Why Statement Coverage Is Not Enough

Technical Backgrounder

A Practical Strategy for Coverage Closure

Jeffrey Barkley, Marketing Manager, TransEDA

INTRODUCTION

Today, many simulators include a built-in code coverage capability. However, what you find when you look closer is that what they offer is only statement coverage. While this is a good starting metric to use it is typically insufficient to really measure coverage closure. This paper will look at common code coverage metrics and what they detect. It will then propose a set of code coverage metrics that should be used to measure your coverage closure. This paper does not include any discussion of FSM or functional coverage metrics.

Coverage Closure

Coverage closure is the process of:

- Finding areas of the HDL code not exercised by a set of tests
- Creating additional tests to increase coverage
- Determining a quantitative measure of code coverage

You use code coverage to assure completeness of your set of tests, not the quality of the design. Code coverage is one of many verification techniques; you should not rely on it alone.

CODE COVERAGE METRICS

The VSI Alliance has defined the following metrics for code coverage tools[i]:

- Statement Coverage — shows how many times each statement was executed
- Branch Coverage — shows which case or if else branches were executed
- Condition Coverage — shows how well a Boolean expression is exercised
- Path Coverage — shows which routes through sequential branching constructs were exercised
- Triggering Coverage — shows whether each process has been uniquely triggered in its sensitivity list
- Toggle Coverage — shows which bits of the signals in the design have toggled

Statement Coverage

The results display the number of times that a line of code was executed during elaboration and simulation. A zero count means this line of code has not been executed during elaboration or simulation.

One argument in favor of statement coverage over other measures is that faults are evenly distributed through out the code; therefore, the percentage of executable statements covered reflects the percentage of faults discovered. However, faults are often related to control flow, not computations. Additionally, we could reasonably expect that designers strive for a relatively constant ratio of branches to statements.

You need to verify that you have tested all of the lines of code. If you have not simulated some code, there is a possibility that the code is wrong.

If your test suite exercises all of the functionality then statements that are not covered indicates code that can potentially be removed from the design saving area.

Where 100% Is Really 0% Statement Coverage

For example, consider something as simple as a concurrent statement assignment. Just running the simulator will cause this statement to be covered since it will always be executed at time 0 when the simulator initializes the circuit.

sigx <= siga AND sigb;	assign sigx = (siga & sigb)

Figure 1 Code for statement coverage example

Any test you write even one which never drives any values on to signals siga and sigb and exits at time 0 will show you 100% statement coverage.

To ensure that concurrent statements are executed other than at time zero you need to be able to exclude initialization from consideration by using a coverage start time. In addition, for the example above even with a coverage start time statement coverage will never tell us if siga or sigb was ever a 0, a 1 or propagated any value other than X to sigx. More sophisticated metrics such as toggle and condition coverage are necessary to gather this information.

Branch Coverage

The results display the number of times that a branch statement was executed during elaboration and simulation. A zero count means that the specified part of the branch evaluated to false every time the branch was executed.

TransEDA
info@transeda.com
www.transeda.com

Branches typically contain statements. Therefore, the reporting structure of branch coverage often brings you through to the same problem area as statement coverage does. In fact, branch coverage points you more directly at the problem area than statement coverage.

Where 100% Statement is 50% Branch Coverage

If you consider the following simple piece of code:

```
PROCESS (xa)                    always@(xa)
BEGIN                            if(xa)
  IF (xa = '1' ) THEN              SA = 1 b1;
     SA <= '1';
  END IF;
END PROCESS;
```

Figure 2 Code for branch coverage example

If you create a test that sets xa to 1 you easily get 100% statement coverage. However, you will not have tested the IMPLIED multiplexers that will be inferred from this code.

You will typically get something like the following figure.

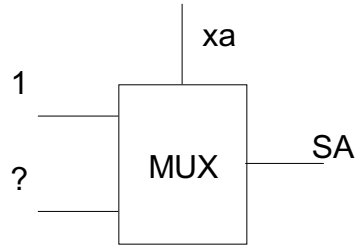

Figure 3 Logic Synthesized From and IF

Since the value of the output SA is not defined in this block for xa set to 0, statement coverage alone would not have pointed out this simple coding error.

Condition Coverage

The results display a percentage of combined values on the expressions that have occurred during elaboration and simulation. Where the condition coverage is poor, the branch will not have been executed at all. In this case, condition coverage, branch coverage and statement coverage may all bring you to the same problem.

You need to verify that you have tested the combinational logic that will be generated by the conditional expression. If you do not test it then you may have problems with control variable values, control logic operators that could result in loss of control. For example if you look at the hardware that is synthesized from a typical IF statement that involves sub-conditions.

Where 100% Statement is 33% Condition Coverage

If you consider the following simple piece of code.

```
PROCESS(xa, xb, xc)                 always@(xa or xb or xc)
BEGIN                                if ((xa  & xb) | xc)
  IF ((xa = '1' ANDxb = '0')            SA = 1 b1;
OR xc = '1') THEN                    else
     SA <= '1';                         SA = 1 b0;
  ELSE
     SA <= '0';
  END IF;
END PROCESS;
```

Figure 4 Code for condition coverage example

If you create a test where to stimulate the vectors shown on the following table.

Xa	xb	xc
0	0	0
0	0	1

This test would give you 100% statement (and 100% branch) coverage, but only 33% condition coverage. You could add many more vectors with xc at 1 and it would not increase your condition coverage.

Xa	xb	xc
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

This is because we have never sensitized the path of the and gate. You will typically get something like the following figure.

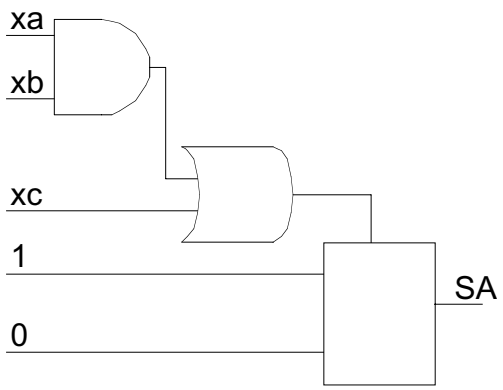

Figure 5 Logic Synthesized from Multiple Subconditions

Statement and branch coverage will tell you that you have tested the multiplexer synthesized for the output SA. However, as you can see this is only a fraction of the total logic.

This expression becomes an AND and an OR gate. Without condition coverage you will have no idea how much of the AND an OR functions have been tested.

Path Coverage

The results display if a path through consecutive branching constructs that are at the same nesting level, has been executed.

You need to verify that you can propagate signals through the multiplexer and demultiplexer or finite state machines that will be

synthesized. Branch coverage or statement coverage will only verify that you have tested the multiplexers not that all signals can be propagated to the output of the process or module.

Where 100% Statement is 25% Path Coverage

If you consider the following simple piece of code:

```
PROCESS (Done, LSB);
BEGIN
  Tmpst:= InitS;
  if Done = '1' then
     Tmpst:= EndS;
  end if;
   if LSB = '1' then
      Tmpst:= AddS;
  end if;
END PROCESS;
```

```
Always @ (Done or LSB);
begin
  Tmpst= InitS;
  if Done
     Tmpst:= EndS;
   if LSB
      Tmpst:= AddS;
end
```

Figure 6 Code for path coverage example

If a test were to stimulate the vector shown on the following table

Done	LSB
1	1

This test would give you 100% statement (and 50% branch) coverage, but only 33% condition coverage. By adding additional vectors as shown below you can get both 100% statement and branch coverage. This has also increased path coverage to 75%.

Done	LSB
1	1
0	1
1	0

However, you still will have not proven that you can propagate the signal InitS to Tmpst, because you need to have both multiplexers properly sensitized.

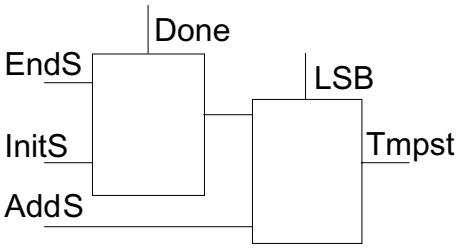

Figure 7 Logic Synthesized from Sequential IFs

There is only one combination that will sensitize the path and propagate the signal and that is when both Done and LSB are 0 as the same time.

Done	LSB
1	1
0	1
1	0
0	0

Triggering Coverage

The results display of the specified signal did, uniquely, cause the sensitivity list to trigger.

You need to verify that there are no timing related problems that might occur at gate level or post layout simulation because of one signal masking another.

Where 100% Statement is 33% Trigger Coverage

If you consider the following simple piece of code:

```
PROCESS(xa, xb, xc)
BEGIN
  IF ((xa = '1' ANDxb = '0')
OR xc = '1') THEN
     SA <= '1';
  ELSE
     SA <= '0';
  END IF;
END PROCESS;
```

```
always@(xa or xb or xc)
  if ((xa  & xb) | xc)
     SA = 1 b1;
  else
     SA = 1 b0;
```

Figure 8 Code for trigger coverage example

If you create a test that stimulates the vectors shown in the following table

xa	xb	xc
0	0	0
0	1	0

This test would give you 100% statement (and 100% branch) coverage, but only 33% trigger coverage. You could add many more vectors going through all possible vectors, achieving 100% toggle coverage and it would not increase your trigger coverage since more than one signal is changing.

xa	xb	xc
0	0	0
0	1	0
1	0	0
1	1	1
0	0	1
0	1	1
1	0	1
1	1	0

In every case xb has changed but its affect has always been masked either by a negating change in xa or masked by the change in xc.

Toggle Coverage

The results display of a signal completed a transition from the specified start value to the stop value.

If your test suite exercises all of the functionality then signals that haven t completed both a complete rise and fall transition may be not controllable and therefore stuck at a value.

100% Statement Coverage 0% Toggle Coverage

As describe in the statement coverage section if you consider something as simple as a concurrent statement. Just running the simulator will cause this statement to be covered since it will always be executed at time 0 when the simulator initializes the circuit.

sigx <= siga AND sigb;	assign sigx = (siga & sigb)

Figure 9 Code for toggle coverage example 1

Any test you create, even a test that never drives siga and sigb or drives each one to a know values by never changes them will show you 100% statement coverage while you will have only achieved 0% toggle coverage.

If you consider the following piece of code:

PROCESS(xa, xb, xc)
BEGIN
IF ((xa = '1' ANDxb = '0')
OR xc = '1') THEN
SA <= '1';
ELSE
SA <= '0';
END IF;
END PROCESS;
	always@(xa or xb or xc)
	if ((xa & xb) | xc)
	SA = 1 b1;
	else
	SA = 1 b0;

Figure 10 Code for toggle coverage example 2

If you create a test as shown by the vectors below, then this test would give you 100% statement (and 100% branch) coverage with the first two vectors and you would observe SA to be both 0 and 1 but still only have achieved 0% toggle coverage. You could add many more vectors and still be at 0% toggle coverage.

xa	xb	xc	SA
0	0	0	0
0	0	1	1
0	1	1	1
1	1	1	1

It is not until we have shown that the signal can make both a 0→1 and a 1→0 transition that we have demonstrated that we can control it.

PRACTICAL COVERAGE GOALS

Each project must choose a minimum percent of code coverage based on available verification resources and the importance of preventing post tape out failures. Clearly, safety-critical devices should have a high goal. You might set a higher coverage goal for unit testing than for system testing since a failure in lower-level code may affect multiple high-level callers.

Using statement coverage, branch coverage or condition coverage you generally want to attain 90% coverage or more. Some people feel that setting any goal less than 100% coverage does not assure quality. However, you expend a lot of effort attaining coverage approaching 100%. The same effort might find more faults in a different testing activity, such as design reviews. Avoid setting a goal lower than 80%.

Representative Coverage Strategy

Choosing good coverage strategy can greatly increase testing productivity.

Your highest level of productivity occurs when you find the most failures with the least effort. Effort is measured by the time required to create test benches, add them to your test suite and run them. It follows that you should use a coverage strategy that increases coverage as fast as possible. This gives you the greatest probability of finding failures sooner rather than later.

One strategy that usually increases coverage quickly is to first attain some coverage throughout the entire design unit being verified before striving for high coverage in any particular area. By briefly visiting each of the modules, or instances, you are likely to find obvious or gross failures early. The idea is to first look for failures that are easily found by minimal effort.

The sequence of coverage goals listed below illustrates a possible implementation of this strategy.

- Execute at least one statement in 90% of the source files.
- Execute 90% of the statements.
- Attain 90% condition coverage.
- Attain 100% condition coverage
- Attain 90% toggle coverage

Notice we do not require 100% coverage in any of the initial goals. This allows you to defer verifying the most difficult areas. This is crucial to maintaining high productivity; achieve maximum results with minimum effort.

---

[i] VSI Alliance Taxonomy of Functional Verification For Virtual Component Development and Integration Version 1.1.1, January 2001 p. 8.