Post-Silicon Code Coverage for Functional Verification of Systems-on-Chip

by

Mehdi Karimibiuki

B.A.Sc., University of British Columbia, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE STUDIES
(Engineering)

The University Of British Columbia
(Vancouver)

August 2012

© Mehdi Karimibiuki, 2012
Abstract

Post-silicon validation requires effective techniques to better evaluate the functional correctness of modern systems-on-chip. Coverage is the standard measure for validation effectiveness and is extensively used pre-silicon. However, there is little data evaluating the coverage of post-silicon validation efforts on industrial-scale designs. This thesis addresses this knowledge-gap. We employ code coverage, which is one of the most frequently used coverage technique in simulation, and apply it post-silicon. To show our coverage methodology in practice, we employ an industrial-size open source SoC that is based on the SPARC architecture and is synthesizable to FPGA. We instrument code coverage in a number of IP cores and boot Linux as our experiment to evaluate coverage — booting an OS is a typical industrial post-silicon test. We also compare coverages between pre-silicon directed tests and the post-silicon Linux boot. Our results show that in some blocks, the pre-silicon and post-silicon tests can achieve markedly different coverage figures — in one block we measured over 50 percentage point coverage difference between the pre- and post-silicon results, which signifies the importance of post-silicon coverage. Moreover, we calculate the area overhead imposed
by the additional coverage circuitry on-chip. We apply state-of-the-art software analysis techniques to reduce the excessively large overhead yet preserve data accuracy. The results in this thesis are valuable data for guidance to future research in post-silicon coverage.
Preface

The two major contributions of this thesis (as discussed in Section 1.2) have been published in a conference paper. There is also a journal paper that is an extension of this thesis. It is accepted recently and to be published. The details of co-authorship for this thesis are outlined below.

The first paper, the research on post-silicon code coverage, was published as [1]. For this paper the concepts and experimental framework were developed collaboratively between Dr. Alan J Hu, Dr. Andre Ivanov, Mehdi Karimibiuki, and Kyle Balston. Mehdi Karimibiuki performed the main research, data generation, and data analysis. Kyle Balston generated the LUT results for overhead calculation. Dr. Alan J Hu made the final preparation of the manuscript and Dr. Andre Ivanov did the final review.

The second paper, the research on post-silicon code coverage for multiprocessors [2], has been accepted and to appear in the journal IEEE Transactions on Computers. This paper accepted June 17, 2012. Again, for this paper, the concepts and experimental framework were developed collaboratively between all authors. Kyle Balston did the data generation and data analysis. Mehdi Karimibiuki
provided the background research and the instrumentation guidelines from [1].
Dr. Alan J Hu did the final draft manuscript, and Dr. Steve Wilton prepared the
final revision of the manuscript. Dr. Andre Ivanov made the final comments and
review.
Table of Contents

Abstract ....................................................... ii
Preface ......................................................... iv
Table of Contents ............................................... vi
List of Tables .................................................... ix
List of Figures ................................................... x
Glossary .......................................................... xii
Acknowledgments ................................................ xiv

1 Introduction .................................................... 1
   1.1 Motivation ............................................... 1
   1.2 Contributions and Thesis Organization .................. 5

2 Review of Previous Work ..................................... 8
   2.1 Post-silicon Coverage ..................................... 8
5.1 Conclusions ........................................... 73
5.2 Future Work ........................................... 74

Bibliography ............................................. 76
List of Tables

Table 1.1  Comparison between pre- and post-silicon verification  
Table 2.1  Tag coverage results  
Table 2.2  Front side bus coverage results  
Table 2.3  Performance monitor event distribution  
Table 2.4  Performance monitor coverage results; Frequency = 5000  
Table 2.5  Performance monitor coverage results; Frequency = 0  
Table 2.6  Performance monitor coverage; final results  
Table 2.7  IBM POWER7 core coverage results  
Table 3.1  IP blocks under test  
Table 3.2  Post-silicon code coverage results  
Table 3.3  Comparison between pre- and post-silicon coverage results  
Table 4.1  Area overhead of coverage monitors  
Table 4.2  Overhead reduction results
List of Figures

Figure 1.1 Time-to-market ........................................ 3
Figure 2.1 Mutation coverage example ................................ 10
Figure 2.2 Mutation test example .................................. 10
Figure 2.3 Tag tracker example ..................................... 13
Figure 2.4 Coverage report obtained from Lisherness et al. ........ 15
Figure 2.5 A conditional statement in a process .................... 18
Figure 2.6 Verma et al. control-oriented path coverage algorithm . . 19
Figure 2.7 Control-oriented coverage results ....................... 21
Figure 2.8 Flow chart to capture data on the front side bus ........ 25
Figure 2.9 Extended execution trace mechanism .................... 31
Figure 2.10 Post-silicon coverage closure flow ..................... 33
Figure 2.11 Statement coverage example ........................... 35
Figure 2.12 Branch coverage example ............................... 36
Figure 2.13 Condition coverage example ........................... 37
Figure 2.14 Condition coverage measurement ...................... 41
Figure 2.15 Expression coverage example ........................... 42
Figure 2.16  Expression coverage report .......................... 43
Figure 3.1  An HDL code with its control flow graph ............... 46
Figure 3.2  Block diagram of LEON3 ................................. 47
Figure 3.3  Xilinx XUP board ...................................... 50
Figure 4.1  An HDL code with its control flow graph ............... 61
Figure 4.2  Pre- and post-dominator trees .......................... 62
Figure 4.3  Basic block dominator graph ............................ 64
Figure 4.4  Merging strongly connected components ................ 64
Figure 4.5  Super block dominator graph ............................ 65
Figure 4.6  Agrawal’s general rule ................................. 66
Figure 4.7  Simple example control flow graph ...................... 69
Figure 4.8  Simple example pre- and post-dominator trees ......... 69
Figure 4.9  Basic block and super block dominator graphs ......... 70
The following are abbreviations and acronyms used in this thesis, listed in alphabetical order:

**SoC**  system-on-chip

**MMU**  memory management unit

**VGA**  video graphics array

**UART**  universal asynchronous receiver transmitter

**FPGA**  field programmable gate array

**JTAG**  joint test action group

**RTL**  register transfer level

**FSB**  front side bus

**EET**  extended execution trace

**LUTs**  look-up tables
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFT</td>
<td>design for testability</td>
</tr>
<tr>
<td>ATPG</td>
<td>automatic test pattern generation</td>
</tr>
<tr>
<td>PMON</td>
<td>performance monitor</td>
</tr>
<tr>
<td>DFT</td>
<td>design for test</td>
</tr>
<tr>
<td>SPARC</td>
<td>scalable processor architecture</td>
</tr>
<tr>
<td>FPU</td>
<td>floating point unit</td>
</tr>
<tr>
<td>MMU</td>
<td>memory management unit</td>
</tr>
<tr>
<td>AMBA</td>
<td>advanced microcontroller bus architecture</td>
</tr>
<tr>
<td>DVI</td>
<td>digital visual interface</td>
</tr>
</tbody>
</table>
Acknowledgments

First, I would like to thank my supervisors, Dr. Andre Ivanov and Dr. Alan J Hu. Without their technical advice, constructive feedback, and financial support, this thesis could not be made. In particular, I kindly thank Dr. Ivanov for always being there to advise whenever I was stuck in my project. In spite of the fact that Dr. Ivanov is the Department’s Head and so has been extremely busy, he has always had time for me and welcomed me in his office to discuss my findings and results, and to give me feedback. His encouragement to employ innovation in my research inspired me a lot during the course of my graduate work. Furthermore, I would like to truly thank Dr. Alan J Hu for spending several hours with me during weekends to work on my thesis as I was working full-time during weekdays. I thank Alan for his generous time and kind support. His great feedback immensely improved my critical thinking and boosted my writing level. I am honored to work with Alan. I thoroughly admire his hard work with me until the end of this thesis.

Second, I would like to thank all my great professors at UBC who served me during my undergraduate and graduate studies. In particular, I would like to thank Dr. David Pulfrey, Dr. Steve Wilton, Dr. Konrad Walus, Dr. Alireza Nojeh, and
Dr. Luis Linares. I always enjoyed sitting in their classes. Their teaching style kept me motivated and let me shine in their classes.

Third, I would like to thank the UBC System-on-Chip staff, Dr. Roberto Rosales, and Mr. Roozbeh Mehrabadi. They always were willing to lend a hand when it was needed. Additionally, I would like to thank the Gaisler team for providing such a great open-source SoC. It allowed us to implement and expand our ideas on a real hardware device.

Four, I would like to thank all the great graduate students in the SoC and the ISD lab. I enjoyed being with them in both sunny and rainy days. Thanks for making the lab a pleasant place to work, research, and study. In particular, I would like to thank Dr. Flavio M de Paula, Mr. Sam Bayless, Mr. Mohammad Beikahmadi, and Mr. Kyle Balston. I give many thanks to Flavio who helped me at the beginning with hardware set-up and software bring-up. I thank Kyle for being there during finishing my project, and during the time that I was away.

Five, I would like to thank Lisherness and Cheng for providing their raw data so that I could reconstruct one of their figures in [3]. Thanks for their generosity in sharing data.

Six, to my mom and dad: I am deeply indebted for their constant support, especially during hard times. I am very grateful to have such a great parents. Without their absolute support and extreme care, I could not be who I am today. I am very proud of myself and my life because of them.

Last and the most important one, to my pretty lady, my lovely wife; to my smart Shima: I truly appreciate her patience and understanding, especially during
the last few months to finish my thesis. I certainly could not finish my thesis without her full back-up. That means a lot and I am thankful to her for the rest of my life.
Chapter 1

Introduction

1.1 Motivation

Today’s integrated circuits are increasingly complex and extremely compact. Many cores with specifically different functionalities in multiple clock domains form a multi-tasking\(^1\) system-on-chip (SoC)\(^2\). Some of these SoCs even support more than one complex protocols\(^3\)— the new PMC-Sierra chip is a feature-rich and sophisticated communication SoC that supports optical and Ethernet protocols [4].

At this high degree of design integration and complicated signal communication, it is very hard to verify the functional correctness of all IP blocks and

---

\(^1\)Multi-tasking refers to a method that different tasks are performed in parallel at one or more CPUs.

\(^2\)A system on chip (SoC) refers to an integrated circuit that comprises multiple functional units (analog and digital). These units communicate to each other with a common on-chip bus. Central processing unit (CPU), memory management unit (MMU), and graphics processing unit (GPU) are some unit examples that may include in a typical SoC.

\(^3\)A protocol is a set of communication rules inside an SoC. A protocol may also set rules about how to exchange information between the SoC and the outside world.
to validate the whole design at system-level. Formal verification does not apply at the system level. Checking the system-level functional correctness of an SoC with model-checking or equivalence proving is impossibly expensive. Even if we run exhaustive tests as our formal solution, it takes unacceptable time, energy and computational resources. Verification by simulation also faces a number of difficulties. Pre-silicon simulation uses a number of directed and random tests to detect bugs. But, since the tests are not exhaustive, simulation tests do not guarantee a bug-free system. Further, with these tests, some of the corner cases are not easy to generate, and even if so, the tests cannot scale to the size of software applications [5]. The main reason is that simulation is slow. It is, at least, four orders of magnitude slower than emulation, and seven to eight orders of magnitude slower than the real chip speed [6]. For example, if we want to test an application that takes only five seconds on a real chip to execute at speed of 1 GHz, it takes a year and a half in simulation to run at speed of 100 Hz. This leaves us no option but to generate some directed-random tests, in the hope of capturing most bugs, if not all. And, then capture remaining bugs that escaped on to the chip in post-silicon validation.

Post-silicon validation must make sure that real applications work fine on the chip and should fix any bugs escaped from pre-silicon. It can cost a lot of money if less attention is given to post-silicon validation and a bug is found after mass-production. For example, in 1994, the FDIV bug found in the floating point unit of the Intel P5 processor, cost about half-a-billion dollars for replacement and recovery [7].
Since post-silicon validation is the last important phase in making sure that all the functionalities of an SoC are operating correctly, considerable resources need to be committed to this step. In particular, validation time and cost has increased with shrinking technology nodes (see Figure 1.1). On average, validation time has increased to over one-third of the chip design development time [8] — this gets worse as we move into smaller technology nodes than 90nm. But even so, we see long documented errata published along with fabricated chips. For example, in the newest generation of AMD processors, the product errata is more than 110 pages [9]. Similarly, Intel has recently published a 55-page functional errata update for its core i-7 desktop processor series [10].

![Figure 1.1: Time-to-Market. Data in this figure are from [11].](image)

The problem is, post-silicon validation does not always work very well. Unlike pre-silicon, post-silicon possesses limited observability (see Table 1.1). We cannot see inside the chip. Our visibility is confined to a limited number of input and output pins, and a few block interfaces. Therefore, it is difficult to check for the effectiveness of our software tests on the chip. If a software run is successful on
a chip, we cannot evaluate how well it has practiced different blocks of the chip. There is no metric to tell us how much of the SoC properties have been exercised. We need an effective metric to give us feedback on the tests that we execute on the chip and tell us how well different properties and functionalities are exercised. We need a metric that is like an “eye-on-chip” for us to report tests’ effectiveness.

Table 1.1: Comparison between pre- and post-silicon verification.

<table>
<thead>
<tr>
<th>Pre-silicon</th>
<th>Post-silicon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excellent observability</td>
<td>Very limited or no observability</td>
</tr>
<tr>
<td>and controllability</td>
<td>and controllability on-chip</td>
</tr>
<tr>
<td>System-level tests run slow</td>
<td>Real software applications run fast</td>
</tr>
<tr>
<td>Cheap or no cost to fix bugs</td>
<td>Very expensive to fix bugs</td>
</tr>
</tbody>
</table>

The solution is coverage. In general, “coverage” is any scheme that measures the thoroughness of validation process. For example, we might measure with coverage what fraction of lines of code are exercised by a test, or what fraction of static timing analyses were successful, or what fraction of electrical faults have been caught, or what fraction of a chip has been exercised by a software execution test.

Coverage is the metric to evaluate validation effectiveness. It measures validation strength and helps the validation team to quickly decide what areas on-chip need deeper care. Coverage gives us the capability to assess and monitor validation. It definitely is the eye-on-chip for the validation engineer. Coverage is a systematic approach in place of many “ansatz” work that validation engineers

---

4 Ansatz means an educated guess that is verified after by the results. In a good problem-solving
apply to evaluate the effectiveness of their tests.

There are a few coverage methodologies proposed for post-silicon validation, namely, mutation coverage, tag coverage, control-oriented coverage, and assertion coverage. As part of the review of the previous work, we will explain these techniques in Section 2.1. As for this thesis, we propose “code coverage” as a practical methodology in post-silicon testing. Code coverage is extensively used in pre-silicon. But in post-silicon no one has done it to date. Code coverage is a syntactic coverage metric that is commonly used as the standard tool in software testing. In practice, code coverage is a primitive technique with expected high results, yet it is an essential coverage methodology [12]. We instrument code coverage on chip with a number of monitors and evaluate our tests’ effectiveness based on the coverage report. In its class, code coverage is competitive compared with assertion coverage, mutation coverage, tag coverage, or path coverage. We choose “code coverage” as our systematic solution to the challenge of evaluating validation effectiveness in post-silicon. Our methodology circumvents the challenge of limited visibility for evaluating tests’ effectiveness on chip.

1.2 Contributions and Thesis Organization

In this thesis, we made two main contributions. The first main contribution is code coverage instrumentation [1]. We propose the use of code coverage in post-silicon and evaluate code coverage effectiveness on-chip. We booted Linux as a typical post-silicon test and measure code coverage for that. We also compare scenario, an ansatz takes different corner cases and constraints into account.
the effectiveness of our post-silicon test with simulation results. We summarize that code coverage can be a standard methodology for evaluating the post-silicon validation effort.

In Chapter 2, we first explain different code coverage techniques that were proposed for post-silicon validation. Afterwards, in that chapter, we explain code coverage in simulation and present different types of code coverage. Then, in the following chapter, we show how we instrument code coverage in hardware. In Chapter 3, we show our evaluation platform and give its specifications. Also, we show and explain our experimental results and compare them with simulation ones.

The second and last contribution of this thesis is to investigate the imposed area overhead by our on chip coverage methodology [1]. In most post-silicon coverage papers, area overhead is not calculated. Area overhead is an important factor for considering a coverage technique on chip. Therefore, we first investigate the area overhead. Then we find ways available from the state-of-the-art software testing to reduce the overhead — to better attract code coverage instrumentation for industrial designs.

In Chapter 4, we first calculate the area overhead on a number of coverage-instrumented IP blocks, and then reduce the unacceptably large area overhead. Our reduction methodology is inspired from Agrawal’s method in software testing [13]. In that chapter, we report our overhead results based on units of flip flops and look-up-tables (FPGA synthesis unit).

Finally, in Chapter 5 we conclude our work and propose future avenues of
research and development in post-silicon validation based on the results obtained in this thesis.
Chapter 2

Review of Previous Work

In this chapter, we review two relevant subjects: (1) prior work on coverage in post-silicon, and (2) software code coverage. First, in Section 2.1, we survey academic and industrial papers on post-silicon coverage. We discuss their solutions and challenges for real industrial designs. Next, in Section 2.2, we introduce software code coverage. We define code coverage and discuss different types of code coverage.

2.1 Post-silicon Coverage

We classify post-silicon coverage techniques into two main categories:

1. Using techniques from pre-silicon verification for post-silicon coverage.

2. Obtaining coverage directly on the chip with specialized coverage collection circuitry.

In the following, we survey the papers of each type.
2.1.1 Post-silicon Coverage Methodologies from Pre-silicon Techniques

There are a number of different coverage solutions proposed for post-silicon validation, which are based on pre-silicon coverage techniques. We next survey these techniques and discuss whether or not they meet the post-silicon coverage goal. In particular, we discuss the following coverage techniques: mutation coverage, tag coverage, control-oriented coverage, and assertion coverage.

Mutation Coverage

A mutation is an intentionally injected error in a program [14]. There is an infinite variety of mutants possible to inject in a program. On one extreme, for example, there could be trivial mutants like typos or similar textural changes such that the code might even not compile — obviously, there is no value in writing tests for such mutants during coverage analyses. And, on the other extreme, there are meaningful mutations worth targeting with writing tests. In particular, incorrect data/reference assignments, wrong operators, or structurally wrong constructs are some examples of valuable mutations that have been investigated in the literature for post-silicon coverage [15]. For example, Figure 2.1(i) provides a piece of simple Verilog code. Figure 2.1(ii) shows a mutant due to incorrect data assignment to the literal x. Figure 2.1(iii), shows a wrong operator used in the code, and Figure 2.1(iv) shows structurally wrong code.

Given a set of mutants, mutation coverage measures what fraction of mutants are detected by a set of tests. If a test can capture an intentionally inserted error (a
module example1
always@(posedge clk)
begin
    signal x = x1;
    signal y = y1;
    signal z;
    if (x>y) then
        z=x;
    else
        z=y;
    endif;
end process;

module example1
always@(posedge clk)
begin
    signal x = x1;
    signal y = y1;
    signal z;
    if (x<y) then
        z=x;
    else
        z=y;
    endif;
end process;

module example1
always@(posedge clk)
begin
    signal x = x1;
    signal y = y1;
    signal z;
    if (x<y) then
        z=x;
    else
        z=y;
    endif;
end process;

module example1
always@(posedge clk)
begin
    signal x = x1;
    signal y = y1;
    signal z;
    if (x>y) then
        z=y;
    else
        z=x;
    endif;
end process;

Figure 2.1: Mutation coverage example. (i) A correct code. (ii) The code with a wrong literal mutant. (iii) The code with an incorrect operator mutant. (iv) The code with a structurally wrong mutant.

proc test
    signal x1 := 0x01;
    signal x2 := 0x02;
    signal y1 := 0x03;
#1 print z;
end proc;

Figure 2.2: Mutation test example.

mutant), it is said that the test kills that mutant. If a test kills many mutants (i.e. the mutation coverage is high), we say it is a good test; otherwise, it needs improvement. For example, Figure 2.2 shows an example that a test can kill mutants in Figure 2.1(iii) and Figure 2.1(iv) but not the one in Figure 2.1(ii). For such an example, our test could only kill two out of our three inserted mutants, therefore, we say our mutation coverage for this simple example is 67%.

Although the purpose of mutation coverage is to write good tests, it does not exactly serve the post-silicon coverage goal. In post-silicon coverage, we would
like to know how much of our hardware model is covered upon running a test. Consider the same example above (Figure 2.1(i)). With mutation coverage, we are not evaluating the coverage of the sample code. Instead, we are evaluating the coverage of mutants based on how many of those are killed by the test. Therefore, high mutation coverage does not mean that the test execution on hardware will achieve high coverage — it just means that the test could kill all mutants that are introduced into the code.

Moreover, mutation coverage is computationally an expensive check. First of all, the number of possible mutants is large [16]. Consider computing mutation coverage for all statements of a state-of-the-art processor[^1]. This might require the insertion of an impossibly large number of mutants — one mutant for every statement. Worse, every mutant must be inserted one at a time. This is because if we insert multiple mutants, we may not know how many mutants a test kills [15]. For example, with the test code in Figure 2.2, we know that mutants in Figure 2.1(iii) and in Figure 2.1(iv) are killed. But, by running that test, we cannot tell whether both errors are present in the code or only one of them exists — our test prints an unexpected value of signal $z$ after one clock cycle which does not give any information whether the error was due to structurally wrong order of assignments or an operational error. This is a very simple example though. Consider a large pool of mutants that can be killed by a single test. We will know a number of those are killed but we cannot tell how many and therefore we cannot compute mutation coverage.

[^1]: A state-of-the-art processor features multi-core multi-tasking properties. It runs at GHz clock frequencies and is power-efficient.
coverage on a single run.

Last but not least, mutation coverage can never be implemented on a real chip. This is a very glaring problem, because if we do so we are going to deliberately break the chip. The closest to chip-like behavior that we can get is to run mutation coverage in emulation. Therefore, mutation coverage is not truly a post-silicon coverage technique.

All in all, even though there is a proposal for mutation coverage implementation in post-silicon [15], we see that due to the above-mentioned shortcomings, this coverage technique is not a practical choice.

**Tag Coverage**

A *tag* is a tracking symbol associated with a statement. Tag coverage reports propagation effects of faulty statements throughout the system. During propagation, the tags might be overwritten, discarded, or ideally they might show up at the output signals. Tags that are not observable at the output signals imply that their corresponding expressions cannot be detected. But, the tags that make it to the system output are collected and reported as successful for coverage. For example, Lisherness and Cheng came up with a tag tracking system such that every data container (i.e. a variable) has a 64-bit tag tracker. This tracker is at first initialized to all zeros. Propagation effect of each variable flips one bit in the tracker. For each assignment, a tag assignment is inserted. The tag assignment is such that the left-hand-side (LHS) tag is the result of the bit-wise OR of the tracker vectors related to the right-hand-side (RHS) operands as well as a distinct tracker
library IEEE; use IEEE.std_logic_1164.all;
entity tag_prog is
  port (
    clk, full, pool, domino, pend, wait, cross, timeout : in std_logic_vector (2 downto 0);
    out1, out2, jtag : out std_logic_vector (7 downto 0));
end entity tag_prog;

Architechture rtl of tag_prog is

  signal full, pool, domino, pend, wait, cross, timeout : unsigned (2 downto 0);
  signal tracker, tr_full : std_logic_vector (7 downto 0) := (others => 0);
  signal tr_pool : std_logic_vector (7 downto 0) := 0x01;
  signal tr_domino : std_logic_vector (7 downto 0) := 0x02;
  signal tr_timeout : std_logic_vector (7 downto 0) := (others => 0);
  signal tr_pend : std_logic_vector (7 downto 0) := 0x03;
  signal tr_wait : std_logic_vector (7 downto 0) := (others => 0);
  signal tr_cross : std_logic_vector (7 downto 0) := 0x04;
  ...
  begin
    process (clk) is
      begin
        full <= pool + domino; tr_full <= tr_pool or tr_domino or 0x05;
        ...
        if ( ... )
          timeout <= pend; tr_timeout <= tr_pend or 0x06;
        end if;
        ...
        wait <= cross; tr_wait <= tr_cross or 0x07;
        ...
      end process;
    out1 <= timeout;
    out2 <= full; tracker <= tr_full or tr_timeout; jtag <= tracker;
  ...
end architecture rtl;

Figure 2.3: Tag tracker example.
bit associated with the LHS variable. Figure 2.3 shows a simple example of this. In this example, the tag tracker associated with the signal variable pool is OR’ed with the tag tracker associated with the signal variable domino, OR’ed with the tag tracker associated with the signal variable full. The same trackers generated for assignments to timeout and wait. If the propagation of tags reaches an observable point like the joint test action group (JTAG) output port in this example, it means that the propagation effect of variables are observable; otherwise, they cannot be monitored. In this example, the trackers of timeout and full are observable. That means that the propagation effect of signals pool, domino, full, pend, and timeout is observable, while the propagation effect of signals cross and wait is not observable. At the end, the number of bits in the tracking system that are set to ‘1’ are collected at all observable points and tag coverage is computed. In our example, tag coverage is 71.4% (5 out of 7 tags observed).

Tag coverage is a method proposed for two specific goals. First, it is used for monitoring the spread of the effect of bugs throughout the system [3]. With tags, the effect of statements are easy to follow across the system and to identify at output signals. For example, Lisherness and Cheng measure tag coverage on an i8039 instruction set simulator (ISS) [3]. Figure 2.4 shows the measurement and also comparison of their results with statement (see section 2.2) and mutation coverage. As can be seen, upon running 91 tests, tag coverage achieves medium coverage results — it is between mutation coverage which is an expensive coverage and statement coverage which is an easy-to-achieve coverage. Figure 2.4 also shows that more than 85% of the tests obtain over ninety percent tag coverage,
**Figure 2.4:** Coverage report for comparison between tag, mutation, and statement coverages. Figure generated by raw data from Lisherness and Cheng [3] (with permission to reproduce).
which means the effect of most potential faulty statements is observable.

Second, tag coverage is obtained for HDL models to debug and make better functional tests [17]. Consider when running a test, tags that are not propagated to the outputs should be examined, and their corresponding statement/line are identified. Moreover, their specific execution paths are identified to determine the blocking statement(s). Finally, the designer may decide to make a legal change in variable values or assignments such that the invisible tag propagates and is seen at the system output. For example, Fallah et al.’s results [17] show coverage on various algorithms such as a traffic controller, a train system relay, a chess queens algorithm, a general purpose processor, and a cache coherence protocol. Their results can be seen in Table 2.1. In their experiment, tag coverage measurement is compared with line coverage. Fallah et al. observe relatively low tag coverage results. For example, for the processor test, even though line coverage is over 90% percent, the tag coverage is only 68%. With coverage that low, presumably, the validation engineer needs to go back to the code and identify those blocking statements with their specific paths to make suggestions about how to improve their tag coverage results. Fallah et al. results for the train system test are also interesting — line coverage is 100% while tag coverage is 77%. This means that even though all lines are hit, not all statements are observable at the system output.

Despite tag coverage having been extensively researched in pre-silicon [3,17], using it for post-silicon is obviously impractical. The main reason is similar to the shortcoming in mutation coverage: tag coverage produces large area overhead.

\footnote{A blocking statement is a statement that prevents the propagation effect of a tag.}
Table 2.1: Tag coverage report and its comparison between line coverage. Data in this table obtained from Table V of [17].

<table>
<thead>
<tr>
<th>Tests</th>
<th>#Lines of code</th>
<th>#tags</th>
<th>tag coverage results based on random tests</th>
<th>line coverage results based on random tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess queens algorithm</td>
<td>220</td>
<td>146</td>
<td>67%</td>
<td>90%</td>
</tr>
<tr>
<td>Train system relay</td>
<td>353</td>
<td>256</td>
<td>77%</td>
<td>100%</td>
</tr>
<tr>
<td>Trafic controller</td>
<td>120</td>
<td>88</td>
<td>67%</td>
<td>90%</td>
</tr>
<tr>
<td>Genral purpose processor</td>
<td>288</td>
<td>198</td>
<td>68%</td>
<td>92%</td>
</tr>
<tr>
<td>Cache coherence protocol</td>
<td>530</td>
<td>346</td>
<td>54%</td>
<td>90%</td>
</tr>
</tbody>
</table>

— for large industrial designs if we are to specify tags for all expressions and statements, we need to reserve large amount of on-chip memory.

Control-Oriented Coverage

Control-oriented coverage [18] is an extension work to tag coverage. Control-oriented coverage identifies all paths that the effect of a tag may propagate through and reports how many of those paths covered. These paths are also known as “faulty paths”, because in the presence of potential errors, different data flow may follow. Control-oriented coverage reports how many of those faulty paths end to an observable point so the propagation effect of the tag is observable along those particular faulty paths. For example, consider the evaluation of the conditional if
Process …

…

if ( $current_address > $0x00ff) then …
  cache := empty;
else
  cache := cache+1;
…

Figure 2.5: A conditional statement in a process.

statement in Figure 2.5. Assume the correct value in current_address is “0x001a”. Then the conditional if predicate is false (0x001a is smaller than 0x00ff). But, let’s see what happens if an incorrect value ends up in the current_address. The incorrect value in current_address may or may not result in an incorrect predicate value. But if it results in an incorrect predicate value (i.e. true), cache is emptied instead of being incremented and a different branching sequence happens. However, a worse scenario is that in the presence of a wrong current_address value, the predicate still evaluates to false. Thus, the fault effect of the wrong current_address value may not propagate and the incorrect value of current_address remains undetected. As a result, an incorrect value for current_address may trigger both true and false of the condition as the faulty paths. Control-oriented coverage captures both paths as faulty paths.

Control-oriented coverage captures two main pieces of information: (1) it reports the number of possible paths in the presence of an error; and (2) it tells whether erroneous data can be detected at an observable point (by their tags) along
Start simulation with identifying all faulty paths due to an error injection [infeasible paths are pruned].

Are all faulty paths due to error injection executed?

Record propagation of the fault along faulty paths.

All tests vectors are executed and simulation is complete?

Compute coverage and report paths that the erroneous data detected.

Figure 2.6: Verma et al. control-oriented path coverage algorithm. Figure reconstructed from Figure 3 of [18].
each path (Figure 2.6 shows the algorithm for control-oriented coverage [18]). As mentioned, in the example in Figure 2.5, control-oriented coverage identifies both directions of the conditional statement as possible faulty paths and tells how many of the two faulty paths are covered (each path sets specific tags along). Figure 2.7 shows control-oriented coverage results of running random test sequences to eight ITC’99 benchmarks[^3]. Control-oriented coverage obtains 100 percent coverage on all benchmarks except the last two. This means that the propagation effect of faults is detectable along all faulty paths. For the last two benchmarks, which coverage achieves low results, almost half of the tags are not detectable along faulty paths. This could be either due to an incorrect selection of faulty paths, or if they are selected correctly, due to the missed propagation effect of tags.

Similar to mutation coverage and tag coverage, control-oriented coverage possesses a number of shortcomings. First, control-oriented coverage is computationally an expensive check. The propagation effect of faults are evaluated one at a time. Worse, there could be many different faults whose faulty paths cannot be found at a single test run (this may happen as a result of non-determinism from internal arbitration, asynchronous communication, or multiple clock domains). Second, control-oriented coverage cannot be achieved on real silicon — obviously, injection of erroneous data to check faulty paths breaks the chip’s functionality. Last but not least, we did not find any information in the literature to perform comparison between control-oriented coverage and other coverage metrics.

[^3]: ITC benchmarks are a number of gate-level/ register transfer level (RTL) programs provided specifically for measurement. The ITC benchmarks are gathered by many companies in electronics industry and used for design for test (DFT) and automatic test pattern generation (ATPG) purposes.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Tags</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM1</td>
<td>195</td>
<td>100</td>
</tr>
<tr>
<td>BM2</td>
<td>66</td>
<td>100</td>
</tr>
<tr>
<td>BM3</td>
<td>156</td>
<td>100</td>
</tr>
<tr>
<td>BM4</td>
<td>168</td>
<td>100</td>
</tr>
<tr>
<td>BM5</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>BM6</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>BM7</td>
<td>111</td>
<td>67.3</td>
</tr>
<tr>
<td>BM8</td>
<td>279</td>
<td>50.1</td>
</tr>
</tbody>
</table>

**Figure 2.7:** Control-oriented coverage results [18].

**Assertion Coverage**

An *assertion* is a predicate (usually a conditional statement) that is always true unless something has gone wrong at that place in the code. Assertion coverage reports how many of the assertions are satisfied and how many are violated. So, with assertion coverage we can check application requirements or specific system functionalities that could go wrong in post-silicon testing [19].

Assertion coverage has two main usages. First, assertion coverage is used for localizing post-silicon bugs [19]. One example of this is writing assertions on module interfaces. During execution of test vectors, we can check whether those assertions are satisfied or violated. The failing assertions help identify which blocks’ boundary are affected. Thus we can identify the exact location where the bug happened (bug localization). This approach experimented on MIPS core with over 20,000 signals, and have been successful in localizing specific bugs in between 278 modules [19]. In addition to bug localizing, extending the use of assertion coverage on hardware is applied to application-specific error monitoring.
for specific processor designs [19].

Second, unlike the previous three coverage techniques that cannot readily be implemented on silicon, assertion coverage is implementable on silicon. However, that costs area investment. The number of assertions on the hardware in addition to the location, where they must be placed to get the effective information, is important. One optimization, possibility, is to use assertions that can detect important crashes. In that case, we may leave assertion monitors on-chip for the following two events: (1) checks for unexpected events that should never happen. For example, an assertion coverage for checking the correctness of a gray-code bus to make sure no more than one bit changes per cycle. (2) checks for events that must finally happen especially during the chip bring-up\(^4\). For example, an assertion for instruction fetch after reset is recommended such that an instruction must eventually be fetched from memory on the following cycles and if not assertion triggers. Boulé and Zilic left a number of assertion monitors for the aforementioned checks in their HDL designs and came up with a synthesizeable hardware model generator that produces correct assertion-based circuits in HDL [20].

2.1.2 Collecting Coverage Directly On-chip with Specialized Collection Circuitry: Two Industrial Case Studies

Section 2.1.1 above surveyed the literature which investigated post-silicon coverage based on pre-silicon techniques; however, none of those techniques have

\(^4\) Chip bring-up is a stage before mass-production of a chip but after the tape-out. At this stage, a test-chip, also known as the first silicon prototype, must pass all important tests — all the necessary software applications must run correctly for producing the intended functionality.
really been implemented in silicon — they were tried in simulation and emulation only. To date, there are very few reports of how coverage is instrumented on real silicon dies — or if there are any, they are not published. This is mainly due to the high cost for additional coverage collection circuitry on-chip — and so as a result, silicon companies are reluctant to put their post-silicon coverage circuitry on chip. In fact, we found only two papers that discuss on-chip coverage collection circuitry. They report the coverage instrumentation and measurement on industrial chips. The first report is from Intel, and the second one is from IBM.

**Functional Coverage Measurement of the Intel Core2 Duo Family**

Bojan et al. [21] at Intel are the first to report instrumenting coverage on an industrial-size processors\(^5\). They measure functional coverage on the Intel Xeon 5100 series and the Intel Core2 Duo mobile processors. They give three enhanced orthogonal approaches by looking at the CPU interfaces and using existing features of the IA-32\(^6\).

In all of Bojan et al.’s work, they confine themselves within the limitations of a commercial processor. In particular, there are three main limiting factors that Bojan et al. take into account. First, the die size is the most important limiting factor that directly affects the cost and yield. Therefore, in most of their work they do not add any instrumentation, and when they do, they add only a few monitors.

---

\(^5\) An industrial-size processor is referred to a leading processor by a major chip manufacturer like Intel, AMD, etc. At the time of this paper, Intel’s industrial-size processors have over 70 million gates and clock speeds up to 3.8 GHz.

\(^6\) IA-32 refers to Intel 32-bit processors’ architecture. IA-32 is the main instruction set architecture that has been developed for all Intel processors [22].
Second, the state of the CPU is visible at a very small subset of events and interfaces [21]. Therefore, there is limited visibility of the internal events for collecting the coverage of the internal nodes. Bojan et al. find ways to measure the coverage based on the observable events at block interfaces. Third, due to the short time-to-market, Bojan et al. need to stay on a tight schedule. They come up with practical coverage techniques that are compatible with the current test platforms that they already have at Intel.

Bojan et al. measure coverage by three main available data vectors of the CPU: (1) the front side bus (FSB); (2) performance monitors (PMON); and (3) extended execution traces (EET). Below, I describe each of these in sequence.

1. Coverage collection by using the FSB contains two stages: first, capturing data that indicates coverage information, and then, processing the coverage information.

In the first stage, the correct information must be captured. To do this, Bojan et al. monitor CPU transactions at the FSB\textsuperscript{7}. To collect the FSB transactions, Bojan et al. employ the same test platform that they used in emulation for on-chip monitoring. The FSB traces are stored by a logic analyzer\textsuperscript{8}. These traces contain coverage monitors that came from the pre-silicon instrumentation [21]. The logic analyzer triggers are configured to capture the correct

\textsuperscript{7}The Front Side Bus (FSB) is the main interface between the CPU and other cores. The FSB protocol has been extensively used and verified since the introduction of the Intel Pentium 4 [21]. Therefore, the FSB is an extremely robust and a stable bus to watch CPU’s behavior.

\textsuperscript{8}A logic analyzer is off-chip circuitry (or in some SoC architectures it can be added as an on-chip peripheral unit) that is designed to capture and take the important signal information out of the chip for evaluation. The logic analyzer usually connects to the host computer via JTAG ports.
coverage data until the logic analyzer buffer is full. After that, the coverage information is dumped from the logic analyzer to the front-end workstation to measure the coverage of the FSB events. The flow chart of this process is shown in Figure 2.8.
Table 2.2: FSB coverage collection results.

<table>
<thead>
<tr>
<th>Number of Cases</th>
<th>FSB Pre-silicon Results</th>
<th>FSB Post-silicon Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 single event cases</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>14000 back-to-back event cases</td>
<td>54%</td>
<td>74%</td>
</tr>
</tbody>
</table>

To compute the coverage, Bojan et al. categorize the coverage data that was obtained from the FSB into two different types of occurrences: (1) the coverage data that was obtained due to the occurrence of single events on the bus; and (2) the coverage data that was obtained due to the occurrence of back-to-back events [21]. In particular, the coverage for occurrence of single events monitors Arbitration, Snoop, and Request & Response acquisitions, while the coverage for occurrence of back-to-back events captures the request pairs that happen in two clock cycles. Table 2.2 shows the number of cases and the coverage results. The main benefit of obtaining coverage data from the FSB is that the post-silicon coverage results are compared against the pre-silicon coverage data (see Table 2.2) obtained in simulation. Therefore, they are able to improve their tests by looking at the two results.

2. In their second post-silicon coverage collection technique, Bojan et al. use performance monitor (PMON) registers [21]. With these registers that are already on the chip, it is possible to monitor and count a number of predefined events inside the CPU. In particular, PMONs are used to indicate whether a specific condition is hit or not. Coverage is measured based on the number of hits, which shows the number of events related to their cor-
Table 2.3: PMON event distribution among clusters. Data in this table are from Table II of [21].

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Number of events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>224</td>
</tr>
<tr>
<td>Memory</td>
<td>41</td>
</tr>
<tr>
<td>Front End</td>
<td>16</td>
</tr>
<tr>
<td>Execution</td>
<td>21</td>
</tr>
<tr>
<td>Out of Order</td>
<td>34</td>
</tr>
<tr>
<td>Power Management</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>343</strong></td>
</tr>
</tbody>
</table>

responding PMONs that are exercised.

In their experiments, Bojan et al. selected a number of events for coverage monitoring. Table 2.3 shows the set of events from each cluster that are selected for coverage measurement. The coverage data that were measured in this project are collected at intervals of six hours of continuous test runs, which is about 450,000 different tests [21]. Initial results show that 36 events out of 343 are not covered — Bojan et al. set a frequency level of 5000 hits per event, which means every event that did not reach that frequency is considered as an inadequately covered event and should be inspected. Table 2.4 shows the coverage results. To extend the analysis for coverage results, Bojan et al. also measured the events that were never triggered (the zero occurrence events). Table 2.5 shows these results. After obtaining the results of the two experiments, they proceeded by inspecting the zero-triggered coverage results first. After inspection, they reported two
Table 2.4: PMON coverage results. Some data in this table taken from Table III of [21].

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Not covered events (threshold frequency of occurrence = 5000)</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>27</td>
<td>87.94%</td>
</tr>
<tr>
<td>Memory</td>
<td>2</td>
<td>95.12%</td>
</tr>
<tr>
<td>Front End</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Execution</td>
<td>4</td>
<td>80.95%</td>
</tr>
<tr>
<td>Out of Order</td>
<td>2</td>
<td>94.11%</td>
</tr>
<tr>
<td>Power Management</td>
<td>1</td>
<td>85.71%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>36</strong></td>
<td><strong>89.50%</strong></td>
</tr>
</tbody>
</table>

Table 2.5: PMON coverage results. Data in this table taken from Table IV of [21].

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Not covered events (threshold frequency of occurrence = 0)</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>12</td>
<td>94.64%</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
<td>97.56%</td>
</tr>
<tr>
<td>Front End</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Execution</td>
<td>3</td>
<td>85.71%</td>
</tr>
<tr>
<td>Out of Order</td>
<td>1</td>
<td>97.06%</td>
</tr>
<tr>
<td>Power Management</td>
<td>1</td>
<td>85.71%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>18</strong></td>
<td><strong>94.75%</strong></td>
</tr>
</tbody>
</table>
main reasons for the zero-covered events: (1) an actual functional bug prevented some events from happening during the run; and (2) some events are for specific usage modes (such as the scan-chain mode\textsuperscript{9}) that require writing directed tests for them. After fixing the bugs, they rerun the tests under the same conditions. Bojan et al. report that their confidence in delivering a bug-free chip increased with the final results of the coverage data.

3. In their last coverage collection solution, Bojan et al. exploit the extended execution trace (EET) mechanism on the processor to collect coverage information. During the normal execution of the CPU, EET records microcode\textsuperscript{10} instructions that get executed in real-time. The idea is to detect the execution of specific microcode instructions inside the EET and signal them to the outside world [21]. For this purpose, additional monitoring features were added to some sequences of the microcode for a number of specific instructions. Whenever a specific instruction is executed, a signal is generated and sent to the JTAG port. Once the JTAG buffer is full, data are dumped into a file for coverage analysis. The system level components of using the EET mechanism for coverage are shown in Figure 2.9. Bojan et al. were able to achieve 100% coverage in this method. However, for functions that were less dependent on microcodes, the coverage was lower

\textsuperscript{9}Scan-chain mode is a test technique. It is to observe the value of the flip-flops inside the chip. In this mode, all flip-flops are wired together, and their value is streamed out as a serial vector format. This is to make sure that all of the flip-flops have the correct value at the specific event instances.

\textsuperscript{10}A microcode is a sequence of low-level instructions for circuit-level operations that controls the processor directly.
Table 2.6: Final PMON coverage results. Data in this table taken from Tables V & VI of [21].

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Not covered events</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(threshold frequency of occurrence = 5000)</td>
<td></td>
</tr>
<tr>
<td>Bus</td>
<td>13</td>
<td>94.20%</td>
</tr>
<tr>
<td>Memory</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Front End</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Execution</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Out of Order</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Power Management</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>96.21%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cluster name</th>
<th>Not covered events</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(threshold frequency of occurrence = 0)</td>
<td></td>
</tr>
<tr>
<td>Bus</td>
<td>5</td>
<td>97.77%</td>
</tr>
<tr>
<td>Memory</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Front End</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Execution</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Out of Order</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Power Management</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>98.54%</td>
</tr>
</tbody>
</table>

(they did not name those functions and did not report the coverage data).

Post-silicon Coverage Instrumentation on IBM’s POWER7 Processor

At IBM, Adir et al. proposed a different technique for post-silicon coverage. They employed their emulation platform to run a number of post-silicon exer-
JTAG Controller
Output file with micro-code labels to measure the coverage.

CPU
Micro-code
EET Custom patch
Transfer to DFT PINS
Debug vector

Output file with micro-code labels to measure the coverage.

Figure 2.9: EET mechanism. Figure reconstructed from Fig. 4 of [21].

Exercisers[11][23]. With this approach, they guarantee excellent post-silicon coverage results, because in emulation they have enough observability and controllability to improve the post-silicon tests.

First, they generate a test suite that ensures high coverage results in emula-

Exercisers are common scenarios/programs that run on the chip. For example, an operating system bring-up is a typical exerciser.
tion. They create a probabilistic\textsuperscript{12} regression suite\textsuperscript{13} as large as a real software application. Then, they run it in emulation and measure coverage. Their coverage reports on what fraction of the monitors that are placed in the HDL model are hit by running that regression suite. They continue improving their regression suite until they get high coverage results in emulation. The refined regression suite that produced high coverage results is used to run on the real silicon. The result of this work is to obtain high quality regression suites that are suitable for post-silicon validation tests and guaranteeing high coverage results. This also means that “coverage measurement on the silicon becomes less important”\textsuperscript{[23]} and so it is fine to remove the coverage monitors (or leave only the “hard-to-hit” monitors) on the chip. Figure 2.10 shows their flow-chart of preparing high quality regression suites for post-silicon coverage.

This method has been used in the verification of the IBM POWER7 core. A number of general purpose exercisers were selected to run on the IBM POWER7 core. Throughout the project, Adir et al. measured coverage multiple times in emulation to improve the quality of their tests and to close coverage holes. Table 2.7 shows their final results. The coverage results of the simulation tests are comparable with the ones obtained by running the exercisers in emulation — obviously the great coverage results of the latter is due to the large number of cycles and the good quality (close to real software applications) of the exercisers. Overall, the

\textsuperscript{12}The term probabilistic is used for a technique that gives information about the occurrence probability of coverage events for every test run.

\textsuperscript{13}A regression suite is a test-case based on previous/older tests. It makes sure that with a new changes to a design (i.e. bug fixes, adding coverage monitors, or adding additional features), the system’s functionality is not broken.
Run tests with existing templates in pre-silicon accelerator/ emulation

Coverage quality is good?

Investigate reasons for coverage holes

Collect and measure coverage information

Improve/refine tests

Yes

Employ important test templates and get ready to run

No

Run regression on silicon

Figure 2.10: Post-silicon coverage closure flow. Figure reconstructed from Fig. 2 of [23].

Table 2.7: IBM POWER7 core coverage results. Data in this table taken from Table 1 of [23].

<table>
<thead>
<tr>
<th>Unit name</th>
<th>coverage in simulation</th>
<th>coverage in emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFU</td>
<td>96.79%</td>
<td>94.99%</td>
</tr>
<tr>
<td>ISU</td>
<td>96.48%</td>
<td>92.78%</td>
</tr>
<tr>
<td>FXU</td>
<td>99.60%</td>
<td>85.85%</td>
</tr>
<tr>
<td>FPU</td>
<td>97.44%</td>
<td>90.20%</td>
</tr>
<tr>
<td>LSU</td>
<td>94.33%</td>
<td>91.04%</td>
</tr>
<tr>
<td>PC</td>
<td>92.51%</td>
<td>55.23%</td>
</tr>
</tbody>
</table>
production and use of exercisers in emulation for post-silicon validation turned out to be a successful project for IBM [23]— it produced high coverage results and gave confidence to the IBM team that their post-silicon bring-up tests are capable of generating excellent coverage results.

2.2 Code Coverage in Simulation

Code coverage measures how much of a body of source code has been exercised. It is a syntactic coverage metric that is commonly used as the standard tool in software testing. In simulation, code coverage reports on how much of the HDL code has been simulated. There are four main types of code coverage: statement, branch, expression, and condition.

2.2.1 Types of Code Coverage

We describe the four common forms of code coverage used in hardware simulation.

- **Statement Coverage**: this metric reports on how many of the statements are hit. For example, Figure 2.11 shows an example of a piece from the LEON3 UART source code [24]. In this figure, there are 27 statements (highlighted). There can be multiple statements in a single line which statement coverage identifies. At the end, statement coverage reports what fraction of all statements are executed by a given test.

- **Branch Coverage**: this metric focuses on branches and conditions (i.e. “if /then /else” and “case”). It collects hits for both True(T) and False(F)
architecture rtl of uart is 
… 
[defined signals and constant] 
… 
begin 
… 
[defined variables] 
… 
1   v := ... := '0'; 
17   end if; 
18   if r.rcnt /= rcntzero then 
19 dready := '1';  
20   end if;  
… 
end process;

Figure 2.11: Statement coverage example. This code is a piece from UART core of the LEON3 open-source SPARC processor [24].
architecture rtl of uart is
...
[defined signals and constant]
...
begin
...
[defined variables]
...
1 v := r; irq := (others => '0'); irq(pirq) := r.irq;
2 v.irq := '0'; v.txtick := '0'; v.rtick := '0';
3 rdata := (others => '0'); v.rxdb(1) := r.rxdb(0);
4 dready := '0'; thempty := '1'; thalffull := '1'; rhalffull := '0';
5 v.ctsn := r.ctsn(0) & uarti.ctsn;
6
7 if fifosize = 1 then
8   dready := r.rcnt(0); rfull := dready; tfull := r.tcnt(0);
9   thempty := not tfull;
10  else
11    tfull := r.tcnt(log2x(fifosize)); rfull := r.rcnt(log2x(fifosize));
12    if (r.rcnt(log2x(fifosize)) or r.rcnt(log2x(fifosize) - 1)) = '1' then
13      rhalffull := '1';
14    end if;
15    if (r.tcnt(log2x(fifosize)) or r.tcnt(log2x(fifosize) - 1)) = '1' then
16      thalffull := '0';
17    end if;
18    if r.rcnt /= rcntzero then
19      dready := '1';
20    end if;
...
end process;

Figure 2.12: Branch coverage example. This code is a piece from UART core of the LEON3 open-source SPARC processor [24].
architecture rtl of uart is
...
[defined signals and constant]
...
begin
...
[defined variables]
...
1   if r.debug = '1' and r.tcnt /= rcntzero then
2       rdata(7 downto 0) := r.thold(conv_integer(r.traddr));
3   if fifosize = 1 then
4       v.tcnt(0) := '0';
5   else
6       v.traddr := r.traddr + 1;
7       v.tcnt := r.tcnt - 1;
8   end if;
9   end if;
10  paddr := "000000"; paddr(abits-1 downto 2) := apbi.paddr(abits-1 downto 2);
11  if [apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1] then
12     dready := '1';
13  end if;
...
end process;

**Figure 2.13:** Condition coverage example. This code is a piece from the UART core of the LEON3 open-source SPARC processor [24].
branches and reports if any of the T or F conditions are not covered. Figure 2.12 shows that our example code has eight branches (highlighted).

- **Condition Coverage**: this metric is an extension to branch coverage. It gives the decision information made about the ternary statements as well as the decision data made in the “if” conditionals. For example, Figure 2.13 shows that there are two if statements that this type of coverage identifies. Condition coverage constructs separate boolean tables for each of the condition expressions in lines 1 and 11. Then it calculates the possibilities and counts the number of hits for each possibility. Figure 2.14 shows an example of how this is done in Modelsim®. Condition coverage reports how many those combinations in that table are covered. In this example, the report says that no combinations of the condition in line 1 were hit. But, all combinations of the condition in line 11 were hit.

- **Expression Coverage**: this metric provides statistics for logical expressions and checks all combinations of their boolean possibilities. Expression coverage, at first, identifies the expressions on the right-hand-side of the assignment statements. Then, it reports how many of those expressions covered. For expressions that are logical, similar to condition coverage, a table is constructed. The report tells how many of those boolean possibilities are covered. For example, Figure 2.15 is a different piece from the LEON3 UART core [24]. Expression coverage identifies the two expressions for this piece in Modelsim®. Figure 2.16 shows how coverage is measured for
2.2.2 About the Usage of Code Coverage for Post-silicon in This Thesis

We employ software code coverage as the basis of our work. Code coverage is a weak coverage model. It is easy to achieve and it does not give much information about the functionality of an HDL model. However, code coverage is actively used in the industry and proven valuable (e.g., [25] specifically highlights the value of code coverage, although acknowledging its weaknesses). Code coverage is a minimum requirement in simulation. It is the necessary test requirement in pre-silicon, and so because of that, if we obtain code coverage on-chip, we have the freedom to compare the two (pre- and post-silicon) results. In fact, being able to compare similar coverage results between the pre- and post-silicon is very valuable, because it gives us a better understanding about the quality of our tests during the verification process.

Moreover, code coverage is a very well-known and long-established coverage model. It is a standard, well-defined, and a well-accepted metric. The definitions of code coverage are precise and completely objective — the functional coverage metrics that we already explained in this chapter require verification expertise for devising efficient coverage targets.

We use code coverage as a lightweight, efficient, and a practical starting point for coverage research in post-silicon. To support our claims, we empirically show that, for post-silicon deployment, code coverage does not have conventional diffi-
culties that other coverage metrics do. We show how we implement code coverage on our hardware with reduced instrumentation. Due to the scope of this project, we limit our post-silicon code coverage findings to the statement coverage and the branch coverage. Therefore, in the following chapters wherever we refer to code coverage, we mean only the statement and branch coverage metrics.
Figure 2.14: Condition coverage measurement.
architecture rtl of uart is

...  
[defined signals and constant]  
...

begin

...  
[defined variables]  
...

1  scaler := r.scaler - 1;
2  if (r.rxen or r.txen) = '1' then
3    v.scaler := scaler;
4    v.tick := scaler(11) and not r.scaler(11);
5    if v.tick = '1' then v.scaler := r.brate; end if;
6  end if;
7  -- optional external uart clock
8    v.extclk := uarti.extclk;
9  if r.extclken = '1' then v.tick := r.extclk and not uarti.extclk; end if;
10  -- read/write registers
11  if (apbi.psel(pindex) and apbi.penable and (not apbi.pwrite)) = '1' then
12      rdata(7 downto 0) := r.rhold(conv_integer(r.rraddr));
13  end if;
...

end process

Figure 2.15: Expression coverage example. This code is a piece from the UART core of the LEON3 open-source SPARC processor [24].
Figure 2.16: Expression coverage report.
Chapter 3

Post-Silicon Code Coverage: Instrumentation and Results

In this chapter, we discuss code coverage in post-silicon as our solution to evaluating validation effectiveness. Although code coverage is easy to observe in simulation, in hardware, due to limited visibility, code coverage is hard to measure. In order to measure code coverage on chip, we come up with an instrumentation approach that gives us the freedom to observe coverage on chip.

In the following section, we describe how we instrumented and collected code coverage on hardware. Then, in Section 3.2, we introduce our hardware system, and in Section 3.3, we show and discuss the experimental results obtained from the hardware.


3.1 Instrumentation

To instrument code coverage on hardware, we first identify basic blocks in our hardware model.

Definition 1. “Basic Block” is a sequence of consecutive statements with a single branch or return statement at the end. The flow of control enters and leaves the basic block without any branching into or out of the sequence.

We granulate our RTL source code in terms of basic blocks. For example, the code in Figure 3.1 is comprised of 12 basic blocks. We allocate a unique bit for every basic block. Therefore, for the code in Figure 3.1, we require 12 unique bits, one for each basic block. When a test runs on our HDL model, we set a basic-block-bit to “one” if the test executes through that basic block, otherwise, the bit stays unset. The idea of “one-unique-bit-per-basic-block” establishes the foundation for statement and branch code coverage monitoring on hardware.

Note that statement coverage collection in our model is slightly different than what we have as statement coverage measurement in software tools that we explained in Section 2.2.1. In software, every statement is flagged, while in our technique, we only flag basic blocks in the code. Consequently, all statements in a basic block are covered if and only if that basic block is covered. All in all, when we report statement coverage, we evaluate and report what fraction of all basic blocks get exercised.

For branch coverage, we would want to see both directions (true and false) of all conditionals. For example, in Figure 3.1, for branch coverage, we would like
Figure 3.1: Example of an HDL code with corresponding control flow graph.

to see both directions of the if statements at basic blocks 2, 5, and 10, as well as all three directions of the case statement at basic blocks 4, 7, and 9. This is similar to branch coverage in simulation that we explained in Section 2.2.1.

3.2 Evaluation Platform

We chose an industrial-scale SoC that we can instrument for coverage monitoring, and then a means to run post-silicon validation tasks on it. In this section we demonstrate that our proposed code coverage methodology works on a real, industrial-scale SoC.

At UBC, we have developed a system-on-a-chip that has a LEON3 core. LEON3
is a 32-bit scalable processor architecture (SPARC)\textsuperscript{1}. We developed such an SoC, built from the open-source IPs that is provided by the Aeroflex Gaisler [24]. The LEON3 processor has an integer unit (SPARC V8, 7-stage pipeline), I- and D-caches, a floating point unit (FPU), and a memory management unit (MMU) [24]. Also integrated into the SoC are peripheral units, including a universal asynchronous receiver transmitter (UART), PS/2 ports, a digital visual interface (DVI), an ethernet, and a flash memory; allowing us to plug in keyboard, mouse, video, and networking [24]. It is roughly comparable to a netbook or tablet device and can be fabricated to 0.18 $\mu$m TSMC ASIC technology [2] at a speed of 400

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.2.png}
\caption{Block Diagram of LEON3 SoC. Diagram taken from GRLIB IP Library User’s Manual [26].}
\end{figure}

\textsuperscript{1} SPARC is a high performance processor architecture. It developed by Sun Microsystems in mid-1987.
MHz. Its maximum operating frequency on our Virtex-5 XC5VLX110T field programmable gate array (FPGA) device is 80 MHz, a typical frequency for FPGA-based designs. The processor is fast enough to boot Linux and run the X11 windowing system. We have also added an on-chip logic analyzer (LOGAN) peripheral core to the SoC [27], which we use to store the values of the basic-block coverage monitor flags, and which we can access via the JTAG port. An architectural view of the LEON3 with corresponding IP cores connected by the advanced microcontroller bus architecture (AMBA)\(^2\) can be found in Figure 3.2.

### 3.3 Experimental Results

To perform our experiments, we selected nine diverse IP blocks on the SoC and instrumented basic-block bit variables in each of their basic blocks to measure coverage. Table 3.1 shows the IP names with brief functionality and number of basic blocks.

After instrumentation, we synthesized our design into the FPGA. The resulting synthesized SoC runs at 75Mhz. As mentioned earlier, this speed is fast enough to run graphical and interactive applications. Figure 3.3 shows our Xilinx XUPV5 (ML509) FPGA prototyping board.

For the post-silicon test, we booted Linux on the SoC running on the FPGA board — a very typical post-silicon test, which would be impossibly slow to run in simulation. Table 3.2 shows the results we obtained by our experiment.

To compare our post-silicon test with the pre-silicon simulation tests, we also

\(^2\) AMBA is an on-chip bus in SoC designs. It was introduced by ARM Ltd. in 1996.
Table 3.1: IP blocks under test.

<table>
<thead>
<tr>
<th>IP Block</th>
<th>Lines of Code</th>
<th>Basic Blocks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2cmst</td>
<td>107</td>
<td>15</td>
<td>I²C Master Controller from AMBA APB</td>
</tr>
<tr>
<td>div32</td>
<td>140</td>
<td>26</td>
<td>64-by-32 Bit Integer Divider</td>
</tr>
<tr>
<td>mmutw</td>
<td>179</td>
<td>28</td>
<td>MMU Table-Walk Logic</td>
</tr>
<tr>
<td>mul32</td>
<td>320</td>
<td>90</td>
<td>Signed/Unsigned 32-Bit Multiplier</td>
</tr>
<tr>
<td>uart</td>
<td>420</td>
<td>102</td>
<td>Asynchronous UART</td>
</tr>
<tr>
<td>mmutlb</td>
<td>421</td>
<td>54</td>
<td>MMU Translation Lookaside Buffer</td>
</tr>
<tr>
<td>svgactrl</td>
<td>472</td>
<td>104</td>
<td>VGA Controller</td>
</tr>
<tr>
<td>mmu</td>
<td>475</td>
<td>62</td>
<td>MMU Top-Level Entity</td>
</tr>
<tr>
<td>iu3</td>
<td>650</td>
<td>128</td>
<td>LEON3 7-Stage Integer Pipeline</td>
</tr>
</tbody>
</table>
run pre-silicon experiments. We simulated the entire design using Aeroflex Gaisler-supplied, short, simple system-level tests [24]. Table 3.3 shows the comparison of our results. Note that since we want both pre- and post-silicon results to be comparable with each other, we calculate the simulation results based on basic blocks too.
Table 3.2: Post-silicon code coverage results.

<table>
<thead>
<tr>
<th>IP Block</th>
<th>Statement</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2cmst</td>
<td>86.0%</td>
<td>81.8%</td>
</tr>
<tr>
<td>div32</td>
<td>89.2%</td>
<td>73.3%</td>
</tr>
<tr>
<td>mmutw</td>
<td>92.9%</td>
<td>94.7%</td>
</tr>
<tr>
<td>mul32</td>
<td>40.3%</td>
<td>35.7%</td>
</tr>
<tr>
<td>uart</td>
<td>90.2%</td>
<td>72.5%</td>
</tr>
<tr>
<td>mmutlb</td>
<td>94.4%</td>
<td>81.0%</td>
</tr>
<tr>
<td>svgactrl</td>
<td>92.7%</td>
<td>90.5%</td>
</tr>
<tr>
<td>mmu</td>
<td>88.7%</td>
<td>85.9%</td>
</tr>
<tr>
<td>iu3</td>
<td>96.1%</td>
<td>95.0%</td>
</tr>
</tbody>
</table>

3.3.1 Discussion

Not surprisingly, statement coverage is over 85% for most blocks, which is considered a good coverage result but is not enough — in industry, it is required that the coverage stays above the 90% line and in some cases over 95% [29]. Branch coverage results are different. In some blocks, branch coverage is over 90% and in others it is below 85%. These results are expected because statement coverage is
Table 3.3: Comparison between pre- and post-silicon coverage results.

<table>
<thead>
<tr>
<th>IP Block</th>
<th>Pre-Silicon Coverage</th>
<th>Post-Silicon Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(System-Level Directed Tests)</td>
<td>(Linux Boot)</td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td>Branch</td>
</tr>
<tr>
<td>i2cmst</td>
<td>90.0%</td>
<td>86.4%</td>
</tr>
<tr>
<td>div32</td>
<td>92.8%</td>
<td>80.0%</td>
</tr>
<tr>
<td>mmutw</td>
<td>90.5%</td>
<td>78.9%</td>
</tr>
<tr>
<td>mmul32</td>
<td>41.2%</td>
<td>39.1%</td>
</tr>
<tr>
<td>uart</td>
<td>88.6%</td>
<td>68.3%</td>
</tr>
<tr>
<td>mmutlb</td>
<td>92.3%</td>
<td>74.6%</td>
</tr>
<tr>
<td>svgactrl</td>
<td>57.6%</td>
<td>32.2%</td>
</tr>
<tr>
<td>mmu</td>
<td>84.0%</td>
<td>63.2%</td>
</tr>
<tr>
<td>iu3</td>
<td>89.8%</td>
<td>69.4%</td>
</tr>
</tbody>
</table>

An easy coverage metric to achieve, while branch coverage is challenging because obtaining different branching scenarios with running one software application on chip, is difficult. There should be more complicated software runs other than the Linux boot to hit as many branching sequences as possible for achieving higher branch coverage results.

The integer unit (iu3) shows statement and branch coverage results of over
95%. This is due to the integer unit being the central core for data fetching, decoding, and executing. The integer unit is one of the busiest cores in LEON3. On the other hand, the multiplier unit (mul32) has particularly poor coverage. At first, we thought that there was not a great deal of multiplication happening in either the simulation testbench, or during a Linux boot. But this turned out not to be true, in this case. Upon more detail analysis of the multiplier unit, and careful code inspection in that block, we realized that the poor coverage was just an artifact of the many different configuration variations embedded in this particular IP block. In practice, the irrelevant configurations would be manually excluded.

We expected that the divider unit (div32) exhibits similar results as the multiplier unit. However, we observed high coverage results from the divider unit (compared with the multiplier unit coverage results), which was surprising. We inspected the divider RTL code and realized that the divider unit has a simpler design than the multiplier unit. We also observed that even a few divisions exercised most of the divider design, which means the Linux boot prompts for a number of different divide operations.

Another significant observation is the overall higher branch coverage in post-silicon emulation compared with pre-silicon simulation. In emulation, there are much greater number of cycles — our Linux boot runs at 75 Mhz for 45 seconds, which translates to 3.4 billion cycles, while our simulation run takes only three thousand cycles (if we wanted to boot linux in simulation, it could take from about 39 days to 10 years depending on the detail and the speed of simulation [25]).
Another particularly significant result that we monitor is for MMU and video graphics array (VGA) blocks. Statement and branch coverages are over 85% in post silicon while in pre-silicon these numbers are 84% and 63% respectively. Obviously, Linux OS, practices more conditional branches of the MMU unit — branches were impossibly hard to cover in simulation. The VGA block, also, shows greater results in post-silicon. This is because there is not much great deal of exercise that can be done in simulation for the VGA block. In post-silicon, our board is interacting with monitor and so is sending and receiving data that practices the VGA unit.

Overall, we conclude that our experiments corroborate existing methodology: even booting the OS achieves high coverage, which explains why ad-hoc post-silicon tests can find many bugs. The coverage achieved pre-silicon is also high and generally very similar to that achieved post-silicon, which explains why pre-silicon simulations are successful in finding most bugs. But most importantly, the coverage is occasionally very low, and very different between pre- and post-silicon tests, which corresponds to escapes — bug escapes that are not detected in pre-silicon simulation.

### 3.4 Chapter Summary

In this chapter, we demonstrated code coverage for evaluating hardware validation effectiveness. We employed an industrial-size SoC and showed that as a proof-of-concept our coverage methodology is practical on such complex designs.

We also compared our post-silicon coverage results with our pre-silicon cov-
verage tests. This comparison shows that in some simulation tests, where corner cases are “hard-to-hit”, post-silicon tests can exercise those corner-cases on hardware and report high coverage results.
Chapter 4

Area Considerations

In this chapter, we investigate the area overhead that our coverage instrumentation imposes.

We also describe how we reduce the overhead using techniques from the software testing community.

4.1 Area Overhead Calculation and Discussion

We calculate the area overhead based on two different metrics: look-up tables (LUTs) and flip flops.

On FPGAs, LUTs are the best unit to report area. There are two main reasons for this: first, the LUT overhead is reported after design mapping and optimization. So, we are sure that there are no further changes due to routing or optimization to the current configuration of the synthesized hardware on FPGA. Secondly, the per-core area results on FPGA are collected based on LUTs. It is not
possible to get the per-core area results based on any other relevant FPGA components such as LUT-FF pairs or slices. The synthesis tool provides the LUT-FF area number only for the whole design. Slices are less representative of true area because they depend on how well the LUTs and flip flops were packed into the slices. For example, it can be possible to increase the overall number of LUTs by 10%, without increasing the number of slices by too much (by just using up space left in the slices already). Similarly, we believe that the number of LUTs better compares to the amount of area used for an ASIC design because ASIC designers do not have to worry about packing things into slices — most ASIC designs are standard-cell based with custom routings and metal fills, so they would not have the unused portions of slices.

Flip flop overhead is also important to us. It is captured after design compilation upon logic synthesis but before any mapping or optimization. There are two reasons why we report additional flip flops due to code coverage instrumentation: (1) at the beginning, when I started capturing area overhead, I realized that since we are adding bit variables for code coverage, it would be interesting to learn how much change there would be in terms of logic components. It turns out that we cannot get this report at the end of synthesis — the tool can provide only a summary of flip flops paired with LUTs for the whole chip. So, the only place we could capture those numbers per block was after HDL synthesis. (2) Moreover based on my experience working at PMC-Sierra, I observe that during layout of an ASIC chip, the number of logic components, including flip flops, is very important (not only for area but also for clock gating and power considerations) and
must be reported for every single block. I also observe that the number of digital components have a utilization limit on every block, since the dedicated area for each block must be strictly met to avoid routing congestion difficulties.

Table 4.1 shows our measurement results. The area overhead ranges from 9.6% to 134.7% for flip flops and 1.24% to 21.74% for LUTs. The large variance comes from code complexity versus the number of flip flops in the original design. For example, there are not many flip flops in the mmu block compared to its combinational circuitry. But, there are many branching in this block resulting in many basic blocks. Therefore, instrumenting all of its basic blocks exceeds the original number of flip flops that it had and results in large flip flop overhead of 134.7%. The iu3, on the other hand, originally contains a large number of flip flops. Instrumenting its basic blocks is not excessive compared to the large number of flip flops that it originally requires. Therefore, additional coverage flip flops only increase the number by 9% of the total flip flops in this core. Overall, as we can follow from Table 4.1 these numbers are large. We will investigate ways to reduce overhead results in Section 4.2.

The overhead numbers are unacceptable for the industrial deployment of code coverage — anecdotally, as I discussed with a variety of experts in industry, they often cite a rule of thumb that the entire area overhead allowed for test circuitry on chip is below 5% and in some cases below 1%. For example, the test circuitry inside Intel’s high performance (frequencies higher than 3GHz) microprocessors is less than three percent [30]. The area overhead for test circuitry inside AMD-K7 processors is 3.25% [31]. In theory, this threshold could conceivably be met
### Table 4.1: Area overhead of coverage monitors.

<table>
<thead>
<tr>
<th>IP Block</th>
<th>Additional Flip-flops</th>
<th>Additional LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2cmst</td>
<td>21.7%</td>
<td>3.53%</td>
</tr>
<tr>
<td>div32</td>
<td>31.0%</td>
<td>5.57%</td>
</tr>
<tr>
<td>mmutw</td>
<td>38.4%</td>
<td>21.74%</td>
</tr>
<tr>
<td>mul32</td>
<td>65.0%</td>
<td>6.25%</td>
</tr>
<tr>
<td>uart</td>
<td>60.0%</td>
<td>18.58%</td>
</tr>
<tr>
<td>mmutlb</td>
<td>61.4%</td>
<td>4.53%</td>
</tr>
<tr>
<td>svgactrl</td>
<td>21.9%</td>
<td>12.85%</td>
</tr>
<tr>
<td>mmu</td>
<td>134.7%</td>
<td>4.18%</td>
</tr>
<tr>
<td>iu3</td>
<td>9.6%</td>
<td>1.24%</td>
</tr>
</tbody>
</table>

on four of our nine instrumented IP cores. The LUT report shows that four IP blocks have below 5% overhead. But overhead in the mmutw, uart, and svgactrl blocks is over 10%. Note that this is just for coverage bits which is part of the entire test circuitry on chip. Therefore, it is hard for us to believe that the overall area overhead for instrumenting code coverage on the whole chip would be below the desired number.
4.2 Area Overhead Reduction

Given the unacceptably high overhead we observed, the next phase of this thesis is to investigate the possible overhead reduction by adapting state-of-the-art techniques from software analysis. We know that code coverage is a long-established concept in the software testing community. There is plenty of research done in reducing the instrumentation overhead for computing code coverage, i.e., in [13], [32], [33], [34], and [35]. In particular, the best optimization solution known for this problem is due to Agrawal [13]. The method analyzes the control flow graph to deduce where certain coverage monitor flags imply other flags, thereby allowing the elimination of flags yet preserving the same coverage information.

We describe the approach in more detail next. Our explanation uses the same example control flow graph from Section 3.1. For ease of following our discussion, we reproduce Figure 3.1 here in Figure 4.1.

4.2.1 Overhead Reduction Using Agrawal’s Method

The Agrawal algorithm relies on the concepts of pre- and post-dominance. These are graph-theoretic relations, assuming specific entry and exit nodes in the control flow graph.

Definition 2. A node \( r \) pre-dominates a node \( s \) if every path from entry \( [1] \) to \( s \) passes through \( r \).

Definition 3. A node \( t \) post-dominates a node \( s \) if every path from \( s \) to exit \( [2] \) passes

\(^1\) By entry node, in HDL, we mean the beginning of a process.
\(^2\) By exit node, in HDL, we mean the end of a process.
module example
    initial
        begin
            s1;
            if (s2)
                case (s3)
                    when s4
                        if (s5)
                            s6;
                        endif;
                    when s7
                        s8;
                    when s9
                        if (s10)
                            s11;
                        endif;
                    endcase;
            endif;
            s12;
        endmodule;

**Figure 4.1:** Example of HDL code with corresponding control flow graph.

The pre-dominance and post-dominance relations are each partial orders [36], i.e. they are (1) reflexive: every node dominates itself; (2) transitive: if r (pre/post) dominates s, and s (pre/post) dominates t, then r (pre/post) dominates t; and (3) anti-symmetric: if r (pre/post) dominates s and s (pre/post) dominates r, then r = s.

For example, in Figure 4.1, node 3 pre-dominates itself (reflexive) and node 7. Node 7 pre-dominates node 8. So node 3 holds the transitive property to node 8. On the other hand, as an example of post-dominance relations, node 7 post-dominates itself but does not post-dominate node 3, because if we look at our control flow graph in Figure 4.1 there are other paths to go from 3 to the exit. Node
3 is post-dominated only by node 12. Another interesting example is node 12 itself. This node is pre-dominated only by nodes 1 and 2. But, reversely, this node post-dominates all nodes.

Additionally, by the definition of the dominance relations, it turns out that the set of (pre/post) dominators of a node are linearly ordered [37]. Therefore, every node except the entry/exit node, has only one immediate (pre/post) dominator. As a result, it is possible to represent the (pre/post) dominator relation as a tree [38]. Figure 4.2 shows the dominator trees for our example. Lengauer and Tarjan devised a fast algorithm to obtain these trees by carrying out depth-first search on the control flow graph [39].

We can reduce our coverage instrumentation points by looking at the pre- and
post-dominator trees. In the pre-dominator tree, if we cover a node, it means that we already have covered the pre-dominators of that node. Similarly, in the post-dominator tree, if we cover a node, it means that we already have covered the post-dominators of that node. As a result, if we instrument only the dominated nodes, covering all those nodes means that we have covered all nodes in the control flow graph. However, not covering a dominated node, we have no information whether its (pre/post) dominators are covered or not. For example, in Figure 4.1, if we cover node 8, then from the pre-dominator tree in Figure 4.2, we can tell that nodes 7, 3, 2, and 1 are covered, and from the post-dominator tree we can tell that node 12 is covered. But, if we do not cover node 8, we in general do not know whether those nodes in the (pre/post) dominator tree are covered or not — i.e. node 3 may or may not be covered. Next, we show that how we resolve this issue.

Agrawal’s algorithm merges the pre- and post-dominator trees to obtain the basic block dominator graph. Figure 4.3 shows the resulting graph. We use the basic block dominator graph to get better optimization for reduced instrumentation of our coverage points. In this graph, we search for strongly connected components (SCCs)\(^3\), e.g., in Figure 4.3, nodes 1, 2, and 12 form a strongly connected component. We merge all the nodes in each strongly connected component, resulting in the graph for our example shown in Figure 4.4. The group of basic blocks in a strongly connected component is called a “Super Block”. We benefit from the fact that each basic block in a super block dominates all the basic blocks in the group.

\(^3\) A strongly connected component contains a set of nodes that from any of them, we can travel to the rest and come back to our initial node.
and so this implies that we need only one coverage flag per super block to achieve the same coverage accuracy as if we had flagged all nodes in the super block.

There is still more overhead reduction possible. Agrawal’s algorithm, simplifies the graph by removing composite edges, i.e., an edge from $u$ to $v$ is removed.
if there is an alternative path from $u$ to $v$. The resulting graph is dubbed the super block dominator graph. (See Figure 4.5.) On this graph, we can expect a similar coverage relationship between the nodes as we observed for the (pre/post) dominator tree [13]:

If a child of a node is covered, that node is covered by the same test.

For example, hitting 11 guarantees that we also covered 1, 2, 12, 3, 9, and 10. Unfortunately, the converse is not always true — if we do not hit 11, we may or may not have hit the dominating blocks. For example, looking back on our control flow graph in Figure 4.1, imagine an execution that runs through nodes 1, 2, 3, 9, 10, and 12. In such a run, node 11 is not hit. That is, if we had instrumented node 11 only, we would not have any coverage information. As a result of this, we would need to know instrumenting which nodes gives us true coverage.

Agrawal’s algorithm solves this issue by working with two main graphs: superblock dominator graph (Figure 4.5) and control flow graph (Figure 4.1). With
Agrawal’s general rule for coverage instrumentation.

Figure 4.6: Agrawal’s general rule for coverage instrumentation.
these two graphs, Agrawal’s general rule (Figure 4.6) says if we choose a node in the super block dominator graph, we have to go back and check that node in the control flow graph. In there, we have to check whether there is any path that goes through that node without going through the children of that node in the super block dominator graph. If the answer is yes, we need to leave our coverage bit in that node. Otherwise, we may remove the bit to save area and yet preserve the coverage data accuracy. For example, let’s choose superblock node 3 in the super block dominator graph. This node has three children: nodes \{4,5\}, \{7,8\}, and \{9,10\}. Now, if we check node 3 in the control flow graph, we see that there is no path from entry to exit that passes through node 3 and not at least some of the nodes 4, 5, 7, 8, 9, and 10. Therefore, we may remove the coverage bit from this node. On the other hand, let’s look at the superblock node \{4,5\}. This node has only one child: node 6. If we go back to the control flow graph, we see that there is a path from entry to exit that goes through nodes 4 and 5 and not the child, node 6. Therefore, we should leave a coverage bit in the superblock node \{4,5\}.

We do this check for every super block in the superblock dominator graph, i.e., in our example, we do this check seven times because there are seven super blocks.

However, since the general rule is computationally expensive, Agrawal finds two exceptional cases that guarantee the expensive check will fail. Therefore, we do not need to go back and check the control flow graph. We can decide we must instrument those nodes by just looking at the super block dominator graph. Here are the two cases:

1. Leaves: these nodes do not have children. So the general rule of Agrawal
cannot apply to leaves. Therefore, leaves in the superblock dominator graph must be instrumented.

2. One-child nodes: similar to leaves, the general rule of Agrawal fails for these nodes. Therefore, there is no need for the expensive check and these nodes must be instrumented. To prove this, let’s assume the check passes, i.e., we do not need to instrument a one-child nodes. That means for the one-child node, there exists no path that does not go through the child of that one-child node while going through that one-child node. In other words, all paths from entry to exit that run through the one-child node, go through the child of that one-child node. This implies that the one-child node pre-dominates the child node. It also implies that the child node post-dominates its parent. Therefore, the two must be in one superblock node, which they are not, and this is a contradiction. Thus in a superblock dominator graph one-child nodes must be instrumented.

To better understand the discussion above, we present a simpler example. Consider the simple code and its control flow graph in Figure 4.7. The entry is to node 1 and the exit is from node 4. There are two possibilities to traverse from node 1 to node 4 (and exiting), which are via nodes 2 and 3. Figure 4.8 draws the pre- and post-dominance trees of this simple example. We merge the two trees to obtain the basic block dominator graph. Afterwards, we merge the strongly connected components and eliminate composite edges. The resulting graph is shown in Figure 4.9. According to the final graph obtained, there are two leaves and no
module example
...
always @ (posedge clk)
begin
  if(s1) then
    s2;
  else
    s3;
  endif;
  s4;
endmodule;

Figure 4.7: Simple example control flow graph.

Figure 4.8: Simple example pre- and post-dominator trees.

one-child node. So, the leaves must be instrumented, and if one of them is cov-
ered, we know right away that node 1 and node 4 have also been covered. But
if none of the leaves are covered, we cannot claim that node 1 or node 4 is not
covered, too. So far, we have saved two expensive checks out of four while we
continue applying the general rule of Agrawal. The only node that is left on the
superblock dominator graph is node \{1,4\}. We check back on the control flow
graph for this node and see whether there is a path from entry to exit that runs
through node 1 and node 4, but not the children of these nodes. The answer is no, because all paths in the control flow graph that go through node 1 and node 4, pass through one of their children. Therefore, node 1 and node 4 need not to be instrumented. We see that in this simple example, we have fifty percent area overhead savings.

Let’s get back to our main example in Figure 4.5. If we instrument basic blocks 1, 4, 6, 7, 9, and 11, we can reconstruct the true coverage information, exactly as if we had instrumented every basic block of Figure 4.1.

### 4.2.2 Results

Table 4.2 shows the area overhead reduction we achieve using Agrawal’s technique. Since the method reduces only the number of monitor flags, the reduction in flip-flops is larger than the overall area reduction that is based on FPGA LUTs. Still, for all IP blocks except one, the area overhead was reduced. In the exceptional case, in the smallest block, i2cmst, the area overhead increased by a single
look-up table, likely due to tool flow variation. The Xilinx synthesis tool use non-deterministic/probabilistic techniques which are not guaranteed to find the same solution. This becomes apparent in i2cmst. Because this block is the smallest block, we could remove only one coverage variable out. But it happened that the tool added a LUT for this small block.

Overall, the reduction ratios vary from 0.47 to 0.94 for flip flops and 0.72 to 1.17 for LUTs. Even with state-of-the-art software analysis techniques, the code coverage monitoring overheads for many blocks is still excessive for most market segments, which suggests that there is a need for exploiting better reduction methodologies to reduce unacceptably large area overhead for post-silicon code coverage.

### 4.3 Chapter Summary

In this chapter, we calculated the area overhead imposed by our code coverage instrumentation on-chip. The results obtained show that code coverage imposes large area overhead on our experimental industrial-scale SoC.

Furthermore, we employed the best known software technique to reduce the unacceptably large area overhead yet preserve post-silicon code coverage data accuracy. Still, the area overhead is large and requires more investigation to be deployable on industrial designs.
Table 4.2: Overhead reduction results.

<table>
<thead>
<tr>
<th>IP Block</th>
<th>Flip-Flop Overhead</th>
<th></th>
<th>LUT Overhead</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>Agrawal</td>
<td>Ratio</td>
<td>Baseline</td>
</tr>
<tr>
<td>i2cmst</td>
<td>21.7%</td>
<td>20.3%</td>
<td>0.94x</td>
<td>3.5%</td>
</tr>
<tr>
<td>div32</td>
<td>31.0%</td>
<td>21.4%</td>
<td>0.69x</td>
<td>5.6%</td>
</tr>
<tr>
<td>mmutw</td>
<td>38.4%</td>
<td>32.9%</td>
<td>0.86x</td>
<td>21.7%</td>
</tr>
<tr>
<td>mul32</td>
<td>65.0%</td>
<td>52.5%</td>
<td>0.81x</td>
<td>6.3%</td>
</tr>
<tr>
<td>uart</td>
<td>60.0%</td>
<td>45.6%</td>
<td>0.76x</td>
<td>18.6%</td>
</tr>
<tr>
<td>mmutlb</td>
<td>61.4%</td>
<td>45.5%</td>
<td>0.74x</td>
<td>4.5%</td>
</tr>
<tr>
<td>svgactrl</td>
<td>21.9%</td>
<td>17.1%</td>
<td>0.78x</td>
<td>12.9%</td>
</tr>
<tr>
<td>mmu</td>
<td>134.7%</td>
<td>63.0%</td>
<td>0.47x</td>
<td>4.2%</td>
</tr>
<tr>
<td>iu3</td>
<td>9.6%</td>
<td>6.7%</td>
<td>0.70x</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis considers the challenges of post-silicon coverage. It gives a reasonable solution to the challenge that accounts for observability and controllability limitations in hardware validation. We successfully obtained real data on the effectiveness of a typical validation task (booting the OS) for a (small) industrial-scale SoC.

The results show: (1) The typical test of booting the OS often achieves high coverage, similar to what is achieved by pre-silicon directed tests, but in some blocks, the coverage is markedly different. This is exactly the main purpose of post-silicon validation in general, and post-silicon coverage measurement, in particular. (2) Code coverage is implementable in post-silicon. However, it imposes large area overhead for their coverage monitors — the area overhead of the cov-
verage monitoring instrumentation in one block reaching 22%. (3) State-of-the-art software analysis techniques reduce the overhead substantially (e.g., a 15% reduction in uart, which was a block with 18.6% overhead), but the remaining overhead is still unacceptably high for practical deployment.

To conclude, in this thesis, the simulation and on-chip coverage experiments proved that code coverage can be used as a standard coverage technique for both pre-silicon verification and post-silicon validation. In pre-silicon, almost all simulation tools have the code coverage feature. For post-silicon, code coverage measures the information accurately with specific on-chip monitors (while with reduced area overhead). Therefore, we can employ code coverage to capture bugs in simulation, but, more importantly, we employ code coverage for bugs escaping into silicon that failed to catch in pre-silicon (in some industrial cases the resulting escapes are extremely expensive). Code coverage satisfies the need in validation and is a solid baseline for post-silicon coverage research.

5.2 Future Work

The following areas are potentially interesting avenues to extend this thesis work:

- First, we believe there is promise for further overhead reduction via more expensive analyses. In the world of software test, the overhead of coverage monitoring is a minor nuisance, so it is not acceptable to do a costly analysis to reduce it further. As a result, techniques from software analysis are very lightweight, analyzing only graph properties of the control-flow graph. In contrast, for post-silicon coverage, a costly analysis that was effective in
reducing overhead would be a worthwhile trade-off.

- A more applied, practical direction for further research is to investigate extremely low-overhead monitoring techniques (e.g., monitoring only very few points such as particularly “hard-to-hit” monitors), and then use our current results to validate the more practical coverage model. If we can get a monitoring scheme with extremely low overhead, yet that correlates well with true post-silicon code coverage, that would be very valuable.

- Finally, it would be valuable to extend these results to other code coverage metrics such as condition coverage and expression coverage. It would also be valuable to compare these results with other coverage models (e.g., path coverage, assertion coverage, etc.).
Bibliography


[4] Technical Communication with PMC-Sierra RTL design team. The product information can be found online


79

→ pages 23


→ pages 34, 35, 36, 37, 38, 42, 47, 50


80
[29] Technical Communication with PMC-Sierra validation team, November 2011. \(\rightarrow\) pages 51


