

# Using Code Coverage to Enhance Design Validation

M. Zia Ullah Khan, Ph.D.  
Sandra R. Clark  
PCI Components Division  
Intel Corporation  
1900 Prairie City Road, Folsom, CA 95630

## 1.0 Abstract

Most modern IC designs implement complex design protocols that must be extensively tested to validate their correct operation. Exhaustive testing of all possible combinations is not feasible due to time and resource constraints. Thus, test designers must devise means to balance the amount of validation coverage and satisfy quality goals.

Code coverage data can be used to determine the quality of test suite by identifying components of a design that are not exercised during simulation. This information is used to guide designers in determining the areas to focus test development effort. The use of various coverage metrics to make judgments on the quality of our test programs is discussed. Experience from several ASIC development efforts is presented.

## 2.0 Introduction

Verification by simulation continues to be the main strategy for validating functionality of most complex ASICs. Test suites are created to verify the presence of desired functionality (the features of a chip) and the absence of undesired functionality (bugs). Most design teams appreciate the need to test designs thoroughly and systematically. To achieve quality, designers must exhaustively test their VHDL code. But to get to market quickly, they must limit the time they spend in simulation. Therefore, despite a desire to improve quality, designers are still faced with the dilemma of choosing between quality and cost. Due to resource constraints, there is a need to limit the task of validation by restricting its size and scope. At the same time, it is extremely important to ensure that there are no validation escapes; the cost of a single escape could be extremely high.

An on-going challenge for test developers is knowing how to ensure their tests and test efforts are being focused to achieve maximum test coverage. Coverage analysis addresses this problem by measuring the proportion of the code which has been executed. Code coverage tools provide a detection mechanism to identify untested VHDL code. As a result, the user can easily determine when there has been sufficient simulation and thereby dramatically improve the overall design team efficiency.

In the following sections we discuss the use of code coverage techniques for measuring the quality of test suites for VHDL-based designs. We describe various types of coverage methods, their relative benefits and limitations in identifying holes in a test suite. We propose a methodology for using these techniques on chip design projects. We present data from projects that used this methodology. Finally, we explore some enhancements that might improve code coverage tools.

### **3.0 Validation Environment**

The PCI Components Division of Intel develops peripheral chipsets for personal computers that enable new technologies and showcase the power and capabilities of Intel microprocessors. These components must work with an industry standard architecture and comply with many existing and evolving bus standards. Most of these chips act as controllers for cache and DRAM memories in a personal computer and manage flow of information in the system using a complex set of interacting finite state machines.

In Intel, we use an elaborate system level simulation (SLS) environment that models the relevant portions of a personal computer system. The VHDL models of new chips being designed are instantiated in this environment. The designers use various facilities provided by the SLS environment to create stimuli that exercises the design under test to verify it's functionality. The details of this environment can be found in [1].

With each new generation of designs the complexity and size is growing significantly making it increasingly difficult to create test benches that exercise all portions of the design. There are no easy means to determine if a test suite completely exercises a chip. Thus a mechanism that provides the designers an indication of the coverage of a test suite is very helpful. Furthermore, identifying the sections of the chip not exercised by the test suite helps direct the designers to focus on those areas.

### **4.0 Code Coverage Mechanism For VHDL Designs**

Gathering code coverage data has been a technique used in the software industry for sometime. With advent of modern hardware description languages the chip design process is beginning to resemble large scale software development. Higher level language constructs are used to define various data transformations in a chip. Thus, the process of testing a design can be thought of as a process to validate the various branch conditions of a high level language construct.

There are many different types of coverage testing that are used: line, branch and path coverage to name a few. The purpose of all these varieties is the same -- to identify code that has not been executed or tested. The underlying assumption is that any unexercised portion of a design is likely to have unexposed bugs.

There are several methods to determining code coverage of VHDL designs. In following sections we describe various coverage mechanism and their use in a validation environment. In our applications, we have used both Vantage's SimCoverage[2] and TransEDA's VHDLcover[3] tools.

#### **4.1 Statement Coverage**

Statement coverage provides the most basic level of analysis. Every executable line of an HDL is considered a unit (or node). During a simulation run, the simulator keeps a record of the number of times every line of code is executed. An executable line of VHDL is considered covered if it has had a transaction during a simulation run. The transaction may or may not cause an event, but the line was executed. After simulation is completed, the execution count data is annotated with the VHDL source code. Any executable line with zero count is identified as unexercised. The designers analyze this information to determine why the line was not executed. This method is a simple way to measure the effectiveness of a test suite by identifying those areas of the code that are not exercised by the test suite.

Although very useful in quickly identifying obvious test suite weaknesses, the statement coverage method suffers from the simplistic approach of only measuring transaction rates of every

executable line of code. 100% coverage does not necessarily mean all the sections of the code have been fully exercised. For instance, consider the following code segment:

```
y <= a OR b;
```

whenever **a** or **b** change value this line will be executed and a transaction count will be registered. Thus, if **a** remained unchanged while **b** did change during simulation, **y** would still have changed its value. Coverage data will indicate that this line was exercised during simulation. However, **a** and the logic feeding into it never really got exercised.

## 4.2 Branch Coverage

Branch coverage provides the next layer of coverage information above statement coverage. This analysis targets statements containing prioritized or parallel transactions including IF and CASE statements. The analysis will identify if each branch, or possible result, of these structures was exercised.

Consider the following code fragment:

```
IF a= '1' THEN
    y <= m;
END IF;
```

This code has two possible branches; the first branch is taken when **a** = '1' and the second is taken when **a** /= '1'. This code would have 100% statement coverage if **a** = '1'. However, the branch coverage would be only 50% because the condition of **a** /= '1' has not been exercised. Given the coding style of this example, only branch coverage would identify this test hole. If the user had written the code with an ELSE clause and signal assignment, statement coverage would have also identified the hole.

It may be mentioned here that the above code would infer a storage element when synthesized. By analyzing the branch coverage, the designer has an early indication of latches and registers being inferred. Thus, any unintended latches or registers in the design can be exposed.

Branch coverage provides useful information regarding test quality. Achieving high branch coverage should be considered a minimum requirement gating more stringent analysis.

## 4.3 Conditional Coverage

Conditional coverage extends the information provided by statement and branch coverage. This analysis summarizes all possible combinations of a signal or branch evaluation. The equations analyzed include conditional signal assignments and expressions in IF and ELSIF statements. Consider the code shown below:

```
y <= a OR b;
```

Conditional coverage identifies if each combination of **a** and **b** was exercised. For this example, if **a** and **b** are of type **bit**, four combinations are analyzed. This information is extremely useful in determining test quality. The condition of only **a** or **b** causing **y** to be 1 could be critical to the model reliability.

The downfall of this analysis is the amount of information can easily become unmanageable. The first reason is if **a** and **b** are of type **std\_ulogic** (an enumerated type with nine possible val-

ues) instead of bit, the permutations would increase to 81. Therefore, the user must have a mechanism to control the range of analysis. The user may determine that for coverage purposes, only the states '1' and '0' are relevant. The second cause of information growth is the number of terms in the evaluation. In this example, there are only two terms, a and b. However, most actual designs frequently have many terms. Consider the next piece of code.

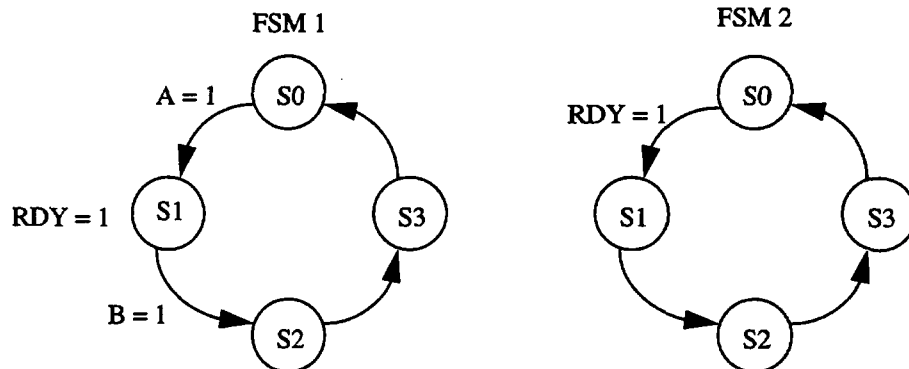
```
y <= ( a AND b AND c ) OR ( d AND e AND f ) OR g OR ( h AND m AND n );
```

Assuming type bit, the condition coverage has  $2^{10}$  combinations. Although condition coverage provides thorough analysis, the designer is overwhelmed with combinations to exercise. In addition, some of the combinations may be irrelevant. For instance, the designer may know only one piece of the OR'ed evaluations will be true at a single instance. For instance, the condition of a, b, c and g equal to '1' may be an impossible combination. For this reason, the coverage tool has a mechanism to control the analysis and provide only relevant information.

Even though condition coverage can be overwhelming in the amount of information, this coverage provides a very good indicator of the test quality. In addition, by analyzing unexercised combinations in conjunction with statement and branch coverage, the designer may eliminate unnecessary code.

#### 4.4 State or Signal Coverage

The next layer of coverage analyzes the state of a model. This coverage analyzes the relationships between multiple state machines or signals. Consider a model that includes two state machines, each having four states. These state machines may be controlling different aspects for an individual transaction. For example, FSM 1 is triggered on signal A becoming '1'. During FSM 1, state S1, the output RDY is assigned '1'. RDY then triggers FSM 2.



It may be critical for the designer to know that FSM 1 was in state S1 while FSM 2 was in states S1, S2, and S3. By identifying and analyzing all relevant combinations, the designer may realize that some of the conditions may not be handled. Furthermore, the designer may discover the FSMs have been over designed to handle impossible combinations.

The test development effort to achieve high state or signal coverage can be substantial. The example provided has only two relatively small state machines. In reality, both the number of FSMs involved in a transaction and the number of states will be much greater. However, this should not diminish the great benefit of this coverage. This coverage is excellent at identifying

boundary conditions within a model.

#### 4.5 Path Coverage

Another method of identifying boundary conditions is path coverage. Path coverage builds on the branch coverage information by tracing the flow of sequential statements. Within a process, path coverage identifies the flow of decisions. For instance, consider the following piece of code:

```
IF a = b THEN
  y <= m;
END IF;
```

```
IF c = '1' THEN
  x <= n;
END IF;
```

There are following four paths in this code:

Path #1	Condition A = B is true and condition C = 1 is true
Path #2	Condition A = B is true and condition C = 1 is false
Path #3	Condition A = B is false and condition C = 1 is true
Path #4	Condition A = B is false and condition C = 1 is false

This is a simple example and it is easy to enumerate all the paths manually. In large designs it may not be feasible to list all such paths and verify if they have been exercised. The path coverage method provides this capability by identifying all paths and measuring their coverage.

Although this type of coverage offers interesting information, our models did not gain any benefit from path coverage. The number of paths is very large and the amount of time required to generate the test scenarios outweighs the anticipated benefits.

#### 4.6 Triggering Coverage

Triggering coverage determines the individual toggling of signals listed in a sensitivity list or wait statement. The analysis identifies if each signal was toggled individually and if at least two of the signals toggled together. Trigger coverage is useful when sensitivity lists or wait statements are used to control the execution of code.

```
PROCESS (a, b)
  call_some_function;
END PROCESS;
```

In the example above, the process will not be executed unless there is an event on **a** or **b**. Given this coding style, statement coverage would indicate if the line was ever exercised. But this is not a real insight into coverage since the statement is inside a process with a sensitivity list. The statement will always be exercised at least once. The designer may really want to know if the process was evaluated due to events on **a** and **b**, only **a**, or only **b**. Triggering coverage would identify these cases.

Although this coverage may be useful in other designs, the analysis did not provide a significant benefit on our test cases. Our models are written for synthesis, so we do not control program execution through a sensitivity list. In addition, in our test cases triggering coverage must be

used in conjunction with glitch filtering. If no glitch detection is performed, a signal may temporarily have an event, but the event does not have any effects due to another signal in the equation. Overall, triggering coverage has not provided a great benefit for our applications.

## **5.0 Using Coverage Metrics**

Code coverage data is easy to use and can quickly highlight possible problem areas. By annotating the coverage data to the HDL source code, it is easy to determine if a line of code has or has not been executed. This is typically enough to identify a hole in the test suite.

Branch coverage assures that every branch alternative in an executable has been exercised at least once. Condition coverage is an extension of branch coverage. It verifies that all possible permutations in any one function have been executed. State and path coverage offer the next layer of coverage. However, making use of all this coverage information is increasingly expensive with decreasing return value.

The most concrete benefit for using coverage data is that it prevents untested pieces of design from being included in the chip. One way to gauge the effectiveness of a test suite is to look at the amount of code coverage it delivers. Using coverage data to highlight unexecuted code forces a designer to examine the test suite and improve the effectiveness of the testing.

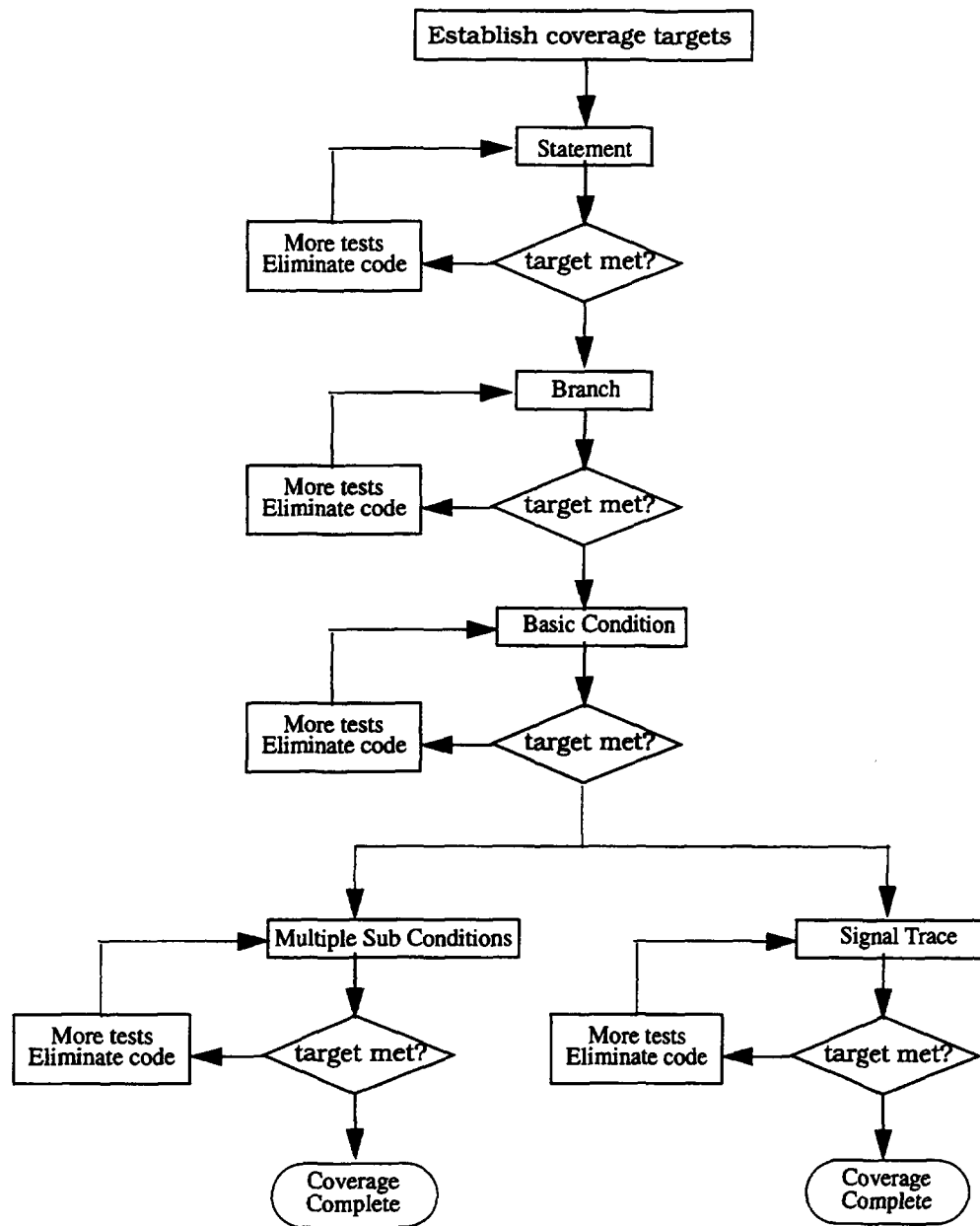
There are more subtle benefits to using coverage data. For instance, designers using a coverage tool can spend time analyzing whether the code that has not been executed is even necessary. After all, one way of dealing with unexecuted code is to remove it! In addition, the coverage analysis provides an early selection of tests for production test development. The high code coverage tests are an excellent basis for fault grading and burn in test development.

The primary danger of using code coverage is to equate 100% code coverage with full validation. First, total coverage cannot be equated with bug free code because the coverage analysis will not identify code which should have been included. Code coverage is not a formal validation tool. Code coverage is a quality indicator of the test suite. Low coverage implies low test quality, high coverage implies only that the code written has been exercised. The second reason total coverage cannot be equated with bug free code is even if the code is exercised, the coverage analysis does not guarantee correctness. The designer must ensure the test was performed properly.

## **6.0 Methodology**

Even though the coverage information will not ensure complete functionality and the amount of information may be tremendous, using coverage data has value and can be a major step towards delivering reliable products to customers. In addition, coverage analysis can be used as a progress indicator for the test development and design. The indicator may be used for both progress tracking and to identify areas of improvement.

A flow chart showing our proposed methodology is shown on next page. The first step is setting the target coverage. The goal of a design team should be 100% analyzed coverage. It is acceptable to have code in the design which has not been exercised. The reasons may include the code is not finished, the test suite is not completed, or the test condition is too difficult to generate in the simulation environment. In addition, the code may not be relevant. For example, error checking code or 'when others' statements may not be important to get 100% measured coverage. When setting this criteria, a designer must determine the balance between effort and quality. The effort includes the writing the test, running the simulations, and analyzing the results. The desired result is to discover issues as soon as possible.



Flow Chart of Proposed Coverage Methodology

Based on this usage model, the designer may choose to set the goal of 90% measured branch, statement, and condition coverage for a wide range of model variations. Achieving 90% measured coverage in these three areas will provide a high level of coverage with an acceptable amount of effort. Achieving the last 10% coverage may require a significant amount of effort. By covering many variations, the designer hopes to identify many problems quickly, then spend focussed efforts on achieving the design release criteria.

Upon determining the coverage criteria, the coverage should begin by evaluating the most basic coverage. This allows a quick screening of the test suite. The amount of information is relatively

small and the designer may easily determine gross test holes. The first level of testing should also have a percentage of coverage in the criteria. It is a good practice to ensure high coverage at this point prior to analyzing more coverage types.

As statement and branch coverage reach the desired goal, the designer should begin to analyze the more detailed conditional coverage. This process can become very resource consuming. As such, the designer must balance the degree of evaluation against resources of people, computers, and time. The resources may be managed by allowing the coverage analysis to identify redundant test suites. Tests offering no additional conditional coverage may be candidates for elimination from test suite.

### 7.0 Results of Using Code Coverage

Various code coverage approaches have been used on Intel projects. Recently, the tool developed by TransEDA was evaluated for collecting code coverage data. Since our validation methodology is robust and multi-tiered, the usage of code coverage was as a final indicator of test quality. Code coverage was generated near the completion of the design only as an analysis.

The lead designs in evaluating this new tool and methodology used statement, branch, and basic condition coverage. The coverage did not include all tests available primarily due to high simulation overhead. These experiments were mainly intended to comprehend the methodology and develop plans to incorporate it into our design flow.

**Table 1: Code Coverage Results**

Project	Gate Count	Statement	Branch	Basic Condition
1	8.5K	94%	90%	92%
2	48K	93%	90%	92%
3	115K	96%	96%	DNA
4	50K	97%	96%	DNA
5	32.4K	93%	93%	DNA

DNA = Did Not Analyze

The time to debug and develop new test cases exponentially increases with model complexity. Therefore, designers use a divide and conquer strategy by partitioning a design into smaller blocks. Code coverage can then be used on the smaller blocks to reduce time and effort. These models can achieve good coverage with minimal throughput time of test writing, simulations, and coverage analysis. In future designs, we plan to use code coverage early in the design cycle. Coverage goals will be set for each level of model development.

### 8.0 Future Enhancements

During our use of the coverage tools, we encountered a few unanticipated issues. The issues came from combining the process of developing a hardware model and with a software code coverage tool. In following sections we present ideas on how this tool can be enhanced to be more

useful to the designers.

One case was the use of components in a VHDL a model. A designer may write one model then use the same model in multiple applications. For example, a counter may be a component which is then instantiated multiple times. In one instance the counter may be a wait state timer, in another instance the counter may be a buffer monitor. Currently, the coverage reports the combined coverage of all instantiations on one model. This means the coverage is not separated by instantiation. Since the coverage is reported based on the model or component, the designer will have a false sense of security in test quality. The preferred coverage method would be to analyze and report each instance of a model separately.

Another area for improvement is the ability to selectively eliminate code from the coverage analysis. Quite often, designers will include VHDL in a model for testing purposes only. This code is not intended to be included in the ASIC, but offers good information during development. If the code is analyzed for coverage, the coverage may be low. The designer must then post process the reports to eliminate extraneous information. In addition, the post processing can be cumbersome as code is modified. A better approach would be to have a mechanism to ignore specified code in order to reduce the amount of information in the analysis.

Finally, the major issue with coverage analysis is the simulation runtime. Even though the implementation of the tools vary, the overhead is three to five times a simulation without coverage analysis. This overhead severely limits the throughput time of simulation and designer analysis of the reports. In many cases, the designer has already moved on to a new model or more tests have been written which cover the untested code. By the time the coverage data is available, it is no longer relevant. A more suitable overhead would be 1.5 to 2 times the non-coverage analysis. This used in conjunction with information reduction would provide a useful analysis in a timely manner.

## **9.0 Conclusions**

Code coverage identifies code that has not been executed. This information is very useful as it can pin point areas where bugs may exist and assists in continual improvement of testing. The danger is associating 100% coverage with 100% functionally correct design. This is not the case. In fact, the goal should be 100% analyzed coverage rather than 100% measured coverage. Analyzed coverage allows for software that has not been executed provided there are explanations as to why that is the case. There are good reasons why having unexecuted code is fine. For instance, some error cases may be too difficult and time consuming to try to recreate in a test case. The rule of diminishing returns can play a factor.

## **10.0 Acknowledgment**

The authors would like to thank PCD employees who ran this tool and the TransEda application engineers who helped with this tool.

## **11.0 References**

1. S. Tayal, A. Moezzi, and E. Magnusson, "System Level Verification of ASIC Chipsets", 6th Annual IEEE International ASIC Conference, 1993, pp 283-287.
2. Vantage Reference Manual, Viewlogic Corp.
3. VHDLcover Users Manual, TransEDA Corp.