2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION TAIWAN

Better Late Than Never
Collecting Coverage from Zeroes and Ones

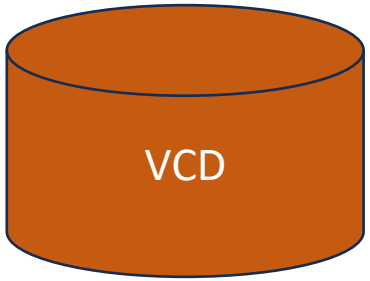
Rich Edelman, Siemens EDA, Fremont, CA US
Tsung-Yu Tsai, Siemens Taiwan, HsinChu City, Taiwan

SIEMENS

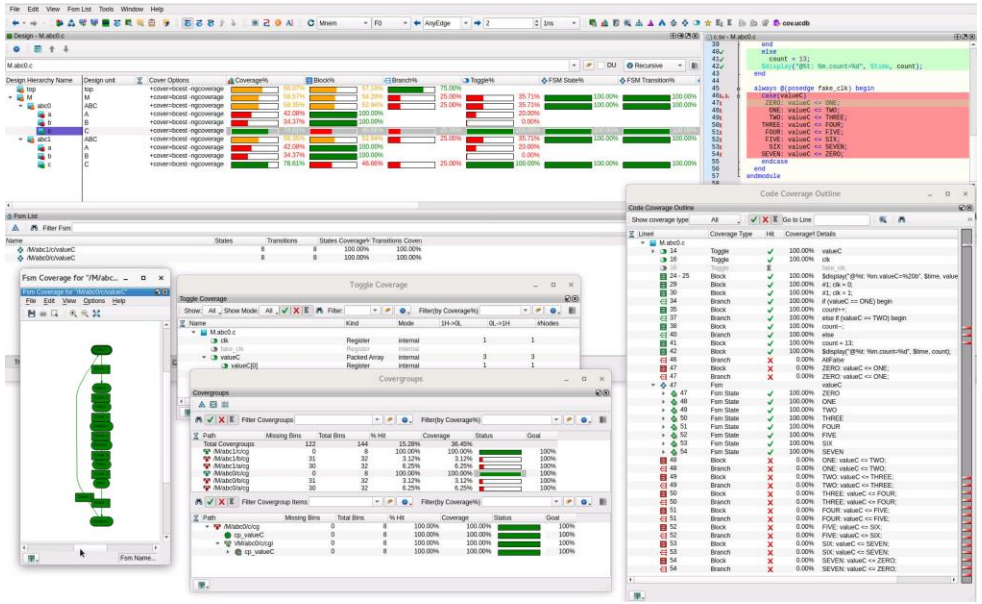
accellera SYSTEMS INITIATIVE

Background

- Files of zeroes and ones
  - ASCII – expected and actual
  - VCD
  - qwave.db
  - WLF
  - other
- Coverage
  - Toggle
  - Block
  - FSM
  - Conditional
  - Functional




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



Concept – Imagination

- Coverage wasn't modeled
  - Management decision - No time in the schedule – maybe it is an FPGA
- But we'd like to know about some of the coverage
- Build a Verilog instance tree that has names like the real design
  - This makes the coverage report easy to read
  - At each instance, populate it with the "sampled value datatypes" (reg[3:0]valueB)
- Build a functional coverage model for the variables.
  - coverpoint, bins, crosses
- Now, assign those zeroes and ones to the variables in turn. Collecting coverage

Conceptual Flow

```
10 010101010110
20 101011101011
30 101010111111
40 000001110001
```

The File Reader

- A “global” variable to hold the line of bits
- Get the filename
- Open the file using $fopen

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

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

    integer fd, code;
    now = 0;

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

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

The File Reader

- Read the whole file
- Use $fscanf
  - Read the time
  - Read the bit vector
- Update the current time
- Apply the bits to the “global” holding bit vector

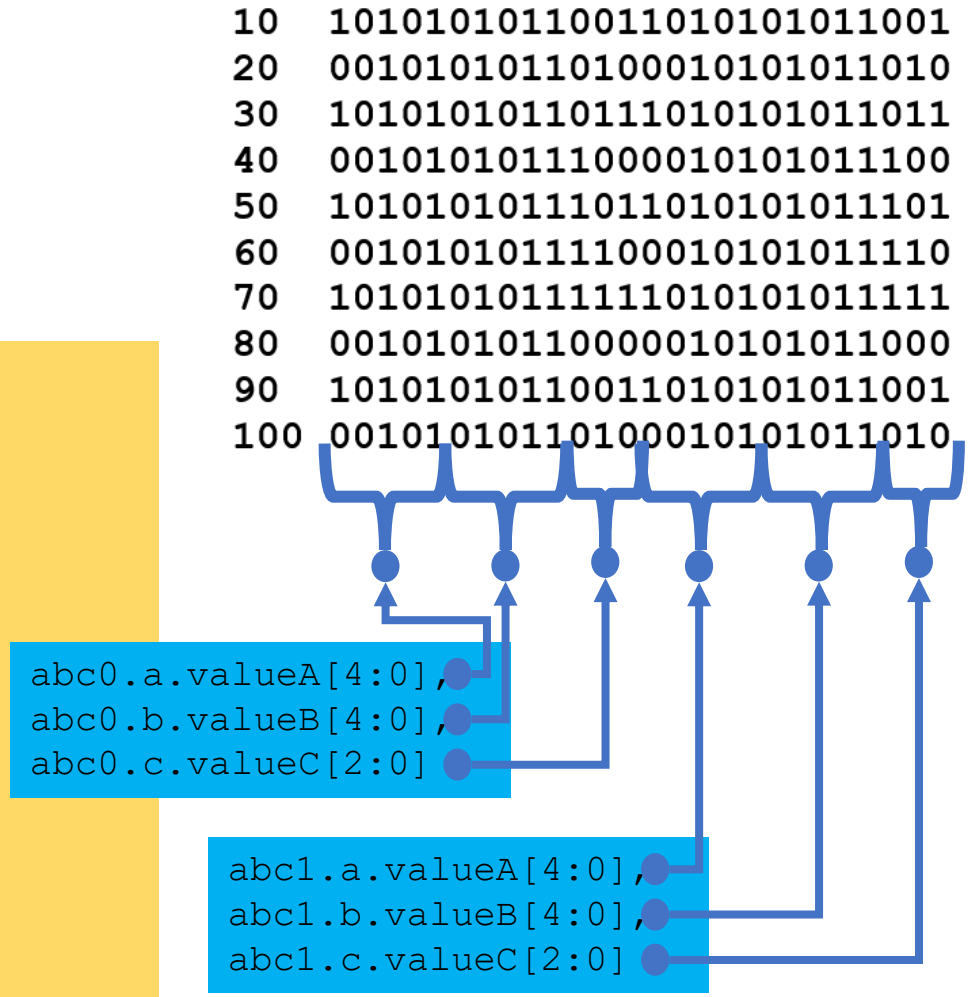
```
    // Loop through each line, one at a time.
    //   1. Update the time
    //   2. Apply the values
    //   3. Repeat for each line
    forever begin
      // Read a line
      code = $fscanf(fd, "%d %b", my_time, my_values);
      if (code == -1) begin
        $finish(2);
      end
      // Update time
      d = my_time - now;
      #d;
      now = now + d;

      // Apply the values to the global
      vector = my_values;
    end
  end
endmodule
```

Assigning the bits

```
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
```



A Simple Functional Coverage Model

- Construct a coverage object ‘cgi’.
- Write a coverage model. Simple in this case.
- Inside the module A(), a value ‘valueA’.
- valueA got assigned by the concatenation above.
- Once valueA changes, trigger a call to sample().

```
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
```

Structs too

- Exactly the same concepts.
- Assign, trigger, call sample()

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

  covergroup cg;
    cp_valueB: coverpoint valueB;
  endgroup

  cg cgi = new();

  always @(valueB) begin
    cgi.sample();
  end
endmodule
```

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

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

  cg cgi = new();

  always @(valueB) begin
    cgi.sample();
  end
endmodule
```

Build a covergroup appropriate for the data type – bit vector or struct – for example

Coverage

- Bit Vector model
- Struct model

Path	Missing Bin	Total Bins	% Hit	Coverage	Status	Goal
/M/abc1/b/cg	31	32	3.12%	3.12%		10
cp_valueB	31	32	3.12%	3.12%		10
VM/abc1/b/cgi	31	32	3.12%	3.12%		10
cp_valueB	31	32	3.12%	3.12%		10
auto[0]				0		
auto[1]				0		
auto[2]				0		
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

Bit Vector

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

Struct

Interesting beyond functional coverage

- This is our “regular” model
- A value is going to be assigned as read from the file
- A covergroup was designed
- cgi.sample() is triggered

- What other coverage can be collected?

```
module C();
  reg [2:0] valueC;

  covergroup cg;
    cp_valueC: coverpoint valueC;
  endgroup
  cg cgi = new();
  always @(valueC) begin
    cgi.sample();
  end
endmodule
```

What about other kinds of coverage?

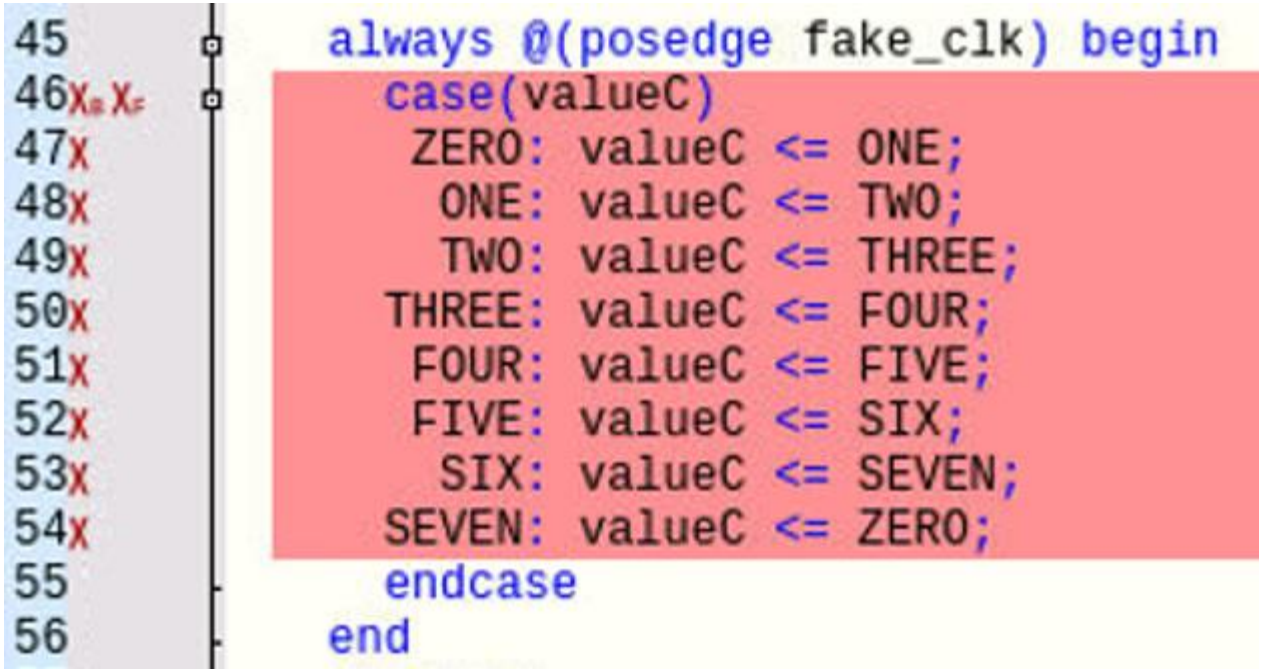
- Write a “fake” FSM in the ‘module C’
- It gets recognized by the compiler / optimizer
- But it never operates – the states are assigned
  - Notice the ‘fake_clk”
    - it doesn’t run

```
module C();
  reg [2:0] valueC;
  reg clk, fake_clk;

  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
```

Interesting beyond functional coverage 1

- Since the state machine doesn't "run"
  - Block / Statement coverage doesn't register

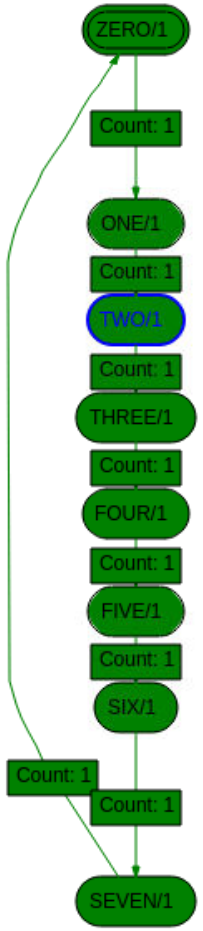


Interesting beyond functional coverage 2

- ‘valueC’ is being assigned
- the state and transition coverage is collected

```
module C();
  reg [2:0] valueC;
  reg clk, fake_clk;

  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
```



Interesting beyond functional coverage 3

- An always block that operates expressions but does NOT change the “values” that are assigned from the file

```
27
28      always begin
29✓       #1; clk = 0;
30✓       #1; clk = 1;
31      end
32
33      always @(posedge clk) begin
34✓       if (valueC == ONE) begin
35✓         count++;
36        end
37✓       else if (valueC == TWO) begin
38✓         count--;
39        end
40✓       else
41✓         count = 13;
42✓       $display("@%t: %m.count=%d", $time, count);
43      end
```

```
module C();
  reg [2:0] valueC;
  reg clk;
  int count;

  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;
end
```

The expressions and branches and lines can be covered – they are conditioned with ‘valueC’ from the file assigns

Coverage Roll-up

- Not all the coverage categories are “valid”
- It “depends”
  - Covergroups/Toggle/FSM – all good
  - Block/Branch – depends on the module/instance

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%

Conclusion

- Coverage is a useful tool for measuring “completeness”
- Even after the fact, a file of zeroes and ones can be used to collect coverage
  - Have all the legal values been used
  - Have all the legal “crosses” between two variables been used
- Building a small structure helps with naming conventions and reporting
- Keep the system simple
- Explore more kinds of coverage that might apply to this scheme

Questions

- Source code available – contact rich.edelman@siemens.com