Computer Design Technology Insight

Controlling Costs through HDL Behavioral Verification Strategies

By Dr. Martin Abrahams

Dr. Martin Abrahams is applications support manager for TransEDA, Ltd., based in Eastleigh, Hampshire, England, the first company to develop hardware code coverage tools for CHDL and Verilog HDLs. Abarhams has more than 14 years experience in ASIC CAD tool development and support, and assisted with development of TransEDA's VHDLcover and Coverplus. He holds a degree in Maths and Physics from Bristol University and a PhD in Maths from Southampton University.

When it comes to ASIC design and manufacturing, the trick has always been to correct design problems before committing to silicon. That's why design verification can be the most important aspect of ASIC design, and hardware testing is more significant that software testing in order to eliminate the expense of having to re-spin the ASIC. However, functional verification is no longer sufficient. Code coverage is adding new capabilities to IC design, highlighting design flaws that are not covered by conventional functional verification.

If you apply the findings of conventional software design studies to ASIC engineering using hardware description language (VHDL or Verilog) code, you can expect to see one to three errors per hundred lines of code. Isolating these errors accounts for fifty percent of a project's labor costs. To effectively control costs and sorten time to market, design verification, including code coverage, should occur throughout the design flow, at various levels of integration. In order to control costs, the trick is knowing the appropriate verification strategies to deploy, when to verify for hardware and when to verify for software, and where different verification strategies can be deployed most economically within the design flow.

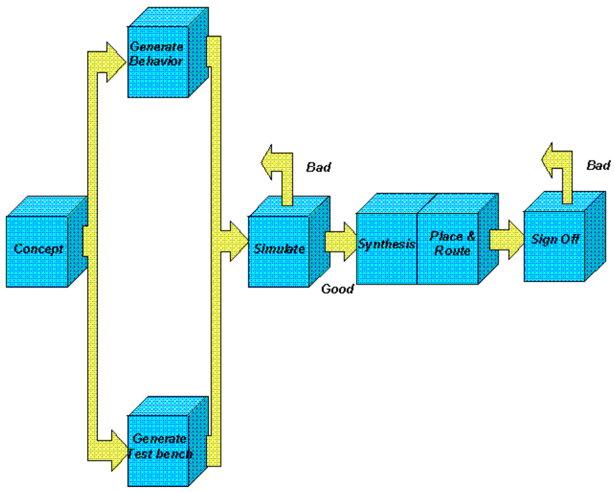
Consider that if you take data gathered from software design studies and apply it to ASIC engineering with a hardware description language (VHDL or Verilog) code, you can expect to see one to three errors per hundred lines of code. Finding these errors accounts for approximately fifty percent of project labor costs, and slows time to market. Verification should occur throughout the design flow, at various levels of integration. In order to control costs, the trick is knowing the appropriate verification strategies to deploy, when to verify for hardware and when to verify for software, and where different verification strategies can be deployed most economically within the design flow. This article will present an ASIC design verification strategy based on module/unit testing, integration testing, system testing, and regression testing, and will describe the four best test approaches that can be used in the design flow – functional, structural, error-oriented and stress/performance.

If you were to apply conventional software studies, such as tiose conventional software studies [1,2] were applied to ASIC designs using VHDL or Verilog [3] code, we could expect about one to three errors per hundred lines of code. Finding errors accounts for approximately fifty percent of project labor costs. Available verification approaches are testing, design reviews/code inspections, prototype/emulation, requirements tracing, and formal verification. Verification occurs throughout the design flow and should occur at various levels of integration. This paper discusses a strategy of module/unit testing, integration testing, system testing, and regression testing. It describes four test approaches (functional, structural, error-oriented and stress/performance).

Design flow

The first stage of an ASIC project is to write a detailed specification document that becomes the basis for design and verification work. The next stages are capture of the ASIC design in Verilog or VHDL as register transfer level code (RTL code), and development of tests to verify the design. Figure 1 shows how the activities fit together into a design flow.

Figure 1: A typical design flow



The RTL code model is the master for all further design and verification work. The RTL code is the input to the synthesis program which implements the design as a gate level netlist. Simulation of the RTL code produces "golden results" against which all gate level simulations will be compared.

Therefore, demonstrating the RTL code is an accurate representation of the requirements of the specification document is essential. Most of the verification effort is put into this activity.

A typical design

A recent project by one of TransEDA's customers was a LAN ASIC, it was typical in size and complexity to many modern ASICs. The design size was 800k gates. There was over 120k lines of source code, contained in 277 VHDL source files. Verification required 650 system simulation test benches, representing over 27 million clock cycles. The company applied the principles of a high volume of system testing of the VHDL code using Mentor Graphics and Synopsys simulators and also used an ASIC emulation system from Quickturn. TransEDA's VHDL code coverage product is used to measure how much of the design model has been simulated.

Applying the software statistic of approximately 2 errors per hundred lines of code, 120k lines of code might contain 2400 errors, so a significant verification effort was required.

Verification Approaches

Verification activities occur throughout the evolution of the product.

There are numerous techniques and tools that may be used in isolation or in combination with each other. In general there are five broad categories that reflects the way most of the verification approaches are described.

Design Reviews include techniques such as walk-through, inspections, and audits. Most of these approaches involved a group meeting to assess a working product.

Testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Formal verification [4] is a collection of techniques that apply the formality and rigor of mathematics to the task of proving the consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution.

Requirements tracing is a technique for insuring that the product, as well as the testing of the product addresses each of its requirements. The usual approach to performing requirements tracing uses matrices.

One type of matrix maps requirements to modules. Construction and analysis of this matrix can help insure that all requirements are properly addressed.

System verification diagrams are another way of analyzing requirement/modules tracing.

One type of matrix maps requirements to test cases. Construction and analysis of this matrix can help insure that all requirements are properly addressed.

A third type of matrix maps requirements to their evaluation approach. The evaluation approaches may consist of various levels of testing and reviews. The requirements/evaluation matrix insures that all requirements will undergo some form of verification.

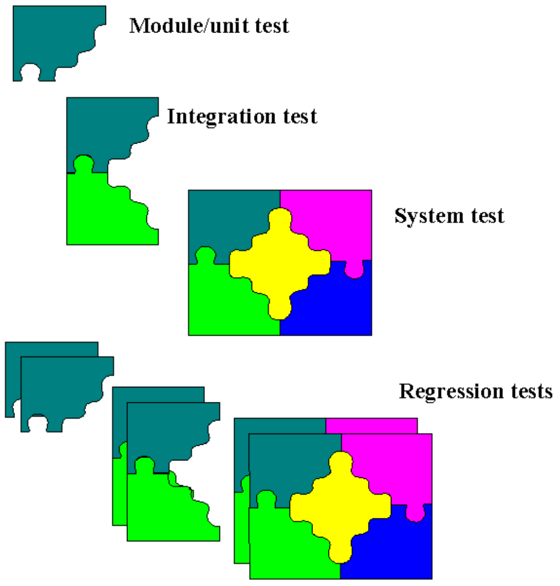
Emulation/Prototyping are techniques for analyzing the expected behavior. These are usually used to analyze requirements and specifications to insure that they reflect the user's needs. Since they are executable, they offer additional insight into the completeness and correctness of these documents.

Testing at each integration level

Testing should be performed at each stage of integration of the components during the bottom up construction of the ASIC design, as shown in Figure 2.

Module testing is the lowest level of testing and involves the testing of a module or unit. The goal of module-level testing is to insure that the component being tested conforms to its specifications and is ready to be integrated with other components of the product.

Figure 2: Levels of test



Integration testing consists of the systematic combination and execution of product components. Multiple levels of integration testing are possible with a combination of components at several different levels. The goal of integration testing is to insure that the interfaces between the components are correct and that the product components combine to execute the functionality correctly. System testing is the process of testing the integrated system to verify that it meets its specified requirements. Practical priorities must be established to complete this task effectively. One general priority is that system testing must concentrate more on system capabilities rather than component capabilities. This suggests that system tests concentrate on insuring the use and interaction of functions rather than testing the details of their implementations. Another priority is that testing typical situations is more important than testing special cases. Test cases should be constructed corresponding to high-probability user scenarios, so giving early detection of critical problems that would greatly disrupt the user.

System Tests should be developed and performed by a group independent from the code developers. System test plans must be developed and inspected in the same rigor as other project elements. Systems tests must be repeatable.

Regression testing can be defined as the process of executing previously defined tests cases on a modified code to assure that the changes have not adversely affected the previously existing functions. The error-prone nature of modifications demands that regression testing be performed. An important regression testing strategy is to place a higher priority on testing older capabilities than on testing the new capabilities provided by the modification. This insures that the capabilities that end-users depend on are still intact. This is especially important when you consider that some studies show that half of all failures detected by end-users after a modification were failures of old capabilities.

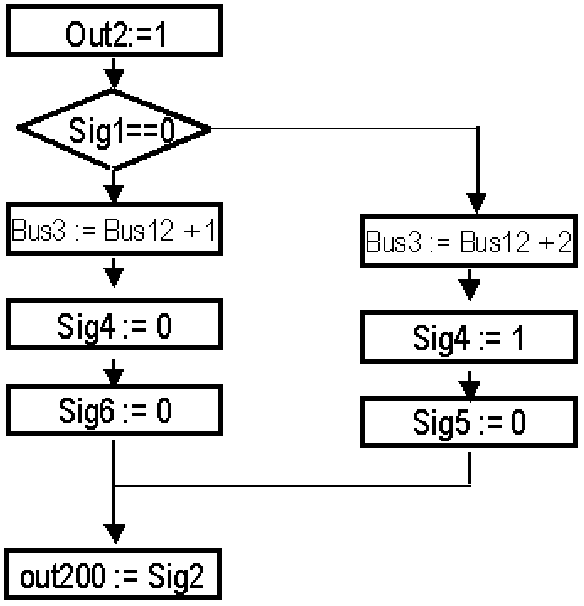
Possible Test Strategies

At each of the levels one or more different testing techniques will be applied as necessary to reach the overall testing goals.

Functional verification is the title for the most commonly used approach to ASIC RTL code testing. Functional verification compares the HDL code functionality with the desired functionality. This is black box testing. Functional verification is carried out by VHDL or Verilog simulation. Actual output vectors from the HDL model are compared with the desired output vectors during simulation. The software world acknowledges that no more than 60% of design errors can be isolated through black box testing.

Structural testing develops test data based on the implementation of the product. This testing occurs on the source code. Structural testing includes data flow anomaly detection, data flow coverage assessment and various levels of code coverage [5]. The goal of structural testing is to exercise each aspect of the implementation, as shown by an example in Figure 3. Structural testing is a form of white box testing where there is visibility of the internal construction of the design.

Figure 3: Structural testing



Error-Oriented testing develops test data by focusing on the presence or absence of errors in the coding process.

The goal of error-oriented testing is to verify that specific errors are not present in the implementation. This is a lessons-learned approach that looks for errors that have been experienced in the past.

Stress/Performance analysis involves analyzing the behavior of the system when its resources are saturated in order to assess whether or not the system will continue to satisfy its specifications. Performance analysis involves insuring that the product meets its specified performance objectives.

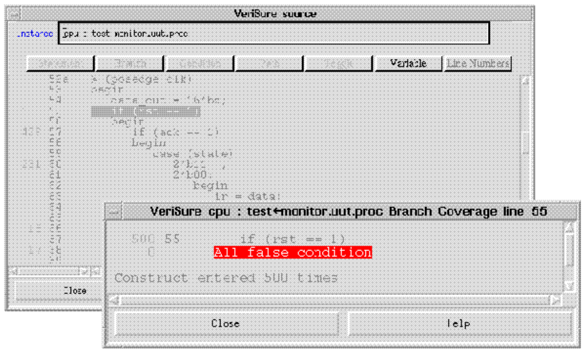
Measurement of test thoroughness

Designers and verification engineers need to know when enough simulation has been completed. They need to know which areas of a design have not been tested, so they can focus test development into these areas. To fulfil this need, HDL code coverage tools are used.

Code coverage is more important to hardware design than it is to software design because the cost of re-spinning the design in Silicon is much higher than re-spinning software.

Code coverage records what code lines, blocks, expressions have been executed during simulation. A display of the code coverage results is shown in Figure 4.

Figure 4: Code coverage results


VeriSure source
Instproc
Variable
Line Numbers
Close
VeriSure cpu : test←monitor.uut.proc Branch Coverage line 55
Close
Help

```
cpu : test monitor.uut.proc
```

```
@ (posedge clk)
begin
  data_out = 'b'bz;
  if (rst == 1)
  begin
    if (ack == 1)
    begin
      case (state)
        2'b11 :
        2'b00:
          begin
            ir = data;
```

```
500 55        if (rst == 1)
  0        All false condition

Construct entered 500 times
```

The unexecuted code has not been tested. Conversely, the code that has been executed has not necessarily been tested. The concerns are that the coverage tool may have recorded either executions of statements when the variables used had ‘X’ values (during initialization), or settling executions at a time point (glitches), or the code execution was not observable at an output [6]. HDL code coverage tools now address the first two points above.

Code coverage needs to be coupled with functional verification. It is necessary to show both that code was executed and correct output values occurred throughout simulation [7].

Typical design results

In the example presented earlier, the following strategy was used. Functional verification was used extensively at the system and regression testing stages. Code coverage was used to record statements and branches executed during functional verification simulations of all test cases. Combination of all 650 coverage results showed that 95% of statement and branches had been executed. 5% of the design code amounts to 6000 lines of source code, leaving 120 possible undetected errors.

Results interpretation

There are two possible interpretations of the coverage results. Code may be unexecuted because either (a) there was no test available for this code, or (b) this code was difficult to reach or unreachable. Because of (a), the verification team reviewed the specification and created additional tests. Circumstance (b) existed also and required new verification approaches. With such an extensive set of system tests, the remaining unexecuted code was difficult to reach with a system test.

Choices were available here, such as development of specific module tests, code inspection of untested code or formal verification of the module. This company chose the route of code inspection.

Conclusions

Hardware testing is more significant than software testing because of the high re-spin cost. Using functional verification alone is no longer best practice. It is possible to measure how much of the model has been executed. Code coverage shows when functional verification is not complete. The benefits of improving the verification strategy are reduced risk of the project having to re-spin Silicon and improved verification productivity through test development directed at untested areas.

References

[1] J. S. Collofello "Introduction to Software Verification and Validation" Software Engineering Institute Module SEI-CM-13-1.1, Dec. 1988.

[2] L. J. Morell, L. E. Deimel "Unit Analysis and Testing" Software Engineering Institute", SEI Curriculum Module SEI-CM-9-2.0, June 1992.

[3] D. E. Thomas and P. R. Moorby "The Verilog hardware description language", Kluwer Academic Publishers, Boston, MA, 2nd Edition, 1994.

[4] David Dill, "What's between simulation and formal verification?", in proceedings of 35th Design Automation Conference, June 1998.

[5] S. P. Riches and M. S. Abrahams "Practical approaches to improving ASIC verification efficiency", ISD Magazine, July 1998.

[6] Farzan Fallah, Srinas Divadas, Kurt Keutzer, "OCCOM: Efficient Computation of Observability based code coverage metrics for functional verification", in proceedings of 35th Design Automation Conference, June 1998.

[7] D. Drako, P. Cohen "HDL verification coverage", ISD Magazine, June 1998.