PLATFORM APPLICATION NOTE

cādence

Using Coverage

GUIDING VERIFICATION TO EFFICIENT COMPLETION

February 2003

TABLE OF CONTENTS

CADENCE Incisive Verification Platform 1

Application Note Overview 1

1 Coverage Goals 2

2 Types of Coverage 2

3 Information Collection 4

4 Coverage Metrics 6

5 Exploration 7

6 Conclusion 8

Appendix A: Transactions 9

Appendix B: Transaction Exploration Searches in Tcl 10

Appendix C: Code Coverage 12

CADENCE INCISIVE VERIFICATION PLATFORM

Verifying today's complex ICs requires the speed and efficiency that can only be provided in a unified verification methodology. The Cadence Incisive verification platform enables the development of a unified methodology from system design to system design-in for all design domains. A unified verification methodology consists of many different tools, technologies and processes all working together in a common environment. The Incisive platform provides the tools, technologies, a common user environment and the support needed to develop a unified methodology. This application note details specific topics for using the tools and technologies in the Incisive platform to help create a unified methodology to verify your design.

APPLICATION NOTE OVERVIEW

Verification is defined as a confirmation of truth or authority and the evidence for such a confirmation[1]. Any verification process is incomplete without evidence and coverage is a critical part of that evidence. Any useful piece of evidence must be captured, collected, correlated, and interpreted in order to determine what the evidence means. By performing these functions, you improve the effectiveness of your verification process and increase your efficiency. The Incisive platform provides various ways of performing these functions.

Coverage is defined as the extent or degree to which something is observed, analyzed, and reported[2]. Before you can observe something, you must look for, look at, or capture whatever it is you want to observe. In verification, you define what needs capturing in the test plan for the design. That definition describes the specific information you need to collect, how that information relates to the conclusions or results you want to obtain, and how you want to report your conclusions and results.

This application note describes the different types of coverage and how, in each type, information is captured, collected, correlated, and interpreted. This application note also explains how the Incisive platform facilitates each of these tasks and produces a timely, efficient, and thorough verification process.

[1] *The American Heritage Dictionary of the English Language*, 4th ed., s.v. “verification”
[2] *The American Heritage Dictionary of the English Language*, 4th ed., s.v. “coverage”

1 COVERAGE GOALS

The primary goal of any complex integrated circuit (IC) verification process is to determine whether a design meets its specification; its secondary goal is to accomplish the primary goal efficiently. To accomplish these goals, you must have a comprehensive test plan that is based on the design's specifications. This test plan drives the capture, collection, correlation, and interpretation of the coverage information.

There are two types of coverage: functional coverage and code coverage. Each type can be broken down into smaller categories, each with its own interpretation as to how well the design has been exercised. In typical methodologies you would

1. Write the test plan.
2. Write the tests.
3. Perform functional coverage to cover everything specified in the test plan.
4. Perform code coverage to identify any redundant or overlooked items.
5. Update the test plan to include any new/untouched scenarios or remove any redundancy.
6. Iterate back through the functional coverage.

At the highest levels, some of this coverage information is valuable in accomplishing the primary goal. However, the majority of the coverage information is focused on the secondary goal. With improved coverage information, you can improve your results and the efficiency of your verification process. As this efficiency grows, you have more time to spend on the most important part of verification – exploring functions and scenarios that were not initially considered in the test plan. These unconsidered scenarios can unmask hard-to-find bugs. You can also find most of these unconsidered scenarios by analyzing the existing coverage information.

This application note explores

- The different types of coverage
- How information is captured, collected, correlated, and interpreted – in general and in the context of the Incisive platform
- Analyzing coverage information to uncover hidden bugs

2 TYPES OF COVERAGE

There are two types of coverage: functional coverage and code coverage. This section discusses the two types of coverage and how each type can be divided into more specific categories.

2.1 FUNCTIONAL COVERAGE

Functional coverage is generally complex and difficult to define; however, it is the starting point in answering the question, "Does this design meet its specification?" Functional coverage brings the specification, the test plan, and the actual tests together by providing information about which functions have or have not been exercised in the design.

There are three types of functional-coverage information.

- Application coverage
- Interface coverage
- Structural coverage

2.1.1 Application Coverage

Application coverage is the highest level of functional-coverage information. Generally, you define application needs when you define your architecture and develop your functional virtual prototype (FVP)[3]. For more information on FVPs, see the *Creating a Functional Virtual Prototype* application note.

You need specific information about the typical application of the design in order to determine the kind of information to capture at this level. The test plan should include the application features that will trigger the capture of this coverage information; these features can be internal and/or external to the ASIC or SoC. The application features can involve multiple interactions across multiple interfaces, sequentially or concurrently. Because this type of coverage information often encompasses the entire design, or at least large portions of the design, it is typically captured at the subsystem, system, and integration levels of verification. Some examples might include a cache hit, an automatic retry, a resend of a bad packet, or initiating functions on peripheral devices.

[3] An FVP is a golden functional representation of the whole DUV and its testbench.

2.1.2 Interface Coverage

Interface coverage is the most common form of functional coverage. You define your needs for interface coverage and implementation at the same time and these needs define the interfaces that will exist in the design; this is a part of the transaction taxonomy. The transaction taxonomy identifies the types and sequences of stimulus and responses for each interface.

Since interfaces are both internal and external to the ASIC or SoC, capturing interface coverage information occurs at the block, subsystem, system, and integration levels of verification. Internal-interface information is captured at the block and subsystem levels; external-interface information is captured at the system and integration levels. Examples include the various types of reads and writes across a bus, such as the peripheral component interconnect (PCI) bus, the advanced micro-controller bus architecture (AMBA) bus, or sending and receiving frames across an Ethernet port.

2.1.3 Structural Coverage

Structural coverage relates closely to the implementation of a design. You define your structural-coverage needs in the micro architecture of particular blocks of the design. Information is captured after block-level testing because of performance cost and because you cannot obtain useful information until the block functions correctly. The following are examples of design elements that capture structural coverage information.

- Finite state machines (FSMs)
- First-in-first-outs (FIFOs)
- Memory elements
- Arbiters
- Handshaking on an interface

Some structural coverage can be done by code coverage, namely state machines.

2.2 CODE COVERAGE

Code coverage is widely-used because, unlike functional coverage, code coverage is easy to use, automatically captured, and essentially predefined. However, code coverage does not sufficiently answer the question, “Does this design meet its specification?”

Code coverage provides valuable information; it tells you what code in your design has been exercised. Without this information, portions of the design might not be exercised at all. However, just knowing design code has been exercised is not sufficient to determine if that code was exercised properly. It could be said that some of the higher forms of code coverage (such as, expression, branch, path, and state machine coverage) can give you more information about your code, but it is dangerous to make this assumption. Using these higher forms of code coverage could lead to false conclusions that can cost large amounts of time and money. Code coverage information *is* very valuable, but it must be utilized appropriately.

Code coverage includes

- Statement or line coverage
- Expression coverage
- State machine coverage

2.2.1 Statement or Line Coverage

Statement or line coverage tells you whether or not a line of code was exercised.

2.2.2 Expression Coverage

Expression coverage provides information about code that involves expressions, typically Boolean expressions. Expression coverage not only tells you whether a line of code was exercised, but it also tells you whether it was exercised in every way possible. For example, an `if` statement might be true in more than one way. Expression coverage indicates whether all the possible combinations of conditions were exercised. Branch coverage, which is an extension of expression coverage, shows which branches of an `if` statement or `case` statement were exercised. Path coverage is similar to branch coverage; it shows the exercised paths through a sequential `if` statement or `case` statement.

2.2.3 State Machine Coverage

State machine coverage provides information on the visited transitions, arcs, or states in a finite state machine. This coverage can also provide the actual path taken through the state machine. You can view this additional information as an extension of path coverage (branch coverage), as explained above.

3 INFORMATION COLLECTION

You must collect functional and code-coverage information before it can be useful. The code-coverage information you collect must be comprehensive, depending on the type of code coverage tool you use. However, this is not the always the case for functional coverage. For functional coverage, you only need to collect the information that is useful in the verification process. The test plan helps you determine which information is useful, but it is not always sufficient. The test plan can only cover the areas that the verification team identifies; unfortunately, the worst bugs usually involve the areas that were not considered. For more information on dealing with hidden bugs, see Section 5, “Exploration.”

You should only collect functional and code-coverage information *after* you have a working test; coverage information has little value without a working test. You can use some of the same functional-coverage information during debug, but the code-coverage information is useless if your test is failing. Most often, you only need to collect coverage information once for a given test; you should not collect coverage information during regression runs because of its performance impact.

This section discusses collecting information efficiently and quickly so that there is time left in the verification process to explore hidden bugs.

3.1 FUNCTIONAL COVERAGE

Functional coverage information is collected at all levels of verification. Functional-coverage information typically takes two forms: run-time information and post-processing information.

Run-time information is calculated information; it includes statistics or ratios about the functions that occurred during the verification. This could be as simple as the total number of times a particular function was exercised or as complex as a latency calculation on an ATM cell passing through a switch. In any case, this kind of information only gives you the result; you do not learn about the supporting components or how the results were calculated. However, this information is available immediately and does not require further analysis.

Post-processing information is raw information and needs to be analyzed. With this kind of information it is difficult to identify what kind of raw information to store and how much to store. The two most common forms of this information are: text/ASCII files, such as a log file, and simulation-waveform databases. Text files must be processed and analyzed. Simulation-waveform databases are often very large and can cause storage space and simulation performance issues. Each event on each waveform being stored must have its information put into the database. The file I/O overhead can be quite large. In some instances it is more efficient to run the simulation twice, the first time without capturing waveform information and the second time capturing only necessary waveform information; this is better than running the simulation once, but having to store all of the waveform information.

In order to use post-processing information efficiently, it cannot just be raw data in a file or waveform information in a database. You must have a different way of storing the information in the verification process. The Incisive platform offers a unique way of storing this information that decreases the impact of the issues discussed previously. The Incisive platform stores this information as transaction[4] information in a waveform database. Transaction information in a waveform database requires less space than typical signal information with all of its transitions. Since in the data is in a database and not an ASCII file, it is easier to extract and analyze.

For information on how the Incisive platform extracts and analyzes the post-simulation information, see Section 4, “Coverage Metrics.”

Transaction information can be generated from multiple sources in the Incisive platform, including hardware description language (HDL) testbenches, SystemC testbenches, and assertions. For more information on these sources, see Appendix A, “Transactions.”

Figure 1 shows how various types of functional-coverage information is collected within the FVP.

[4] A transaction is a single, conceptual transfer of high-level data or control. Transactions are defined by their start time and end time; all of the information associated with the transaction are stored as attributes.

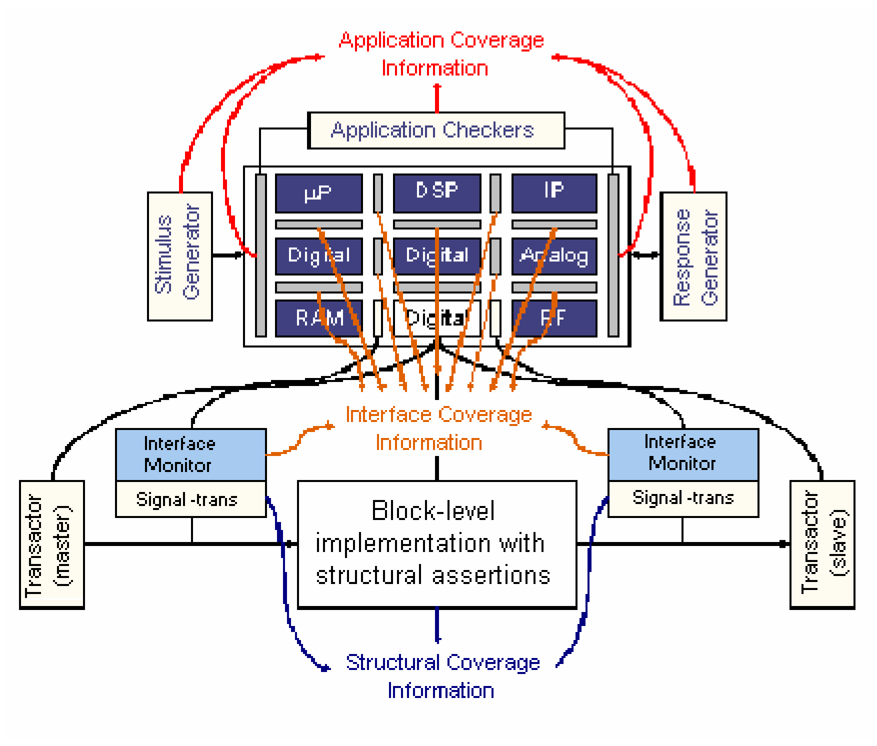

Figure 1 Functional-coverage Information Collection

3.1.1 Application Coverage

For application coverage, you need to collect run-time and post-processing information. Self-checking testbenches and application-level assertions can collect run-time information. Testbenches record this information in log files. However, run-time information that is generated by testbenches or assertions should also be recorded as transaction information in the waveform database that the Incisive platform uses. The flexibility of the transactions makes this possible. Although transactions are most commonly used at the interface level, they are also relevant at other levels of abstraction — like the application level. When transaction information is available after the simulation, you have access to the results of any checking that was done during run time and the information that went into those results.

3.1.2 Interface Coverage

For interface coverage, you need to collect both run-time and post-processing information. As with application coverage, testbenches and interface-level assertions can collect run-time information. However, at this level, transaction information is extremely valuable and should be used extensively. Generally, interface operations map to transactions very well; this enhances and reduces debug. Similar to application coverage, transactions for interface coverage are generated by both the testbenches and the assertions.

3.1.3 Structural Coverage

Structural coverage is closely related to the actual implementation; it lends itself more to run-time information. Known implementations confine the possible interactions, failures, and intended behavior to a much smaller space. For structural coverage, assertions provide most of the information. Transactions, which were generated by assertions, might also be used at this level of coverage.

3.2 CODE COVERAGE

Code-coverage information is collected based on instrumentation of the design netlist. This instrumentation, which is typically performed automatically by the code coverage technology, facilitates the collection of code-coverage information. The Incisive platform's code-coverage technology allows for flexibility within the information-collecting process. You can choose to

- Collect information from only certain parts of the netlist, based on the flexibility and capability of the code coverage tool.
- Or, you can collect all the information and then filter the result.

However, you should consider the performance impact of capturing redundant information. Code coverage can have anywhere from a small (10% or less) to a significant (50% or more) impact on simulation run time.

4 COVERAGE METRICS

This section discusses the correlation and interpretation of the coverage information. Now that you have captured and collected all of the necessary coverage information, you need to understand what the information means. This understanding involves two areas: correlation and interpretation.

Correlation aids the efficiency of the verification process by determining how the coverage information relates to the task at hand. Correlation is done for each individual test and across all tests. If duplicate or unnecessary information is present, it might be possible shorten overall verification time by eliminating some tests or eliminating the generation of information. Merging coverage information across the entire test suite provides the necessary global picture of the verification process.

Once you have determined the relevancy of the information, you must analyze and interpret that information into metrics that detail the progress of the verification process and identify areas that still need work. The results of this analysis can be presented in many forms. It might be as simple as a number that represents a count of a particular operation or as complex as a percentage of the total set of operations that have been tested. This primarily monitors how much work has been done and how much is still left; it does not detail how well the work has been done. Coverage metrics, including functional coverage and code coverage, are primarily quantity measurements — not quality measurements. Functionality might have been tested, but that does not mean that functionality actually operated correctly. However, due to the transaction information obtained by the Incisive platform, you can determine some quality measures of functional coverage.

4.1 FUNCTIONAL COVERAGE

The complex nature of today's ASICs and SoCs require directed-random testing. There are too many possibilities to explore and not enough time to do so; a design can become obsolete before you can test every possible scenario. With directed-random testing, you can test many functions and scenarios without the overhead required to guarantee that every possible one has been tested. Because it is not feasible to cover everything, it is critical to ensure that coverage information has been collected for the important functions and scenarios stated in the test plan. The better the information correlates to the functions and scenarios identified for testing, the easier it will be to interpret whether progress is being made.

4.1.1 Application Coverage

Application-coverage metrics are the closest in not only determining whether functionality was exercised, but whether it was exercised correctly. These metrics are at a high enough level that some of the functionality being tested must have operated correctly to have operated at all. These metrics are often derived from a collection of run time information, transactions, and sequences of transactions from various levels. This collection determines the high-level applications that occurred. This level might have less metrics than the interface level, but formed conclusions are more directly tied to the overall goals of the design. Underlying pieces that have been tested can operate properly individually, but can cause issues when joined together; these issues will show up first in the application-coverage metrics.

4.1.2 Interface Coverage

Interface-coverage metrics relate closely to transactions and sequences of transactions. Interface-coverage metrics can help answer questions like:

- Was every memory location written to and then read from?
- Did each port in the switch have a packet sent through it?
- Did the PCI bus interface have each combination of a transaction followed by another transaction?

When an interface is not sufficiently exercised, based on the goals of the test plan, it needs more testing. This might entail adjusting some constraints on existing directed-random tests; or, it might require new tests in any of the three main realms: directed, random, or directed random.

4.1.3 Structural Coverage

Structural-level metrics are primarily based on events. Structural-coverage metrics can help answer questions like:

- Was the state visited?
- Did the FIFO fill?
- Was the bus request acknowledged?
- Was the correct input selected on the multiplexer?

Structural-level metrics are usually specific requirements rather than qualitative measurements. You can easily interpret these numbers to determine areas that need more testing. For example, if a FIFO was filled only once, it might be good to fill it a few more times for more confidence in its robustness.

4.2 CODE COVERAGE

Code-coverage metrics are very straightforward and are valuable in determining what areas of the design have been neglected in the verification process. These metrics identify areas where adjustments need to be made so that the entire design is exercised. However, these are the only conclusions you can make from this information.

5 EXPLORATION

This section discusses exploring areas in a design that were not considered in the test plan. The complex nature and size of today's ASIC and SoC designs makes verification of all possible scenarios all but impossible. Therefore, verification processes must be efficient and must cover as many scenarios as possible.

When you have transaction information available, as discussed in Section 3, "Information Collection," you can explore functional features that were not defined in the functional coverage information. Transaction information also allows you to identify whether these new features were exercised and to interact with these features; this helps recognize areas that need further analysis to determine proper functionality.

Exploration usually takes place at the application and interface levels of functional coverage because these areas have a large number of possible scenarios. Code coverage and the structural level of functional coverage do not require exploration because these areas are conclusive.

For example, if you have a network switch with four input ports and four output ports, you would typically

1. Perform structural testing by making sure all the basic structural elements of each block that make up the switch are operating correctly.
2. Perform functional testing by making sure each port operates correctly when both good and bad cells are presented.
3. Perform application testing by mimicking a real-world scenario and randomly sending multiple frames or cells to each port.

You might find and fix bugs through the structural and interface testing, but find a new issue at the application level. For example, when the switch is stressed, frames or cells get dropped. This problem is difficult to resolve because you know each element *works*, but is failing when everything is combined. You know you need to perform more exploration when your coverage information says that all the important functions have been covered, but you still receive errors at the application level. There might be scenarios that were not identified by the verification team; or, there is a structural problem that works fine under light loads, but cannot handle heavy loads (such as a FIFO that is not big enough). Due to time constraints, it is difficult and unreasonable to test all these scenarios in order to find that one exceptional scenario. However, if you don't find the scenario, it can cost you more time if problems arise in the manufactured design.

Exploration takes the information collected from all the tests, each guided by the test plan to determine a specific area of functionality, and uses that information to explore new scenarios that show up. These new scenarios can be as important as others for various reasons.

- It might be a new interaction that no one has experienced before. Thus, the verification team might not even recognize it as a problem.
- Similar scenarios might have been seen in the past, but added has caused something to be a little different.

In either case, it is difficult identify areas that you have never considered before and exploration can bring these scenarios forward. Run-time information has no supporting information and is not useful for exploration. However, the transaction information collected in the Incisive platform's waveform database is useful for exploration. These transactions might not have all of their supporting information either, but they have enough information to give a fairly accurate picture of how the design was functioning throughout the test.

In the example above, exploration identified the issue of dropped frames or cells. Without this capability and the transaction information to analyze, this issue might not have been found. Without exploration, the only thing that might have been identified would be situations where the network using this switch gets busy and the switch keeps asking for frames or cells to be present because it dropped them. The performance of the switch would have degraded in a high traffic situation, possibly compromising the specified quality of service.

The Incisive platform makes exploration possible by using the transaction and signal information in the simulation waveform database. The Incisive platform performs this exploration using searches written in a superset of Tcl to query the database and extract results (see Appendix B, "Transaction Exploration Searches in Tcl"). The search might be as simple as counting the number of a particular type of transaction or as complex as determining the latency of a particular sequence of transactions as it progressed through the design. Exploration is instrumental in uncovering scenarios that might have serious implications if not found and corrected.

6 CONCLUSION

Coverage plays a critical role in any verification process. Without exploration

- Holes in the test suite can go unnoticed and cause major errors in the design.
- Redundancy in the test suite, which can weaken performance and waste time, might be overlooked.

Coverage is effective in improving the efficiency and speed of the verification process; the Incisive platform is vital in improving the coverage process. SystemC brings the full power of C++ to the verification space and enables coverage to be collected in multiple forms, from calculated information to transactions at either the application or interface level. Transactions provide abstraction, where coverage is collected easily and in an understandable form, and the intuitive correlation and interpretation of coverage information.

Coverage, along with assertions and the transactions generated by those assertions, provides the broadness needed to encompass the entire design or any subset of that design. Exploration of coverage information seeks out the elusive, unknown bug. And finally, all the necessary code coverage features are provided to ensure nothing is missed.

APPENDIX A: TRANSACTIONS

In the Incisive platform, transactions are created using the Simulation Database Interface (SDI). There are several forms of this interface, including an HDL version and a C++ version. You can also create transactions from sequential assertions; all this information is stored in the simulation waveform database.

The following sections point to examples of executable code that show the transaction capabilities of the Incisive platform.

HDL TRANSACTIONS

The SDI interface for HDL is a collection of functions that utilize PLI to extract information from the HDL simulation and put that information into the database as transactions.

NC-Verilog examples are available in the installation tree at:

```
<install_dir>/tools/transrecord/sdi-verilog/examples
```

NC-VHDL examples are available in the installation tree at:

```
<install_dir>/tools/transrecord/sdi_ncvhdl/examples
```

For other examples or more information on HDL transaction recording, you can contact any Incisive platform application engineer (AE) or refer to other Incisive platform documentation. There are some command line options necessary for SDI recording with HDL. Both the examples and the documentation specify the use of these options.

SYSTEMC TRANSACTIONS

The SDI interface for SystemC (C++) is a collection of classes that allow all the capability of the HDL PLI functions and can extend these capabilities to allow more flexibility.

For examples of transaction recording using SystemC, go to the TestBuilder website at http://www.testbuilder.net. For downloadable SystemC verification extensions, go to the TestBuilder website or the SystemC website at http://www.systemc.org. For other examples or more information on transaction recording, you can contact any Incisive platform AE or refer to other Incisive platform or SystemC documentation.

ASSERTION TRANSACTIONS

Transaction information from assertions is generated by specific options to the probe command. Specifically, when creating a probe on an assertion or set of assertions (by using the –assertion option to the probe command), transactions from sequential assertions in the set are produced by also adding the –transaction option to the probe command. The documentation of the probe command gives the specific details.

A tutorial on assertions is available from SourceLink, http://sourcelink.cadence.com. See the *Using Assertion*s application note and the Incisive platform documentation for more assertion details.

APPENDIX B: TRANSACTION EXPLORATION SEARCHES IN TCL

Transaction exploration searches are written in a superset of the Tcl language. The power and flexibility of this language provides the capability to do anything from simple searches to complex searches.

For more details on using transaction exploration and writing searches, see the Incisive platform documentation. More examples are available in the installation tree at `<install_dir>/tools/txe/examples`. The transaction exploration engineering notebook is also available in the installation tree; from a browser, go to

```
<install_dir>/doc/txeEngrNtbk/index.html
```

SIMPLE SEARCH

This is an example of how to count transactions; this example is from the transaction exploration engineering notebook.

```
apply {
     fiber top.processor {
       trans_type * {
             accept
       }
     }
}
table {
     col instruction [trans_type]
     col items [items]
     sort instructions
}
```

This search produces the following table.

instruction	items
add	86
brz	117
diag	16
eq	36
flush	1
halt	1
incr	62
jp	70
load	36
lt	81
mul	8
setl	84
store	42

CROSS-PRODUCT SEARCH

This example is from the transaction exploration engineering notebook.

```
init {
     set registers {r0 r1 r2 r3 r4 r5 r6 r7}
     cross src1 $registers
     cross src2 $registers
}
apply {
     fiber top.processor {
        trans_type * {
               cross src1 [attribute src1]
               cross src2 [attribute src2]
               cross -increment
        }
     }
}
table {
     cross -row src1 -col src2 -goal 30 -dontcare {$src1 == "O/B" || $src2 == "O/B"}
}
```

This search produces the following table.

	src2	r0	r1	r2	r3	r4	r5	r6	r7
src1									
r0		0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*
r1		0/30*	0/30*	54/30	16/30*	0/30*	0/30*	0/30*	0/30*
r2		57/30	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*
r3		24/30*	0/30*	32/30	0/30*	0/30*	0/30*	0/30*	0/30*
r4		0/30*	0/30*	0/30*	0/30*	0/30*	36/30	0/30*	0/30*
r5		0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	34/30	0/30*
r6		0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*
r7		0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*	0/30*

APPENDIX C: CODE COVERAGE

The Incisive platform provides all the necessary technology for code coverage collection and analysis; its GUI offers the following features.

- Graphically-sorted summary, which helps you quickly identify untested blocks within the design.
- Color-coded and user-definable FSM viewer, which helps you quickly identify untested states and arcs within the design.
- Color-coded and synchronized HDL-source and coverage reports, which gives you a quick coverage-model view of what has and has not been exercised.

For more information on code coverage and its use, see the Incisive platform documentation.