# FPGA Emulation for Critical-Path Coverage Analysis

by

Kyle Balston

B.A.Sc., Simon Fraser University, 2010

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

**Master of Applied Science**

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

October 2012

# Abstract

A major task in post-silicon validation is timing validation: it can be incredibly difficult to ensure a new chip meets timing goals. Post-silicon validation is the first opportunity to check timing with real silicon under actual operating conditions and workloads. However, post-silicon tests suffer from low observability, making it difficult to properly quantify test quality for the long-running random and directed system-level tests that are typical in post-silicon. In this thesis, we propose a technique for measuring the quality of long-running system-level tests used for timing coverage through the use of on-chip path monitors to be used with FPGA emulation. We demonstrate our technique on a non-trivial SoC, measuring the coverage of 2048 paths (selected as most critical by static timing analysis) achieved by some pre-silicon system-level tests, a number of well-known benchmarks, booting Linux, and executing randomly generated programs. The results show that the technique is feasible, with area and timing overheads acceptable for pre-silicon FPGA emulation.

# Preface

Preliminary work related to this thesis has been published in a conference paper and a journal paper. The work itself will be published as an invited conference paper. The first paper, Post-Silicon Code Coverage Evaluation with Reduced Area Overhead for Functional Verification of SoC, was published as [33]. In this paper we manually added on-chip monitors to extract statement and branch coverage information from a uniprocessor LEON3 system running on a Xilinx FPGA. Mehdi Karimibiuki implemented the coverage monitors as well as designed and carried out the experiments with the help of Kyle Balston. The paper was written collaboratively with the help of Dr. Alan Hu and Dr. André Ivanov.

The second paper, Post-Silicon Code Coverage for Multiprocessor System-on-Chip Designs, was published as [6]. In this paper we extended [33] to target a multiprocessor LEON3 system and collected data showing how statement and branch coverage was achieved with respected to processor cycle count. For this paper, Kyle Balston took the lead on the design and execution of the experiments. The paper was written collaboratively by Kyle Balston and Mehdi Karimibiuki with the help of Dr. Alan Hu, Dr. André Ivanov and Dr. Steve Wilton.

The work that forms the basis of this thesis will be presented in a third paper, Emulation in Post-Silicon Validation: It's Not Just for Functionality Anymore [5]. Kyle Balston performed the research, data collection, data analysis, and writing under the supervision of Dr. Steve Wilton and Dr. Alan Hu with industry guidance from Amir Nahir of IBM.

# Table of Contents

# List of Tables

# List of Figures

# Glossary

| | |
|---|---|
| **ASIC** | Application-Specific Integrated Circuit |
| **BRAM** | Block Ram |
| **DRC** | Design Rule Check |
| **DSP** | Digital Signal Processor |
| **FPGA** | Field-Programmable Gate Array |
| **IC** | Integrated Circuit |
| **LUT** | Lookup Table |
| **NCD** | Native Circuit Description |
| **SoC** | System-on-Chip |
| **SSTA** | Statistical Static Timing Analysis |
| **STA** | Static Timing Analysis |

|  |  |
|---|---|
| **VHDL** | VHSIC hardware description language |
| **XDL** | Xilinx Description Language |

# Chapter 1

# Introduction

## 1.1 Motivation

Integrated Circuit (IC) complexity continues to increase as predicted by Moore's Law [52], allowing chip designers to deliver more functionality on a given die, at lower manufacturing cost and at smaller power windows. However, it also makes it increasingly difficult to detect and fix design errors simply due to the sheer number of transistors on a single chip. Today, modern chips contain upwards of 3 billion transistors [58], which together can form an entire computing system, often called a System-on-Chip (SoC) design. Before being fabricated, these multi-billion transistor SoC designs are verified to be correct in model-based verification without a physical chip, known as pre-silicon verification, and after fabrication, with a release-candidate physical chip in post-silicon verification. Together these two styles of testing dictate when a chip is ready to go to manufacturing, and into the hands of consumers. However, proper testing of these chips is imperative as even a single error escaping to manufacturing can cost a company millions in product recalls and irreparably tarnish a company's reputation.

## 1.2 Historic Design Errors

One of the most well-known examples of a design error that made it into a consumer processor is the infamous FDIV bug in Intel's P5 microarchitecture [57]. The bug caused certain floating-point operations and operand combinations to produce incorrect results (e.g., the result returned for $\frac{4195835}{315727}$ was incorrect beyond four digits). Unfortunately, the bug was not discovered until the processor was in widespread use and only exhibited errors under rare conditions (approximately 1 in 9 billion operand pairs would result in an error at double-precision [14]). Because of this, Intel was initially hesitant to issue product recalls until it became highly publicized. In total, this cost Intel an estimated \$475 million [68]. However the damage to their reputation was also substantial (particularly among the scientific computing community). More recently, Intel's 2011 recall of the Cougar Point chipset due to an issue with its SATA ports cost Intel an estimated \$700 million [20]. AMD, another large semiconductor company, is not without its own share of issues. In 2007, an error in the translation lookaside buffer (TLB) of the B2 stepping of their Barcelona processors could lead to system lockup [16], until AMD disabled it altogether in a BIOS update, causing some workloads to experience a 5-20% performance penalty. These design errors escaped to production, costing many times more than if they were detected during pre- or post-silicon validation.

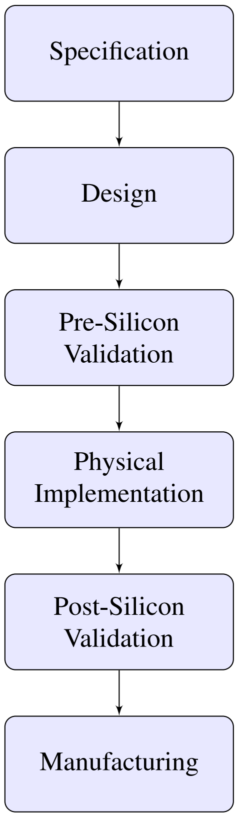


**Figure 1.1:** ASIC design flow.

## 1.3 ASIC Design Flow

To illustrate how integrated circuit designers verify their designs and identify design errors, this section gives a high-level overview of the Application-Specific Integrated Circuit (ASIC) design and verification process. A more in-depth analysis of the ASIC flow can be found in Section 2.1.

An ASIC design typically starts with specifying the functional and non-functional properties of the design, shown as 'Specification' in Figure 2.1. Note that, at this stage, a module's underlying implementation details are not specified.

Specification is often followed by the implementation stage, shown as 'Design' in Figure 2.1. In the design stage, designers write code (usually VHDL or Verilog) to implement the design or module while conforming to the specification detailed in the previous stage.

Throughout the implementation process, pre-silicon validation is performed at the block level. Once the design has been fully implemented, top-level pre-silicon validation is started, shown as 'Pre-Silicon Validation' in Figure 2.1. Ideally, all design errors would be discovered at or before this stage, however this is often impossible to accomplish. Pre-silicon validation makes use of simulation, static timing analysis, formal verification, and emulation to verify the correctness of the design. Pre-silicon simulation is performed throughout the ASIC flow: simulating the hardware description language directly (RTL simulation), after synthesis (gate simulation) and after place and route (timing simulation). In general, RTL-level simulation is used more extensively than other forms of simulation because it runs quicker and can more easily determine the root cause of detected design errors. However, lower level simulations such as timing simulation are also common because they can more accurately model the system in exchange for reduced throughput. Formal verification techniques can be used to mathematically prove a property of a design, however these techniques suffer from scalability problems for large designs. Emulation uses prefabricated programmable hardware devices such as Field-Programmable Gate Arrays (FPGAs) to act as a stand-in for the fabricated chip. These devices run orders of magnitude faster than simulation but do not provide full visibility of internal signals [28, 46]. These pre-silicon techniques are further detailed in Section 2.1.2.

The physical implementation stage follows, where the placement and routing of a design is determined. This physical description is then given to a foundry, which produces a number of silicon prototypes.

Post-silicon validation tests the prototype chips at normal operating speed under expected operating conditions. Unlike simulation, post-silicon techniques generally suffer from poor visibility into the system state during execution. Since chips which are physically larger are more expensive to fabricate, on-chip monitoring logic must be minimized in order to reduce cost. Often it takes more than one silicon iteration to produce a production-ready chip.

Once the chip passes post-silicon validation, it is ready to be sent to manufacturing for mass production. Each produced chip is subject to manufacturing tests to ensure each chip is functional and speed binning to categorize produced chips by their maximum operating frequency. Manufacturing tests check for fabrication issues related to specific dies (e.g., manufacturing defects) and not systemic issues (e.g., design errors).

### 1.3.1 Differences Between Pre- and Post-Silicon Validation

In general, pre-silicon techniques are slow and/or approximate but have high observability into the system state. Conversely, post-silicon techniques have very

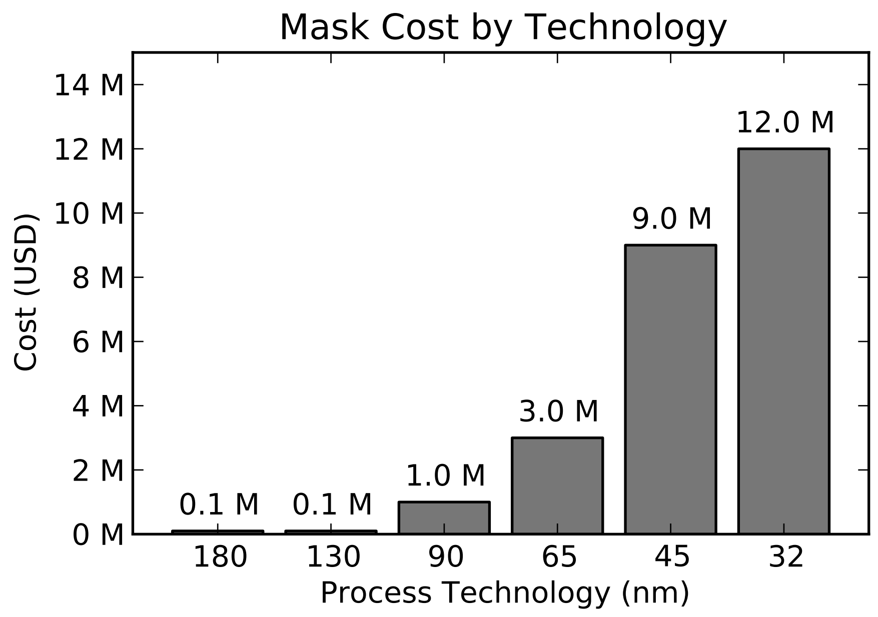


**Figure 1.2:** Mask production cost by process technology [29].

low visibility into the system state but run at or near the design's native operating frequency.

Unfortunately, relying on post-silicon validation to find bugs is expensive due to increasing mask development costs. As shown in Figure 1.2, mask costs now represent a significant cost to IC development. Worse still, typical fabrication processes can easily take four to eight weeks to complete [44], slowing the throughput of post-silicon validation and increasing time to market. To mitigate these costs, it is important to perform as much validation as possible in pre-silicon (before fabrication) and to maximize the number of design errors found with each iteration of the fabricated chip. Commercial tools are available to accelerate this

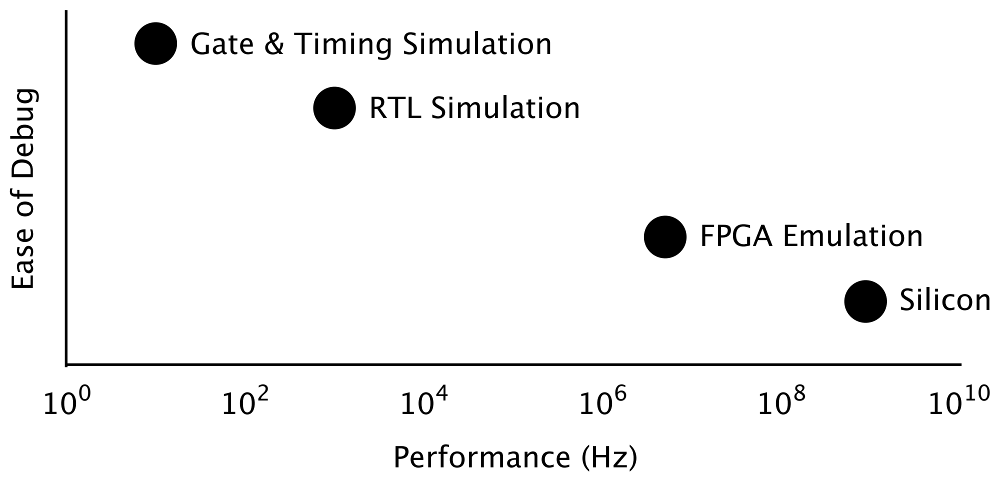


**Figure 1.3:** Ease of debug and performance comparison of validation techniques (performance data from [23]).

process [3, 62, 71].

### 1.3.2 Emulation — Bridging Pre- and Post-Silicon Validation

Emulation has become an important technology for the validation of large ICs. Although designers make extensive use of software simulation when verifying their designs, simulation runs approximately eight orders of magnitude slower than actual silicon [53]. At this speed it would take more than three years to simulate just one second of silicon execution. For example, during pre-silicon validation of the Pentium 4 processor, Intel found that in a two year period between initial RTL system-level integration and A-step tapeout, their compute farm (which by the end of the project was several thousand nodes large) had accumulated only 230 billion simulation cycles [9], approximately two minutes of execution on a

1.5 GHz processor.

### 1.3.3 FPGA Devices

FPGAs are ICs that have been fabricated to contain flexible programmable logic. This programmable logic allows FPGAs to implement arbitrary logic, albeit at a slower speed and higher unit cost (at high volume production) than can be provided by a fixed-function IC. In contrast, they do not require lengthy fabrication turnaround times or expensive mask costs, making them ideal for low-volume applications. As fixed-function ASIC validation has become more costly, one such low-volume application for FPGAs has emerged: FPGA-based emulation of ASICs for validation and prototyping purposes.

### 1.3.4 FPGA-based Emulation

FPGA-based emulation technology allows for much higher validation coverage in pre-silicon testing by allowing designers to re-target their designs to one or more FPGAs and 'execute' the design at orders of magnitude faster than simulation. For example, Intel's Atom processor (composed of 47.2 million transistors) was mapped to a single FPGA running at 50 MHz [66], and their performance-oriented i7 Nehalem processor (transistor count ranges from 177 to 1170 million transistors [17]) was mapped to five FPGAs running at 520 kHz [60]. While slower than the fixed-function chip, these speeds are fast enough to perform common validation tasks such as booting an operating system — tasks which are infeasible in simulation. In fact, even at 520 kHz, all 230 billion simulation cycles accumulated before tapeout for the Pentium 4 processor could be covered in just over five days by a single FPGA emulation system.

Because the underlying implementation is so different between the integrated circuit and the FPGA-based emulation, emulation typically captures only the functional behaviour of the design. Due to this difference, FPGA emulation is not typically appropriate for designs which make heavy use of asynchronous or analog circuits. When designers re-map their digital circuits to one or more FPGAs, they must compile their designs using the FPGA vendor's tools, which may make different optimization decisions than the ASIC synthesis tools. In addition, the pre-fabricated nature of an FPGA means that certain structures (e.g., memories, multipliers) may be implemented very differently on an FPGA than on an ASIC. Furthermore, the physical implementation of a Lookup Table (LUT) is radically different from a standard cell or full-custom gate; this means important properties like timing, critical paths, robustness to process and environmental variation, noise margins, etc. are all different in the emulated design.

An obvious example of such a property is timing. Designers often need to know the maximum operating frequency of the chip and if there is any way of increasing performance by modifying the frequency or shortening critical paths that limit speed improvements. Designers regularly employ pre-fabrication timing verification techniques such as Static Timing Analysis (STA) and Statistical Static Timing Analysis (SSTA). These techniques provide valuable information regarding potential timing problems in a design, but are only approximate — the paths predicted to be critical do not correlate perfectly with the true critical paths [7], hence the need for expensive (but also more accurate) post-silicon timing validation [51].

Many of the aforementioned physical properties (e.g., tolerance to voltage and temperature variations, accelerated aging, etc.) can be tested via timing tests, by varying some parameter and measuring at what frequencies the chip still behaves correctly (e.g., the classic 'shmoo' plot obtained by running the chip at different voltages and measuring what frequencies give correct behaviour). This indicates that post-silicon tests related to timing are useful for covering many types of properties that are not well tested in pre-silicon.

### 1.3.5 Conventional Emulation Goals

Traditionally, emulation has only been used for validation of functional behaviour; the conventional wisdom dictates that emulation cannot help with validating other properties, particularly properties which are the focus of post-silicon validation such as timing. In this dissertation, we show otherwise — emulation *can* be used as an important tool to assist validation of more than just functional behaviour. In particular, we focus on timing validation and the task of quantifying the *critical-path coverage* of a validation plan. We define the critical-path coverage of a validation plan as the proportion of the critical and near-critical circuit paths that are exercised during the execution of the plan. As will be discussed in Section 3.2, measuring this quantity is essential, both to guide the creation of suitable post-silicon validation tests, and to determine how thoroughly a design has been validated before it is put into high-volume production. Our technique involves adding a small amount of instrumentation (*coverage monitors*) in strategic locations. Rather than add monitors to the actual silicon (which would have significant area overhead), we add them to an FPGA-based emulation of the design. By measuring coverage during emulation, we can obtain a metric for the timing coverage of our verification plan when applied to the actual chip during post-silicon validation. A key challenge is that the critical paths of the original ASIC will likely be different than the critical paths of the emulated version.

Note that our approach does not replace post-silicon timing validation, but instead complements it by measuring how effectively it is being done. Nor are we performing timing analysis of FPGAs themselves [43], which would give little insight into the timing of the ASIC. Instead, we are measuring the quality of the post-silicon timing *tests*, by using FPGA emulation.

## 1.4 Research Objective and Contributions

To summarize the previous section, the goal of this research is to *show how FPGA-based emulation can be used to measure non-functional properties of integrated circuits, concentrating on critical-path delay coverage*. The main contributions of this dissertation are as follows:

1. We propose a general toolflow for using FPGA-based emulation to support

validation of post-silicon properties.

2. We instantiate a specific instance of this flow: using emulation to measure critical-path coverage of post-silicon validation tests.

3. We demonstrate this instance on a complex SoC using common post-silicon validation tests, including booting Linux and running long random instruction streams.

4. We report critical-path coverage data for these post-silicon validation tests.

5. We describe the academic tools developed to implement this methodology and release them for use by the research community.

## 1.5 Research Approach

To address the above objectives, we designed and implemented a system to automatically instrument the open-source LEON3 SoC design [2] with path coverage monitors for the paths deemed most critical by the Xilinx Timing Analyzer [73]. We ran various typical pre- and post-silicon tests against the SoC, and recorded the coverage reported by our coverage monitors.

Due to the limited interoperability of the Xilinx toolflow with non-standard processes (e.g., many of the Xilinx tools are not re-entrant so a partially placed-and-routed design cannot be integrated back into the toolflow), we developed a tool to generate a structural VHDL description from a placed-and-routed Xilinx design, allowing us to easily instrument paths described structurally by the STA tool for coverage. We integrated our coverage monitors as VHDL into this description and resynthesized the design with the Xilinx tools, allowing us to easily monitor signals specified by the Xilinx Timing Analyzer as timing critical. The tool allows for integration of arbitrary VHDL code alongside a structural description of the original design.

## 1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of background topics and related work. Chapter 3 describes the ideal coverage monitor instrumentation flow and gives an example of a coverage monitor targeting a critical path. Chapter 4 presents the toolflow used in this dissertation's research to approximate the ideal flow shown in Chapter 3 and details the XDL to VHDL converter developed to allow automatic integration of coverage monitors. Chapter 5 demonstrates the feasibility of coverage monitors in emulation systems by performing critical-path coverage instrumentation on a LEON3 SoC and reporting coverage achieved for common post-silicon tests. Chapter 6 concludes the dissertation by providing a summary and discusses limitations for the toolflow presented as well as future work that could be used to address these limitations.

# Chapter 2

# Background and Related Work

In this chapter, we first give a high level overview of the ASIC design flow in Section 2.1, followed by a discussion of pre- and post-silicon coverage metrics in Sections 2.2 and 2.3, respectively. Section 2.4 describes how the work done for this dissertation differs from previous work.

## 2.1 ASIC Design Flow

As discussed in Section 1.2, design errors in ASIC designs can be very costly. As such, verification is central to the ASIC design process; before a chip is sent to the foundry, it is imperative that it has been tested as thoroughly as possible. Ideally, the first fabricated iteration of the chip, ‘first silicon’, would be bug-free and be ready to send to manufacturing. Unfortunately, increasing chip complexity has caused the number of first silicon successes to decline from 39% in 2002, to

33% in 2004 and 28% in 2007 [69]. Estimates for the total amount of design time and resources spent on verification vary (55% [69], 80% [39], 90% [59]), but they all agree that verification (pre- and post-) is critical to the ASIC design flow and important for reducing the number of silicon iterations required before sending a design to manufacturing.

### 2.1.1 Stages of ASIC Design

This section gives a high-level overview of the ASIC design process detailed in Figure 2.1. Each stage of the design process is shown on the left in bold, followed by its description.

**Specification** details the functional and non-functional properties of the design. It should be at sufficient granularity to ‘pass’ or ‘fail’ a given HDL implementation without dictating how it must be implemented. For example, a specification could state that on every clock cycle, a module will take an `A` and `B` input, and output `C`, which is `A + B`.

**HDL** is a VHDL, Verilog or other hardware description of the specification. This stage is responsible for transforming the specification into a design by converting it to RTL code. It details how the specification is functionally implemented (but not the underlying transistor technology). For example, it could describe `C = A + B` as a simple ripple-carry adder, or a more complicated carry-lookahead adder.

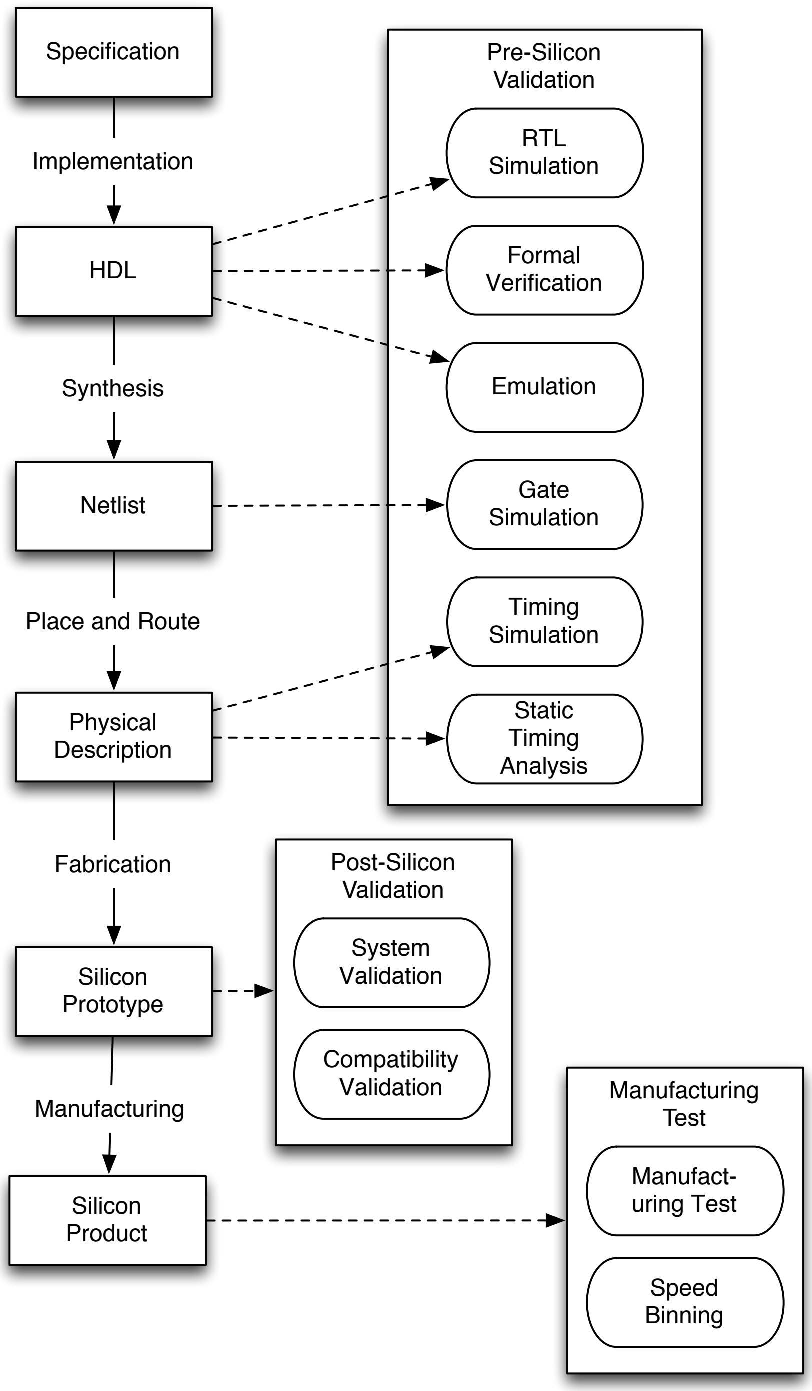


**Figure 2.1:** ASIC design flow.

**Netlist** files comprise the logic connections between a set of gates defined by a standard cell library. Synthesis converts the HDL code into these gates while preserving the functional behaviour of the RTL and continuing to satisfy design constraints.

**Physical Description** is usually divided into two main stages: placement and routing. The placer determines the placement of standard cells, described by the netlist, onto a region representing the ASIC. The router determines a set of connections between these placed cells. The resulting layout will be validated with a Design Rule Check (DRC) and other physical verification checks to ensure it conforms to the foundry's design rules.

**Pre-Silicon Validation** ensures the design is functionally correct using a combination of simulation, formal verification and emulation-based techniques. Pre-silicon validation is discussed in more detail in Section 2.1.2.

**Silicon Prototype** chips are received from the foundry for use with post-silicon validation. Fabrication is an expensive and time-consuming process, costing upwards of $10 million USD for a modern 32 nm process (as shown in Figure 1.2), making it imperative that the number of silicon iterations be minimized.

**Post-Silicon Validation** involves validating a hardware design after fabrication, but before large-scale manufacturing, with a physical chip. Post-silicon validation generally has low visibility into system signals. However, it runs at full chip execution speed and can validate all aspects of the design (since the chip is a release-candidate). Post-silicon validation is discussed in more detail in Section 2.1.3.

**Silicon Product** is the final product that will be shipped to end-users.

**Manufacturing Test** includes conventional manufacturing test, which checks for defects caused by the manufacturing process; and speed binning, which categorizes chips based on their performance characteristics. These two types of manufacturing test are described in further detail in Section 2.1.4.

### 2.1.2 Pre-Silicon Validation

Pre-silicon validation is the problem of finding design errors without using a physical chip. It forms the backbone of IC validation and is responsible for detecting most design bugs. The most common forms of pre-silicon testing are simulation, formal verification and emulation.

#### Simulation

Simulation is employed to validate all but the simplest designs (and usually even those too). It involves supplying a series of test vectors, often generated from constrained random parameters, to a testbench that will verify functional correctness of a design (or a design's modules) compared to the designer's expected output. It allows for complete visibility into all signals of the design and can detect a deviation from expected top-level ports and internal signal values the instant they occur.

Complete visibility allows for full signal traces leading up to any deviation. These traces make diagnosing and fixing bugs much easier than it would be otherwise (and indeed, this visibility is a primary advantage of pre-silicon techniques over post-silicon). However, simulation is very slow compared to real-chip execution, requiring tests to be very short and therefore not representative of real-world use. Furthermore, simulation models are usually slow and/or approximate for complex electrical interactions and timing behaviours.

There are three main types of simulation: RTL, gate, and timing. RTL simulation of the design ensures the implemented design functionally conforms to the specification by simulating the HDL written by designers. It allows for full visibility into system signals during execution but runs much slower than real-chip execution and does not take into account detailed implementation information (such as timing). Gate simulation ensures that the implementation inferred by the synthesis tools still conforms to the specification, and is functionally equivalent to the functional implementation. It allows for full visibility into system signals during execution (however these gate-level signals can be non-trivial to map back to the original RTL signals) but runs slower than RTL simulation. Timing simulation ensures that the placed-and-routed implementation of the design conforms to the specification under realistic timing conditions. It allows for full visibility into system signals during execution (however as in gate-level simulation, these signals can be non-trivial to map back to the original RTL signals). Timing simulations runs slower than gate-level simulation because it includes additional timing, placement and routing information. In all cases, these techniques are only approximate and may not match the behaviour seen on the physical chip.

Coverage, as will be discussed in Section 2.2, is integral to simulation. Achieving coverage closure, the point at which 100% of one's coverage goals[1] have been met, is one of the most important milestones for the pre-silicon verification team [1]. This exemplifies how coverage-driven verification is already understood to be an important part of silicon validation.

### Formal Verification

Formal verification is the process of validating a design by ensuring design intent (expressed through specifications) is preserved through to implementation [19, 22, 34, 54]. Once a design or block is successfully verified (as in [32, 37]), it is mathematically guaranteed to satisfy the validated properties. Before a complicated design (or perhaps a module in the design) is formally verified, it has to be converted into a simpler, verifiable format which is representative of the unit as a whole. However, even when the design is converted to this verifiable format, formal verification still often suffers from capacity limits. Furthermore, there are often questions about the completeness of the specifications and accuracy of the constraints.

[1]This is not necessarily the same as reaching 100% coverage which, depending on the circuit design and coverage metric, may be impossible.

### Static Timing Analysis

Static Timing Analysis (STA) is a method for computing the worst- and best-case timing performance of a circuit without performing rigorous circuit simulation [27, 36, 40, 48, 50]. STA relies on conservative delay modelling of gates and interconnect valid for a range of operating conditions and die variation. It allows the analysis to scale with the size of the circuit, however it relies heavily of the accuracy of the delay models and cannot determine if detected paths are false (i.e., it cannot determine if some of the detected paths cannot occur during circuit operation) [10].

### Emulation

As pre- and post-silicon validation costs continue to increase [30, 35, 56], emulation has steadily increased in popularity. In modern IC development, it is common for a design to be converted to run on an FPGA for testing and prototyping purposes (e.g., [24, 47, 60, 64, 65, 66]). The development of the FPGA system is usually supported by commercial tools for creating on-chip debugging infrastructure (e.g., [3, 63, 71]). This FPGA design will run orders of magnitude faster than simulation, usually quickly enough to be plugged into a working system in place of the not-yet-fabricated IC. Here it can be validated against live data from other devices and long-running system-level tests which are not feasible in simulation.

However, FPGA emulation is not without limitations. While ASIC and FPGA designs share a common language to describe logic (e.g., VHDL and Verilog), there are many ASIC constructs which do not map well to FPGAs (and vice-versa). These differences require the HDL of an ASIC design to be modified before it can target an FPGA (as described in [60, 66]). If not done carefully, this may change functional behaviour of the design and substantially reduce the value of emulation.

### 2.1.3 Post-Silicon Validation

Obviously, all errors are ideally caught before fabrication, but as discussed in Section 2.1.2, pre-silicon validation has a few fundamental limitations. The biggest of these, pre-silicon execution speed compared to real-chip execution and the difficulty of properly modelling complex electrical effects, can be addressed by post-silicon validation where designers test their chips post-fabrication, *in situ*, allowing for much more extensive and realistic tests [11, 30, 51].

However, post-silicon validation is not without its own limitations: it has challenges related to observability and controllability of internal signals. Commercial tools have emerged to aid in post-silicon debug at modest area overhead (e.g., [62]), however there is still room for improvement. Limitations in post-silicon validation are in large part due to cost-savings; ideally no debug or validation logic would be present on the final chip since it does not add any value for the user and adds to the overall fabrication cost of the chip (by requiring additional chip area) though in some cases this additional cost is seen as necessary for risk mitigation. Unfortunately, even with modern pre-silicon techniques, at least some debug and validation logic is required to enable and increase the effectiveness of post-silicon validation. One solution is to fabricate a ‘debug version’ of a chip with a large amount of validation logic to be used only during testing. However, increasing mask costs (shown in Figure 1.2) make it important that validation be achieved as close as possible to, and ideally at, first silicon. Furthermore, it would still be imperative to validate the production chip before it is handed off to manufacturing.

Many bugs caught in post-silicon validation are pure functional errors that could have been caught pre-silicon. Other bugs, however, relate to electrical or physical properties of the chip that can only be truly validated in silicon because these behaviours are too complex to be modelled with perfect accuracy in pre-silicon. Even in the ideal case where pre-silicon validation detects all functional errors, post-silicon validation would almost certainly be required before large-scale manufacturing. Regardless of the tools and techniques used to exercise the fabricated chip, it is often very difficult to quantity the effectiveness of post-silicon tests.

### 2.1.4 Manufacturing Test & Speed Binning

It is important to distinguish our approach from delay fault testing for manufacturing test [38]. Many underlying concepts (e.g., path delay, sensitizability, critical paths, etc.) are the same, but the application is different. In high-volume manufacturing test, the design is assumed to be functionally correct and the goal is to catch manufacturing defects which vary from chip-to-chip. Manufacturing test usually uses scan-chain based techniques which load test vectors into registers deep in the design and later verify that the output of the circuit at another scan-chain register contains the correct value given that test vector. Post-silicon validation usually focuses on techniques which interact with the system through its top-level ports whereas manufacturing test usually does not usually extend well to this type of top-level testing and is usually focused on testing at a much lower level.

In manufacturing test, unlike post-silicon validation, test time per die must be kept extremely low. In contrast, in post-silicon validation, the goal is to catch subtle design errors that escaped pre-silicon validation, so extremely long-running tests (e.g., trillions of cycles of running OS and application software), with the die in an actual system, are used. Our goal is to be able to assess the quality of such long-running tests.[2]

Speed binning refers to the testing process used to quickly but conservatively estimate performance of a physical SoC [15] by estimating its maximum operating frequency, often through scan-chain based techniques [8]. This allows IC producers to sell a range of processors that are the same silicon design, but with differing performance characteristics due to fabrication variation. Superficially, speed binning seems very related to the problem which this dissertation seeks to address. Although tests used in speed binning are usually functional, they are also very short and have been designed to exercise the most delay sensitive parts of

[2] Of course, in post-silicon validation, short targeted tests are also run on automated test equipment, but such tests are typically generated with coverage information known by construction, and therefore are not the focus of this thesis.

a design. Furthermore, since speed binning tests are short, their coverage can be pre-computed using simulation. In some cases, however, manufacturing tests may be sufficiently extensive such that simulation may not be feasible. In post-silicon validation, the tests are almost always longer running than manufacturing tests and exploratory in nature. Since the achievable cycle-count is so much higher in post-silicon validation than in pre-silicon simulation (due to slow simulation speed) or in manufacturing test and speed binning (since each fabricated chip must be tested, reducing testing time directly corresponds to cost-savings), the amount of designer effort devoted to generating effective (but short) tests in post-silicon validation is lower. In general, post-silicon tests make use of the increased cycle count by employing constrained random tests. As such, this dissertation focuses on how emulated platforms can provide additional insight into post-silicon test characteristics and allow for effective test generation without requiring the amount of tester effort currently needed to design test suites for manufacturing test and speed binning.

As coverage-based techniques for generating effective post-silicon tests mature, it may be possible to use them to help design speed binning tests. However, for large production runs, it would likely be difficult to match the reduced testing times of a designer-crafted test suite. Since these savings are multiplied by every fabricated chip, the sunk engineering cost of test design is usually a good investment.

Recently, there has been renewed interest in reducing the effort required to design speed binning tests by using structural scan chain-based tests [49] but the time required to load scan chains with test vectors is non-trivial, increasing overall testing time.

## 2.2 Pre-Silicon Coverage Metrics

There are many metrics by which test coverage can be measured. Many have been designed to be specific to a certain stage of silicon validation: pre-silicon, post-silicon, or manufacturing test. In general, however, coverage-based techniques are in wide use in pre-silicon simulation and manufacturing test, but are used less commonly at other stages of validation.

Code coverage and functional coverage are extensively used during pre-silicon validation to evaluate the effectiveness of testing. Ideally, designs must reach 100% statement, branch and functional coverage, however exceptions are made if it is known that a given coverage point is unreachable or not important.

### 2.2.1 Types of Pre-Silicon Coverage

In this section, we will briefly discuss seven types of coverage that are often used in simulation:

**Statement Coverage:** reports what fraction of HDL statements are executed. In most coding styles, each line corresponds with a statement. However, this is not strictly required. An example of statement coverage is illustrated in Section 2.2.2.

**Branch Coverage:** reports what fraction of branches created by decision statements such as `if → elsif → else → end if` and `case` blocks are executed (note that an `if → end if` block has an implied empty `else` branch). An example of branch coverage is shown in Section 2.2.2.

**Condition Coverage:** reports what fraction of boolean sub-expressions in a decision statement (e.g., `if`) that have evaluated to both `true` and `false` values. For example, "`if (var_a = '1' and var_b = '2') then`" has two sub-expressions: "`var_a = '1'`" and "`var_b = '2'`".

**Path Coverage:** reports what fraction of combinations of branches have been executed. For example, a block with two `if` statements, one after another has four paths since each `if` statement has two possible values, `true` and `false`.

**Functional Coverage:** reports what fraction of user-configured signal values, statements, properties, or cross products of these coverage points are executed. It is used extensively in industry. However, to be effective it requires careful implementation by a designer with deep domain knowledge into the module under test.

**Mutation Coverage:** reports on the percentage of tests which detect an error which is intentionally inserted into RTL [55, 70]. For example, a mutation of the original RTL may include referencing the wrong variable or modifying operators. A test is said to have 'killed' the mutation, if it successfully detects the change. A test that detects many mutations (i.e., one that has high mutation coverage) is likely more comprehensive and thorough than one with low mutation coverage. Unfortunately, mutation coverage relies heavily on the quality of the mutations generated and it is complicated to implement effectively.

**Tag Coverage:** reports what fraction of signal assignments have an effect on output signals (i.e., how many signal assignments are observable at the output) [18, 21, 45]. For every signal assignment, a tag signal corresponding to that signal is also updated, allowing the propagation of signal assignments to be traced. A test with high tag coverage would cause many signal assignments to be detected at the module's output. This type of coverage, while similar to statement coverage, is much more difficult to attain because the effect of many statements will only be reflected in a module's output under very limited circumstances.

### 2.2.2 An Example of Statement and Branch Coverage

For example, in Figure 2.1, if `a = 0` and `b = 0`, statement coverage would be 4/6 = 67% because statements 1, 2, 3, and 6 were executed. The process for branch coverage is similar but instead of counting statements, it counts branches created by conditionals (such as `if` and `case`). The branch coverage for the code in Figure 2.1 would only be 2/4 = 50% since branch 1 (the main program branch) and branch 2 were the only branches executed.

```
setup(); // Statement 1 & Branch 1
if (a == 0) { // Statement 2 & Branch 2
    do_something(); // Statement 3
} else if (b == 1) { // Statement 4 & Branch 3
    do_something_else();// Statement 5
} // There is an implied else branch here
// else { }  Branch 4

teardown(); // Statement 6
```

**Listing 2.1:** Simple example which illustrates statement and branch coverage techniques

## 2.3 Post-Silicon Coverage Metrics

The design of effective coverage metrics for physical chip-based validation techniques such as pre-silicon emulation or post-silicon validation is still an open research problem. As such, coverage techniques for these platforms are not nearly as established as they are in pre-silicon simulation or manufacturing test (where the limited per-chip testing time allows coverage to be pre-computed using simulation). Furthermore, the coverage metrics used in manufacturing test are not well suited for chip-based verification due to their focus on detecting manufacturing faults, which vary from chip to chip, and not functional bugs, which are systemic to the design itself. Instead, coverage metrics common in simulation (e.g., code coverage, assertion coverage, and mutation coverage) are often used in post-silicon verification. Since these techniques are used extensively in pre-silicon verification, they could be used to provide a universal view of coverage from pre- to post-silicon verification.

### 2.3.1 Related Work — On-Chip Coverage Monitors

Most coverage research has largely focused on how coverage metrics can be used in simulation and emulation, and not on how coverage monitors can be effectively implemented in silicon. In [11], however, the authors implemented on-chip coverage monitors for use during post-silicon validation of Intel's Core 2 Duo processor family. They focused on monitoring important, as dictated by a designer with insight into the chip, functional coverage points that were difficult to achieve in pre-silicon. Unfortunately, due to the constraints of minimizing on-chip monitoring overhead, only very minimal coverage information was extracted. It is interesting to note that in order to reduce area overhead, the coverage monitor data was accessed through the performance monitor registers already present on the processor. These performance monitors are normally used to monitor important performance metrics such as cache misses, power states, and instructions per cycle [61].

In [1], the authors add coverage monitors to their emulated design for use with the post-silicon test suite. During pre-silicon emulation, a constrained random post-silicon test suite (which is much too long to be simulated) is executed, and the coverage is recorded. This coverage is expected to match the coverage achievable by the same tests during post-silicon validation. Using this system, a team can tune the post-silicon tests to guarantee high coverage before the processor is fabricated. As in [11], the tests were limited to functional coverage. Furthermore, the area overhead for implementing these monitors was not evaluated and some details were considered proprietary and not disclosed.

In [33], we manually added coverage monitor flags alongside targeted IP cores, and extracted statement and branch coverage. Unfortunately, we found the monitoring system imposed a large area overhead on the design. To mitigate this, we used state-of-the-art software coverage analysis techniques to losslessly reduce the overhead of our monitors. These area optimizations were successful and provided an area decrease of up to 28% at no quality cost. Even with these optimizations, we hypothesized that the system would not be feasible for implementation on an ASIC due to area constraints.

In [6], we extended the work done in [33] to a multiprocessor SoC design. The coverage monitor system was improved such that the logic analyzer used could record a timestamp along with our coverage information, allowing us to analyze how coverage changed over time between the two processors of our SoC as they were booting Linux. Additionally, some optimizations were made in how our coverage monitors were implemented which resulted in some area savings. We used Synopsys Design Compiler to estimate the area overhead for the coverage monitor system on an ASIC. As expected, we confirmed the coverage monitor area overhead was too high for implementation on an ASIC.

## 2.4 Timing Validation

The work done for this dissertation, while similar the work done in [6] and [33], is focused on solving a fundamentally different problem: timing validation. Timing validation is a substantially different problem than functional coverage, requiring non-functional properties to be validated. Furthermore, the previous work focused on the feasibility of integrating coverage monitor into ASIC chips, whereas this work primarily focuses on coverage monitor implementation for pre-silicon FPGA emulation.

In this dissertation, we focus on critical-path coverage, which considers a circuit path (from the output of a terminating primitive like a flip-flop to the input of a terminating primitive) covered when the path has been exercised under input vectors for which it will experience worst-case timing conditions. Since we are only interested in paths which may cause the circuit to exhibit incorrect behaviour under suboptimal timing conditions (i.e., paths with a lot of timing slack can take longer than expected without any ill effect), we only consider paths which we believe are on or near the critical-path (as dictated by a STA tool).

The research done for the dissertation seeks to explicitly quantify the coverage of specific timing paths in a design. In order to ensure correct timing behaviour, it is necessary to ensure that the critical paths of a circuit are adequately exercised post-silicon. Our method quantitively measures the effectiveness of post-silicon verification plans in exercising paths which have been identified as at or near the critical path with pre-silicon timing analysis. Due to inaccuracies in pre-silicon timing analysis and die-to-die manufacturing variation, it is not possible to determine which subset of these paths will be critical on a given physical chip. The overall goal is: given a chip that has been fabricated, and a verification plan (software to run on the processor and other realistic input stimuli), determine to what extent the timing-critical paths of the circuit are exercised. More precisely, we seek to measure a coverage metric which we define to be the proportion of the $n$ longest paths which have been exercised at least once during the execution of the verification plan. (In our experiments in Section 5.2, $n = 2048$, but $n$ can be chosen arbitrarily.)

# Chapter 3

# Proposed Flow

This chapter describes an idealized toolflow for automatic integration of critical-path coverage monitors. Although our techniques could be applied to any large integrated circuit, we focus on large processor-based SoCs since the interaction between hardware and software provide significant verification challenges. We begin in Section 3.1, where we motivate critical-path coverage monitors through a discussion on timing validation. In Section 3.2, we highlight the role FPGA-based emulation can play in obtaining critical-path coverage information of a post-silicon test suite. In Section 3.3, we describe the composition of the coverage monitors and discuss how the fundamental implementation differences between the ASIC and FPGA versions of the chip do not pose a significant barrier to extracting coverage information. Section 3.4 summarizes the chapter.

## 3.1 Problem Statement

The performance of ICs is in large part dictated by its maximum operating frequency. This maximum operating frequency describes the shortest clock period that a design can reliably use. If a design is running faster than this frequency, a circuit path's output may not have resolved by the time it is read by a flip-flop or other memory device (i.e., a setup time violation). Or, in the case of a hold time violation, the input signal to a flip-flop may change too soon after a clock transition, causing an incorrect value to be read. In either case, the output may be erroneously recorded by the device, likely causing incorrect functional behaviour in the design [12]. Unfortunately, determining this frequency is non-trivial due to the variation between given fabricated chips as well as the limitations of timing analysis techniques. Ignoring variation in fabrication for now, the maximum operating frequency of a chip for a given voltage and temperature pair could be determined using exhaustive circuit simulation. This approach quickly becomes infeasible for all but the smallest designs. Instead, Static Timing Analysis (STA) techniques are used to conservatively estimate the expected timing of a chip without requiring simulation. If the design has already been placed-and-routed, STA can take into account this information for a more accurate estimate. A variant of STA, Statistical Static Timing Analysis (SSTA), replaces the timing values used in STA with probabilistic distributions, allowing the tool to report a probabilistic distribution of possible circuit outcomes.

STA techniques are only approximate; to ensure timing performance, validation must be done with the physical silicon. Furthermore we cannot simply use STA techniques to determine the single most critical path to be targeted during post-silicon validation because variation in fabrication may cause other paths to become more critical. Post-silicon timing validation is further complicated because it is not simple for tests to target specific timing-critical paths. This is particularly true for post-silicon tests because they are almost always too long to be simulated. Coverage monitors could be added to the ASIC design to aid in targeting paths, but this poses a substantial (and usually untenable) area cost.

## 3.2 Overall Flow

One way to determine whether a path has been covered in hardware validation is to add small coverage monitors to each path of interest; during the run, these coverage monitors could trip if the corresponding path is exercised, and at the end of execution, the state of these monitors could be read, and the coverage metric calculated. The problem with this approach is the area overhead of the coverage monitors. Although these coverage monitors could be removed in the release version of the chip, doing so would change the timing and electrical behaviour of the chip, perhaps leading to different paths becoming critical, nullifying the value of the results.

Our approach takes advantage of the prevalence of FPGA-based emulation. In the following discussion, we use the term *ASIC version* of the design to refer to the version of the design before emulation (the version that will eventually be implemented as a chip and shipped) and the term *emulation version* of the design to refer to the version of the design that has been re-mapped to FPGAs for validation and prototyping purposes.

Our flow uses the *emulation version* to determine the timing coverage metric of the *ASIC version*. More precisely, our approach is as follows. Using STA techniques, we identify the *n* longest paths in the ASIC version. If information regarding false paths is available, those paths can be discarded at this stage. Similarly, if there are other paths of particular interest to the design and validation engineers, they can be added at this stage. Once we have identified the paths of interest in the original design, we create the emulation version by mapping the circuit to FPGAs (e.g., as described in [60, 66]). We then instrument the emulated design, as will be described in Section 3.3. This instrumentation monitors the targeted paths, and determines when each has been exercised. At the end of the run, the coverage information can be read from the instrumentation, and an overall timing coverage metric calculated. This is the coverage that is expected when the same verification plan is run on the ASIC version of the design. The proposed flow is shown in Figure 3.1.

In the above flow, it is important that the paths be determined using the original design, since once the design has been re-targeted to an FPGA for emulation, the paths that are critical may change. It is very possible that the paths we target are *not* the critical paths during emulation, but they are the ones we care about, since they are the critical paths of the chip that we intend to ship.

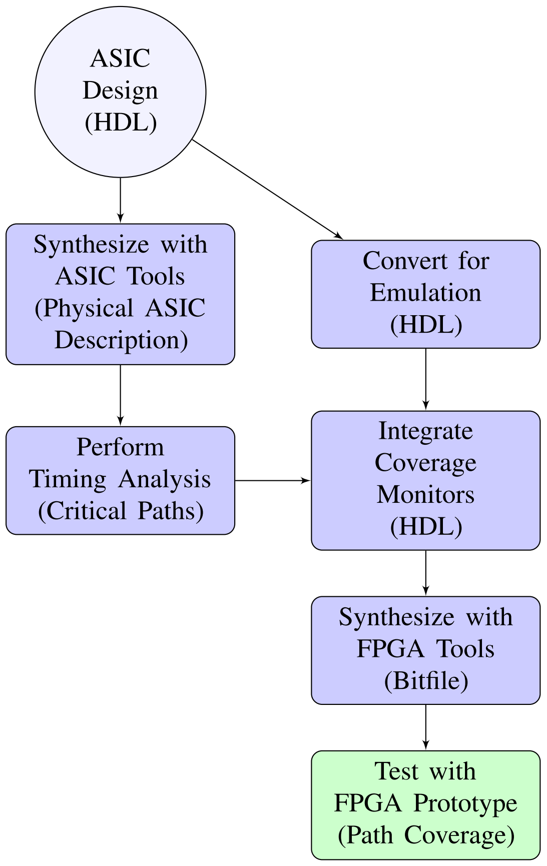

ASIC Design (HDL)
Synthesize with ASIC Tools (Physical ASIC Description)
Convert for Emulation (HDL)
Perform Timing Analysis (Critical Paths)
Integrate Coverage Monitors (HDL)
Synthesize with FPGA Tools (Bitfile)
Test with FPGA Prototype (Path Coverage)

**Figure 3.1:** Toolflow for adding path coverage monitors to the FPGA prototype of an ASIC design. Each stage's output is shown in parentheses.

## 3.3 Instrumentation

Key to the flow described in Section 3.2 is the instrumentation that is added to the emulation version of the circuit. Because we are adding our instrumentation to the emulation version of the circuit only, the area and delay overheads do not translate to the shipped chip. However, minimizing overhead is still important, since the overhead directly relates to the size of the FPGA emulation system and the speed at which emulation tests can be run.

As described in Section 3.2, we obtain a set of timing-critical paths from the static timing analysis of the ASIC version of the circuit, and then instrument these paths in the FPGA version of the circuit. The instrumentation is designed to record whether, for each path, the path has been exercised. A monitored path is considered exercised, and therefore covered, when a transition on the input of the path causes there to be a transition along every element of the path. To instrument a path, we add:

a) A transition detector at the start of the path,

b) Boolean difference [13] logic to calculate the sensitization condition [4] for each logic element along the path,

c) A bit to record whether the path has been entirely sensitized.

An example is shown in Figure 3.2, where the logic function, $f$, is $(\overline{A+B}+C) \cdot (\overline{A}+\overline{B})$. From this logic, its sensitization function can be calculated to be $B \cdot C$. This makes intuitive sense as when $B$ = '1' the upper OR gate's $\overline{B}$ input will be '0', allowing signal $A$ to control that gate's output. This is similarly true for the final AND gate; when $C$ = '1' its lower input will be '1', allowing the targeted path to control the value of $f$. Figure 3.2(a) shows the original path, and Figure 3.2(b) shows the instrumented circuitry, including the logic to compute the sensitization condition. Figures 3.3 and 3.4 demonstrate how the coverage monitor shown in Figure 3.2 can be modified to be sensitized only for rising or falling edges, respectively. This is achieved by modifying the comparator which detects the transition on the first element of the path.

Further modifications could be made to allow detection of fault observability by selectively injecting errors whenever the path has been exercised (i.e., whenever coverage is achieved) as shown in Figure 3.5. If the injected error is found to be observable, it should be similarly observable on an ASIC system with a timing fault affecting that path (though in the case of multiple faults, the fault may not be observable). However, to be practically useful, this modification would have to be extended to allow error injection to be enabled on a per-path basis since the fault may only be observable many clock cycles after the error is injected. In this thesis, we sensitized the coverage monitors for rising and falling edges (i.e., the monitoring circuitry shown in Figure 3.2), and did not instrument for error injection.

It is important that we add instrumentation circuitry to the RTL representation of the circuit, before it is optimized and mapped by the FPGA tools. It is very possible that after mapping to the FPGA, the original paths either do not exist (because of the optimizations performed by the FPGA tools) or because parts of the original paths have been encapsulated into hard blocks (such as DSP blocks which are common in all modern FPGAs). Note, however, that the inclusion of this instrumentation does not prevent the FPGA tool from optimizing the original circuit as it sees fit. In the example of Figure 3.2, once the instrumentation has been added to the RTL version of the circuit, the tool is free to optimize the original path in any way (including removing it from the circuit entirely). The instrumentation circuit will still record whether the path would be exercised in the ASIC version of the circuit, even if that path does not exist in the emulation version.

The area overhead of this instrumentation is proportional to the number of paths instrumented and the length of each path. For each segment along the path, the sensitization function is guaranteed to require no more lookup tables than a gate-level implementation of the original path, since the sensitization function at each stage requires fewer inputs than the original function at each stage. However, in some cases, it may be that the original function can be optimized effectively by the FPGA CAD tools (for example, a multiply function may be efficiently mapped to an embedded DSP block), while the instrumentation logic cannot be optimized as efficiently. In the worst case, if the instrumentation circuitry requires signals that would otherwise have been encapsulated inside a DSP block, then the circuitry generating these signals will have to be replicated in the FPGA soft logic. If this occurs, it may tend to increase the overhead of our instrumented circuit compared to the same circuit without instrumentation beyond that which would be expected due to the extra logic. However, as we will show in Section 5.2, experimentally, this rarely occurs.

The delay overhead in the emulation version is very small. Although some signals will have increased fanout, because FPGA interconnect is typically fully buffered [42] the overall timing impact will be small. The specific frequency overheads are quantified in Section 5.2.3.

## 3.4 Chapter Summary

This chapter described the ideal flow for integration of critical-path coverage monitors into emulated designs. Section 3.1 began by introducing and motivating the problem this dissertation addresses. Section 3.2 highlighted the value FPGA-based emulation can provide for post-silicon validation. In Section 3.3, we described the coverage monitors themselves and discuss the challenges posed when using emulated systems to provide information on the silicon product.

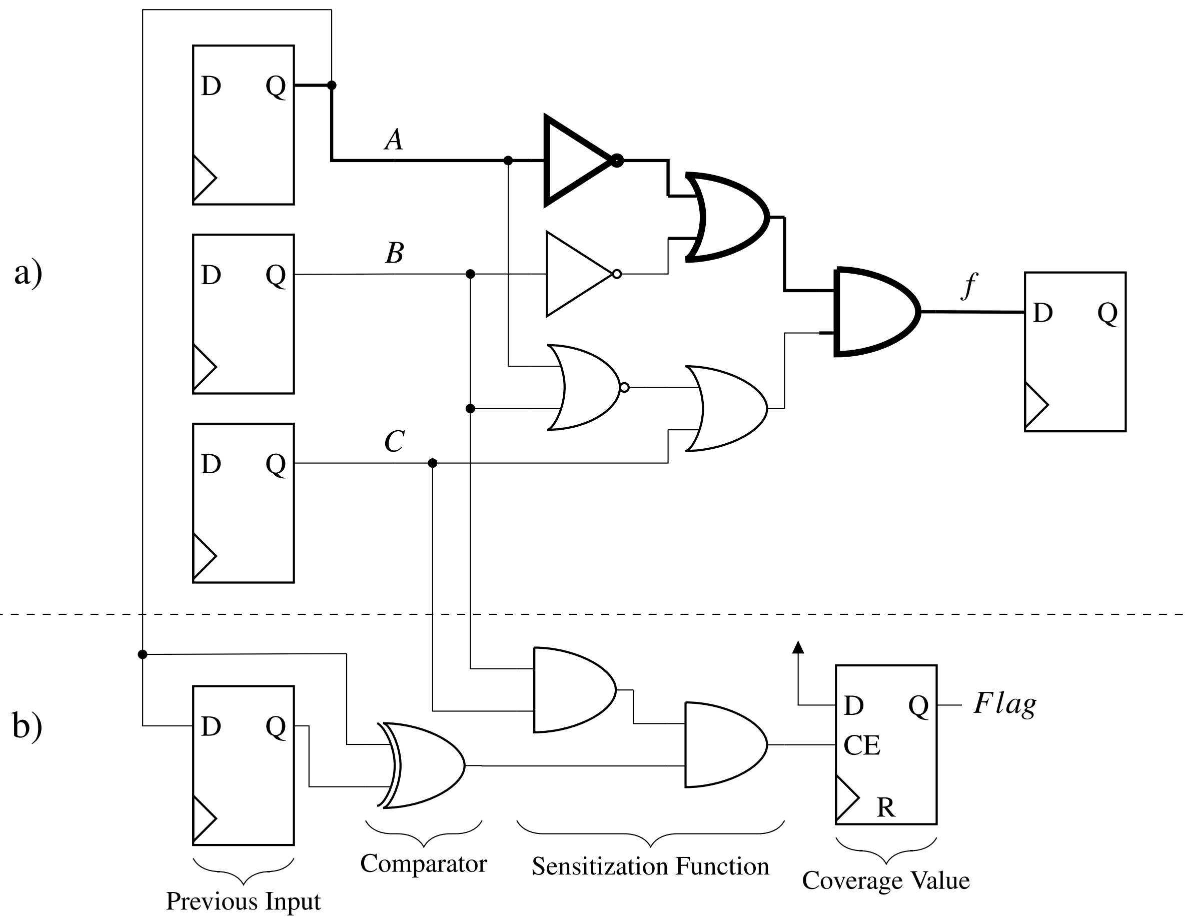


**Figure 3.2:** Example of a critical path, a), instrumented to monitor coverage by b) for bolded path $f$ initiated by a rising or falling transition on input $A$.

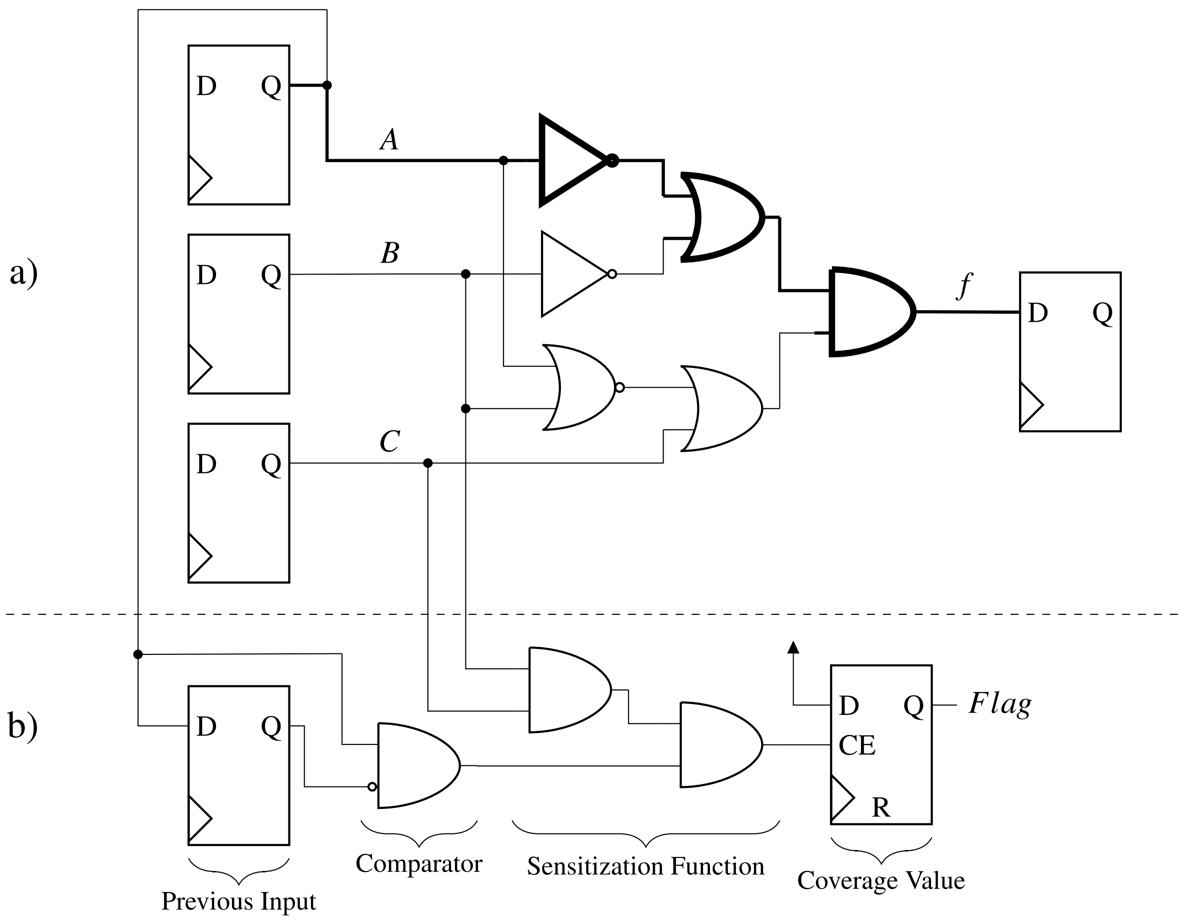


**Figure 3.3:** Example of a critical path, a), instrumented to monitor coverage by b) for bolded path $f$ initiated by a rising transition on input $A$.

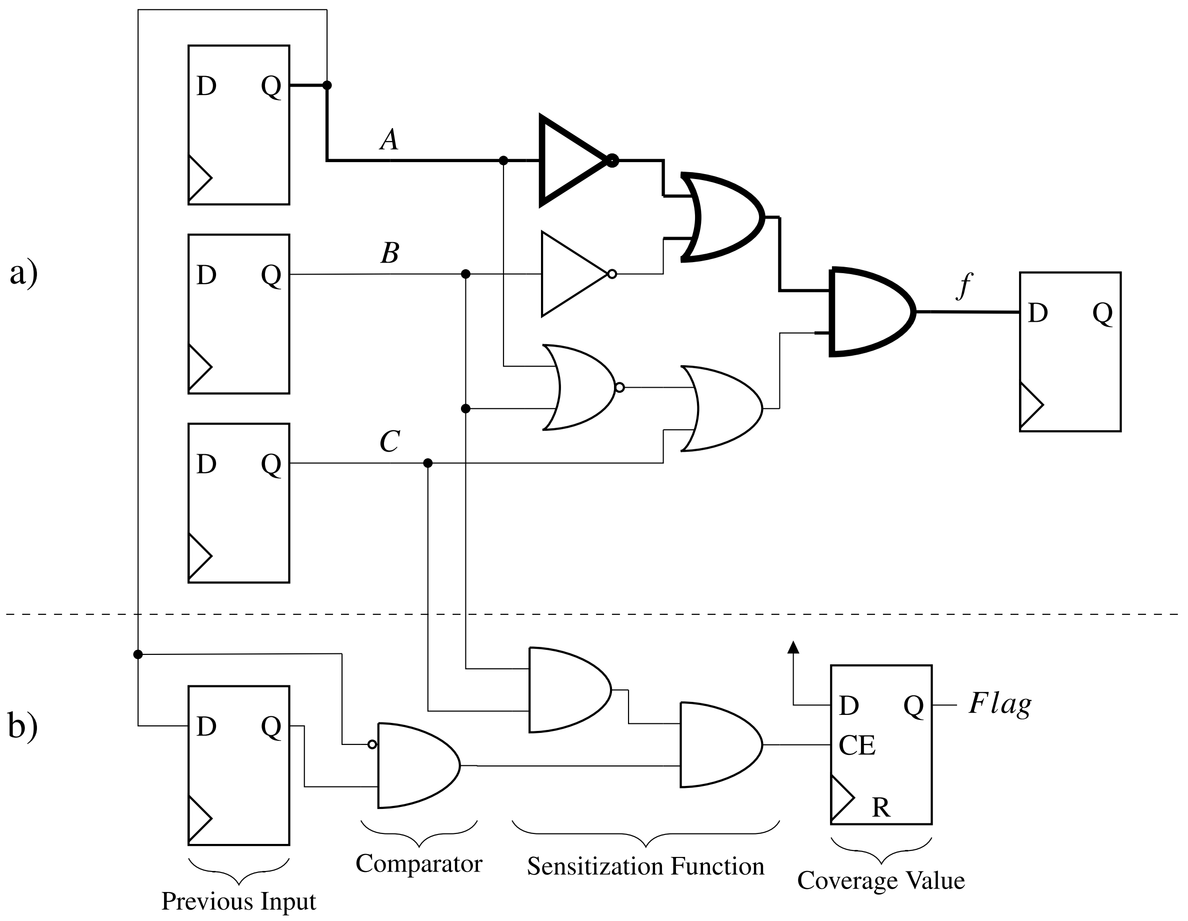


**Figure 3.4:** Example of a critical path, a), instrumented to monitor coverage by b) for bolded path $f$ initiated by a falling transition on input $A$.

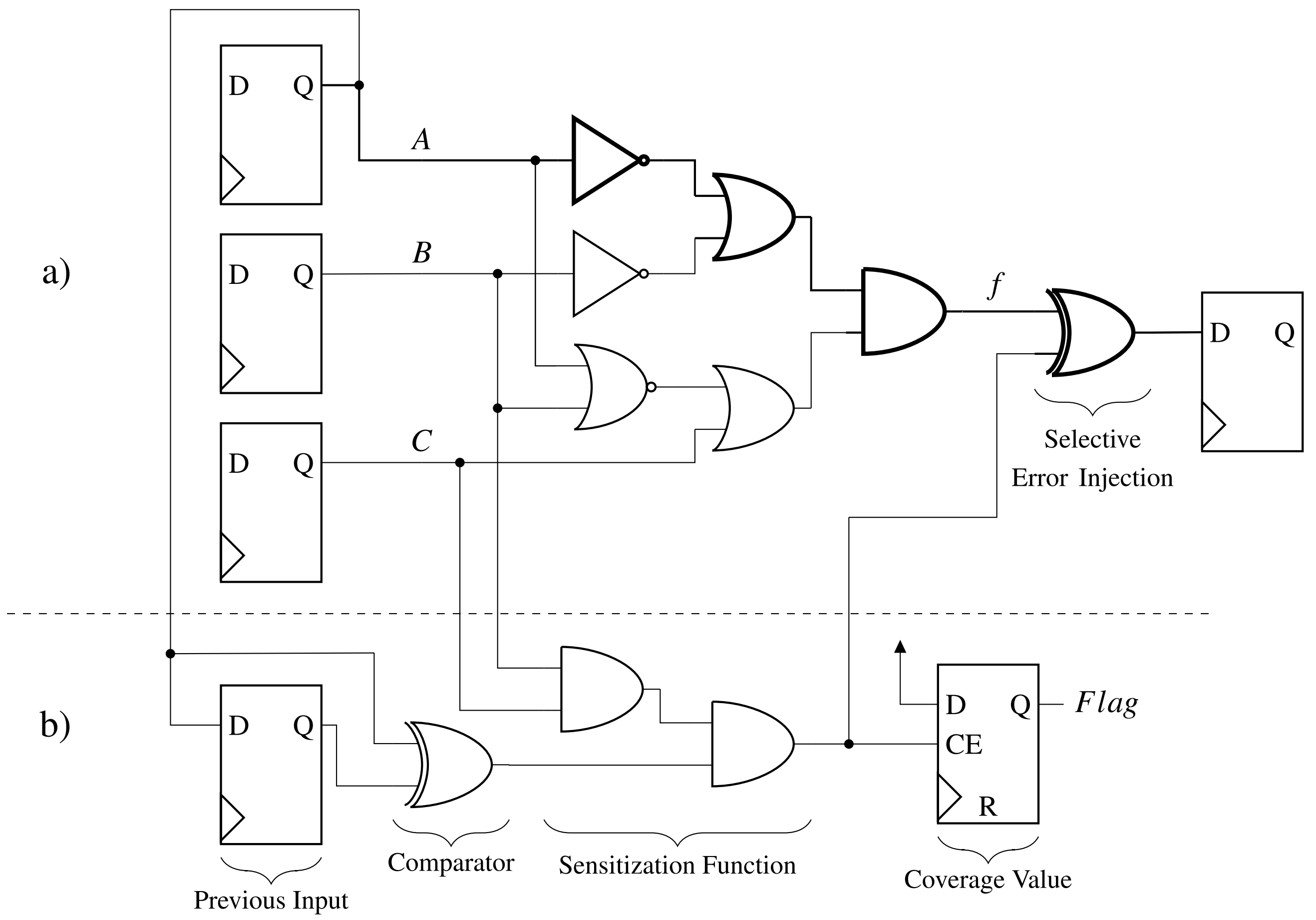


**Figure 3.5:** Example of a critical path, a), instrumented to monitor coverage by b) for bolded path $f$ through input $A$ with additional circuitry allowing for error injection whenever coverage is achieved.

# Chapter 4

# Implementation

In this chapter we introduce the toolflow implemented to automatically instrument hardware designs with critical-path coverage monitors. We describe the toolflow used to approximate the ASIC emulation process and describe our implementation of the XDL to VHDL converter used to facilitate automatic coverage monitor integration, including a detailed characterization of the six main classes of XDL instances encountered.

## 4.1 Approximated Flow

Because the focus of this work is the design and implementation of the coverage monitors introduced in Section 3.3, as well as the investigation of the coverage achieved for some common post-silicon tests, and not the verification of a processor-centric SoC *per se*, we have shortened the proposed verification flow in two ways:

a) by starting with a design that is already known to be synthesizable to FPGA,

b) by integrating coverage monitors into the design at the gate-level, not the RTL level.

This modified flow is shown in Figure 4.1. After synthesis, we perform timing analysis on the placed-and-routed circuit description. The timing analyzer reports the slowest *n* paths to the converter tool from which path coverage monitors are generated. These coverage monitors are integrated into a new structural VHDL description generated from the placed-and-routed XDL description of the circuit. This new VHDL description is then synthesized using the full Xilinx toolflow. All constraints on top-level ports, signals or instances should still apply to the structural circuit description, however constraints for signals or instances deep in the design hierarchy may need to be modified to account for differences in generated names.

Starting with a design that is already synthesizable to an FPGA, particularly one that can be fabricated to ASIC technology, is a very good approximation to using an FPGA-emulated ASIC design. However, it causes our flow to differ in one significant way: there is no ASIC design from which to obtain timing paths. Instead, we use timing results from the design natively synthesized for the FPGA. While it is expected that these timing paths will differ in content from those derived for an ASIC version, they should not differ substantially in composition. The FPGA timing paths used in our approximated flow could be easily exchanged for ASIC timing paths. It is important to highlight that this simplification was made to facilitate our experimentation; in a ‘real’ application of the flow, ASIC timing paths would be used.

We integrated our coverage monitors at the gate-level because, to the best of our knowledge, there are no full-featured, open-source VHDL parsers available for adaptation to our uses. In contrast, the two biggest FPGA vendors, Altera and Xilinx, both provide toolkits which allow experimental integration with their tools. For this thesis, we used the RapidSmith API from Brigham Young University [41] for parsing and iterating over the Xilinx Description Language (XDL) format, Xilinx’s human readable gate-level circuit description. Parsing XDL is much easier than parsing VHDL because it is a simple, structural description of a circuit written by an automated tool and not a language created for circuit designers to write directly.

### 4.1.1 Automated Integration of Coverage Monitors Using XDL

A key aspect of our ideal and approximated flows is the two CAD tool passes: the design first must be synthesized, placed, and routed (ideally, with the ASIC tools) for static timing analysis to determine likely critical paths of the ASIC version; then this design is instrumented for coverage and processed a second time by the FPGA tools to create the emulation version. Accordingly, to automate the flow, we need a mechanism to extract the output from the first step, add instrumentation, and then input this into the FPGA tools for the second step.

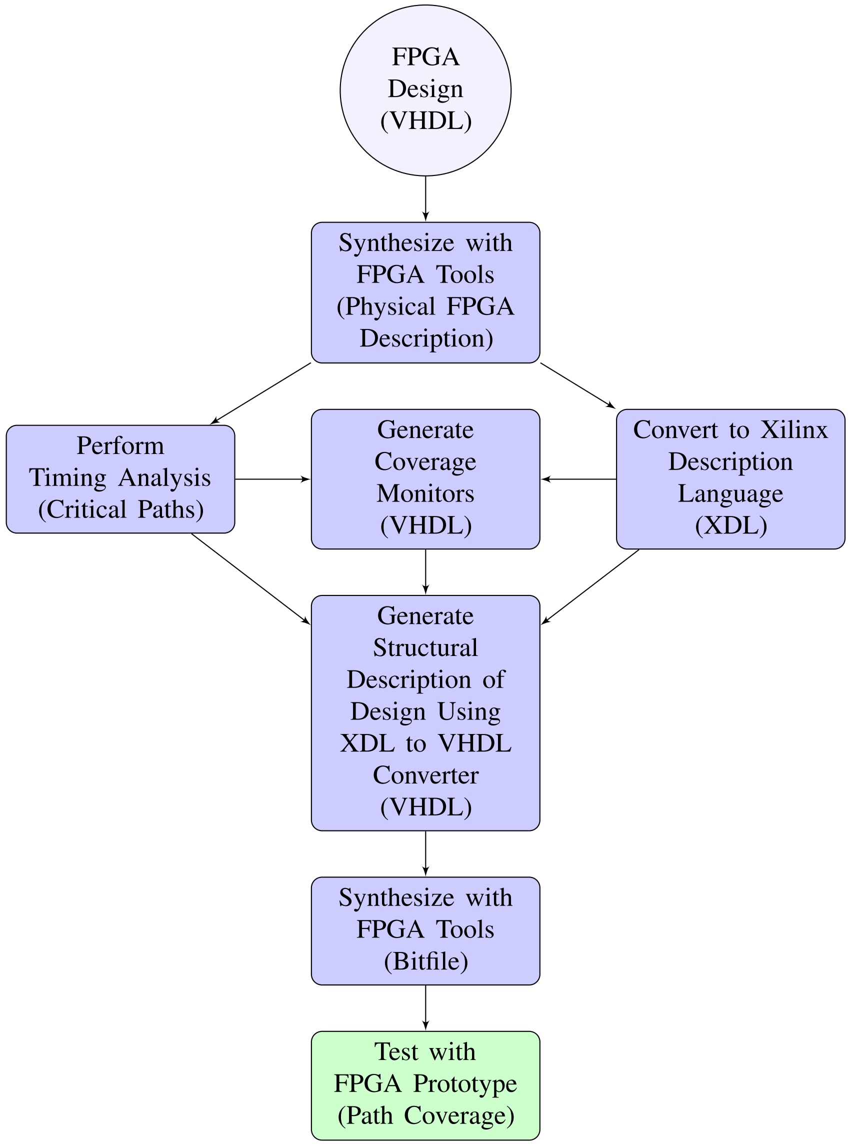

FPGA Design (VHDL)
Synthesize with FPGA Tools (Physical FPGA Description)
Perform Timing Analysis (Critical Paths)
Generate Coverage Monitors (VHDL)
Convert to Xilinx Description Language (XDL)
Generate Structural Description of Design Using XDL to VHDL Converter (VHDL)
Synthesize with FPGA Tools (Bitfile)
Test with FPGA Prototype (Path Coverage)

**Figure 4.1:** Toolflow used in this thesis to automatically add path coverage monitors to an FPGA design. Each stage's output is shown in parentheses.

Xilinx provides unofficial support for an interface to export their proprietary Native Circuit Description (NCD) files to a human-readable XDL file in terms of their fundamental FPGA primitives and the nets that connect them. This allows designs to be imported and exported freely between the major FPGA synthesis stages: *mapping*, *placement* and *routing*. However, in recent versions of the Xilinx tools, the *mapping* and *placement* stages have been integrated, making import/export after mapping but before placement impossible. In these versions, it is possible to export a placed description and discard the placement information, but there is no way to pass a mapped-but-unplaced description to the Xilinx placer.

XDL can describe an unplaced, a placed-but-not-routed, or a placed-and-routed physical description of a circuit on a Xilinx FPGA. These XDL files are composed of *instances* which represent the Xilinx building blocks such as slices, input/output buffers, DSPs and Block Rams (BRAMs), and *nets* which describe how they are connected together. Listing 4.1 shows a sample instance with net connections for a combinational slice which implements the $\overline{A_5} \cdot A_6$ function in the `A` lookup table (described by the slice's `A6LUT` attribute).

We use this netlist information to generate an equivalent VHDL description of the XDL file alongside additional monitoring logic. This new VHDL description will be functionally equivalent to the original design but will include our coverage monitors. This task is made possible by instantiating Xilinx FPGA primitives through the VHDL libraries provided by Xilinx. However, in most cases the XDL describes the FPGA primitives differently than can be directly instantiated in

VHDL. For example, XDL describes combinational logic in terms of lookup table attributes within a slice instance, but the Xilinx libraries must instantiate slices in terms of their components (such as lookup tables, muxes, and carry chains). Similarly, the BRAM primitives that XDL describes include pins which are held at GND or VCC but the BRAM instances that can be instantiated in VHDL only include ports which are used. For example, a RAMB18X2SDP instance's RDADDR and WRADDR vectors are 6 signals wider as described in XDL than can be instantiated in VHDL. These differences are further described in Section 4.2.

In general, this process is fairly mechanical. However, it requires careful analysis to ensure that each instance is converted correctly. We used an industrial formal equivalence tool to ensure that the VHDL description generated from the XDL was equivalent to the original circuit.

## 4.2 XDL to VHDL Converter

After converting our placed-and-routed NCD to XDL as discussed in Section 4.1.1, we found that the XDL contained fourteen types of instances. Ideally, converting these XDL instances to the Xilinx VHDL components would be trivial but as shown in Table 4.1, this is not often the case; three of these fourteen XDL components are identical to their VHDL counterpart (shown as "No Changes Required") but nine differ marginally (shown as "Attribute Translation", "Fixed Location" and "Ports Offset") and two differ greatly (shown as "Composed of Many VHDL Instances"). The techniques for converting these classes of XDL instances are described in Sections 4.2.2 through 4.2.7.

```
inst "ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_3" "SLICEL",
placed CLBLM_X1Y87 SLICE_X1Y87,
cfg "
// shared slice attributes
CEUSED::#OFF // no chip enable
CLKINV::CLK // clock is not inverted
COUTUSED::#OFF // carry chain Cout is not used
PRECYINIT::#OFF // carry chain Cin is not used
REVUSED::#OFF // no reverse set/reset signal
SRUSED::#OFF // no set/reset signal
SYNC_ATTR::ASYNC // has asynchronous reset and clear

// attributes for lookup table (LUT) A
A5LUT::#OFF // slice is not in dual 5LUT mode
A6LUT:ddrsp0.ddrc0/ddrc/ddrc/ndr_refctr(0)1:#LUT:O6=(A5*A6)
    // logic is (A5 and A6)
ACY0::#OFF // no AX pin passthrough or 5LUT needed
AFF:ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_0:#FF // flip-flop (FF
   ) is enabled
AFFINIT::INIT0 // initial value of FF is '0'
AFFMUX::O6 // input to FF is output of 6LUT
AFFSR::SRLOW // set/reset signal in normal mode
AOUTMUX::#OFF // no additional AMUX signal needed
AUSED::#OFF // output pin which bypasses FF is not used
';

net "ddrsp0.ddrc0/ddrc/ddrc/ndr_refctr_share0000(0)" ,
outpin "ddrsp0.ddrc0/ddrc/ddrc/Madd_ndr.refctr_share0000_cy
   (3)" AMUX ,
inpin "ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_3" A5 ,
; // other inpins are omitted for brevity

net "ddrsp0.ddrc0/ddrc/N63" ,
outpin "ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_10" A ,
inpin "ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_3" A6
; // other inpins are omitted for brevity
```

**Listing 4.1:** Representation of a SLICEL instance in XDL along with its nets.

```
-- ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_3 logic=(A5*A6)
ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_3_lut_A6_output <= (
   ddrsp0_ddrc0_ddrc_ddrc_ndr_refctr_share0000(0) and
   ddrsp0_ddrc0_ddrc_N63);

-- SLICEL ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_3
ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_0_inst : FDCPE
generic map (
    INIT => '0') -- initial value of FF is '0'
port map (
    Q => ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_0,
    C => clkml, -- clock is chip-level clkml signal
    CE => '1', -- chip is always enabled
    D => ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_3_lut_A6_output,
    CLR => '0', -- never in clear state
    PRE => '0'  -- never in preset state
);
```

**Listing 4.2:** VHDL snippet translated from XDL shown in Listing 4.1

### 4.2.1 Automatically Generating Java Objects from Xilinx Libraries

RapidSmith object definitions are automatically generated from the VHDL libraries included with a default Xilinx installation using a Python script developed for this research. Those object definitions allow instances that are identically described in XDL and the Xilinx VHDL libraries to be converted without programmer intervention. Additionally, these definitions provide a basis for a user to describe how unknown XDL instances can be converted to equivalent VHDL instances. Official Xilinx documentation for these VHDL instances can be found in the Virtex-5 Libraries Guide for HDL Designs [72].

### 4.2.2 XDL Instances Requiring No Changes

Three XDL instances, `DCM_ADV`, `RAMB36_EXP` and `TIEOFF`, can be directly converted to VHDL instances by the XDL to VHDL script because they do not require any of the special handling described in Table 4.1.

### 4.2.3 XDL Instances Requiring Simple Attribute Name Translation

Some XDL instances types, such `BSCAN` and `BUFG`, are very nearly directly compatible with their VHDL counterparts. In these cases, they simply require a simple attribute name translation. For example, the `BUFG`'s input port is named `I` in XDL, but `I0` in the VHDL library (though in general they are not this trivial). The complete list of the seven XDL instance types that require attribute translation can seen in the "Attribute Translation" column of Table 4.1.

### 4.2.4 XDL Instances Requiring Fixed Locations

The XDL to VHDL converter tries to ensure that top-level signal and instance names for its generated VHDL will be identical to those in the original design. This allows the design's constraint files (which, among other things, specify where individual pieces of logic should be placed and specific timing requirements for nets) to be equally applicable to the original design and the converter-generated VHDL. However, some XDL instances must be constrained to specific locations even if this behaviour has not been specified in the constraints file. This can

**Table 4.1:** Characteristics of XDL Instances

| Instance | No Changes Required | Attribute Translation | Fixed Location | Ports Offset | Composed of Upper & Lower VHDL Instances | Composed of Many VHDL Instances |
|---|---|---|---|---|---|---|
| BSCAN | | ✓ | ✓ | | | |
| BUFG | | ✓ | | | | |
| DCM_ADV | ✓ | | | | | |
| IDELAYCTRL | | | ✓ | | | |
| ILOGIC | | ✓ | | | | ✓ |
| IOB | | ✓ | | | | ✓ |
| IODELAY | | | ✓ | | | |
| OLOGIC | | ✓ | | | | ✓ |
| RAMB18X2 | | ✓ | | ✓ | ✓ | |
| RAMB18X2SDP | | ✓ | | ✓ | ✓ | |
| RAMB36_EXP | ✓ | | | | | |
| SLICEL | | | | | | ✓ |
| SLICEM | | | | | | ✓ |
| TIEOFF | ✓ | | | | | |

be because those locations may have direct connections to other instances (e.g., `IDELAYCTRL`), logic may have been added which relies on predetermined bus positions (e.g., `BSCAN`) or because that location has a known delay to another location (e.g., `IODELAY`). These three XDL instance types have their other behaviours documented in Table 4.1.

### 4.2.5 XDL Instances with Ports Offset Compared to VHDL Instances

Some XDL instances such as `RAMB18X2SDP` and `RAMB18X2` have their port values offset relative to the VHDL instances (e.g., the signal with index six of the RDADDR and WRADDR signals of the `RAMB18X2SDP` module index correspond with the port at index zero of the VHDL instance.

### 4.2.6 XDL Instances Composed of Upper and Lower VHDL Instances

These XDL instances are composed of two VHDL instances of the same type. This is done because these hardware primitives are fracturable (e.g., BRAM primitives such as `RAMB18X2` and `RAMB18X2SDP` can be 36 Kb in size, or split into two separate 18 Kb BRAMs). These instances add a suffix, usually '_U' or '_L', to the attribute names to distinguish between the upper and lower instances.

### 4.2.7 XDL Instances Composed of Many VHDL Instances

Some XDL instances types such as `ILOGIC`, `IOB`, `OLOGIC`, `SLICEL` and `SLICEM` are composed of many different VHDL instances. For example, a `SLICEL` instance is composed of 4 six-input LUTs, 4 flip-flops, 3 dedicated multiplexers, a 4-bit carry chain logic block, and sufficient routing to connect these blocks in various ways. The configuration for the blocks in the `SLICEL` and the description of the instance's interconnect is described by XDL attributes, but the behaviour of these attributes is undocumented and often not intuitive. The complicated nature of these XDL instances posed a significant challenge during the implementation of the XDL to VHDL converter.

## 4.3 Coverage Monitor Instrumentation

The critical-path coverage monitors implemented for this dissertation are very similar to the idealized coverage monitors described in Section 3.3. A sample coverage monitor instantiation is shown in Listing 4.3. This monitor instruments a path which begins at the `Q` output of lookup table A for the slice instance previously shown in the XDL and VHDL snippets of Listings 4.1 and 4.2.

When the sensitization condition (`SENSITIZATION_CONDITION(1600)`) is satisfied, we know that a transition on the input (`INPUT(0)`) will be reflected along every circuit connection between every element in the targeted path. This transition, since it begins at the head of the critical path and propagates through every element along the critical path, reflects the worst-case delay for that path.

```vhdl
-- Coverage Monitor Instantiation
pathCoverageUnit1600 : PathCoverageUnit
-- id=126291 delay=2.255
generic map (WIDTH => 1)
port map (
    INPUT(0) => ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_0, -- net
       name is ddrsp0.ddrc0/ddrc/ddrc/dr.refctr_0
    CLK => clkml,
    RESET => gpioi_din(8), -- coverage can be reset with a
       pushbutton
    COVERED => TIMING_PATH_COVERAGE(1600),
    TRIGGER  => TRIGGER(1600),
    SENSITIZATION_CONDITION => SENSITIZATION_CONDITION
       (1600)
);

SENSITIZATION_CONDITION(1600) <= (((( not ('0') and (
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_3 and not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_0))) xor (( not ('1')
   and (ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_3 and not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_0))))) and (((( not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_5 and (
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_7 and ( not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_4 and ( not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_2 and (
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_6 and ('0'))))))) xor
   (( not ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_5 and (
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_7 and ( not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_4 and ( not
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_2 and (
   ddrsp0_ddrc0_ddrc_ddrc_dr_refctr_6 and ('1')))))))))
);
```

**Listing 4.3:** Sample coverage monitor for VHDL instance shown in Listing 4.2

Once this has occurred at least once, we mark the path as exercised. The path will remain covered until explicitly reset by the tester.

Obviously the logic for the coverage monitor sensitization function could be optimized, particularly where signals have been `AND`ed with constant ‘0’ or ‘1’ values. However, we found that the FPGA CAD tools easily optimized the unoptimized logic function. As such the overall circuit quality was identical regardless of the quality of the sensitization function, though admittedly, this may affect overall CAD tool runtime.

## 4.4 Reading Coverage Information Off-Chip

We used Xilinx’s logic analyzer, Chipscope [71], to read our coverage information off-chip. The logic analyzer data is composed of the coverage flags, as well as the current processor cycle count. This allows us to pinpoint the processor cycle at which an individual coverage point is achieved.

In order to easily gather coverage information from all the coverage monitors, we added an additional ‘trigger’ signal to the monitors which would activate for a single clock cycle once the coverage monitor achieved coverage. The `OR` of all coverage monitor trigger signals acted as the trigger for the logic analyzer; whenever the trigger signal is high, the coverage and current clock cycle needs to be recorded by the analyzer.

## 4.5 Chapter Summary

This chapter presented the toolflow we used to automatically instrument a hardware design with critical-path coverage monitors. We began by detailing the key differences between the ideal and implemented toolflows, and describing the reasons why these approximations are appropriate for our research. We then described the XDL to VHDL converter used in our toolflow and discussed the differences discovered between the XDL and VHDL instances when describing the same FPGA primitive. We showed the VHDL used to implement a coverage monitor for a given critical path and describe how the Xilinx logic analyzer was used to read coverage information off-chip.

# Chapter 5

# Results

This chapter presents the test setup used to validate the coverage monitor system. We describe the SoC design targeted by our coverage monitors, and the benchmarks evaluated by these monitors. We discuss the area and frequency overheads of these monitors and show how the coverage achieved by benchmarks varies during benchmark execution.

## 5.1 Test Setup & Benchmarks

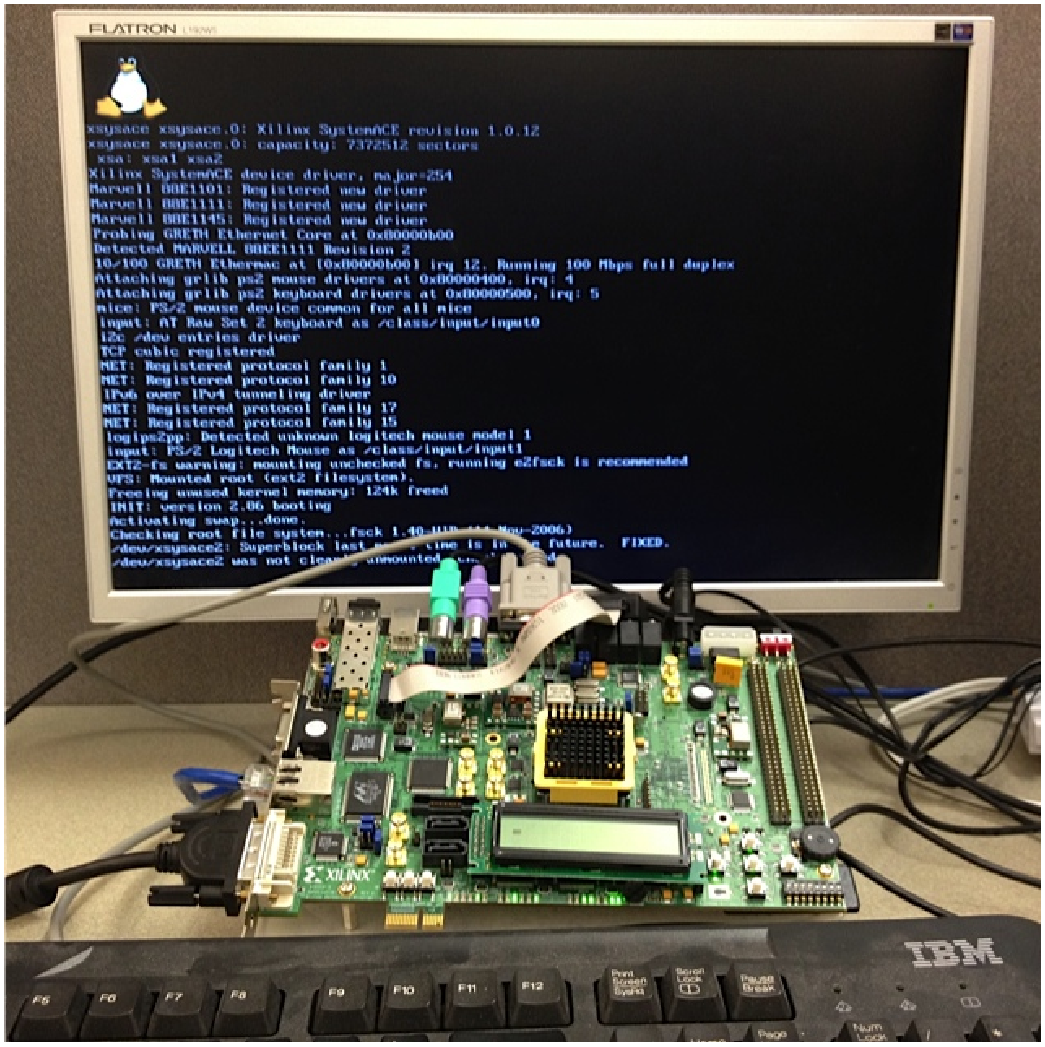
The system-on-chip which we have used is composed of the open-source LEON3 SPARCv8 processor, provided by Aeroflex Gaisler, and various peripherals supplied by the open-source community. The system configuration is detailed in Table 5.1. It is roughly comparable to a netbook or tablet device and can be fabricated to 0.13 µm ASIC technology at a speed of 400 MHz. Its maximum oper-

**Table 5.1:** LEON3 Configuration

| | |
|---|---|
| System Frequency | 70 MHz |
| Hardware Multiplier | 5-cycle (pipelined) |
| Hardware Divider | 35-cycle (pipelined) |
| Floating-point Unit | IEEE-754 compliant GRFPU<br>3-cycle add/subtract/multiply (pipelined)<br>16-cycle divide (not pipelined)<br>24-cycle square root (not pipelined) |
| Instruction Cache | 16 kB; 2-way set associative; 32B lines; LRU |
| Data Cache | 8 kB; 1-way set associative; 16B lines; LRU |
| MMU | Combined instruction/data TLB<br>Incremental replacement scheme<br>8 TLB entries; 4 kB page size |
| DDR2 SDRAM Controller | 190 MHz; 130 ns tRFC |
| DDR2 SDRAM Capacity | 256 MB |
| Ethernet | 100 Mbit/s |
| SVGA | 1024x768 resolution |

ating frequency on our Virtex-5 XC5VLX110T FPGA device is 70 MHz, a typical frequency for FPGA-based designs. The processor is fast enough to boot Linux and run the X11 windowing system, the board can see be seen booting Linux in Figure 5.1.

We selected the five IP blocks shown in Table 5.2 to instrument for timing-



```
xsysace xsysace.0: Xilinx SystemACE revision 1.0.12
xsysace xsysace.0: capacity: 7372512 sectors
 xsa: xsa1 xsa2
Xilinx SystemACE device driver, major=254
Marvell 88E1101: Registered new driver
Marvell 88E1111: Registered new driver
Marvell 88E1145: Registered new driver
Probing GRETH Ethernet Core at 0x80000b00
Detected MARVELL 88EE1111 Revision 2
10/100 GRETH Ethermac at [0x80000b00] irq 12. Running 100 Mbps full duplex
Attaching grlib ps2 mouse drivers at 0x80000400, irq: 4
Attaching grlib ps2 keyboard drivers at 0x80000500, irq: 5
mice: PS/2 mouse device common for all mice
input: AT Raw Set 2 keyboard as /class/input/input0
i2c /dev entries driver
TCP cubic registered
NET: Registered protocol family 1
NET: Registered protocol family 10
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
NET: Registered protocol family 15
logips2pp: Detected unknown logitech mouse model 1
input: PS/2 Logitech Mouse as /class/input/input1
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 124k freed
INIT: version 2.86 booting
Activating swap...done.
Checking root file system...fsck 1.40-[illegible]-Nov-2006)
/dev/xsysace2: Superblock last [illegible] time is in [illegible] future.  FIXED.
/dev/xsysace2 was not clea[illegible] unmounted [illegible]
```

**Figure 5.1:** LEON3 SoC running on a Virtex-5 FPGA shown booting Linux.

**Table 5.2:** IP Blocks under Test

| IP Block | Lines of RTL | Description |
|---|---|---|
| ddr2spa | 1924 | DDR2 Controller |
| gwfpwx | Netlist Only | IEEE-754 Compliant FPU |
| iu3 | 3131 | 7-stage Pipelined Integer Unit |
| mmu | 1929 | Memory Management Unit |
| mul32 | 24477 | (Un)signed 32-bit Multiplier |

path coverage. All IP blocks were available in VHDL except for the floating-point unit (FPU) which was only available in a netlist format. One advantage of the XDL-based coverage monitor integration (as opposed to integrating monitors at the RTL level) is that it allows us to instrument designs that do not have their source available.

## 5.2 Results

### 5.2.1 Coverage

Path coverage results for the five IP blocks shown in Table 5.2 were obtained for the top 2048 most critical paths, as determined by the Xilinx Timing Analyzer, for the tests shown in Table 5.3. These tests were chosen to be representative of what is generally run during the post-silicon stage of testing: various long-running system-

**Table 5.3:** Post-Silicon Validation Tests Used in Experiments

| Name | Description |
|---|---|
| dhry | Dhrystone [67] is a synthetic integer benchmark used to calculate general processor performance that does not stress cache performance |
| hello | The classic ‘Hello, World’ program consisting of a single printf statement |
| linux | Booting an embedded operating system (built using Buildroot 2012.02 with version 3.0.4 of the Linux kernel) |
| random | A series of 10,000 random programs generated by Csmith [74] |
| stanford | Eight small benchmarks gathered by John Hennessy and modified by Peter Nye, including integer matrix multiply, sorting, and permutation algorithms [26] |
| systest | Pre-silicon system-level tests for the LEON3 processor, supplied by Aeroflex Gaisler |

level tests, random tests, and application software. Included for comparison is the set of pre-silicon system-level tests supplied by Gaisler for the LEON3 processor.

Table 5.4 shows the coverage achieved for each test on each block, and Figures 5.2–5.6 plot the coverage obtained over time on these blocks. The dominant overall trend is that more cycles translate into better coverage, hence emphasizing the value of long-running post-silicon tests. For example, the `linux` benchmark tends to achieve the highest coverage, but the long-running `random` benchmark gradually catches up. We also see that coverage often comes in bursts, when an application starts a new phase of computation that performs different operations. There are also some peculiarities, e.g., the `random` benchmark does particularly well on the DDR controller. We believe this is because the benchmark is loading in a series of programs one-after-another from memory, so it exercises the DRAM extensively very early on. Similarly, the `stanford` benchmark does particularly well on the multiplier, presumably due to the matrix multiplication program in the benchmark suite. The `FPU` core has particularly low coverage for all tests. This is because many of the benchmarks used do not focus on floating-point arithmetic, and due to the complexity of the core, the number of elements in a timing path is very large (the average number of elements is 24.2, as shown in Table 5.6). It is interesting to note that this is the only test in which the pre-silicon `systest` outperforms all other tests. For a tester, this indicates that it would be useful to create a constrained random test based on the pre-silicon test. These results also confirm past work which indicates that testing floating-point units is very difficult, and that formal verification may be the best way of verifying such a core [25, 31]

Beyond overall coverage, the monitors give detailed path-level coverage information — we know exactly which paths were covered, and when. For example, an obvious question a validation engineer would ask is whether any test is dominated by the others. The coverage monitors allow us to answer this question. For example, Table 5.5 shows, for each test, how many paths were covered by that test, but not by some other test. For example, looking at the first row, second column, we see that `dhry` covered 945 paths that `hello` did not. Conversely, in the second row, first column, `hello` covered 24 paths that `dhry` did not. The last row and

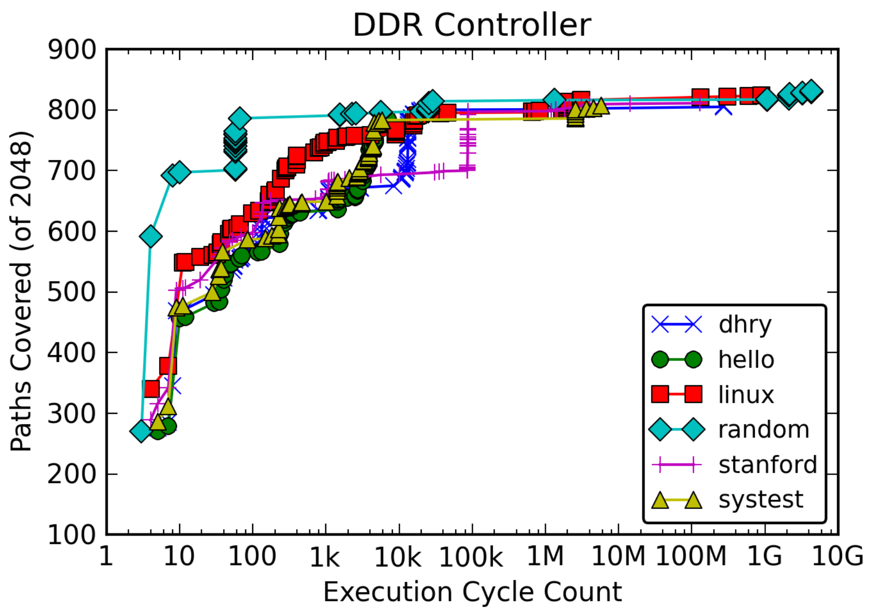


**Figure 5.2:** Coverage achieved over time for the DDR controller (ddr2spa).

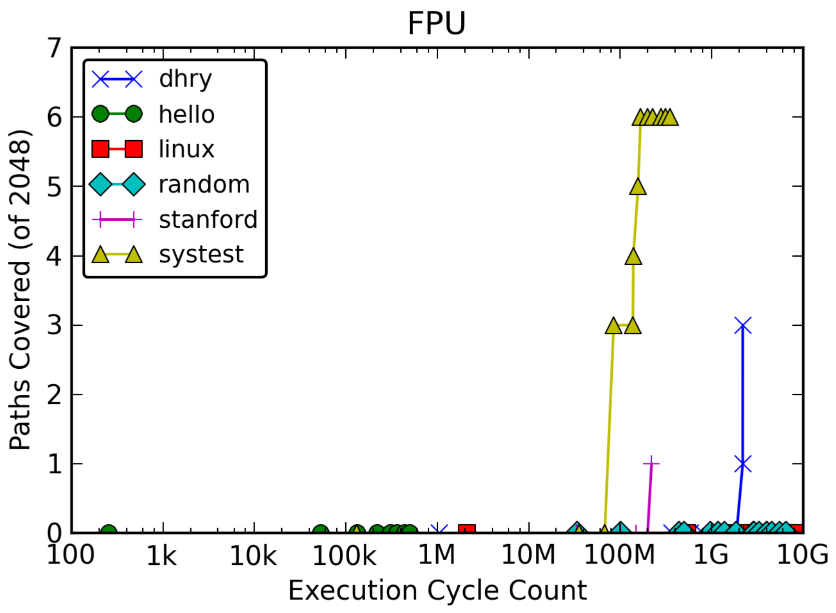


**Figure 5.3:** Coverage achieved over time for the floating-point unit (grfpwx).

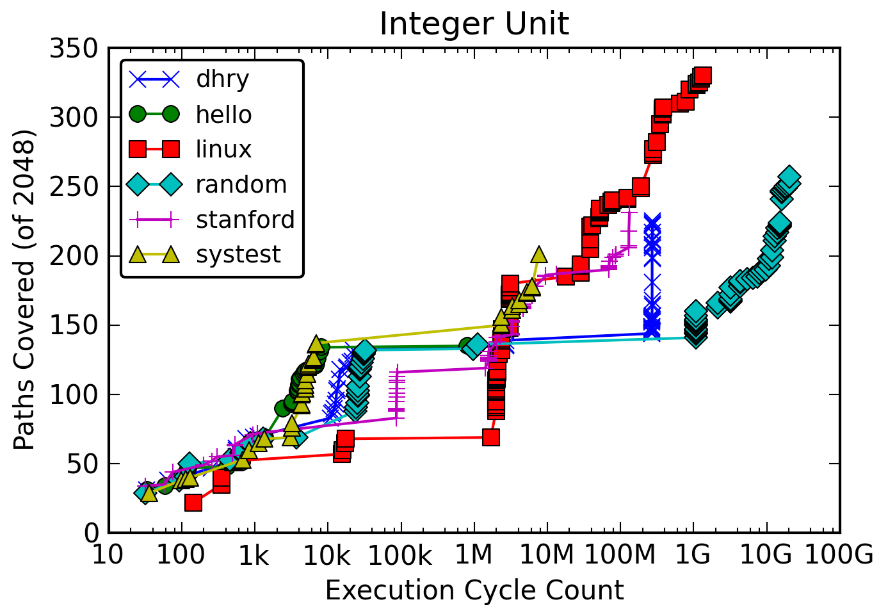


**Figure 5.4:** Coverage achieved over time for the integer unit (iu3).

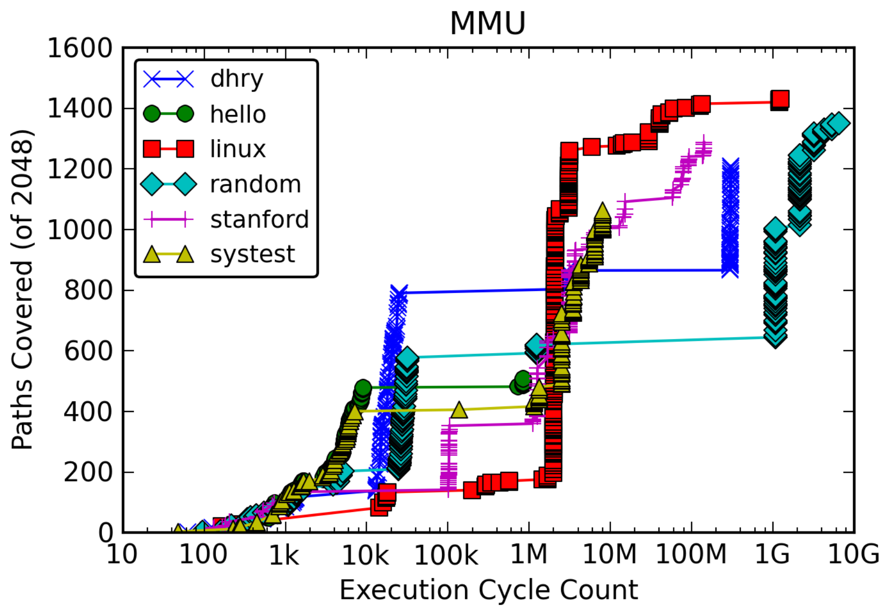


**Figure 5.5:** Coverage achieved over time for the memory management unit (mmu).

**Table 5.4:** Path Coverage Results (2048 Most Critical Paths)

| IP Core | dhry | hello | linux | random | stanford | systest |
|---|---|---|---|---|---|---|
| DDR Controller | 807 | 785 | 811 | 807 | 831 | 824 |
| FPU | 3 | 0 | 0 | 0 | 1 | 6 |
| Integer Unit | 225 | 135 | 330 | 257 | 231 | 201 |
| MMU | 1210 | 508 | 1431 | 1351 | 1288 | 1063 |
| Multiplier | 175 | 71 | 271 | 325 | 334 | 247 |

column compare each test to all the others. For example, the last row, first column shows that all the tests except for `dhry` covered 556 paths that `dhry` missed. Similarly, the first row, last column shows that `linux`, `random` and `systest` all covered paths which all other tests missed. In practice, this information is invaluable because it gives testers insight into the effectiveness of their tests and allows them to prioritize their test suites accordingly.

### 5.2.2 Area Overhead

Table 5.6 shows the area overhead of our instrumentation, in FPGA LUTs. It is also useful to visualize the area overhead per monitored path as the number of monitored paths increases. We generated monitors for 512, 1024, 1536, as well as 2048 paths, with the area overheads shown in Figure 5.7. From the figure, we see that the area per monitored path is roughly constant, as predicted. Indeed, it tends

**Table 5.5:** Coverage Achieved by One Test but not by Another

| | | ...which was not achieved by this test | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | dhry | hello | linux | random | stanford | systest | All Others |
| Coverage achieved by this test... | dhry | 0 | 945 | 7 | 24 | 56 | 366 | 0 |
| | hello | 24 | 0 | 2 | 1 | 9 | 110 | 0 |
| | linux | 443 | 1359 | 0 | 185 | 263 | 594 | 101 |
| | random | 368 | 1266 | 93 | 0 | 132 | 517 | 30 |
| | stanford | 301 | 1175 | 72 | 33 | 0 | 458 | 10 |
| | systest | 270 | 935 | 62 | 77 | 117 | 0 | 14 |
| | All | 556 | 1477 | 120 | 212 | 311 | 652 | |

to drop off slightly, due to a combination of shorter paths as we monitor less critical paths, and greater sharing and optimization between paths being monitored. This shows that we can always scale back the number of paths monitored if emulation area overhead becomes problematic or conversely, we can scale up the number of monitored paths to make use of unused FPGA resources.

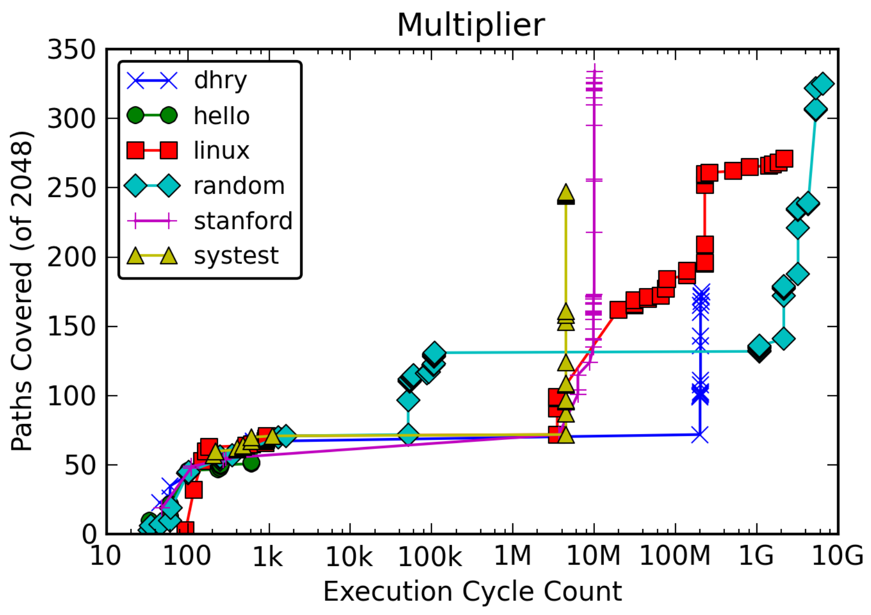


**Figure 5.6:** Coverage achieved over time for the multiplier (mul32).

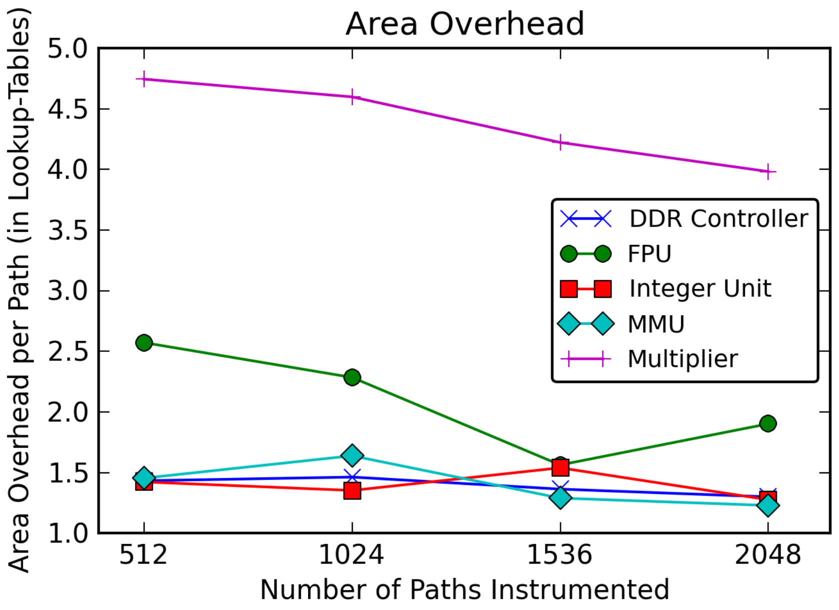


**Figure 5.7:** Area overhead per path shown as the number of targeted paths increases.

**Table 5.6:** Area Overhead to Monitor 2048 Critical Paths

| IP Block | Average Path Length (Xilinx primitives) | Area Overhead (Slices) |
|---|---|---|
| DDR Controller | 2.5 | 2654 |
| FPU | 24.2 | 2580 |
| Integer Unit | 7.4 | 1430 |
| MMU | 3.2 | 1946 |
| Multiplier | 14.6 | 2918 |

### 5.2.3 Frequency Overhead

The delay overhead imposed by our coverage monitors, although small, has been quantified in Table 5.7, ranging from no overhead to 1.94% . This overhead is due to the increased fanout on monitored signals. We believe that increasing the effort level of the Xilinx place and route tools would decrease this overhead. Alternatively, monitoring one fewer element along a timing path should reduce the frequency overhead further at a cost of slightly overestimating coverage. It is important to note that these monitors are much less costly on an FPGA than on an ASIC because FPGA interconnect is typically fully buffered [42].

**Table 5.7:** Frequency Overhead for 2048 Monitors

| IP Block Instrumented | Maximum System Frequency |
|---:|---:|
| None | 70.12 MHz |
| DDR Controller | 69.84 MHz |
| FPU | 68.64 MHz |
| Integer Unit | 69.24 MHz |
| MMU | 70.19 MHz |
| Multiplier | 70.18 MHz |

## 5.3 Chapter Summary

In this chapter, we demonstrated the feasibility of coverage monitors for the emulated version of a small but industrial-sized System-on-Chip design for a post-silicon-style test suite (as well as with pre-silicon tests provided by Aeroflex Gaisler, the SoC's industrial backer). We show how coverage is achieved over time on each IP core for every test, and show that in no case did the coverage achieved by a single test dominate all other tests. In addition, we presented the area and frequency overheads for these coverage monitors on our design.

# Chapter 6

# Conclusion

## 6.1 Dissertation Summary

This thesis presents a novel coverage-monitor-based approach for assessing the quality of post-silicon timing validation tests based on instrumenting a pre-silicon emulation version of the design. Our approach does not require a fabricated chip before beginning the creation and evaluation of post-silicon tests. Quantitatively evaluating post-silicon tests is important because it reduces the number of fabrication iterations in the post-silicon testing cycle and reduces the likelihood of emergency hotfixes or product recalls due to design errors because it allows post-silicon test quality to be increased with respect to an objective measurement.

In our experiments, we demonstrate the ability of our approach to measure the coverage achieved by a variety of typical post-silicon validation tests on a non-trivial System-on-Chip design. The timing overhead (impacting the speed of emulation) of the approach is small, between no overhead and 1.94%, depending on the IP core instrumented, and the area overhead is controllable, being proportional to the number of paths monitored. In general, the coverage achieved for our test suite was low, indicating that the tests evaluated are not very well suited for testing ASIC timing paths. Alongside sets of well-known software packages such as these, sets of constrained-random tests are usually designed to specifically exercise the critical paths in the system. The coverage information itself is useful because it shows which parts of the tests achieved coverage and provides a foundation for generating constrained random or directed tests that make use of this information.

## 6.2 Limitations

In this section, we identify a number of limitations of the research described in this dissertation. For each limitation, we propose how it could be addressed in future work.

### 6.2.1 Reliance on XDL

Ideally, this research would make use of a VHDL or Verilog parser to identify possible coverage points as well as integrate coverage monitors into the design. However, there are no robust open-source parsers of this kind available to the academic community. If such a tool were made available, it could easily be used to extend this body of work to coverage monitors integrated at the HDL level. Most importantly, it would allow this research to easily target any FPGA vendor's products. This research, as it stands, is reliant on Xilinx tools to provide a particular circuit-level description which is only supported by Xilinx and which, due to its unofficial status, may not be supported in the future.

In industry, however, an open-source parser would not be necessary. If a CAD provider wanted to design a tool to include coverage monitors into emulated designs, they could make use of their existing infrastructure for interacting with HDL.

### 6.2.2 Toolflow

Our current toolflow has a few limitations which can be addressed by future work. Firstly, the flow only has support for the Virtex-5 series of FPGAs. While it would not be difficult to port the existing flow to another Xilinx FPGA, it would be ideal if it could be generalized to support an architecture file that would describe the FPGA architecture (e.g., how to map XDL instances to VHDL instances). Unless the generated XDL instances fundamentally change[1], an architecture file would allow FPGAs which are similar to current FPGAs to be supported without requiring changes in the XDL-to-VHDL converter itself.

Furthermore, the generated VHDL description is in a single file. For our trivial example bundled with the converter, the generated VHDL description is only 508 lines, however the generated VHDL for the LEON3 SoC is over 200,000 lines, resulting in longer synthesis times than normal. Dividing the VHDL descrip-

[1]While it is possible for Xilinx to change the XDL instances generated by their `xdl` tool, it is unlikely given that these instances have remained stable for the last ten years.

tion into different files based on the hierarchy inferred from the XDL description would reduce file sizes and decrease CAD runtime, although it would complicate the converter and make it more difficult to debug designs which were generated incorrectly.

### 6.2.3 Register Retiming

Some circuit optimizations, such as register retiming, cause the circuit paths described by the ASIC RTL (before optimization) and the final circuit description (after optimization) to differ. If the circuit used for FPGA emulation is generated from the ASIC RTL description, the paths described by the STA tool will not be the same as those on the FPGA. One possible solution to this problem is to generate the FPGA circuit from the post-optimization ASIC description, similar to what was done in this thesis. However, to reduce tool runtime and increase circuit performance, it would be ideal to synthesize the FPGA description from behavioural RTL and not from a structural ASIC description. Another possible solution is to modify the coverage monitors to measure coverage across FPGA register boundaries when ASIC retiming is performed, however this would require additional flip-flops and a way to map post-optimization paths back to paths in the RTL. Retiming of the FPGA registers can be done without risk of affecting coverage because the retiming will only be done when it will not affect the top-level functional behaviour of the circuit.

### 6.2.4 Non-Determinism

Non-determinism during silicon validation is a major detriment to effective testing because it can make it difficult to identically reproduce specific test cases. To mitigate non-determinism, post-silicon tests are usually ran many times over a long period of time. In our case, non-determinism makes it more difficult to correlate coverage achieved during FPGA emulation with expected coverage on the ASIC. During this research we found that the final coverage values achieved by a given test did not vary, however, we did find there to be some small variation in the cycle count during which specific coverage points were achieved while booting Linux.

### 6.2.5 Area Optimizations

There are several straightforward but time consuming optimizations that could be made to our current toolflow to reduce area overhead. These optimizations include compression of coverage information, the use of lightweight monitors accompanied with a controller to unify data and handle off-chip data transfer, or the use of lossy sensitization functions. If recording the specific clock cycle in which coverage is achieved is not critical, this additional area overhead, albeit small, could be eliminated.

## 6.3 Future Work

We believe that this dissertation shows that coverage monitors in emulation systems can be used to complement post-silicon validation without additional overhead in the final product. However, there is little research into the development of coverage metrics specifically tailored for post-silicon validation (particularly those which can be effectively implemented on an emulation system). Given the value that coverage monitors in emulation can provide at low cost, it would be useful to identify coverage points appropriate for post-silicon validation that could be used to complement the functional coverage points currently specified by designers.

Specifically for timing validation, there are several promising avenues for further research. First, it would be useful to prune the paths reported by the static timing analyzer for functional reachability. This would make the coverage metric more accurate and reduce the number of coverage monitors which have sensitization conditions that will never be satisfied. Secondly, the current coverage monitor implementation provides a pessimistic view of coverage, since there are conditions under which the worst-case delay is experienced but the sensitization function is not satisfied. It may be useful to develop an alternative coverage monitor that provides a more optimistic view of coverage, with the true coverage lying somewhere between. Thirdly, our current monitors record only whether a path has been sensitized; additional monitors could be added to record whether the result is observable. Finally, coverage information could be used as a feedback mechanism to help develop better tests, e.g., via genetic algorithms.

## 6.4 Public Distribution

The XDL translation scripts have been made publicly available and can be found at http://ece.ubc.ca/~kyleb/files/xdl_translation_scripts.zip.

# Bibliography

[1] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *Proceedings of the 6th International Haifa Verification Conference (HVC)*, pages 60–75. Springer, 2011. → pages 20, 30

[2] Aeroflex Gaisler. LEON3 multiprocessing CPU core. http://www.gaisler.com/doc/leon3_product_sheet.pdf, 2012. Accessed July 2012. → pages 12

[3] Altera. SignalTap II embedded logic analyzer. http://www.altera.com/literature/hb/qts/qts_qii53009.pdf, 2012. Accessed May 2012. → pages 7, 21

[4] D. B. Armstrong. On finding a nearly minimal set of fault detection tests for combinational logic nets. *Transactions on Electronic Computers*, EC-15(1): 66–73, February 1966. → pages 39

[5] K. Balston, A. J. Hu, S. J. E. Wilton, and A. Nahir. Emulation in post-silicon validation: it's not just for functionality anymore. In *Proceedings of the 2012 High Level Design Validation and Test Workshop (HLDVT) [invited paper]*. IEEE, 2012. To be published. → pages iv

[6] K. Balston, M. Karimibiuki, A. J. Hu, A. Ivanov, and S. J. Wilton. Post-silicon code coverage for multiprocessor system-on-chip designs. *Transactions on Computers*, 99, 2012. → pages iii, 31

[7] P. Bastani, B. N. Lee, L.-C. Wang, S. Sundareswaran, and M. S. Abadir. Analyzing the risk of timing modeling based on path delay tests. In *Proceedings of the 2007 International Test Conference (ITC)*. IEEE, 2007. → pages 10

[8] D. Belete, A. Razdan, W. Schwarz, R. Raina, C. Hawkins, and J. Morehead. Use of DFT techniques in speed grading a 1 GHz+ microprocessor. In *Proceedings of the 2002 International Test Conference (ITC)*, pages 1111–1119. IEEE, 2002. → pages 24

[9] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 244 – 248, 2001. → pages 7

[10] J. Bhasker and R. Chadha. *Static timing analysis for nanometer designs: a practical approach*. Springer Verlag, 2009. → pages 21

[11] T. Bojan, M. Arreola, E. Shlomo, and T. Shachar. Functional coverage measurements and results in post-silicon validation of Core 2 Duo family. In *Proceedings of the 2007 International High Level Design Validation and Test Workshop (HLVDT)*, pages 145–150. IEEE, 2007. → pages 22, 30

[12] T. Chaney and C. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *Transactions on Computers*, 100(4):421–422, 1973. → pages 35

[13] A. C. L. Chiang, I. S. Reed, and A. V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *Transactions on Computers*, 21(2):189–195, February 1972. → pages 39

[14] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational aspects of the Pentium affair. *Computational Science & Engineering*, 2(1):18–30, 1995. → pages 2

[15] B. Cory, R. Kapur, and B. Underwood. Speed binning with path delay test in 150-nm technology. *Design & Test of Computers*, 20(5):41–45, 2003. → pages 24

[16] DailyTech. Understanding AMD's TLB processor bug. http://www.dailytech.com/Understanding++AMDs+TLB+Processor+Bug/article9915.htm, 2007. Accessed June 2012. → pages 2

[17] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: recording microprocessor history. *Communications of the ACM*, 55(4): 55–63, April 2012. → pages 8

[18] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the 1996*

*International Conference on Computer-Aided Design (ICCAD)*, pages 418–425. IEEE Computer Society, 1997. → pages 28

[19] R. Drechsler. *Advanced formal verification*. Springer, 2004. → pages 20

[20] EETimes. Intel finds design error in chip. http://www.eetimes.com/electronics-news/4212699/Intel-finds-design-error-in-chip, 2011. Accessed June 2012. → pages 2

[21] F. Fallah, S. Devadas, and K. Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *Transactions on Computer-Aided Design*, 20(8):1003–1015, 2001. → pages 28

[22] A. Goel and W. Lee. Formal verification of an IBM CoreConnect processor local bus arbiter core. In *Proceedings of the 37th Design Automation Conference (DAC)*, pages 196–200. ACM, 2000. → pages 20

[23] J. Goodenough and R. Aitken. Post-silicon is too late: avoiding the $50 million paperweight starts with validated designs. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 8–11. ACM/IEEE, 2010. → pages 7

[24] M. Gschwind, V. Salapura, and D. Maurer. FPGA prototyping of a RISC processor core for embedded applications. *Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):241 –250, April 2001. → pages 21

[25] J. Harrison. Floating-point verification. *Formal Methods*, pages 631–631, 2005. → pages 67

[26] J. Hennessy and P. Nye. Stanford integer benchmarks. 1988. → pages 66

[27] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development*, 26(1): 100–105, January 1982. → pages 21

[28] E. Hung and S. J. E. Wilton. Scalable signal selection for post-silicon debug. *Transactions on Very Large Scale Integration (VLSI) Systems*, (99):1, 2012. → pages 5

[29] J. Hurtarte, E. Wolsheimer, and L. Tafoya. *Understanding fabless IC technology*. Newnes, 2007. → pages 6

[30] D. Josephson and B. Gottlieb. The crazy mixed up world of silicon debug. In *Proceedings of the 2004 Custom Integrated Circuits Conference (CICC)*, pages 665 – 670, October 2004. → pages 21, 22

[31] R. Kaivola and N. Narasimhan. Formal verification of the Pentium 4 floating-point multiplier. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, page 20. IEEE Computer Society, 2002. → pages 67

[32] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, and E. Reeber. Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In *Proceedings of the 2009 Conference on Computer Aided Verification (CAV)*, pages 414–429. Springer, 2009. → pages 20

[33] M. Karimibiuki, K. Balston, A. J. Hu, and A. Ivanov. Post-silicon code coverage evaluation with reduced area overhead for functional verification of SoC. In *Proceedings of the 2011 High Level Design Validation and Test Workshop (HLDVT)*, pages 92 –97. IEEE, November 2011. → pages iii, 31

[34] C. Kern and M. Greenstreet. Formal verification in hardware design: a survey. *Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999. → pages 20

[35] J. Keshava, N. Hakim, and C. Prudvi. Post-silicon validation challenges: how EDA and academia can help. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 3–7. ACM, 2010. → pages 21

[36] T. Kirkpatrick and N. Clark. PERT as an aid to logic design. *IBM Journal of Research and Development*, 10(2):135–141, 1966. → pages 21

[37] B. A. Krishna, A. Sullerey, and A. Jain. Formal verification of an ASIC ethernet switch block. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 13–20. IEEE, 2010. → pages 20

[38] A. Krstić and K. Cheng. *Delay fault testing for VLSI circuits*, volume 14. Springer, 1998. → pages 23

[39] H. Krupnova and G. Saucier. FPGA-based emulation: industrial and custom prototyping solutions. *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 68–77, 2000. → pages 15

[40] Y. Kukimoto, M. Berkelaar, and K. Sakallah. Static timing analysis. *Logic Synthesis and Verification*, pages 373–401, 2002. → pages 21

[41] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings. Rapid prototyping tools for FPGA designs: RapidSmith. In *Proceedings of the 2010 International Conference on Field-Programmable Technology (FPT)*. IEEE, December 2010. → pages 49

[42] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and single-driver wires in FPGA interconnect. In *Proceedings of the 2004 International Conference on Field-Programmable Technology (FPT)*, pages 41–48. IEEE, 2004. → pages 41, 73

[43] J. Levine, E. Stott, G. Constantinides, and P. Cheung. Online measurement of timing in circuits: for health monitoring and dynamic voltage & frequency scaling. In *Proceedings of the 2012 Conference on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–116. IEEE, 2012. → pages 11

[44] M. Lin. *Introduction to VLSI Systems: A Logic, Circuit, and System Perspective*. CRC Press, 2011. → pages 6

[45] P. Lisherness and K. Cheng. An instrumented observability coverage method for system validation. In *Proceedings of the 2009 High Level Design Validation and Test Workshop (HLDVT)*, pages 88–93. IEEE, 2009. → pages 28

[46] X. Liu and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1338–1343, 2009. → pages 5

[47] S. L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium in a complete desktop system. In *Proceedings of the 2007 International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 53–59. ACM/SIGDA, 2007. → pages 21

[48] G. Martin, J. Berrie, T. Little, D. Mackay, J. McVean, D. Tomsett, and L. Weston. An integrated LSI design aids system. *Microelectronics*, 12(4): 18–22, 1981. → pages 21

[49] T. McLaurin. Creating structural patterns for at-speed testing: a case study. *Design & Test of Computers*, (99):1, 2012. → pages 26

[50] T. McWilliams. Verification of timing constraints on large digital systems. In *Proceedings of the 17th Conference on Design Automation (DAC)*, pages 139–147. IEEE, 1980. → pages 21

[51] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 12–17. ACM, 2010. → pages 10, 22

[52] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, April 1965. → pages 1

[53] A. Nahir, A. Ziv, and S. Panda. Optimizing test-generation to the execution platform. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 304 –309, February 2012. → pages 7

[54] A. Nordstrom. Formal verification — a viable alternative to simulation? In *Proceedings of the 1996 International Verilog HDL Conference (IVC)*, pages 90–95. IEEE Computer Society, 1996. → pages 20

[55] A. Offutt and S. Lee. An empirical evaluation of weak mutation. *Transactions on Software Engineering*, 20(5):337–344, 1994. → pages 27

[56] P. Patra. On the cusp of a validation wall. *Design & Test of Computers*, 24 (2):193–196, 2007. → pages 21

[57] D. Price. Pentium FDIV flaw — lessons learned. In *Micro*, volume 15, pages 86 –88. IEEE, April 1995. → pages 2

[58] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski. A 32 nm 3.1 billion transistor 12-wide-issue Itanium processor for mission-critical servers. In *Proceedings of the 2011 Solid-State Circuits Conference Digest (ISSCC)*, pages 84 –86. IEEE, February 2011. → pages 1

[59] C. Rowen. Reducing SoC simulation and development time. *Computer*, 35 (12):29–34, 2002. → pages 15

[60] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang. Intel Nehalem processor core made FPGA synthesizable. In *Proceedings of the 2010 International Symposium on Field-Programmable*

*Gate Arrays (FPGA)*, pages 3–12. ACM/SIGDA, 2010. → pages 8, 21, 22, 37

[61] B. Sprunt. The basics of performance-monitoring hardware. *Micro*, 22(4): 64 – 71, July/August 2002. → pages 30

[62] Tektronix. Clarus SoC post-silicon validation solution. http://www.tek.com/embedded-instrumentation/clarus-soc-post-silicon-validation-solution, 2012. Accessed July 2012. → pages 7, 22

[63] Tektronix. Clarus prototyper: multi-FPGA debug suite. http://www.veridae.com/index.php/products/clarus.html, 2012. Accessed May 2012. → pages 21

[64] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte. The MOLEN polymorphic processor. *Transactions on Computers*, 53(11):1363 – 1375, November 2004. → pages 21

[65] H. Wang, X. Gao, Y. Chen, D. Tang, and W. Hu. A multi-FPGA based platform for emulating a 100m-transistor-scale processor with high-speed peripherals. In *Proceedings of the 2010 International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 283–283. ACM/SIGDA, 2010. → pages 21

[66] P. Wang, J. Collins, C. Weaver, B. Kuttanna, S. Salamian, G. Chinya, E. Schuchman, O. Schilling, T. Doil, and S. Steibl. Intel Atom processor core made FPGA-synthesizable. In *Proceeding of the 2009 International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 209–218. ACM/SIGDA, 2009. → pages 8, 21, 22, 37

[67] R. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984. → pages 66

[68] C. Williams. Intel's Pentium chip crisis: an ethical analysis. *Transactions on Professional Communication*, 40(1):13 –19, March 1997. → pages 2

[69] Wilson research group. IC/ASIC functional verification study. 2010. → pages 15

[70] M. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the 1988*

*Workshop on Software Testing, Verification, and Analysis*, pages 152–158. IEEE, 1988. → pages 27

[71] Xilinx. ChipScope pro. http://www.xilinx.com/tools/cspro.htm, 2012. Accessed May 2012. → pages 7, 21, 60

[72] Xilinx. Virtex-5 libraries guide for HDL designs (UG621). http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/virtex5_hdl.pdf, 2012. Accessed July 2012. → pages 54

[73] Xilinx. Timing analyzer overview. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/pta_c_overview.htm, 2012. Accessed July 2012. → pages 12

[74] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM/SIGPLAN, 2011. → pages 66