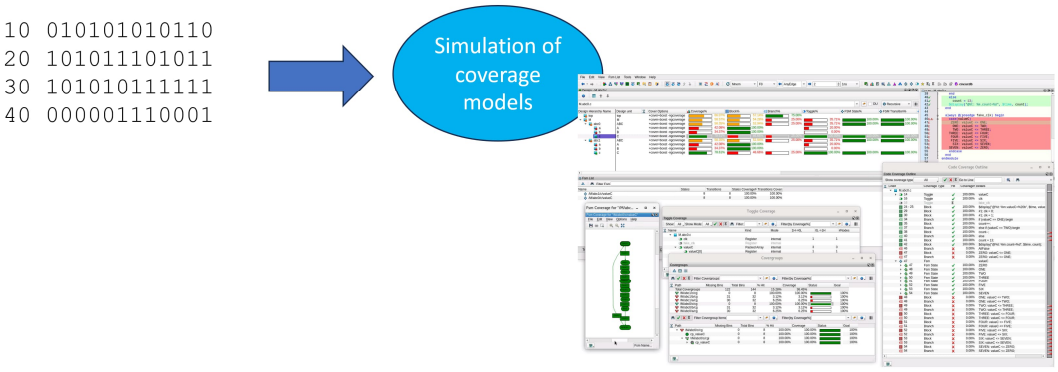


Figure 1: Simple Flow – test vectors to coverage report

II. The Captured Values

Values can be captured in any format, including VCD, qwave.db, and other wave formats. For this paper, a simple ASCII file of ones and zeroes with a time are captured. These captures can come from simulation, emulation, prototyping, abstract high level model or any place at all.

```
10  10101010110011010101011001
20  00101010110100010101011010
30  10101010110111010101011011
40  00101010111000010101011100
50  10101010111011010101011101
60  00101010111100010101011110
70  10101010111111010101011111
80  00101010110000010101011000
90  10101010110011010101011001
100 00101010110100010101011010
```

This simple format can be easily read with a Verilog program using simple $fscanf().

```
bit [1023:0] my_values;
...
code = $fscanf(fd, "%d %b", my_time, my_values);
```

For a larger testcase, more bits can be used, or multiple lines per time slot, multiple files or other solutions. Instead of a simple table of ones and zeroes, a waveform file could be used. But these require special handling (readers and writers). A simple file of ones and zeroes can be generated from many different sources and languages.

The main idea of the captured values is simplicity. Capture the values that are important and keep it simple. For example, register values or port connections to the security device. The captured values lose their names and any structure in the test vector file. Those names and how they are organized is important for the coverage model and simulation – the right values must get assigned to the right variables in the simulation so that the right coverage report will be generated.

III. The Coverage Models

The coverage models are covergroups that might have been written in the original simulation. There is nothing special about them – any covergroup constructs can be used. The coverage models are built in a module that represents where a signal might have been in the original design (`top.m.abc.b.valueB`). Inside that module, a coverage model is added.

First, just the module B, and the 5 bits of valueB. Then add the 'always block' and the covergroup, the covergroup construction and the call to sample. Our DUT top level "control" module, will set the value 'valueB'. When it does get set, then call the covergroup sample() routine.

This coverage model is simple. Just cover the bits and construct one of the covergroup objects.

```
module B();
  reg [4:0] valueB;

  covergroup cg;
    cp_valueB: coverpoint valueB;
  endgroup

  cg cgi = new();

  always @(valueB) begin
    $display("@%t: %m.valueB=%20b", $time, valueB);
    cgi.sample();
  end
endmodule
```

With this coverage model and declarations, a coverage report as below can be generated

Path	Missing Bin	Total Bins	% Hit	Coverage	Status	Goal
/M/abc1/b/cg	31	32	3.12%	3.12%		100%
cp_valueB	31	32	3.12%	3.12%		100%
VM/abc1/b/cgi	31	32	3.12%	3.12%		100%
cp_valueB	31	32	3.12%	3.12%		100%
auto[0]				0		1
auto[1]				0		1
auto[2]				0		1
auto[3]				0		1
auto[4]				0		1
auto[5]				0		1
auto[6]				0		1
auto[7]				0		1
auto[8]				0		1
auto[9]				0		1
auto[10]				0		1
auto[11]				1		1
auto[12]				0		1
auto[13]				0		1
auto[14]				0		1
auto[15]				0		1

Figure 2: Coverage Report for 'reg[4:0] valueB'

That coverage is too simple – the bit vector is a status register. Defining the 'valueB' variable as a packed struct, and the covergroup is changed to have coverpoints for each field of the struct.

```
module B();
  typedef struct packed {
    reg [1:0] status;
    reg       intr;
    reg [1:0] count;
  } csr_reg_t;
  csr_reg_t valueB;

  covergroup cg;
    //cp_valueB: coverpoint valueB;
    status: coverpoint valueB.status;
      intr: coverpoint valueB.intr;
     count: coverpoint valueB.count;
  endgroup

  cg cgi = new();

  always @(valueB) begin
    $display("@%t: %m.valueB=%20b", $time, valueB);
    cgi.sample();
  end
endmodule
```

The coverage report now has more details – details about each field of the struct.

Path	Missing Bin	Total Bins	% Hit	Coverage	Status	Goal
/M/abc1/b/cg	7	10	30.00%	33.33%		100%
status	3	4	25.00%	25.00%		100%
intr	1	2	50.00%	50.00%		100%
count	3	4	25.00%	25.00%		100%
VM/abc1/b/cgi	7	10	30.00%	33.33%		100%
status	3	4	25.00%	25.00%		100%
auto[0]				0		1
auto[1]				1		1
auto[2]				0		1
auto[3]				0		1
intr	1	2	50.00%	50.00%		100%
auto[0]				1		1
auto[1]				0		1
count	3	4	25.00%	25.00%		100%
auto[0]				0		1
auto[1]				0		1
auto[2]				0		1
auto[3]				1		1

Figure 3: Coverage report for 'struct packed valueB'

The fidelity and detail of the coverage model will dictate the detail of the coverage report. Simply knowing "the big picture" may be sufficient, but by refining the model on successive runs, more coverage report details will become available.

IV. The "Top Of The DUT"

A DUT top level needs be built to hold the module-instance skeleton that is holding the values.

```
module M();
  ABC abc0();
  ABC abc1();
  // When the intermediate vector changes, assign its contents to the
  // underlying values - deep in the hierarchy or on the top
  always @(top.vector) begin
    $display("@%t: Vector=%20b", $time, top.vector);
    {
      abc0.a.valueA, abc0.b.valueB, abc0.c.valueC,
      abc1.a.valueA, abc1.b.valueB, abc1.c.valueC
    } = top.vector;
  end
endmodule
```

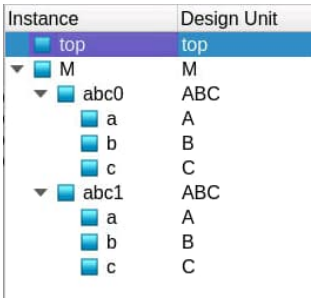


Figure 4: The Coverage Model Hierarchy

The DUT top module has two jobs. First to instantiate the instances below. And second, to wait for the file reader to update the vector.

This DUT top module instantiates two instances of the module ABC, which in turn instantiates one each of module ‘a’, module ‘b’ and module ‘c’.

The DUT module is a “top” for simulation.

The value read from the file is put into a “global” in the top level module named ‘vector’. The DUT top module simply references it as top.vector. The DUT top module then assigns that large bit vector to a concatenation of the proper signals in the proper order. This process is fraught, since the signal bit width in the DUT model must match from the concatenated values in the captured test vector file.

With careful attention and some debug messages, it is easy to manage any issues.

After the assignment to the concatenation, the values deep in the hierarchy have been set, and their ‘always blocks’ will trigger the call to the sample() routine.

V. The File Reader

The file reader is very simple. See the Appendix for the complete source code.

The concept is simple. Read a complete line from a test vector file using $fscanf(). Assign the bit vector to a temporary variable. Advanced time as needed. Assign the temporary variable value into the “top.vector” variable which is being waited on by the coverage model.

It compiles and runs easily as below. The two tops (top and M) taken together are the file reader and the DUT coverage model.

```
vlog a.sv b.sv c.sv abc.sv dut.sv t.sv
vopt -o opt top M -debug +designfile -ngcoverage +cover=sbftce -fsmdebug
vsim -c opt -do "coverage save -onexit cov.ucdb; run -all; quit -f" -qwavedb=+signal \
   -coverage +i=datafile.txt

visualizer design.bin qwave.db -ucdbfile cov.ucdb
```

VI. Writing the Coverage Model

The concept is simple. An outside entity (the test vector reader) manages to assign a value in a hierarchy, then "coverage is sampled". Writing covergroups is straightforward. But additional coverage can be attained with some careful thinking.

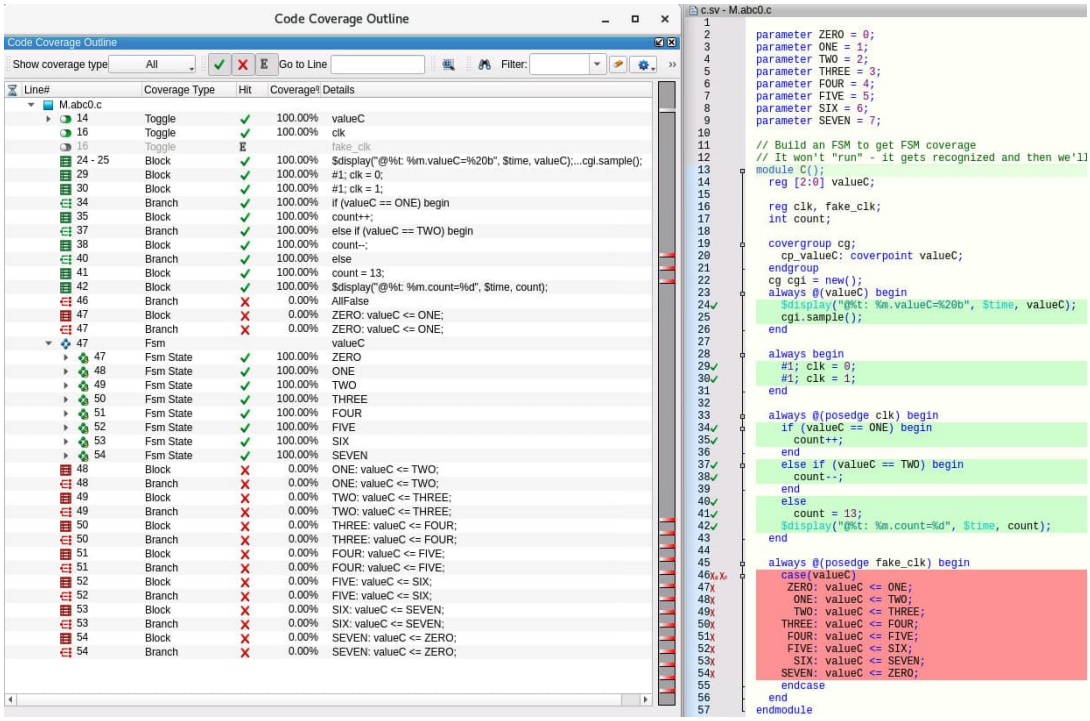
In the module 'c', there is an FSM built. It must be built so that it is recognized and FSM coverage can be reported. But that FSM should not ever execute – the coverage model should never change any value that it is covering. Changes to variables being measured for coverage must only come from the test vector file assignments.

Toggle coverage will be useful, but no line or block coverage, since the execution of the lines are artificial (not part of the chip, instead part of the coverage model). Branch coverage can be attained, but is a little tricky to write. The branches need to execute in the coverage simulation without changing any covered variables.

Design Hierarchy Name	Design unit	Cover Options	Coverage%	Block%	Branch%	Toggle%	FSM State%	FSM Transition%	Covergroup Bin Hit%	Covergroup%
top	top	+cover=bcest -ngcoverage	66.07%	57.14%	75.00%					
M	M	+cover=bcest -ngcoverage	58.57%	54.28%	25.00%	35.71%	100.00%	100.00%	15.27%	36.45%
abc0	ABC	+cover=bcest -ngcoverage	58.35%	52.94%	25.00%	35.71%	100.00%	100.00%	15.27%	36.45%
a	A	+cover=bcest -ngcoverage	42.08%	100.00%		20.00%			6.25%	6.25%
b	B	+cover=bcest -ngcoverage	34.37%	100.00%		0.00%			3.12%	3.12%
c	C	+cover=bcest -ngcoverage	78.61%	46.66%	25.00%	100.00%	100.00%	100.00%	100.00%	100.00%
abc1	ABC	+cover=bcest -ngcoverage	58.35%	52.94%	25.00%	35.71%	100.00%	100.00%	15.27%	36.45%
a	A	+cover=bcest -ngcoverage	42.08%	100.00%		20.00%			6.25%	6.25%
b	B	+cover=bcest -ngcoverage	34.37%	100.00%		0.00%			3.12%	3.12%
c	C	+cover=bcest -ngcoverage	78.61%	46.66%	25.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Figure 5: Hierarchical Coverage Roll-ups

In the screenshot below, the left is a coverage outline – for the module instance 'c'. The red highlighted lines on the right are not covered – that state machine never executed. That's expected. The if-then-else on lines 33 to 43 are covered – that code did execute, but didn't change any covered variables – so it is ok to use.



VII. Conclusion

Using a simple test vector file of ones and zeroes, and a simple skeleton hierarchy and simple coverage models a useful coverage report can be extracted. Without this coverage report, it's unknown how effective the test was at achieving the necessary coverage. The coverage cannot be as complete as running originally in simulation or emulation, but it can still be used to judge risk.

The source code is contained in Appendix I and II or contact the author for a source code copy.

VIII. References

[1] SystemVerilog - 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, https://ieeexplore.ieee.org/document/8299595

IX. APPENDIX I – THE DUT COVERAGE MODEL

```
// ============================================
// FILE: a.sv
// ============================================

// Each module of interest must be built
// Module A has a 5 bit value
// A covergroup should be built - any complexity
// An always block to call sample() and
//  print a debug message

module A();
  reg [4:0] valueA;

  covergroup cg;
    cp_valueA: coverpoint valueA;
  endgroup

  cg cgi = new();

  always @(valueA) begin
    $display("@%t: %m.valueA=%20b",
      $time, valueA);
    cgi.sample();
  end
endmodule


// ============================================
// FILE: b.sv
// ============================================

module B();
  reg [4:0] valueB;

  covergroup cg;
    cp_valueB: coverpoint valueB;
  endgroup

  cg cgi = new();

  always @(valueB) begin
    $display("@%t: %m.valueB=%20b",
      $time, valueB);
    cgi.sample();
  end
endmodule
```

```
// ============================================
// FILE: dut.sv
// ============================================
module M();
  ABC abc0();
  ABC abc1();

  // When the intermediate vector changes,
  //  assign its contents to the
  //  underlying values - deep in the hierarchy
  //  or on the top
  always @(top.vector) begin
    $display("@%t: Vector=%20b",
      $time, top.vector);
    {
      abc0.a.valueA, abc0.b.valueB, abc0.c.valueC,
      abc1.a.valueA, abc1.b.valueB, abc1.c.valueC
    } = top.vector;
  end
endmodule
```

```
// ============================================
// FILE: c.sv
// ============================================

parameter ZERO = 0;
parameter ONE = 1;
parameter TWO = 2;
parameter THREE = 3;
parameter FOUR = 4;
parameter FIVE = 5;
parameter SIX = 6;
parameter SEVEN = 7;

// Build an FSM to get FSM coverage
// It won't "run" - it gets recognized and
//  then we'll set the values
module C();
  reg [2:0] valueC;

  reg clk, fake_clk;
  int count;

  covergroup cg;
    cp_valueC: coverpoint valueC;
  endgroup
  cg cgi = new();
  always @(valueC) begin
    $display("@%t: %m.valueC=%20b",
      $time, valueC);
    cgi.sample();
  end

  always begin
    #1; clk = 0;
    #1; clk = 1;
  end

  always @(posedge clk) begin
    if (valueC == ONE) begin
      count++;
    end
    else if (valueC == TWO) begin
      count--;
    end
    else
      count = 13;
    $display("@%t: %m.count=%d", $time, count);
  end

  always @(posedge fake_clk) begin
    case(valueC)
     ZERO: valueC <= ONE;
      ONE: valueC <= TWO;
      TWO: valueC <= THREE;
    THREE: valueC <= FOUR;
     FOUR: valueC <= FIVE;
     FIVE: valueC <= SIX;
      SIX: valueC <= SEVEN;
    SEVEN: valueC <= ZERO;
    endcase
  end
endmodule

// ============================================
// FILE: abc.sv
// ============================================

module ABC();
  A a();
  B b();
  C c();
endmodule
```

X. APPENDIX II – THE TEST VECTOR READER & TEST VECTOR FILE

```
// ================================================
// FILE: t.sv
// ================================================
module top();

  bit [1023:0] vector; // A "global" variable to hold the line JUST read.
                       // The DUT-coverage-model uses this to assign the parts.

  initial begin
    bit [1023:0] my_values;
    string filename;
    longint my_time;
    longint now;
    int d;

    integer fd;
    integer code;
    now = 0;

    if (!$value$plusargs("i=%s", filename))
      filename = "testfile.txt";
    $display("...processing '%s'", filename);

    // Open the "values" file
    fd = $fopen(filename, "r");

    // Loop through each line, one at a time.
    //   1. Update the time
    //   2. Apply the values
    //   3. Repeat for each line
    //
    forever begin

       // Read a line
       code = $fscanf(fd, "%d %b", my_time, my_values);
       if (code == -1) begin
         $display("INFO: Reached EOF on input file %s", filename);
         $finish(2);
       end
       // Debug - echo the line read
       $display("READ @%0d %0b", my_time, my_values);

       // Update time
       d = my_time - now;
       #d;
       now = now + d;

       // Apply the values to the intermediate vector
       vector = my_values;
       #0; // Yield - let the assign happen and
           //   any sample(), before processing a new line
    end
  end
endmodule



// ================================================
// FILE: datafile.txt
// ================================================
10  1010101011001101010101011001
20  0010101011010001010101011010
30  1010101011011101010101011011
40  0010101011100001010101011100
50  1010101011101101010101011101
60  0010101011110001010101011110
70  1010101011111101010101011111
80  0010101011000001010101011000
90  1010101011001101010101011001
100 0010101011010001010101011010
```