Rapid Instrumentation for Debug and Verification of Circuits on FPGAs

by

Fatemeh Eslami

B.Sc., Shahid Beheshti University, 2009

M.Sc., University of Victoria, 2012

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

August 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Rapid Instrumentation for Debug and Verification of Circuits on FPGAs

submitted by Fatemeh Eslami in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
in Electrical and Computer Engineering.

Examining Committee:

Dr. Steven Wilton, Department of Electrical and Computer Engineering
Supervisor

Dr. Shahriar Mirabbasi, Department of Electrical and Computer Engineering
Supervisory Committee Member

Dr. Martin Ordonez, Department of Electrical and Computer Engineering
University Examiner

Dr. Ryozo Nagamune, Department of Mechanical Engineering
University Examiner

Additional Supervisory Committee Members:

Dr. Karthik Pattabiraman, Department of Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Field-Programmable Gate Array (FPGA) technology is rapidly gaining traction in a wide range of applications. Nonetheless, FPGA debug productivity is a key challenge. For FPGAs to become mainstream, a debug ecosystem which provides the ability to rapidly debug and understand designs implemented on FPGAs is essential. Although simulation is valuable, many of the most elusive and troublesome bugs can only be found by running the design on an actual FPGA. However, debugging at the hardware level is challenging due to limited visibility. To gain observability, on-chip instrumentation is required.

In this thesis, we propose methods which can be used to support rapid and efficient implementation of on-chip instruments such as triggers and coverage monitoring. We seek techniques that avoid large area overhead, and slow recompilation of the user circuit between debug iterations.

First, we explore the feasibility of implementation of debug instrumentation into FPGA circuits by applying incremental compilation techniques to reduce the time required to insert trigger circuitry. We show that incremental implementation of triggers is constrained by the mapping of the user circuits.

Second, we propose a rapid triggering solution through the use of a virtual overlay fabric and mapping algorithms that enables fast debug iterations. The overlay is built from leftover resources not used by the user circuit, reducing the area overhead. At debug time, the overlay fabric can quickly be configured to implement desired trigger functionalities. Experimental results show that the proposed approach can speed up debug iteration runtimes by an order of magnitude compared to circuit recompilation.

Third, to support rapid and efficient implementation of complex triggering ca-

pabilities, we design and evaluate an overlay fabric and mapping tools specialized for trigger-type circuits. Experimental evaluation shows that the specialized overlay can be reconfigured to implement complex triggering scenarios in less than 40 seconds, enabling rapid FPGA debug.

The final contribution is a scalable coverage instrumentation framework based on overlays that enables runtime coverage monitoring during post-silicon validation. Our experiments show that using this framework to gather branch coverage data is up to 23X faster compared to compile-time instrumentation with a negligible impact on the user circuit.

Lay Summary

Field-Programmable Gate Arrays (FPGAs) are programmable integrated circuits that can be used to prototype complex designs, such as processors, or to accelerate computationally intensive applications such as artificial intelligence algorithms. When incorrect behaviour is observed, finding the root cause (also known as debugging) is complicated by a lack of observability, due to the limited resources that can be exploited to observe internal signals. The above challenge is exacerbated by FPGAs very long compilation time, which is required to map a design onto FPGAs. The lengthy compilation time can take several hours and significantly reduce the productivity of the debug process.

In this thesis, we propose techniques for improving existing debugging techniques to provide observability in order to help designers to identify design errors and ensure the correct functionality of designs implemented on FPGAs without requiring long recompilations, accelerating debug process.

Preface

The research contributions presented in this thesis are based on work published in various workshops and conferences, and a journal article. In all of these contributions, I was primarily responsible for designing solutions, development, and performing experiments. This was done under the guidance and direction of my supervisor Dr. Steve Wilton. Dr. Wilton also was responsible for editing and writing segments of the manuscripts. I had the collaboration of Dr. Eddie Hung, from Invionics, for Chapter 6.

Below are publication details for each chapter:

- Chapter 3
  - Incremental distributed trigger insertion for efficient FPGA debug [42]: Fatemeh Eslami and Steven Wilton, IEEE International Conference on Field Programmable Logic and Applications (FPL), 2014.
- Chapter 4
  - An adaptive virtual overlay for fast trigger insertion for FPGA debug [44]: Fatemeh Eslami and Steven Wilton, IEEE International Conference on Field Programmable Technology (FPT), 2015.
  - Enabling Effective FPGA Debug using Overlays: Opportunities and Challenges [42]: Fatemeh Eslami and Steven Wilton, Workshop on Overlay Architectures for FPGAs (OLAF), 2016.
- Chapter 5

  - An Improved Overlay and Mapping Algorithm Supporting Rapid Triggering for FPGA Debug [43]: Fatemeh Eslami and Steven Wilton, ACM SIGARCH Computer Architecture News (HEART), 2017.
  - Rapid Triggering Capability using an Adaptive Overlay during FPGA Debug: Fatemeh Eslami and Steven Wilton, ACM Transactions on Design Automation of Electronic Systems (TODAES), To be published in 2018.

- Chapter 6
  - Extending Post-Silicon Coverage Measurement Using Time-Multiplexed FPGA Overlays [46]: Fatemeh Eslami, Eddie Hung, and Steven Wilton, IEEE European Test Symposium (ETS), 2018.

Table of Contents

List of Tables

List of Figures

Glossary

ASIC Application-Specific Integrated Circuit

CAD Computer-Aided Design

FPGA Field-Programmable Gate Array

HDL Hardware Description Language, e.g. Verilog, VHDL

HLS High-Level Synthesis

IC Integrated Circuit

LC Logic Cluster

LE Logic Element

LUT Look-Up Table

RTL Register-Transfer Level

Acknowledgments

First and foremost, I would like to express my gratitude to my academic advisor, Steve Wilton for his endless patience, encouragement, and unwavering commitment that gave me the opportunity to grow both technically and personally. I would not have gone half of this way without his invaluable guidance and support.

I owe a special thanks to Eddie Hung from Invionics Inc. for his support, thoughts, and insights (and as a never ending source of great ideas)- I truly learned a lot from you. I would also like to thank my committee members, Shahriar Miarbbasi, Karthik Pattabiaraman, Mieszko Lis, Martin Ordonez, Ryozo Nagamune, and Kiarash Bazargan for their valuable feedback.

I am thankful to my colleagues in the SoC lab for creating a friendly research environment. This includes Jeffrey Goeders, Parinaz Tehrani, Shadi Asadi, Al-Shahna Jamal, Jose Pinilla, Pavan Bussa, Hossein Omidian, Sarah Mashayekhi, Assem Bsoul, and several others.

To my family, you have been there for me through fear, uncertainty, and doubt in spite of the geographical distance between us. Thanks for making this distance bearable. Azam, thank you for being such supporting sister. Mostafa, thank you for supporting me, and for believing in me. This thesis is dedicated to the loving memory of my father and to my mother.

Chapter 1

Introduction

1.1 Motivation

In the nanometer era, producing correctly-working complex Application-Specific Integrated Circuits (ASICs) designs is becoming more expensive and time-consuming, primarily due to costly fabrication and the amount of engineering verification required to avoid unaffordable device re-spins [2]. Traditionally, circuit designers use software simulators, such as Mentor's ModelSim [3], to verify and debug hardware designs before fabrication; this is called *pre-silicon debug*. Although simulation-based verification is valuable, it is not enough. Many errors can only be observed when a design is running on the actual hardware for several reasons: (1) many bugs only emerge after long runtimes, and simulation is orders of magnitude slower than real silicon, (2) many "corner case" behaviours (i.e. those that are impractical to describe with a test-bench) may need real workloads before they can be observed, and (3) the most difficult bugs are often in the interfaces between a design and neighbouring chips; only by verification and debugging the system *in-situ* running on

the actual hardware such bugs can be found. This task is often called *post-silicon verification* (also known as post-silicon debug) [34, 48, 103, 104]. Post-silicon verification involves ensuring the design correctness. If an unexpected behaviour is observed, it is necessary to identify the root cause and fix design errors.

To avoid costly re-spins, post-silicon validation is often performed by implementing the design using a prototype built from one or more FPGAs. FPGAs provide the ability to reconfigure the design during validation, allowing for the evaluation of potential design variants, design updates, or fixing found bugs without the need of a costly re-spin. Increasingly, these FPGA implementations are being used as an initial version of the product, and only migrated to an ASIC if volumes warrant. Additionally, running a design at speed on an FPGA prototype with real-world stimulus allows for far more exhaustive and system-level tests (e.g. booting an operating system) that are infeasible in simulation.

Recent years have seen tremendous interest in using FPGAs for computationally intensive applications. Hardware acceleration is now entering the mainstream, as evidenced by Intel's acquisition of Altera, Amazon's incorporation of FPGAs in their EC2 F1 web service, and other efforts to bring FPGA technology into the cloud [10, 37, 73, 92, 109, 117].

Nonetheless, the widespread use of FPGAs in such services is limited due to their very long compilation time which can take several hours or even days for a large and complex design. Traditional hardware designers may be willing to accept long design and debug cycles, however, application designers using FPGA technology to accelerate software applications may not. These designers may expect software-like compile times, and similar support for debug and optimization. Although there has been much work on developing design compilers to support

designers that want to accelerate parts of their computation algorithms, designers need an effective mechanism to rapidly debug, optimize, and understand their designs.

Debugging consumes an increasingly high percentage of an entire project development lifecycle and has become a critical bottleneck, motivating the need for more effective and efficient verification technologies to improve productivity and decrease development time to meet time-to-market demands [80, 104]. As a result, both industry and the research community are now moving to considering entire debug "ecosystems" which provide the ability to rapidly debug and verify the functionality of designs implemented on FPGAs.

The primary focus of this thesis is on improving existing verification and debugging techniques in order to help designers to identify *functional bugs* and ensure the correct functionality of designs implemented on FPGAs for prototyping or accelerators used in production systems. Examples of functional bugs may be incorrect logic specifications in the RTL code, incorrect state transition, or IP misconfiguration.

1.1.1 FPGA Debug Instrumentation

Once the designer observes unexpected behaviour at the outputs of the system, he or she needs to start debugging to identify the root cause of the bug and fix design errors. Unlike manufacturing tests and pre-silicon verification that have benefited from automated methods and standard coverage metrics, post-silicon debugging is still designer-dependent and ad-hoc, meaning that designers with deep knowledge of the design are required to find the root cause of unexpected behaviour. Rather than trying to pin-point the root cause automatically, our approach focuses

on techniques to provide meaningful signal data behind unexpected behaviour to the designer, who can then use this information to find the bug and its root cause. Once the root cause has been identified, steps can be taken in order to fix the bug.

Fundamentally, identifying the root cause of a bug requires investigating and understanding the design's behaviour in detail. In simulation, the designer can observe all signal waveforms over the course of design execution. However, in a hardware implementation, the designer does not have access to all signal data due to a limited number of I/O ports. A widely-used approach to gain observability during debugging is to employ on-chip *debug instrumentation*. This instrumentation records the behaviour of a set of selected signals in on-chip memory blocks (also called *trace buffers*), coordinated by trigger circuitry that determines when data should be recorded. After execution, this recorded data is read out, and used in a custom tool to replay the behaviour, hopefully leading to insights into the operation of the circuit. Due to limited on-chip memory, the recorded data contains information only for the selected signals during a fraction of the circuit execution. To gain additional information about the operation of the circuit, the designer may need to change the selected signals or refine the trigger event to capture a different slice of circuit execution, which is known as a *debug iteration*. Depending on the complexity of the bug or the designer's expertise, several debug iterations may be required to find the root cause of unexpected behaviour.

In commercial tools [74, 119, 127] as well as academic work [29, 53, 83], the instrumentation is added to the user design before compilation. The instrumentation is then compiled along the user circuit using the FPGA's general-purpose logic and routing resources. Once the instrumentation is inserted, if the designer wishes to modify the instrumentation, another recompilation is necessary. The long com-

pilation times that are inherent to FPGA devices increase the time to perform a debug iteration. For very large designs, this can be prohibitive (often called a "go home event") which can severely limit debug productivity. Further, recompiling a design may often lead to slightly different implementation and timing behaviour which may cause a bug to disappear or change [81, 116]. Although FPGA vendors have made gains at reducing compile time in recent years, compile times are still long; therefore, a method of enhancing debugging of designs implemented on FPGAs that does not require frequent recompilations is required.

1.1.2 FPGA Coverage Instrumentation

During verification, it is important to evaluate the quality of test-cases that exercise the design and identify unexplored areas of the design using coverage metrics. Several standard coverage metrics and methods exist for measuring coverage of simulation-based verification. For example, code coverage measures what fraction of the source code (in terms of statement, branch, etc.,) has been executed by the tests, assertion-based coverage indicates how well a design's behaviour indicated by the designer has been tested, and mutation-based coverage measures what fraction of injected bugs are detected during simulation-based verification. Although these metrics can be easily measured with only little overhead to the simulation, it is not sufficient to completely guarantee the correct operation of a chip, either due to extremely long runtimes or its inability to mimic real-world interfaces in a pre-silicon testbench.

Since running the designs on FPGAs enables the designers to run far more complex and realistic tests compared to simulation, it should be leveraged for coverage analysis as well. Currently, post-silicon verification methods and tools are not

structured for coverage analysis and are only used to check functional correctness of the design. Quantifying the coverage of tests during post-silicon verification is difficult; this difficulty is rooted in the lack of visibility into the internal operation of a running chip [104].

On-chip monitors are a straightforward solution for increasing observability and measuring post-silicon coverage. However, implementing a large number of monitors (hundreds or even thousands) on-chip along with the user circuit can consume significant area and may make it infeasible to provide a complete coverage analysis. Although FPGAs enable implementing a limited number of monitors in multiple debug sessions, changing monitors requires a recompilation, recompiling can be very slow, often on the order of hours (or longer); this significantly limits verification productivity. Further, adding these monitors may perturb placement and routing of the user circuit, possibly changing the behaviour of the design. In this thesis, we aim to address these limitations.

1.1.3 FPGA Overlay Fabrics

In recent years, the concept of an *overlay* has emerged as a promising technology to tackle FPGAs long compilation time, and may become key to ensuring that FPGA technology is successful as it moves to the mainstream. Overlay fabrics, also called *intermediate fabrics*, are virtual configurable architectures implemented on top of an FPGA. Overlay fabrics provide a trade-off between flexibility and compile time; by constructing fabrics optimized for specific domains or circuit types, much of the flexibility can be removed leading to faster compilation times. A growing amount of research has investigated overlays for specific applications [26, 31, 39, 77, 78, 82, 86, 90, 97]. However, overlays are limited by their area and

performance overhead that prevented the realistic use of them in practice. The area and performance overheads are mainly because of FPGA resources used to enable the programmability of the overlay fabrics, and that overlay fabrics are typically designed without careful consideration of the underlying FPGA architecture. In this thesis, we aim to address this limitation and explore overlay architectures for an in-system debug ecosystem.

1.2 Contributions and Research Overview

1.2.1 Contributions

The main contributions of this thesis are novel techniques that support rapid and efficient implementation of on-chip *analysis instruments (such as trigger circuits and coverage monitoring circuits)* suitable for an in-system debug ecosystem, in a way that does not require large area overhead, and frequent design recompilations, enhancing FPGA debug productivity. More specifically, we make the following contributions:

- Addressing the feasibility and challenges of using incremental compilation techniques for inserting a trigger circuitry without changing the mapping (placement and routing) of the user circuit and without performing a full recompilation between debug iterations.
- A non-intrusive instrumentation framework that exploits virtual overlays to provide rapid triggering capabilities enabling rapid debug iterations during FPGA debug.
  - An adaptive strategy and CAD techniques to non-intrusively construct

the virtual overlay fabric on top of the user circuit only using leftover resources after user circuit compilation while preserving its original mapping.

  - An overlay architecture and CAD techniques suitable for rapid implementation of arbitrary combinational trigger circuits.
  - An overlay architecture and CAD techniques specialized for efficient implementation of state-based triggering functionalities.

- A scalable and area-efficient coverage instrumentation framework suitable for FPGA-based validation, which enables runtime implementation of on-chip coverage monitors.

1.2.2 Research Overview

Chapter 3 explores applying incremental compilation techniques to insert debug circuitry without performing a full compilation during debugging to enable fast debug iterations. We used incremental compilation techniques to implement trigger functions on top of an already placed-and-routed circuit without modifying the underlying circuit using only resources not used by the user circuit; we preserve packing, placement, and routing of the original user circuit when inserting the debug circuitry; this is important since it ensures that the original user circuit is compiled as normal and its mapping is fixed and does not change as the instrumentation is inserted, changed, or removed. This also allows designers to preserve low-level optimizations and timing closure of the original circuit as well as avoiding the CAD noise that is inherent to FPGA heuristic CAD algorithms. Our approach has two phases: *pack-place phase* and *routing phase*. In the *pack-place*

phase phase, the goal is to *distribute* the trigger logic to spare logic resources on the FPGA. In the *routing phase*, spare routing resources are used to make all the required connections in the trigger circuitry using incremental routing techniques. We will show that using incremental compilation techniques for trigger implementation significantly reduces debug iteration times. However, the ability to insert trigger circuits in this way depends on (1) the spare logic resources, (2) the spare routing resources, and (3) the mapping of the user circuit. This directly motivates exploiting overlays for implementation of trigger circuits. The contribution from this chapter was published in [42].

Using the techniques from Chapter 3, we found it was often difficult to make the required connections when implementing trigger circuits. To address this, Chapter 4 presents an alternative approach. This approach uses a *virtual overlay architecture* that is constructed only once after user circuit compilation but can be reconfigured multiple times to implement different trigger circuits. Importantly, this does not require a recompilation of the entire user circuit between debug iterations. In Chapter 4, we present an adaptive strategy to construct the overlay fabric on top of the user circuit while preserving its original mapping. The overlay architecture is designed as a 2D torus and is made efficient by carefully matching the overlay to the underlying FPGA architecture, and adapting the overlay to use only the resources left unused by the user circuit. In this way, the downside of many overlay approaches is avoided and the overhead can potentially be reduced to zero.

During debugging, an algorithm is required to rapidly configure the overlay fabric. Hence, we propose a routing-aware simulated-annealing based placement algorithm to place a trigger circuitry into the overlay while optimizing for routability of the placement solution. This increases the likelihood of a successful trigger

insertion. Since the overlay is pre-synthesized, the algorithm that maps the trigger function to the overlay architecture can be fast.

We will show that the overlay architecture provides enough flexibility to implement different combinational triggering scenarios an order of magnitude faster compared to the traditional flow. Furthermore, it should be noted that any increase in the circuit critical path delay is temporary; the operating frequency of the circuit can be set as normal when the debug instrumentation is not required. The contribution from this chapter was published in [44].

While the overlay architecture and mapping algorithms presented in Chapter 4 provide support for rapid implementation of arbitrary combinational functions, it does not support more complex triggering capabilities, including sequential (state-based) trigger functions. Such complex trigger circuits are essential to ensure that the designer is able to make the best use of the limited trace buffer capacity. Hence, Chapter 5 presents an overlay architecture that is specialized and optimized to implement state-based trigger functions during debug. The architecture is designed to be adequately flexible to implement a wide variety of such functions. The architecture is also optimized to be area-efficient when implemented on top of a user circuit. To implement the logic part of trigger circuits, the overlay contains a number of cells arranged in multiple levels in a triangular reduction-network pattern. To support sequential part of trigger circuits, the overlay also contains a bank of flip-flops connected to the overlay cells. This chapter also presents several new CAD techniques that adapt to the topological characteristics of this specialized overlay for mapping complex trigger functions onto the overlay. Trigger mapping is divided into two steps: first, the combinational portion is mapped to the overlay cells using a gradual and simultaneous placement and routing algorithm to ensure a

legal mapping solution. Secondly, the flip-flops in the trigger circuit are placed into the flip-flop banks of the overlay fabric and connect to the combinational portion of the circuit.

We will show that our customized overlay fabric can be reconfigured to implement different combinational and sequential triggering scenarios in less than 40 seconds for our benchmark circuits, enabling rapid debug iterations. Furthermore, we compare our approach to a commercial tool, Intel's SignalTap II tool [74]. We find that our approach for trigger insertion during debugging can outperform the general-purpose incremental compilation feature of the Intel Quartus Prime tool in terms of runtime and area overhead. The contribution from this chapter was published in [43] and an extended version was submitted to a journal with minor revisions.

Chapter 6 describes a *scalable* and *adaptable* coverage instrumentation framework for FPGA-based coverage analysis which greatly reduces the area overhead of on-chip monitors. We made the observation that we can *re-purpose existing FPGA-based on-chip debug cores to facilitate on-chip coverage monitoring*. In this approach, an overlay fabric is implemented on the FPGA along with the user circuit; this fabric is flexible enough that it can be used to rapidly cycle through coverage monitor functions through multiple runs without requiring a slow recompile. Importantly, the fabric is added after the user circuit is placed and routed, using only leftover resources that are not used by the user circuit. The architecture is adaptable in that it can adjust to the number of logic blocks and routing tracks not used by the user circuit; if there is ample space available, a large fabric that supports many monitors in parallel can be used, while if space is tight, a smaller fabric that implements fewer simultaneous monitors can be used.

To achieve this, we employ an overlay fabric on top of the user circuit using FPGA spare resources after user circuit compilation. Because the overlay is optimized for coverage monitoring, the time to compile monitors to the overlay fabric is fast and the overlay can be reconfigured at runtime to implement monitors in a time-multiplexed fashion to gather coverage data during the entire design execution. We will show that using our approach to implement all monitors is up to 23X faster compared to compile-time instrumentation with a negligible circuit delay increase. The contribution from this chapter was published in [46].

1.3 Thesis Organization

In Chapter 2, we will discuss existing post-silicon debug techniques in more detail and provide an overview of how debug instruments are implemented and used in post-silicon debug for both ASICs and FPGAs. This chapter then reviews the most common coverage metrics used in simulation-based verification, and reviews prior work in implementing on-chip monitors for post-silicon validation. This chapter also reviews FPGA overlays and prior work that described overlay architectures and tools suitable for specific application domains to address FPGA long compilation times. Chapter 3 explores applying incremental compilation techniques to insert debug circuitry without performing a full compilation and addresses challenges of distributing trigger functions over leftover resources without modifying the mapping of the user circuit. Chapter 4 proposes a non-intrusive instrumentation framework that enables rapid implementation of trigger circuits through the use of debug overlays. This chapter then presents an adaptive strategy to map debug overlays onto underlying FPGA resources. Finally, Chapter 4 presents an overlay architecture and CAD techniques suitable for implementing arbitrary com-

binational trigger circuits into the overlay fabric. Chapter 5 presents a customized overlay architecture and mapping algorithms specialized for implementing state-based trigger functions.

Chapter 6 proposes a scalable instrumentation framework for implementing coverage monitoring in a time-multiplexed fashion suitable for FPGA-based coverage analysis. Finally, Chapter 7 summarizes the contributions of this thesis and provides directions for future work.

Chapter 2

Background and Related Work

Post-silicon validation can be divided into two important tasks: (1) collecting signal data to verify design behaviour and find the root cause of observed bugs; (2) assessing how exhaustively the design has been exercised during validation using coverage metrics which can provide useful feedback to identify which areas of the design may be inadequately tested. Performing these tasks during post-silicon validation is complicated by a lack of *observability* due to the limited number of pins that can be used to observe design behaviour and collect data.

This chapter first describes how on-chip debug instrumentation is used to provide visibility into the design and discusses some of the key challenges of implementing debug instruments during post-silicon debug in Section 2.1. It also discusses several efforts from academia and industry on enhancing visibility and efficient implementation of debug instrumentation for ASICs and FPGAs. This section provides core information for the contributions of Chapters 3, 4, and 5 of this thesis. Next, Section 2.2 discusses pre-silicon simulation-based coverage metrics, and provides a detailed survey of the approaches that use on-chip coverage

monitors to provide visibility for measuring coverage at post-silicon. The latter provides information related to the contribution of Chapter 6 of this thesis. Finally, Section 2.3 describes overlay fabrics for FPGAs and reviews the existing work in this area as FPGA overlays play an important role in the contributions from this thesis.

2.1 Post-Silicon Debug Instrumentation

For many types of bugs, such as straightforward functional errors, software simulation is suitable. For other bugs, however, simulation-based verification may be too slow to expose the bug, especially for large and complex designs that include both software and hardware components. For many timing and interface errors, it may be impossible to observe the bug until the design is implemented on a hardware executing at speed *in-situ* using realistic test cases, such as booting an operating system that is too slow to be simulated. For these reasons, many bugs can only be found by observing the circuit implemented and running in hardware interacting with other system components. A post-silicon validation process is, therefore, an essential part of any integrated circuit design flow [34, 48, 103, 104].

This thesis focuses primarily on post-silicon debugging techniques that aim to ensure design correctness against errors caused by human designers as opposed to manufacturing defects. Manufacturing test is applied to every single device to detect fabrication defects (e.g. stuck-at faults). Unlike manufacturing testing, there are no standard bug models for bug scenarios during post-silicon verification. Bugs that appear at post-silicon are often corner-case bugs caused by subtle design errors and accurate modeling of such bugs may be very difficult [34, 55, 104]. During post-silicon verification and debugging, real-world long-running tests (such as

booting an operating system or application software) are applied to the design running at-speed and the design behaviour is observed. When incorrect behaviour in a running chip is observed, it is very difficult for designers to deduce the root cause of the behaviour due to poor observability into internal data. Two main approaches exist to enhance on-chip visibility: scan-based instrumentation and trace-based instrumentation.

Scan-based instrumentation is a technique used by manufactures to test for manufacturing defects in ASICs. A scan chain is a technique that involves connecting all the flip-flops of a circuit sequentially, providing the ability to observe or control the stored values. Since these scan chains are fabricated into ASIC designs, they can be reused for debugging functional errors [54, 122]. However, some FPGAs (such as Xilinx Virtex-5) have built in support for scan chains. If there is no built in support, Wheeler et al. showed that adding them requires on average 84% additional chip area [123]. One major drawback of scan-based techniques is that it only provides a single snapshot of the states. Hence, the circuit must be stopped at the right time to be able to extract values stored in flip-flops. Extracting the data is slow, and halting the circuit every time the designer wants to observe the stored values in flip-flops makes real-time observation impossible; the process of viewing one flip-flop using device readback techniques can take 2-8 seconds [61, 75, 76]. In [128], the author proposed scan-based techniques in which shadow flip-flops or latches are configured to periodically provide a snapshot of the circuit state during system operation without halting the circuit. However, their techniques cannot avoid the substantial area increase.

The most common technique to provide visibility without interrupting the circuit execution is trace-based instrumentation. As shown in Figure 2.1, this instru-

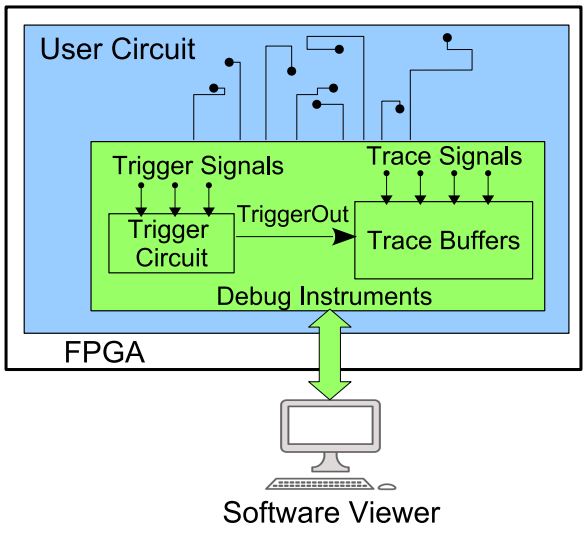


Figure 2.1: On-chip debug instrumentation including trace buffers to capture data, and a trigger circuitry to control capturing

mentation typically consists of *trace buffers*, which are on-chip RAM blocks used to store a history of internal signals (*trace signals*) for later interrogation. This data can then be provided to the user using a graphical user interface.

Since on-chip memory is limited, trace buffers can capture a subset of circuit signals for a limited number of cycles. Since this recorded data is critical in root cause analysis, it is important to ensure that the collected data is the most relevant to the unexpected behaviour that has occurred. To ensure that the collected data is useful, a routing network is typically provided to connect a subset of internal signals to the trace buffer, and *trigger circuitry* is included to identify events during the circuit execution at which data should be collected and stored in the trace buffers [29, 53, 74, 83, 85, 101, 119, 127]. This latter flexibility is of particular importance to the main contributions in this thesis. As shown in Figure 2.1, trigger circuitry consist of three parts: (1) the trigger function, which determines when

trace buffers should stop recording, (2) trigger signals, which are signals from the user circuit and are selected by the designer to construct the trigger condition, (3) a single trigger output, which is the output of the trigger function, and drives control logic within the trace buffers. Trigger circuits can be used in several ways. One option is that trigger circuit could be used to identify the point at which the trace buffers should *start* recording data, in which case, after a trigger event occurs, the trace buffer fills to capacity and then is frozen. Alternatively, the trigger circuit could be used to identify the point at which recording should *stop*; in this case, the memories can be configured as a circular buffer, continuously over-writing the oldest data until the trigger event occurs, at which point the buffers are frozen. In either case, after the trigger occurs, the trace buffers maintain a short history of selected signals which can help the designer uncover the cause of observed bugs. Trigger circuits can be combinational or sequential. As a specific example, a simple trigger function might determine when an internal bus carries a specific bit pattern; a more complex trigger might be a state machine that determines when a pre-determined packet is received on an input stream.

Trace-based instrumentation is commonly used in ASICs [4, 122]. Since these on-chip memories have a limited storage capacity, the designer must pre-select a number of internal signals to be connected to trace-buffers before chip fabrication. Several methods have been proposed to make a trade-off between the amount of visibility, the flexibility of changing selected signals after fabrication without a costly re-spin, and the chip area required to implement the debug instruments. Mentor Certus [101] provides proprietary observation networks to allow designers to instrument a large subset of signals during compilation; any subset of these signals can be selected for debug at runtime [110]. In [110, 111], a programmable

logic core has been embedded into the design along with an access network to provide the ability of connecting signals of ASICs designs to the trace buffer pins after fabrication. The authors in [112] proposed a debug infrastructure to support chips with multiple clock and voltage islands. In [5], the authors proposed a distributed reconfigurable infrastructure along with a signal routing network to implement debug structures. In [94, 95], the authors explored other types of trace interconnection networks and proposed a tree-like routing network to provide more flexibility for debug.

Data compression techniques have been also used to compress the real-time debug data to further enhance the storage efficiency of trace buffers [12, 13, 16, 41, 52, 108]. Several trace signal selection and state restoring methods have been proposed to maximize restoring un-traced signals using traced signals [7, 17, 19, 56, 63, 91, 98].

To further improve observability, several approaches were developed based on a combination of trace-based debugging and periodic scan chain captures. The idea is that a set of trace signals are stored in every cycle, whereas a set of scan signals are dumped across multiple cycles [18, 84]. In [113], the authors used several smaller scan chains with different dumping periods and proposed signal selection algorithms to assign the signals to different scan chains in order to maximize the number of states that can be restored.

2.1.1 FPGA Debug Flow

FPGAs can be configured after fabrication which provides a unique opportunity in that the instrumentation can be recompiled as the debug instrumentation changes during the debug process. Figure 2.2 shows a typical FPGA debug flow. In this

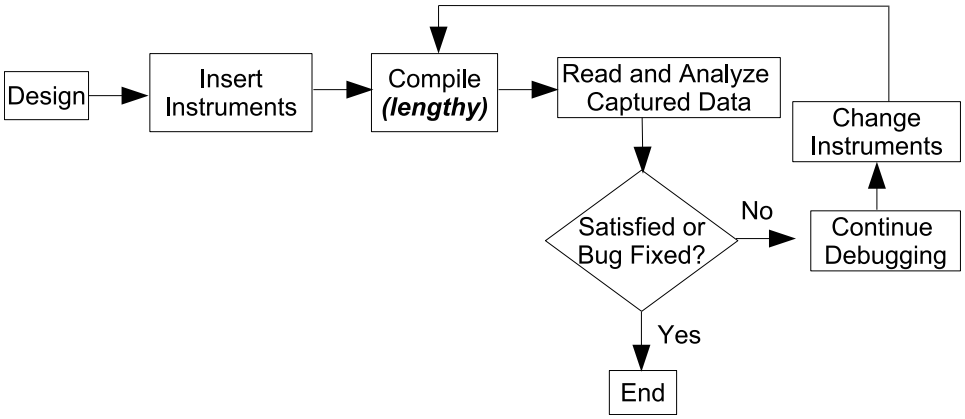


Figure 2.2: Typical FPGA debug flow in which instrumentation is inserted before compilation

flow, the instrumentation is inserted into the design before compilation. The design is compiled using standard tools, and runs as normal. As the circuit is running, the instrumentation records the behaviour of the selected signals in the trace buffer. After the run is complete, the designer can off-load this information for viewing and analysis, often using a waveform viewer.

During the hunt for the root cause of a bug, the designer may need to frequently refine his or her view of how the circuit operates, and may wish to re-run the circuit, recording a different set of internal signals or using a different trigger function condition so as to record behaviour in a different portion of the execution. This flow has three major limitations. First, a recompilation of the design is required whenever the trigger function or set of traced signals is changed during debug iterations. Recompiling today's large and complex designs can be extremely slow (several hours or even days) which can severely limit debug productivity. Second, recompiling can result in a different mapping (placement and routing), which may cause a bug to temporarily disappear or lead to different unexpected behaviour ("crash-

ing in a different way"). Third, the tool will treat the instrumentation and user circuit as equally important and optimize both together. This means, for example, if the critical path is in the debug instrumentation, the tool will not work as hard to optimize the timing performance of the user circuit. Moreover, it also means the debug instrumentation uses those FPGA resources that could be used by user circuit, rather than using only logic resources that the user circuit does not require.

Xilinx's Vivado [127] and Intel's Quartus Prime [74] use general-purpose incremental compilation to avoid full recompilation if the debug core is updated during debug iterations. Intel's SignalTapII [74] requires the designer to compile the debug core with the design as a separate partition and recompiles this partition separately if the property of the debug core is updated using general purpose incremental compile. Similarly, Synopsys's Identify [119] requires recompilation if reinstrumenting introduces new logic to these debug cores. However, because these are general-purpose techniques — in that they are designed to accelerate changes being made to the original circuit (e.g ECO changes), as opposed to simply adding new read-only instrumentation — these techniques can still be slow, especially if significant changes to the debug instrumentation are made, and often fall short of the software-like turnaround times many designers desire. Additionally, the entire design must be loaded into the memory of the workstation running the CAD tool. Therefore, incremental compilation techniques specific to debug instrumentation is required to enhance on-chip debug productivity.

Academic works have considered incremental routing techniques that connect signals to trace buffers or output pins without a full recompilation. In [58], the authors proposed inserting trace-buffers into a Xilinx FPGA at compile-time. Then, they applied bitstream modifications to route the trace signals to trace-buffers.

Hung and Wilton [64, 66] used incremental compile techniques to accelerate trace insertion and signal visibility. They speculatively inserted the debug instruments into the unused resources of FPGAs while preserving the original circuit mapping. They also described a network for rapidly routing trace signals to trace buffers without a recompilation; they created a virtual overlay network using unused multiplexers within the routing fabric and a routing algorithm has been used to route trace signals to the trace buffer. In [68], the same authors described a network flow based routing algorithm that incrementally connects the maximum number of requested signals to trace-buffers. This was made possible by the unique opportunity in signal tracing in which all trace buffer inputs are logically equivalent. In [87] the authors performed a signal parameterization on the user circuit and inserted multiplexers with parameterized selection bits to guide signals to trace buffers. However, those techniques do not address the incremental re-implementation of the trigger circuitry. The necessity to consider both logic and routing makes this a significantly harder problem.

In [69, 70], the authors used an incremental approach to insert additional logic (trigger circuitry in [70] and self-monitoring circuitry in [69]) to existing circuitry by finding a *contiguous* region of unused logic blocks that fits the entire additional logic. The difficultly with this approach is that on a highly-utilized FPGA, it may be difficult to find a region large enough, and with the right shape, to implement the additional logic. It may be possible to "reserve" space for an additional circuitry when compiling the existing circuit, but this interferes with the original circuit mapping or the design may no longer fit into the device. In addition, this would require knowing the size of the additional logic; if the size of the additional logic changes, and requires more logic elements, a complete recompile may be neces-

sary.

In [65], Hung and Wilton focused on the flexible implementation of the interconnect between user circuit and trace buffers, and provided a mechanism to implement only simple trigger circuits by using a portion of trace buffers, where trigger signals are connected to the address lines and different values on trigger signals will cause the RAM block to return a stored value. However, this approach only supports basic bitwise pattern-matching trigger functions. Commercial tools support state-based triggering. In Synopsys's Identify [119] and Xilinx's Vivado [127], state machines with between 2 and 16 states are possible, while in Intel's SignalTapII [74], state machines with between 2 and 20 states are supported. The range of number of trigger conditions that can be created is 1 to 16 in Synopsys's Identify [119], and 1 to 10 in Intel's SignalTapII [74]. Xilinx's Vivado [127] allows up to three conditional branches in each state. However, these capabilities must be specified before compilation and adjustments to these properties requires another recompilation.

HLS Debug

High Level Synthesis (HLS) tools have become available from FPGA vendors such as Intel [72] and Xilinx [126], as well as open-source academic LegUp project [30] to allow designers to translate a high level language, such as C/OpenCL, to HDL. This allows application designers to benefit FPGA technology to accelerate software applications without focusing on the hardware implementation details. Several research works presented techniques for debugging HLS-generated circuits and used trace-buffers to record data and later mapped the values back to the original source code for circuit execution replay [29, 51, 53, 105, 106].

In [105, 106], the authors inserted the debug instrumentation at the C level. In their approach, variables in C code are assigned to new top-level pointers. The HLS tool then converts these pointers to top-level ports in the generated circuit. These ports can be connected to the trace-buffers to record signal data. The authors used the HLS scheduling information to determine the states in which a signal is computed and record signals only in those states. In [29], instead of instrumenting the C code, the debug circuitry is added after the RTL is generated by the HLS tool. In this approach, a database was used to store the information about the original source code variables to the HLS generated circuit signals. Intel SignalTapII [74] logic analyzer was used to record a subset of signals into trace buffers. When the trigger condition fires, the recorded data is related back to the original source code variables to enable source-level debugging. Instead of using existing embedded logic analyzers that are built for any RTL hardware designs, Goeders and Wilton presented a customized instrumentation based on the information provided within the HLS tool and record data only if the signals are updated [51, 53]. They also used signal restoration techniques to improve trace-buffer utilization resulting in improved visibility compared to existing logic analyzers.

These techniques mainly focused on optimizing debug circuitry for recording longer execution histories to enhance observability. However, the lengthy compilation times of the back-end flow have been ignored. Changing the variables to be observed requires a full recompilation of the design and debug instrumentation. To avoid a full recompilation, Bussa et al. used the incremental design flow in a commercial FPGA CAD tool to re-instrument the design between debug iterations [27]. This leads to a 40% reduction in debug turn-around times compared to a full recompilation. Recently, Jamal and Wilton used HLS scheduling infor-

mation to provide highly customized architectural support for selective variable tracing, selective function tracing, and conditional buffer freeze limited to variable assignment in order to maximize trace buffer utilization [79]. To enable these capabilities, an architecture with limited ability to program is compiled with the user's design at compile time. During the debug, bits internal to the instrumentation can be changed to enable those capabilities without performing a full recompilation. However, as discussed in Section 2.1.1 compiling the design with the instrumentation prevents the tool from fully optimizing the user circuit and may change the critical path of the user's design. This may change the behaviour of the user's circuit.

In above mentioned works, a unique opportunity in instrumenting HLS-generated circuits is that they are created by the HLS tool and the HLS scheduling information within the HLS tool can be used to create and optimize circuit-specific debug circuitry for efficient use of hardware resources to enhance observability [105]. In contrast, this opportunity does not exist in optimizing existing debug cores used for RTL hardware designs which is the focus of this dissertation; the limitation comes from the fact that these debug cores are designed to support *generic* RTL circuits designed by a hardware designer and there is no knowledge of the behaviour of the circuit.

2.2 Coverage Metrics for Functional Verification

Coverage metrics enable designers to measure the adequacy of the design functional verification effort and to identify which parts of the design have been adequately verified and which parts need more verification. Test coverage has emerged as an essential metric for evaluating the effectiveness of both conventional simulation-

based verification and post-silicon validation. Since some of the coverage metrics used in simulation-based verification can be adopted for post-silicon validation, some of the common coverage metrics used in simulation-based verification are discussed in Section 2.2.1. Then, Section 2.2.2 provides a discussion on the prior works and methods for measuring coverage during post-silicon validation.

2.2.1 Simulation-based Coverage Metrics

There are a variety of coverage metrics to evaluate the effectiveness of tests during simulation-based verification. We will briefly discuss the prominent ones in this section.

Code Coverage

Code coverage is the most common coverage metric used in simulation-based verification and is inspired from software testing. Code coverage quantifies how well the design HDL code is exercised by the test-cases during simulation. The main code coverage metrics include statement/line coverage, branch coverage, and toggle coverage. Statement/line coverage reports whether a line of HDL code is executed at least once during simulation. Branch coverage reports whether all possible branches of if–else, case , and ternary operator (?:) statements are executed. Toggle coverage reports whether a signal has a transition during simulation. These metrics can be gathered automatically in simulation and does not require extra effort by designers.

Assertion Coverage

An assertion is a design property that should not be violated during simulation runs. An example assertion is ”acknowledge signal *ack* should become 1, one cycle after

the request signal *req* becomes 1". Assertions perform automatic checks and fire immediately when an internal error (malfunction) is detected, hence the designer knows where to start debugging [47]. Assertions are carefully crafted by a designer with an extensive knowledge of the behaviour of the design. Property Specification Language (PSL) and System Verilog Assertions (SVA) are the two commonly used languages for describing assertions [1, 28, 59]. Assertion coverage reports which assertions were successful and which ones failed in simulation. However, writing a comprehensive set of assertions that cover all functional aspects of the design behaviour imposes a significant challenge and there is no standard way to determine whether the assertion set is complete.

Mutation Coverage

Mutation-based verification involves injecting artificial bugs (known as *mutants*) into the RTL description of the design and checking whether it is caught during verification [9, 125]. For example, a mutant comprises of modifying operators (e.g replacement of "+" with "-"), or incorrect assignment statements. Mutation coverage reports what fraction of the mutants are caught during verification. Mutation coverage relies on the quality of the mutants and is computationally expensive since it requires injecting a large number of mutants, and each mutant must be inserted one at a time.

2.2.2 Post-Silicon Coverage Monitors

As mentioned before, although simulation-based pre-silicon verification is ubiquitous, it is not sufficient to completely guarantee the correct operation of a design, due to its long run-times, and its reliance on realistic testbenches to stimulate the

design. This justifies the need for a post-silicon validation process that is an essential part of any integrated circuit design flow.

Unlike pre-silicon verification and manufacturing tests that benefit from well-established coverage metrics and automated methods and tool flows, there is no standard metric and method to quantify the effectiveness of post-silicon validation tests. Evaluating post-silicon coverage is challenging due to the lack of visibility into the internal operation of integrated circuits.

Several approaches have been proposed for inserting on-chip coverage monitors and coverage measurement in ASICs. In [15], the authors manually inserted code coverage monitor flags alongside a multiprocessor SoC design code, and concluded the area overhead of on-chip monitors can be prohibitive, in one block reaching 22%. In [14], the authors used an emulated version of a design to measure path coverage of a validation plan for an ASIC. In [49], the authors instrumented a design in emulation to gather statistics and then used this data to choose a subset of coverage monitors for implementing on silicon. In [22], the authors instrumented Intel's Core 2 processor family for post-silicon coverage monitoring. However, only minimal coverage information was extracted due to area overhead of on-chip monitors. In [6], the authors used their emulation platform for the IBM Power7 to run processor-specific coverage monitors, providing an estimate of coverage by the same test in post-silicon. However, the area overhead of the coverage monitors was not disclosed.

Although assertions have been widely used in pre-silicon verification, assertion languages (e.g PSL, SVA) are not necessarily synthesizable to be re-used in post-silicon verification. Assertion synthesis tools and optimization techniques for silicon have been developed [23–25, 107]; however, on-chip assertion monitors

have not been widely used in practice since the silicon area overhead required to implement hardware assertion monitors limits the number of assertions that can be inserted on-chip. In [50], the authors proposed a technique (TMAC) that employs an embedded FPGA (eFPGA) centralized in a SoC design along with a signal routing network to enable assertion checking for post-silicon bug detection. However, the routing network area overhead grows with the number of signals to be monitored for assertion checking. In [120], the authors have shown that the area associated with adding all discovered assertions to a circuit can easily exceed 20 times that of the area of the circuit itself. Therefore, they proposed a methodology to rank assertions with the objective of increasing the likelihood of bit-flip detection and based on user-specific area constrains and only synthesized a subset of them for on-chip monitoring.

As an alternative to on-chip coverage monitors, in [45, 88] the authors performed trace analysis on signals recorded on trace-buffers to check whether an assertion failed. They proposed trace signal selection algorithms from an assertion coverage prospective to record those signals that are important for assertion checking. However, due to the limited capacity of trace-buffers, these methods can perform assertion checking only for the recorded cycles, instead of complete design execution.

In the above mentioned works, adding a large number of hardware coverage monitors to a design consumes a significant amount of chip area and these solutions are limited to a small number of important coverage monitors, and making a thorough evaluation of coverage infeasible. While ASIC designs require additional area for inserting debug instruments and coverage monitors, circuits implemented on FPGAs typically do not use all pre-fabricated logic and routing resources; this

spare capacity presents an opportunity to be used for debug and coverage analysis. Less work has been done on exploiting FPGA spare resources for inserting analysis instruments such as triggers and coverage monitors.

2.3 Overlay Architectures

Overlay fabrics are virtual reconfigurable architectures implemented within the user logic of an FPGA device. Using overlay fabrics optimized for a specific-class of circuits, much of the flexibility can be removed through abstraction of the physical FPGA details leading to faster compilation times and portability. That is, the designer only needs to map the application onto this pre-synthesized programmable architecture which is a simpler problem and is orders of magnitude faster (100x-1000x speedup) than invoking the FPGA vendor CAD tools to map the application to the fine-grained resources in FPGA. Because overlay fabrics are virtual, portability is also achieved across any FPGA device that can implement the overlay architecture. However, overlays are limited by their area and performance overhead. A growing amount of research has investigated overlays for specific applications and proposed fine grained and coarse grained overlay architectures.

A fine grained overlay (that is "FPGA on an FPGA") is defined as a virtual FPGA that is implemented by resources within a physical FPGA [26, 62, 86, 97]. In [97], the authors proposed a virtual fine-grained architecture to provide portability. However, the proposed virtual FPGA approach results in 100X more hardware usage compared to implementing the circuit directly onto an FPGA. The area overhead of fine-grained virtual architecture, ZUMA, was reduced to 40X by using physical LUTs to implement virtual LUTs and routing multiplexers [26]. In [86], the authors proposed a fine-grained overlay architecture providing portability for

custom instruction set extensions of softcore CPUs; the overlay interconnect was directly mapped to the physical switches resulting 2X-5X area overhead compared to direct implementation of custom instruction extensions into an FPGA with an order of magnitude area overhead reduction compared to related approaches.

As the fine-grained overlays are very costly to implement primarily due to their fine-grained interconnection network, coarse-grained overlays have been proposed to offer fast compilation, hardware design at a higher level of abstraction, improved programmability and runtime management. Coarse-grained overlays range from an embedded processor-style fabric which can be programmed using software [33, 115, 129], to an infrastructure that is specifically optimized for accelerator-type circuits [57, 77, 78, 93] or collections of small processing units [31, 32]. Nonetheless, these overlays suffer from a high area overhead compared to a customized FPGA circuit implementation due to the resources required to provide the programmability of the overlay architecture, particularly the routing network. For example, in [90, 118], the authors presented coarse-grained virtual FPGA fabrics, which are array of processing units specialized for computation of common image-processing kernels. Although this resulted a compilation time speedup of 554x compared to an FPGA CAD tool, the area overhead of this approach was 2.5X of the area of a circuit directly implemented on an FPGA, and the virtual interconnect was identified as the main source of the overhead. In [90], the authors explored the design space of the virtual interconnect and optimized the virtual interconnect to reduce the area by approximately 50% at the cost of 16% reduction in routability compared to the previous work. Additionally, optimizating and reducing the flexibility of the virtual interconnect resulted in 2.4X speed up in compilation time compared to the previous work. Similarly, in [38], the authors used an overlay fabric specialized

for control flow of common image-processing kernels. This approach required 17x more logic elements compared to Verilog implementation of the circuit.

We will show in this thesis that fine-grained debug overlays with reduced flexibility in the routing network can be made efficient by carefully matching the overlay to the underlying fabric, and adapting the overlay to use only the resources left unused by the user circuit. In this way, the area overhead can potentially be reduced to zero.

2.4 Summary

In this chapter, an overview of the existing post-silicon validation techniques for ASICs and FPGAs was provided. To counter the limited observability of hardware and provide support for root-cause analysis, the most common technique is to use trace-based instrumentation containing trace buffers to record data for analysis. To make a more efficient use of these limited on-chip memories, trigger circuitry is used to identify the time at which recording in a trace buffer should start and/or stop. It was explained that in traditional FPGA debug systems, debug instruments are inserted into the design before the design is mapped to the FPGA. The set of signals that are to be recorded and the trigger condition must be determined when the core is inserted. As the engineer refines his or her view of a suspect bug, he or she may wish to change the signals being recorded, or the time window in which data is stored in the trace buffer, meaning the entire design needs to be recompiled, imposing significant compile time overhead and consequently increasing debug cost; this chapter also reviewed work related to these limitations.

Most previous debug techniques have focused on increasing visibility into a circuit by collecting more signal data to help designer to identify the root cause of

an unexpected behaviour; there has been very little attention to the efficient insertion of trigger logic in FPGAs. The necessity to consider both logic and routing makes trigger implementation a significantly harder problem. Hence, this thesis mainly focuses on methods that provide efficient and rapid triggering capabilities without modifying the user circuit, eliminating the need of a full recompilation resulting in rapid debug iterations.

This chapter also reviewed some of the common coverage metrics used in simulation-based verification. It provided a discussion on the prior works and methods for collecting coverage data during post-silicon validation. Adding on-chip coverage monitors imposes significant area overhead. Through the contribution of this thesis, we demonstrate an efficient, and flexible framework that can be used to implement on-chip coverage monitors for code coverage measurement while reducing the area overhead of on-chip coverage monitors during FPGA-based validation.

Chapter 3

Incremental Compilation Techniques for FPGA Debug

As mentioned in Chapter 2, in traditional debug systems, any change in the debug circuitry requires a full recompilation, imposing significant compile time overhead and consequently increasing debug cost. This can be addressed using incremental routing techniques, such as those in [64, 66] to reclaim unused routing resources to incrementally build trace networks without modifying the user circuit. However, those techniques do not address the incremental re-implementation of the trigger circuitry. This chapter explores the feasibility and challenges of applying incremental-compilation techniques to insert the trigger circuitry without requiring a full recompilation.

The proposed debug flow is shown in Figure 3.1. If a designer wishes to modify the trigger circuitry, incremental techniques are used to only insert the trigger circuitry rather than recompiling an entire design. Besides avoiding the undesir-

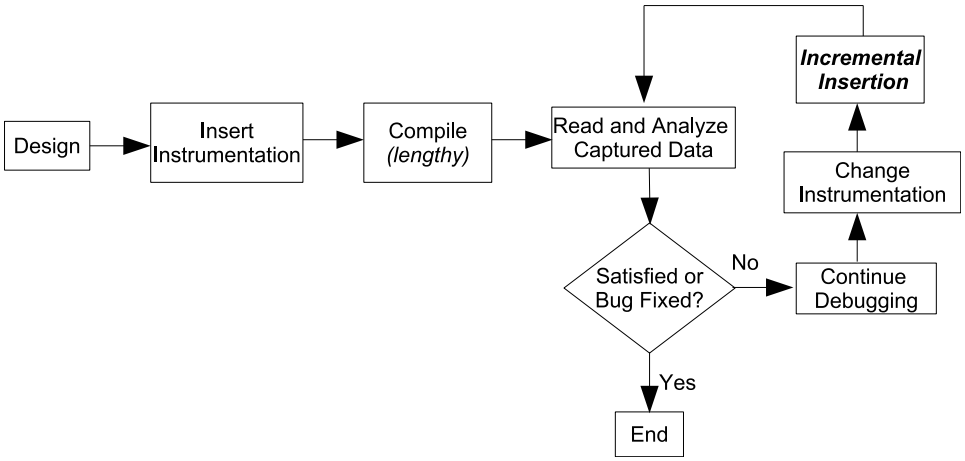


Figure 3.1: Incremental trigger insertion flow.

able compile-time penalty, this technique has the advantage of not modifying the mapping of the user design during debugging.

In our approach, which we call *incremental distributed trigger insertion*, we distribute the logic cells that make up the trigger function across logic elements in the FPGA fabric that are not used by the user circuit. These logic elements may not be contiguous. Intuitively, this will allow the trigger logic to take better advantage of any left-over space after mapping the user circuit, adapting to changes in the size or make-up of the trigger function. After placing the logic cells in unused locations, they must be connected to each other and to the user circuit using incremental routing techniques. The fact that we are not allowing any changes in the user circuit constrains the routing; using logic elements in parts of the design in which congestion is high may result in routing failure. Section 3.1 presents our incremental techniques for inserting the trigger circuitry. Section 3.2 describes our evaluation methodology and Section 3.3 evaluates our techniques using the academic tool VPR [96]. Lastly, Section 3.4 will provide a conclusion and closing

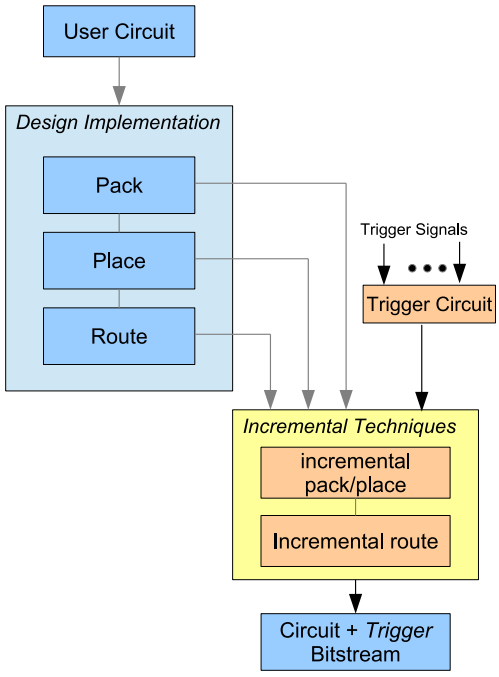


Figure 3.2: Overview of incremental trigger insertion CAD flow.

remarks. Parts of this chapter were published in [42].

3.1 Incremental Distributed Trigger Insertion

Figure 3.2 shows the procedure that we use to implement incremental trigger insertion. In this figure, the left side shows the user design compile flow. First the user circuit is packed and placed into FPGA fabric and then the circuit is routed. On the right side of Figure 3.2, our incremental trigger insertion flow is performed. In this chapter, we assume that the user specifies a trigger function and selects a subset of signals from all internal signals of the user circuit as inputs of the trigger function (in our experiments, we vary the size of the trigger function and hence the number of trigger input signals). We assume that the trigger circuit is parsed

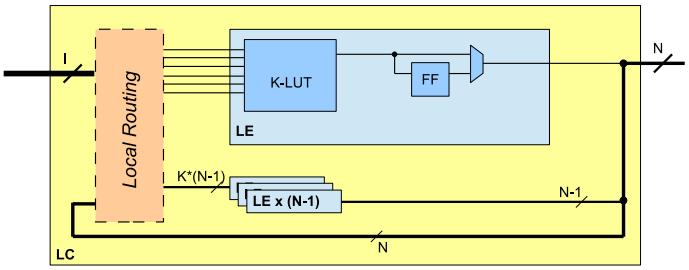


Figure 3.3: Overview of a Logic Cluster (LC) and Logic Element (LE).

and converted to a technology-mapped netlist using a technology-mapping algorithm such as [102]. The goal of our technique is to distribute the logic cells of the trigger netlist to spare logic resources on the FPGA fabric (we refer to this step as the *pack-place phase*), such that all of the required connections can be made in the *routing phase*. The insertion is constrained to not modify the packing, placement, and routing of the original user circuit – we do not allow the rip-up and re-route of any user circuit nets. Therefore, it is crucial to consider routability in trigger logic pack-place phase in order to lead to a successful routing. An overview of the physical FPGA architecture and the algorithms for pack-place phase and incremental routing phase are provided below.

3.1.1 Target FPGA Architecture

We assume the following FPGA architecture in this chapter, however, the overall technique will apply to any FPGA. We assume an FPGA consisting of a large number of Logic Clusters (LC). Using the terminology of [20], each LC itself contains N Logic Elements (LE) connected together through a local routing network as shown in Figure 3.3. Each LE consists of a K-input Look-Up Table (LUT) and a flip-flop. The number of inputs to each cluster is I. The LC output pins are con-

nected to global routing wires, and hence, provide connectivity between the LEs' output and the global interconnect. A family of FPGA architectures can be described using these three parameters: N, K, and I. In this chapter, we assume an FPGA architecture similar to the Intel Stratix IV device described in Section 3.2.

3.1.2 Incremental-Distributed Timing-Aware Trigger Pack-place

In this subsection, we describe how we select which unused LEs in our FPGA fabric will be used to implement the trigger logic. Our approach is to (1) select a *seed* location based on the positions of all the user selected trigger signals that will be tapped to implement the trigger function and (2) position all trigger logic cells in unused slots near the seed.

We first describe how we select the seed location. Intuitively, we wish to place the trigger logic close to the logic elements that drive the trigger signals to minimize the wirelength of the trigger circuits as well as the impact on the user circuit timing. Conveniently, since the trigger logic is inserted after the original circuit is mapped and fully routed, both the position and the timing slack of each trigger input is fully known. Therefore, to find a location for the seed, we find the location of the source of each trigger input signal in the user circuit, and then weight these positions by the signal's timing criticality (a signal with a higher criticality has a lower slack, as described in [20]). An example is shown in Figure 3.4; in this example, *Signal2* has a higher criticality than *Signal1*, meaning the seed location (x=4, y=2) is closer to the source of *Signal2* than to the source of *Signal1*.

Next, we choose sufficient unused LEs as close to the seed location as possible. Figure 3.4 shows our approach; after locating the seed point, we pack the remaining trigger logic in the empty logic elements surrounding this point in a spiral fashion,

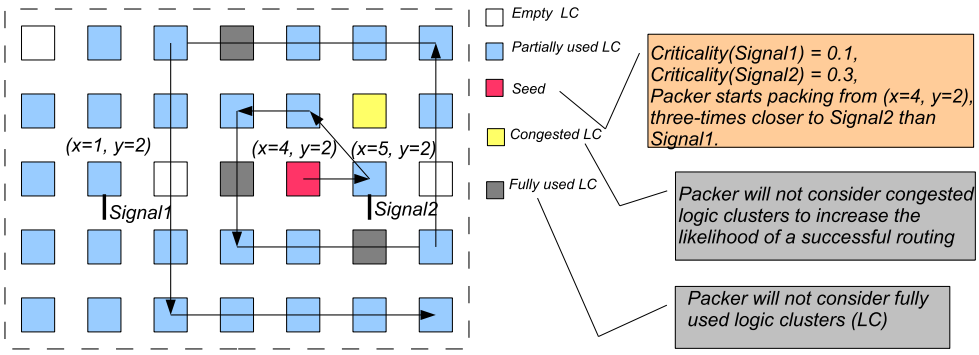


Figure 3.4: Finding spare LCs around the seed.Packer will not consider fully used logic clusters.

using an algorithm that will be described in the next subsection. The use of this spiral placement strategy is intended to keep the logic elements implementing the trigger logic as close together as possible. Although an iterative placement strategy could be employed, such a strategy would have a longer runtime.

3.1.3 Two-level Congestion Awareness in Trigger Pack-place

As described previously, it is important to consider routability in the pack-place phase in order to prevent routing failure in routing phase. If the routing resources around the seed point are heavily used by the user circuit, it may result in failure in trigger logic routing phase. To address this issue, we added two levels of congestion awareness to the pack-place phase: *LC-level congestion awareness* and *LE-level congestion awareness*. Algorithm 1 summarizes the two-level congestion aware pack-place phase.

After finding the seed point as described above, trigger logic packer-placer finds a LC with unused LEs. Then, the *CongestionLevel1* estimates the congestion of the routing nodes connected to the candidate LC's output pins. The congestion is

Algorithm 1: Two-level congestion-aware pack-place algorithm

```
Input:  trigger_cells_to_place /* list of trigger logic cells to be
        packed-placed*/
Output: Determine whether trigger logic has been packed-placed into
        FPGA
/*select a logic cluster as the seed location to start pack-place*/
LC = calculate seed point
while trigger_cells_to_place is not empty do
    if CongestionLevel1(LC) <threshold then
        T = select a trigger logic cell from trigger_cells_to_place
        unused_LE_list = unused LEs in LC
        foreach LE in unused_LE_list do
            if LE is not congested then
                pack T in LE of LC
                remove T from trigger_cells_to_place
                if trigger_cells_to_place is empty then
                    return True
                T = select a trigger logic cell from trigger_cells_to_place
    LC = go to next logic cluster
    if LC is empty then
        return False
```

determined by counting the number of LC's routing nodes used by the user circuit. If this congestion is larger than a predetermined threshold, then this LC is not used to implement the trigger logic; Figure 3.5 shows an example of where a LC was not selected to implement trigger logic.

Even if an LC has a congestion value lower than the predetermined threshold, it is still possible that the individual LEs inside the LC may suffer from high congestion on their outputs; using a highly congested LE can make it impossible to route its output through the global routing. Therefore, each LE in a selected LC is checked to see if its output is connected to fully used routing nodes, and if so, this

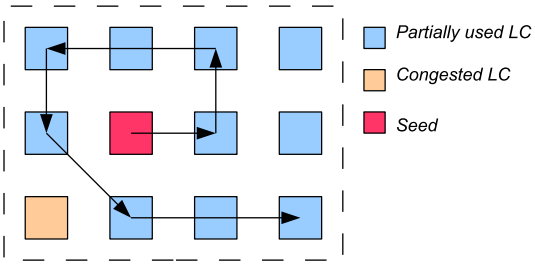


Figure 3.5: Skipping a highly congested LC.

LE is not used. Once a non-congested LE has been found inside a non-congested LC, the packer-placer inserts the trigger logic inside the LE (lines 6-8 in Algorithm 1. The algorithm repeats until sufficient logic elements have been selected to implement the trigger function.

3.1.4 Incremental Routing

After the logic elements have been selected, they are connected to each other, the trigger inputs are connected to the appropriate signals in the user circuit, and the trigger output is connected to on-chip memory blocks. We use the incremental routing technique described in [66] to implement the required connections. This technique uses routing tracks and switches that are not used by the user circuit to implement the required connections. Since routing can only use routing tracks that have not been used by the user circuit, congested circuits may lead to routing difficulties. To improve routability, we enhanced the routing algorithm to use unused LUTs as route-throughs. During routing, in the forward wavefront expansion phase, if an unused LUT is encountered, the router is able to expand the search wavefront through the LUT to its output pin.

Note that it is also necessary to connect the trace signals to the trace buffers as well; although we do not perform this step in this chapter, techniques such as those

in [64] can be used.

3.2 Evaluation Methodology

In this chapter, we investigate the feasibility of our approach using open-source VPR FPGA mapping tool, which is part of the academic Verilog-To-Routing (VTR) project [96]. Using an open-source tool was necessary because demonstrating our approach requires low-level resource manipulation to incrementally insert the trigger circuit into the bitstream after the user circuit has been compiled, whereas commercial tools do not provide this ability as device information is proprietary. Additionally, using VPR we can investigate the impact that changes in the routing supply may have on the technique's overall effectiveness.

3.2.1 CAD Flow

We extended VPR to support incremental-distributed trigger insertion flow as shown in Figure 3.6. In this figure, the left side shows the user design compile flow. The design is parsed and converted to a technology-mapped netlist and *.blif* file is produced. Logic elements are then packed into clusters, producing a *.net* file. VPR then places the LCs onto a minimum size FPGA fabric and then routes the circuit, producing *.place* and *.route* files.

On the right side of Figure 3.6, we show how incremental trigger insertion is performed on top of a fully place-and-routed user circuit. This new flow, makes no modification to the initial packing, placement and routing of the user circuit. Similar to the original flow, the trigger circuit is parsed and converted to a technology-mapped netlist. Then, the pack and place phases are performed by timing aware and congestion aware algorithms presented in Section 3.1. The algorithm from

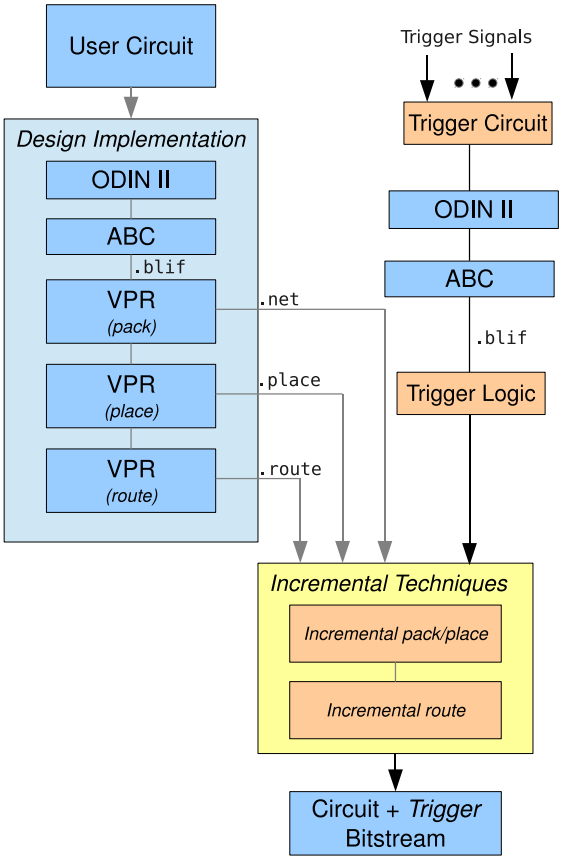


Figure 3.6: Incremental trigger insertion CAD flow.

that section was incorporated into T-VPACK. After inserting the trigger logic, any spare RAM memories are reclaimed as trace buffers.

Finally, the trigger circuitry is routed using the directed search algorithm used in VPR. We modified the directed search algorithm to ignore adding routing nodes that are fully occupied by the user circuit to the routing priority-queue; this reduces the search space during trigger routing. Since the trigger circuit is very small compared to the user design, the compilation time of the trigger circuit is significantly lower than a full design recompilation.

Table 3.1: Benchmark Summary.

Circuit	6-Input LUTs	FFs	W_{min}	I/O Used	I/O All	Logic Cluster Used	Logic Cluster All	RAM Used	RAM All	Global Signals	Free LCs (%)	Free LEs (%)
bgm	32384	5362	80	289	2400	4111	4200	0	120	23476	2	25
LU8PEEng	22634	6630	92	216	1962	2667	2745	45	80	16643	3	18
LU32PEEng	76211	20898	128	216	3552	9105	9213	150	252	55873	1	17
mcml	101858	53736	86	70	3200	7350	7400	38	208	52226	0	9
mkDelayWorker32B	5590	2491	76	1064	1344	916	1302	41	42	4686	30	47
mkPktMerge	232	36	44	467	832	18	494	15	16	515	96	22
or1200	3054	691	74	779	800	288	475	2	12	2602	39	11
raygentop	2148	1423	60	544	640	277	280	1	9	2126	1	35
stereovision0	11460	13405	52	354	1312	1240	1271	0	30	8358	2	31
stereovision2	29943	18416	92	331	2848	5889	5963	0	154	35386	1	57

3.2.2 Experimental Setup

Using the tool flow described above, we packed, placed, and routed a set of heterogeneous benchmark circuits that are supplied with the VTR project onto the smallest FPGA array that can accommodate the circuit using the default VPR architecture based on the Intel Stratix IV characterized by logic cluster size $N = 10$, look-up table size $K = 6$, cluster input and output flexibilities of $F_{c-in} = 0.15$ and $F_{c-out} = 0.10$, respectively, channel segment length $L = 4$, and inputs per cluster $I = 33$. For each benchmark, an FPGA size was chosen to be the smallest square that fit the benchmark circuit.

The benchmark circuits used in this chapter are shown in Table 3.1. In this table, W_{min} is the minimum channel width for which the circuit (without any debug instrumentation) can be fully routed; this quantity is often used as a proxy for the routability of a given architecture using a given CAD tool. The table also shows the total available resources in the minimum-size FPGA which the circuit fits into as well as the resources occupied by the user circuit. The number of global signals (both combinational and sequential) is also listed; these are the signals from which we choose trigger inputs. The column labeled *Free LCs* shows the percentage of the available LCs which are completely empty. When running VPR, we have disabled the default –allow-unrelated-clustering feature; this better mimics packing

in a commercial tool [67]. For each circuit, unused RAM blocks in the FPGA are reclaimed as trace buffer memories as in [66]. Circuits *mkDelayWorker32B* and *mkPktMerge* use most of the available memories of the FPGA. For *bgm*, *stereovision0*, and *stereovision2*, all available RAM blocks are reclaimed as trace buffers since they are not used by the user circuit. The column labeled *Free LEs* shows the percentage of unused LEs inside LCs which have been partially used by the user circuit.

In the results presented in the next section, we assume a trigger function that is a bitwise AND of between 32 to 1024 trigger signals. The trigger inputs are randomly selected from the circuit's global signals; we use multiple runs for each test and choose a different set of trigger signals for each run. As described in Figure 3.6, the trigger circuit is converted to a technology-mapped netlist.

3.3 Evaluation

3.3.1 Trigger Insertion

To investigate the effect of a relatively congested routing scenario on trigger insertion, we assumed a channel width of $W_{min} + 20\%$. In this case, we were able to successfully insert all 32 to 1024 input trigger functions for all benchmarks listed in Table 3.1, with some exceptions. Inserting a trigger function with more than 256 inputs was not successful due to routing congestion for *mkPktMerge* and *raygentop* because these benchmarks are too small to support such large triggering functions. For example, *mkPktMerge* has only 515 global signals, and inserting a trigger function with size of 256 inputs requires adding 335 nets (which is more than 50% of signals in the entire design) that imposed a very high routing congestion that could

not be resolved during incremental routing phase. Inserting a trigger function with size of 1024 failed for *or1200* due to the same reason (up to 512 was successful). Inserting the 1024-input trigger for *stereovision0* and *stereovision2* also failed because of high congestion during the incremental routing phase.

To determine how our technique works in situations with less routing congestion, we repeated the experiments with the channel width set to $W_{min} + 30\%$. By increasing the channel with to $W_{min} + 30\%$, we are able to successfully insert all trigger functions, with the exception of inserting trigger functions with 512, and 1024 inputs for *mkPktMerge* and *raygentop*, respectively. As discussed before, this is because these circuit were too small and inserting such a large trigger circuits resulted in very high routing congestion that could not be resolved during incremental routing phase.

As described in Section 3.1.3 we predetermined a threshold value for evaluating the congestion of each LC. We experimentally found that a congestion threshold of 60% for trigger functions with size 32 to 512 was enough to successfully insert the trigger function. For a trigger function with size 1024, we assumed a congestion threshold of 50% in the pack-place phase.

3.3.2 Effect on Flow Runtime

Figure 3.7 shows the total execution time (in seconds) and the breakdown for Pack-place phase v.s. Routing phase for benchmarks assuming a trigger with 256 inputs. We choose 256 inputs since it was the largest trigger size that was successfully inserted for all benchmark for W_{min} + 20%. In Figure 3.8, we show how the trigger insertion runtime using our incremental approach is reduced compared to the traditional flow where a recompilation is required to implement the trigger circuit.

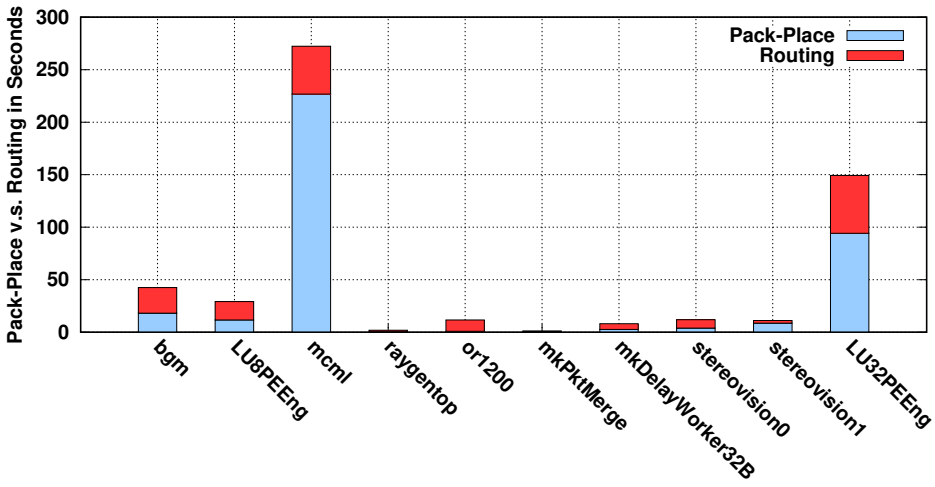


Figure 3.7: Trigger insertion runtime breakdown.

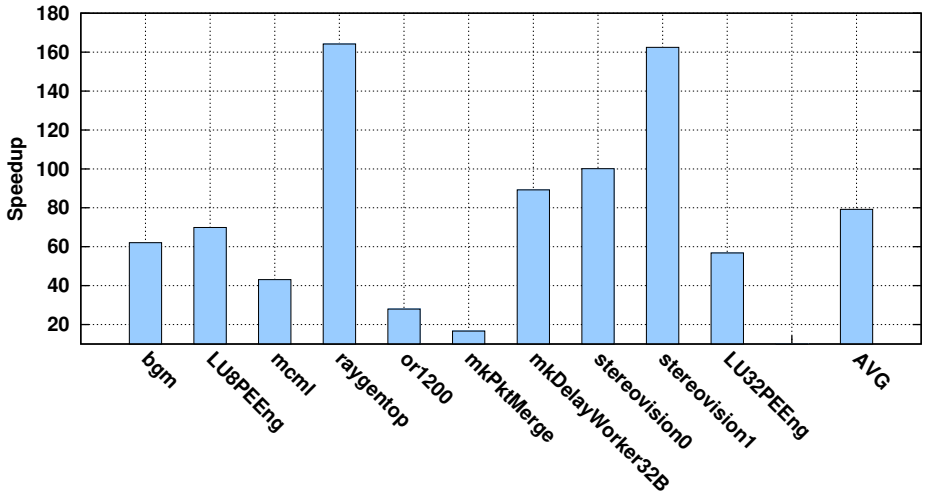


Figure 3.8: Incremental trigger insertion runtime saving in comparison to circuit recompilation.

As shown, on average, incremental-distributed trigger insertion is 80X faster than a full recompilation. This is mainly due to avoiding re-executing computationally expensive parts of the mapping flow for the user circuit.

Table 3.2: Effect of Trigger Insertion on Circuit Critical Path Delay (ns).

		Trigger Circuit Size					
Circuit	Original Delay	32	64	128	256	512	1024
bgm	19.71	23.33	23.84	23.68	24.13	24.65	26.03
LU8PEEng	89.63	89.57	91.32	94.55	96.35	97.09	97.28
LU32PEEng	91.93	92.05	92.58	92.96	93.53	93.61	97.55
mcml	66.58	66.58	66.58	70.12	70.6	72.26	72.46
mkDW	6.46	7.83	8.4	8.8	9.36	9.83	10.3
mkPktMerge	4.93	4.98	5.35	5.82	5.76	6.32	-
or1200	11.6	13.27	14.4	14.6	15.15	15.9	16.8
raygentop	4.22	5.38	6.42	6.76	7.32	8.07	-
stereovision0	3.47	5.52	5.8	6.77	6.9	7.81	8.78
stereovision2	11.93	15.81	16.66	16.62	17.37	17.7	18.4

3.3.3 Effect on Circuit Delay

The critical path delay of a circuit increases if the delay of any routes in trigger circuitry (including the routes between logic elements of trigger function, the routes form trigger signals to the trigger logic clusters, the routes from trigger output to all the trace buffers) is greater than the original circuit critical path delay. Therefore, circuits with a longer critical path delay are less sensitive to the trigger insertion since it is less likely that the inserted routes due to trigger circuitry become longer than the circuit critical path.

Table 3.2 presents the critical path delay of each benchmark circuit before and after inserting different size trigger circuits. As shown in Table 3.2, circuits with a short critical path delay, such as *or1200*, *mkDelayWorker32B*, *raygentop*, and *mkPktMerge*, experience higher increase in delay after inserting the trigger function. On the other hand, *LU8PEEng*, *LU32PEEng*, and *mcml* are less sensitive to the trigger insertion due to a longer critical path delay. Investigating the critical path delay of *bgm*, *stereovision0*, and *stereovision2*, we found that the critical path

of these circuits becomes longer mainly because all available memory blocks in the FPGA array are reclaimed as trace buffers. Hence, the single trigger output connects to all these memories across the chip creating a longer critical path.

3.4 Summary

This chapter explored the feasibility of using incremental compilation techniques for inserting debug logic, in a way that does not require a full design compilation. We distribute the logic cells that make up the trigger function across logic elements that are not used by the user circuit. After placing trigger logic in unused locations, they are connected to each other and to the user circuit using incremental routing techniques. From the study described in this chapter, we can conclude that incremental compilation techniques can be used to accelerate debug iterations. However, the success of our incremental-distributed approach in inserting trigger circuits over spare resources of a fully placed-and-routed design such that its mapping is completely preserved depends on routing congestion in the user circuit. To improve our techniques in using incremental compilation techniques for rapid triggering, the contributions in Chapter 4 are concerned with increasing the likelihood of a successful trigger implementation.

Chapter 4

Rapid Triggering Framework using FPGA Overlays

Chapter 3 explored using incremental compilation techniques for trigger implementation using spare resources after user circuit compilation. The main limitation of this approach was that it was often difficult to make the required connections when inserting trigger logic into a circuit because of the routing congestion in the user circuit.

In this chapter, we aim to increase the likelihood of making successful trigger implementation by improving the routability of trigger insertion by introducing a new instrumentation framework. To achieve this, a virtual overlay architecture is compiled to an FPGA only once after user circuit compilation, and then rapidly configured to implement instrumentation at debug time as shown in Figure 4.1. Although overlays can be notoriously inefficient, we will show that ours can be made efficient by carefully matching the overlay to the underlying fabric, and adapting

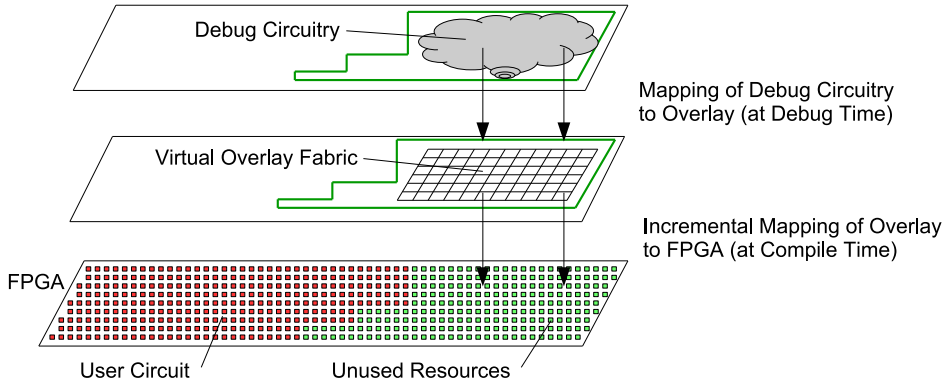


Figure 4.1: Overview of using a virtual overlay for FPGA debug.

the overlay to use only the resources left unused by the user circuit. In this way, the overhead can potentially be reduced to zero.

This chapter is organized as follows. Section 4.1 describes our new FPGA debug flow. Section 4.2 describes the architecture of our overlay. Section 4.3 presents a non-intrusive and adaptable approach for building the overlay architecture by reclaiming spare resources on FPGAs while preserving the underlying user circuit implementation. Section 4.4 presents CAD techniques to rapidly configure the overlay architecture, implementing the trigger circuitry. Section 4.5 details the experimental methodology and steps that were performed to evaluate our overlay architecture and CAD techniques. Section 4.6 presents the results from these experiments in terms of flexibility, runtime, and area overhead. This chapter is summarized in Section 4.7. Parts of this chapter were published in [44].

4.1 Debug Overlay Flow

Figure 4.2 shows our debug framework flow. There are two major phases: (1) compile-time; (2) debug-time.

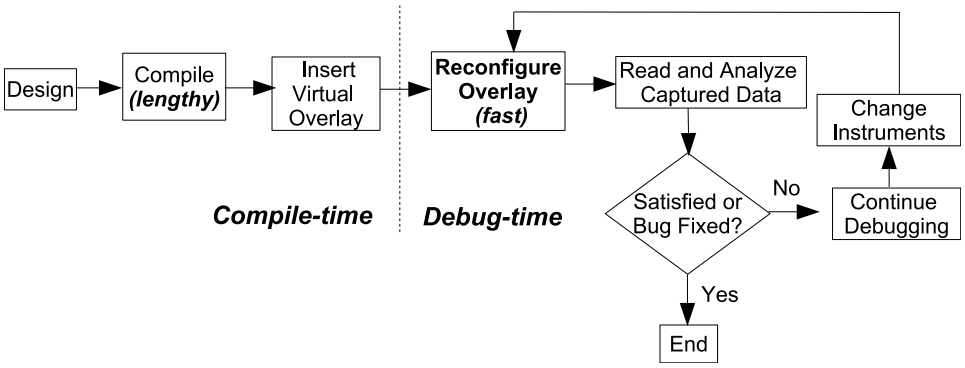


Figure 4.2: Proposed debug flow using an overlay for FPGA debug: compile time and debug time phases.

Compile time. First the user circuit is fully compiled onto FPGA fabric without any debug instrumentation. Then, the user circuit is frozen, and the overlay fabric is incrementally constructed, only once, using only those resources (logic blocks, memories, and routing tracks) that are *not used by the user circuit*. Because of this separation, we ensure that the user circuit characteristics are not affected by the instrumentation. This also allows us to adaptively size the amount of instrumentation (and hence its flexibility) depending on how much of the FPGA is unused by the user circuit.

Debug time. During each debug iteration in which a new trigger function is required, the function implemented in the overlay fabric is encoded in a set of FPGA configuration bits; these bits can be updated possibly using partial reconfiguration. A mapping algorithm is required to map the debug circuitry to this set of configuration bits; importantly, this algorithm is much faster than a full recompilation of the user circuit and instrumentation. This improves debug productivity, both by reducing the debug turn-around time, and also by ensuring that the user circuit is fixed and does not change as the instrumentation is changed. Since the overlay

architecture has a pre-synthesized routing network, the trigger mapping algorithm can be optimized to find a routable solution at debug time, whereas in the incremental approach presented in Chapter 3 there was not such an overlay to guide the routing.

4.2 Proposed Overlay Architecture

In this section, we describe our overlay architecture that is optimized to implement small logic functions that we expect to be commonly used as trigger functions. The architecture is designed to be flexible enough to implement a wide variety of such functions, area-efficient, and adaptable enough to be implemented on top of a user circuit.

The architecture consists of a square grid of *cells*. Each cell contains between four and n_o logic elements (LEs) connected using a fully-connected crossbar network inside of the cell. Each LE consists of a lookup table (LUT) and a flip-flop. Each look-up table has k_o inputs. Although other options for implementing overlay cells are possible [36, 121], building an overlay around LUTs has a number of advantages. First, a look-up table is universal and can implement any logic function of its inputs, simplifying the interconnect between the cells. Second, we can leverage a large body of lookup-table technology mapping work [35, 102]. Third, as we will show in Section 5.1.2, this strategy allows for an efficient mapping of the overlay to the underlying FPGA. We define $n_o = N$ where N is the size of each logic cluster in the target FPGA architecture (in our experiments, we assume $N = 10$). Hence, cells in the same overlay fabric may be of different sizes between 4 and N; as will be described in the next section, this provides a mechanism to adapt the overlay to the availability of empty LEs in the underlying user circuit implemen-

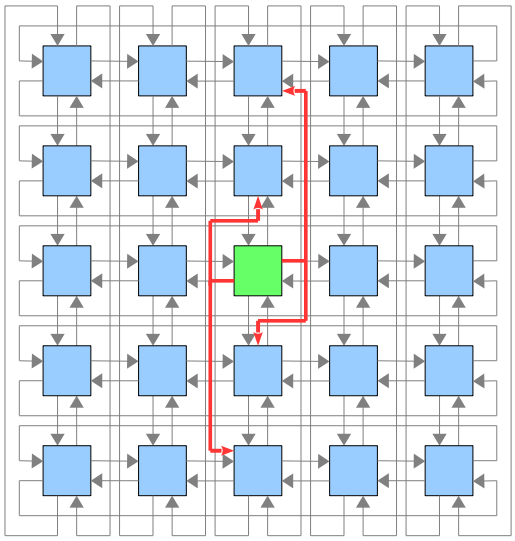


Figure 4.3: Simplified overlay architecture.

tation. For the same reason, we define $k_o = K$ where K is lookup-table size in the target FPGA architecture, in our experiments, we assume $K = 6$.

As shown in Figure 4.3, cells are connected using a 2D torus. Each cell has twelve input pins and four output pins that send and receive signals to and from neighbouring cells. Each output signal is connected to three sinks: one *primary* sink and two *secondary* sinks. The primary sink connects the output pin to the one of the cell's direct neighbour (e.g. to the north side). The primary sink for each of the four outputs connects to a different nearest neighbour (since this is a torus, each cell has four nearest neighbours). Although the primary sinks would be sufficient to provide complete connectivity between all cells in the overlay, additional routing flexibility is provided by the two secondary sinks. The first secondary sink of each output connects to a different nearest neighbour than the primary sink (e.g. the east side). The second secondary sink connects to a cell that is 1-hop away (not a nearest neighbour cell). This pattern is shown in Figure 4.3. We experimented

with higher and lower fanout counts for each output signal, however, we found this strategy gave a good balance between flexibility of the routing network and the routing congestion when implementing the routing network on the underlying FPGA.

As described above, some cells will contain more than four logic elements. Those logic elements do not drive any other cell; instead, they only drive other logic elements within the same cell through the local feedback crossbar.

4.3 Overlay Implementation Strategy

In this section, we describe our CAD techniques for mapping the overlay architecture on top of the user circuit without modifying the packing, placement, and routing of the underlying user circuit – we do not allow the rip-up and re-route of any user circuit nets, nor any relocation of any user circuit blocks.

Rather than describing the overlay fabric in a hardware description language (HDL) and compiling it using the normal FPGA CAD tools, we instead perform a direct mapping of the overlay primitives to the underlying FPGA primitives. For example, the overlay fabric consists of a number of look-up tables; we directly map each of these look-up tables to a look-up table in the underlying FPGA. Similarly, each routing path in the fabric is mapped to a specific routing path in the underlying FPGA. This leads to a much more efficient overlay implementation than compiling a HDL version using normal CAD tools. Furthermore, this technique avoids a downside of many overlay approaches: the high area overhead of implementing an "FPGA on an FPGA" is avoided. However, implementing such an overlay needs to employ custom incremental compilation tools. Details of the algorithm for overlay implementation are provided below.

```
1: procedure OverlayPlacementAndRouting(FPGA Resources)
2:     N = SelectLCsWithSpareLEs(FPGAResources)
3:     nxn = CreateLargest2DTorus(N)
4:     InititalPlacement(nxn)
5:     OverlayPlacement = SA-Placer()
6:     FirstLevelRouting=PathFinder()
7:     OverlayRouting=PathFinder(FirstLevelRouting)
```

Figure 4.4: Overlay placement and routing pseudocode.

4.3.1 Algorithm to map Overlay on top of User Circuit

Selecting exactly which look-up tables and routing resources in the underlying FPGA are used to implement the overlay fabric is a CAD problem; our algorithm to perform this is *adaptive* and *best-effort* in that (a) the number of cells in the overlay depends on the number of unused logic resources, (b) each cell in the overlay is sized according to the number of logic elements (LEs) unused within each logic cluster (LC), and (c) if certain connections in the overlay are difficult or impossible to implement, they are removed from the overlay and not implemented. Figure 4.4 shows the pseudocode for our mapping algorithm; details are provided below.

Selection

Given the mapping of the user circuit, the algorithm first identifies all logic clusters inside the FPGA which contain *at least* four unused logic elements (unused by the user circuit) as well as at least 12 unused input pins. This is due to the fact that each cell in the 2D Torus has four output pins that send and receive signals to and from neighbouring cells. These logic clusters represent potential sites in which the overlay can be implemented. We define the *size* of each site as the number of

empty logic elements in the corresponding cluster (this will be between four and the total number of logic elements in the cluster). We then prune out those sites in which the output pins are connected to routing tracks that have already been used by the user circuit.

We then choose some subset of these sites and create the largest possible overlay fabric out of the selected sites (in the experiments in this chapter, we first select *all* potential sites to maximize the size of the implemented overlay fabric, and later show the impact of relaxing this number somewhat). This overlay consists of a grid of cells; each cell corresponds to one site. The size of the site determines the size of the corresponding cell (recall from the previous section that the number of logic elements in each cell varies across the overlay). The result is a logical grid describing the overlay.

Placement

Since, at this stage, there is a one-to-one mapping between sites with sufficient space and cells in the overlay, we could simply map each cell to one site. However, it is likely that the available locations unused by the user circuit do not lie in a grid-like pattern; to adapt to the underlying pattern of available cells, we perform a simulated annealing placement algorithm to position the cells in the overlay onto available sites in the fabric. The cost function used by the annealer is the total wirelength of the overlay wires which is estimated by the bounding box heuristic. We only consider swaps which result in legal placements; a cell that consists of b logic elements can only be moved to a site that contains at least b empty LEs. This ensures that the final placement result is legal.

Creating Overlay Routing Network

After overlay placement, we attempt to create all connections between the cells. We employ the routability-driven PathFinder [100] algorithm to iteratively resolve routing congestion. Since we are restricted to using routing resources left unused by the user circuit, it may not be possible to find a legal routing solution in which *all* overlay wires are implemented. This is not a problem; in this case, we construct a fabric with "missing" connections. The mapping of the trigger function to the overlay fabric can attempt to construct a mapping that does not use any of the missing connections. This *best-effort* approach for mapping allows us to implement larger overlay fabrics than would otherwise be possible, since 100% routing is not required. To implement this best-effort routing, we perform a legalization heuristic after Pathfinder has completed (i.e. after 15 routing iterations). This heuristic iteratively discards illegal connections (from the overlay) that are overusing the largest number of routing resources until a legal routing solution is achieved. In this way, our approach can be used for user circuits that already stress the FPGA routing.

To minimize the impact of removing connections from the overlay routing network, we prioritize the connections in the overlay routing network, and route the network in two steps. We first use Pathfinder to route the *primary sinks* of each connection (primary sinks are defined in the previous section). We then use the same algorithm to route the *secondary sinks* of each connection. This approach ensures that the primary sink connections have higher priority for using routing resources than the secondary connections.

4.4 CAD for Trigger Mapping

At debug time, the overlay architecture can be configured to implement the desired trigger circuitry. This section describes the algorithm that maps the trigger circuitry to the overlay architecture.

The CAD tasks that must be performed to map a trigger circuit to the overlay are the same as in the traditional FPGA mapping flow: packing, placement, and routing [20]. However, as described in Section 4.3, the overlay architecture has limited flexibility in the routing network. Placement without considering the routing network can result in an un-routable solution. Therefore, it is important that the placement algorithm is *routing-aware* to make the placement solution more suitable for the given routing architecture.

4.4.1 Routing-aware Placement Heuristic

We choose simulated annealing to perform the placement of the trigger onto the overlay, due to its success in the FPGA placement field [20, 21, 40, 99] as well as its ability to target irregular architectures. The cooling schedule of the routing-aware placer is an adaptation of VPlace [99]. Initially, each logic element of the trigger netlist (i.e. LUT) is assigned to a random unoccupied logic element inside the overlay. Our algorithm is LE-based, in that individual LEs are swapped (as opposed to cluster-based placement solutions as in [20]). This allows LEs to migrate between clusters, providing better routability in the final solutions. Although this is more computationally-intensive, the trigger circuits being placed are relatively small (typically hundreds of logic elements).

The cost function used in the placement algorithm is the *product* of two terms: the *routing hops* cost and the *net routing* cost. Each will be described below.

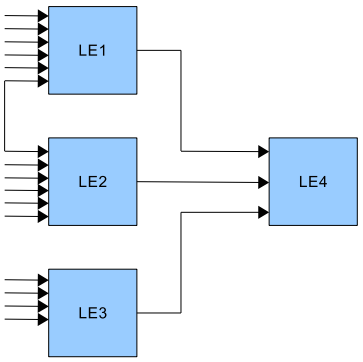


Figure 4.5: Simple trigger netlist.

Routing Hops Cost

Cells that are not neighbouring can be connected by routing through one or more intermediate cells, and configuring these intermediate cells as pass-through buffers. A placement solution that requires fewer routing hops is more likely to be routable than one that requires more routing hops since it requires fewer overall wires. To estimate the Routing Hops Cost, we use the bounding box size of a net, and sum this over all nets.

Net Routing Cost

Figure 4.6 shows how a simple trigger netlist of Figure 4.5 can be mapped to overlay cells. LE1 and LE2 are placed in the same cell, sharing an input pin as well as connecting to LE4 directly. Placing LE3 in the same overlay cell as LE1 and LE2 will result in routing failure; in case (1) the output pin of the LE of the overlay cell is only connected to the local routing crossbar and can not be connected to LE4. In case (2), although the output pin of the LE of the overlay cell can indirectly reach LE4 using one routing-hop through C3, all input pins of C1 are occupied by LE1, and LE2, resulting in an unavailable input pin for LE3. Placing LE3 in C4, can

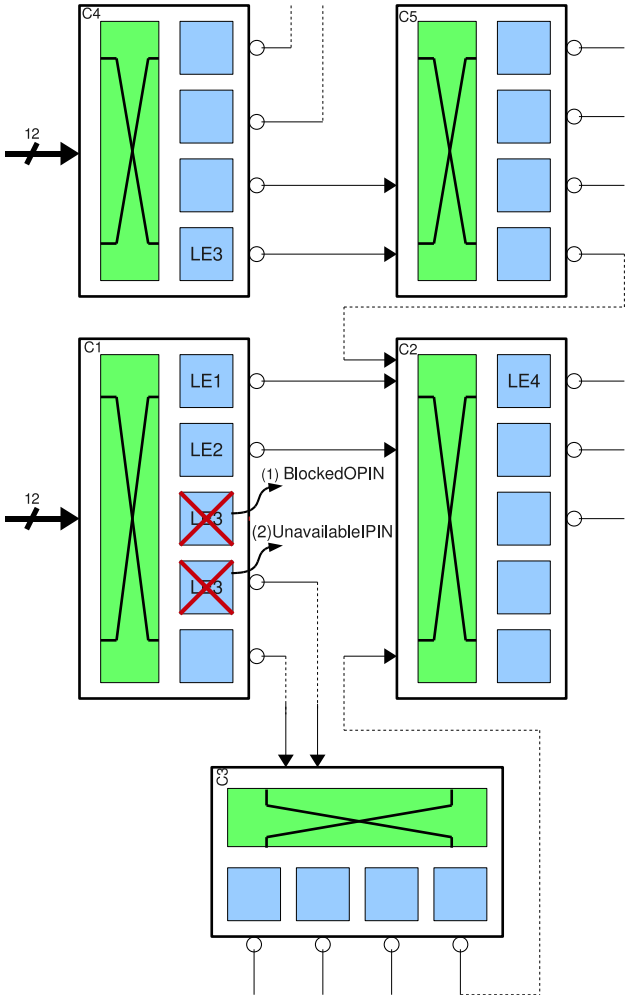


Figure 4.6: The trigger netlist of Figure 4.5 mapped to logic elements of overlay cells.

indirectly connect to LE4, resulting a routable trigger mapping.

The net routing cost guides the annealer to a routable solution. The routing cost of each net described by Equation 4.1.

$$\begin{aligned} NetRoutabilityCost = \alpha \times NumIndirectSinks + \\ \beta \times NumBlockedOPINs + \\ \beta \times NumUnavailableIPINs \end{aligned} \tag{4.1}$$

The overall routability cost is the sum of this quantity over all nets.

Indirect sinks are slightly penalized to encourage LEs to move to the same cluster or neighbouring clusters to make use of the intra-cluster connections or pre-routed connections of the overlay, respectively. Blocked output pins are heavily penalized since they will result in an immediate routing failure. Unavailable input pins are heavily penalized for the same reason. This term also encourages LEs that have common input signals to move to the same cluster to make better use of the input pins of each cluster. We have experimentally found that setting α to 5 and β to 20 gives good results.

We employ an additional optimization during the swap evaluation; if the swap will move an LE to a location that its output pin can not connect to the overlay routing network, which results in a blocked output pin, we alleviate unroutability by relocating the LE to another unoccupied location inside the same cluster. In this way, the output is connected to the overlay routing network resolving the blocked output pin issue. Consequently the swap still has a chance to be accepted toward an optimized implementation (case (1) and (2) in Figure 4.6).

4.4.2 Routing Augmentation

During routing, most connections are between neighbouring cells, and routing is trivial. For nets between non-neighbouring cells, we use a routing algorithm based on the routability-driven PathFinder [100] algorithm. The goal of this algorithm is to minimize the number of *routing hops* of each net, thereby maximizing the routability of the circuit.

4.4.3 Merging user Circuit with Trigger Circuit

The final step is to connect trigger signals in the user circuit to the inputs of the trigger circuitry and connect the output of the trigger circuitry to unused RAM blocks reclaimed as trace buffers. As described in Section 3.1.4, we use an incremental routing algorithm using unused LUTs as route-throughs to implement the required connections. Since these RAM blocks could be physically located in different sites across the chip, connecting the output of trigger circuitry to all of these RAM blocks can introduce a new long critical path. To minimize the effect of inserting trigger circuits on circuit critical path delay, the output of the trigger circuit is pipelined.

4.5 Methodology

In this chapter, we evaluate our techniques using VPR, which is part of the open-source academic Verilog-To-Routing project [96]. Using an open-source tool was necessary because demonstrating our approach requires low-level resource manipulation (e.g. to incrementally insert the overlay fabric into the bitstream after the user circuit has been compiled using left-over resources), whereas commercial tools do not provide this ability as device information is proprietary. Moreover,

Table 4.1: FPGA Architecture Used Based on Intel Stratix IV.

FPGA Architecture Parameter	Meaning	Value
N	Logic cluster size	10
K	Inputs per look-up table (fracturable)	6
I	Inputs per cluster	33
L	Channel segment length	4
F_{c-in}	Cluster input flexibility	0.15
F_{c-out}	Cluster output flexibility	0.10

we have chosen to demonstrate our techniques on VPR, so that we can investigate the impact that changes in the channel width and spare logic resources may have on our techniques.

Using VPR 6.0, we packed, placed, and routed a set of heterogeneous benchmark circuits onto an architecture based on Intel Stratix IV characterized in Table 4.1.

For each benchmark, an FPGA size was chosen to be the smallest square that fit the benchmark circuit. We use options to group only related logic elements into a cluster, and perform the placement and routing on the smallest FPGA array that can fit the benchmark circuit. Only related logic elements are packed into a cluster, mirroring behaviour of industrial tools [67].

Information about each circuit and the results of this mapping are shown in Table 4.2, including the number of LUTs and flip-flops of each circuit, the number of logic clusters and memory resources in the smallest FPGA array that fits each benchmark, and the resources occupied by the user circuit. Circuit *mkDelayWorker32B* uses most of the available memories of the FPGA. Other benchmarks use most of the available logic clusters of FPGA. The ability to construct an overlay depends heavily on the available resources after the user circuit is mapped onto

Table 4.2: Benchmark Summary.

Circuit	6-Input LUTs	FFs	FPGA Size	W_{min}	Logic Cluster Used	Logic Cluster All	RAM Used	RAM All	Free(%) LCs	Free(%) LEs
bgm	32384	5362	75x75	80	4111	4200	0	120	2	25
LU8PE	22634	6630	61x61	92	2667	2745	45	80	3	18
LU32PE	76211	20898	111x111	128	9105	9213	150	252	1	17
LU64PE	147556	39552	153x153	156	17591	17595	293	475	0	17
mcml	101858	53736	100x100	86	7350	7400	38	208	0	9
mkDW	5590	2491	42x42	76	916	1302	41	42	30	47
stereo2	29943	18416	89x89	92	5889	5963	0	154	1	57

FPGA. The column labeled *Free LCs* shows the percentage of the logic clusters which are left completely unused by the user circuit. The column labeled *Free LEs* shows the percentage of the unused logic elements inside logic clusters which are partially used by the user circuit.

For each circuit, unused RAM blocks in the FPGA are reclaimed as trace buffer memories. For *bgm* and *stereovision2*, all available memory blocks are reclaimed as trace buffers.

4.6 Experimental Results

4.6.1 Establishing the Overlay Architecture

First, we evaluate our algorithm for creating the overlay architecture. The ability to construct an overlay depends heavily on (a) the size of the overlay that is being created (as a fraction of the available resources after the user circuit is mapped) and (b) the availability of routing tracks to implement connections and avoid congestion. In this section, we vary both to understand both the limits of the technique and the influence these two factors have on the ability to create an overlay.

Figure 4.7 shows the compile time required to map (placement, primary and

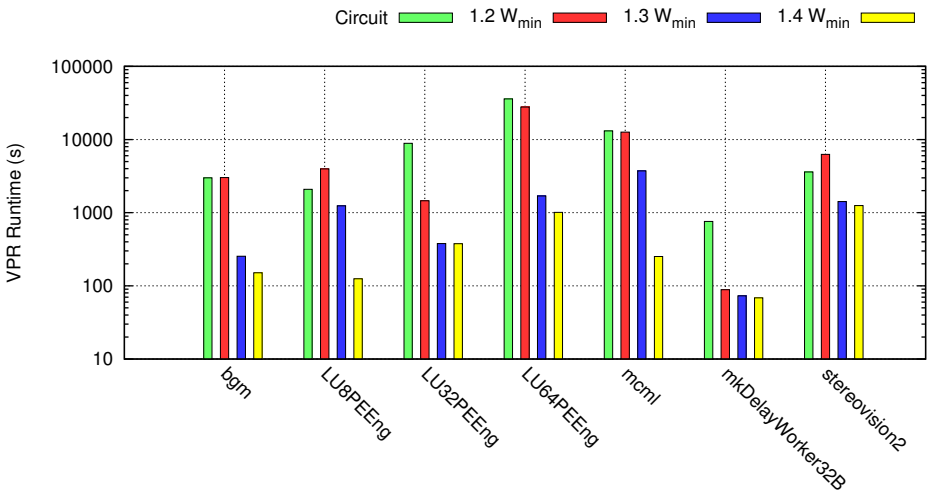


Figure 4.7: Compile-time: VPR runtime for establishing a maximally-sized overlay architecture as a function of channel width. Note that the y-axis uses a log scale

secondary sinks routing) a maximally-sized overlay architecture ($\text{Overlay}_{\text{max}}$) into FPGA spare resources for each benchmark circuit (note that the vertical axis uses a logarithmic scale). Results are shown for four values of the channel width: $W = 1.2W_{min}, 1.3W_{min}$, and $1.4W_{min}$, where W_{min} is the minimum channel width required to route each user circuit. Intuitively, the ratio W/W_{min} is a measure of the amount of *routing slack* available in the architecture. An architecture with $W/W_{min} = 1.2$ is one where there is very little slack, while an architecture with $W/W_{min} = 1.4$ has significant routing slack. Academic architecture studies tend to use a value of W/W_{min} of approximately 1.3 to reflect realistic routing demand as in [8, 96].

Figure 4.7 also shows the time to compile the original user circuit for comparison. The graph shows that increasing the channel width results in a substantial decrease in the overlay compile time due to decreased routing congestion. The overlay compile time overhead is less than 10% of the user circuit compile time for

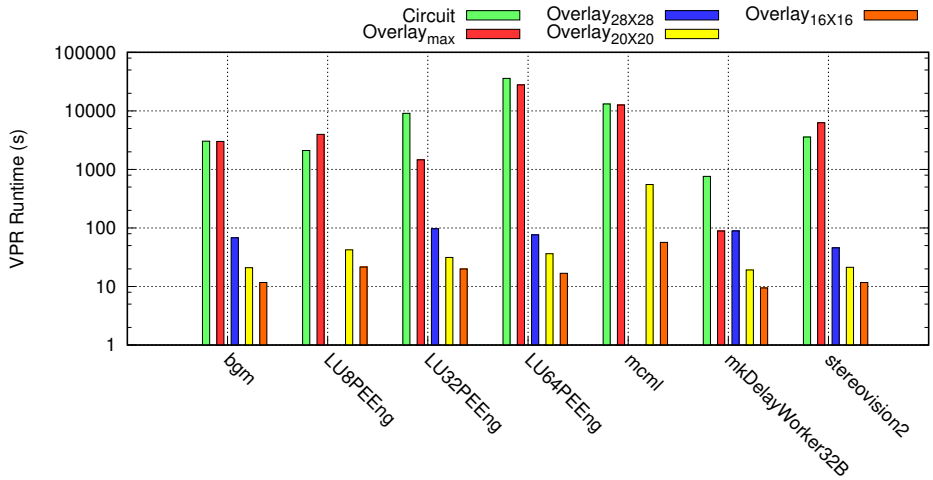


Figure 4.8: Compile-time: VPR runtime for establishing overlay architecture as a function of overlay size for 1.2W$_{min}$

small benchmarks such as *mkDelayWorker32B*. However, the compile time of mapping the overlay architecture is close to the circuit compile time for large benchmarks at a channel width of 1.2W$_{min}$. This is because these large benchmarks highly utilize LCs of FPGA resulting in routing congestion around LCs selected as overlay cells.

Figure 4.8 shows the same thing, but as a function of the overlay size, with W fixed at $1.2W_{min}$. Overlay$_{28X28}$ means an overlay fabric containing 784 overlay cells. Since the Overlay$_{max}$ of *mcml* and *LU8PEEng* has only 729 and 529 cells, respectively, it is not possible to create an overlay fabric with 784 cells (Overlay$_{28X28}$) for these benchmarks. As the results show, smaller overlays compile significantly faster, both because there are fewer cells to place and route, and also because the congestion is reduced. Although small overlays may be more common in practice (for small trigger circuits), the results show that our technique can map even very large fabrics, including those that fill the entire unused portion

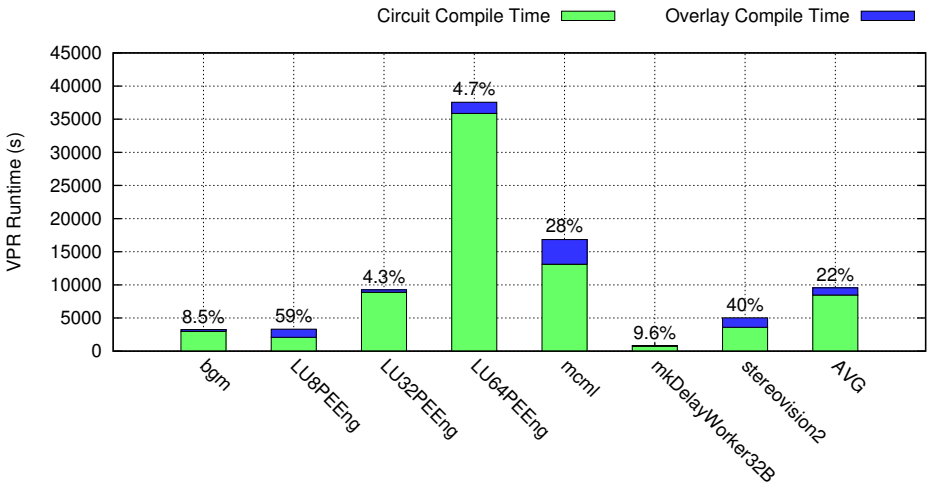


Figure 4.9: Compile-time: VPR runtime overhead for establishing overlay architecture for $1.3W_{min}$.

of the FPGA.

Figure 4.9 summarizes the compile time (the vertical axis is a linear scale) assuming a maximally-sized overlay ($Overlay_{max}$) on an architecture with a channel width of $1.3W_{min}$. On average the additional overhead of our overlay is 22% increase in circuit compile time; this reaches 59% for *LU8PEEng*. Although the designer must wait for place-and-route to implement the overlay architecture, the compilation time is only needed once to build the overlay and is amortized by the runtime savings made over multiple debug turns with the same circuit.

One of the unique aspects of the algorithm from Section 4.3 is that it is *adaptive*; that is, if congestion prevents some nets from being routed, they will be left out of the overlay, creating a slightly less flexible structure. To understand the extent to which nets are being dropped, Table 4.3 shows the average number of input pins per overlay cell that were successfully routed as a function of W and the overlay size for each benchmark circuit. Ideally, this number would be twelve

Table 4.3: Average Number of Input Pins per Overlay Cell.

	$\text{Overlay}_{\text{max}}$			1.2W_{min}		
Circuit	1.2W_{min}	1.3W_{min}	1.4W_{min}	$\text{Overlay}_{\text{28X28}}$	$\text{Overlay}_{\text{20X20}}$	$\text{Overlay}_{\text{16X16}}$
bgm	10.9	12	12	12	12	12
LU8PE	6.8	11.8	12	-	12	12
LU32PE	12	12	12	12	12	12
LU64PE	11.9	12	12	12	12	12
mcml	10.4	11.5	12	-	11.5	12
mkDW	12	12	12	12	12	12
stereo2	11.5	11.9	12	12	12	12

(four primary sinks and eight secondary sinks as in Section 4.2). As the results show, for most cases, the overlay is completely routed (the number of input pins is 12), however, for small values of W or very large overlay sizes, this number is smaller for some circuits. Note that $\text{Overlay}_{\text{max}}$ of *mcml* and *LU8PEEng* has less than 784 cells, $\text{Overlay}_{\text{28X28}}$. The average number of input pins per overlay cell for *LU8PEEng* is significantly lower than other benchmarks at channel width of $1.2W_{min}$; however, the number approaches 12 as the overlay size decreases, since a smaller overlay means there are more tracks available for routing, hence less congestion. For *mcml*, the number of input pins does not reach 12, even for an overlay size consisting of 400 overlay cells ($\text{Overlay}_{\text{20X20}}$); this is because even for the smaller fabric, there is still significant routing congestion at channel width of $1.2W$.

4.6.2 Trigger Mapping

In this subsection, we evaluate our algorithm for mapping a trigger circuit to the overlay architecture. In these experiments, we use a maximally-sized architecture on an FPGA with $W = 1.3W_{min}$. From the previous section, it is clear that for some

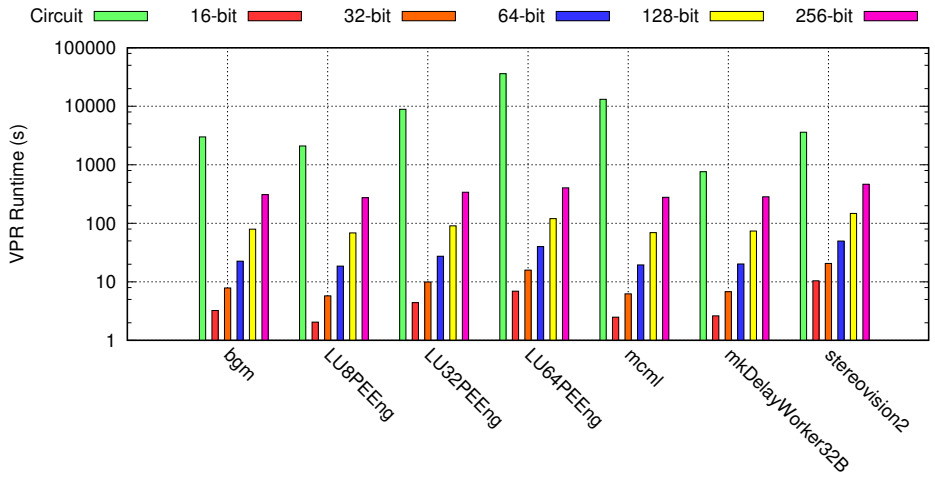


Figure 4.10: Debug-time: trigger mapping runtime assuming $\text{Overlay}_{\text{max}}$ for different size comparators.

of the benchmark circuits, this configuration leads to fewer than twelve inputs per cell (on average), however, the algorithm discussed in Section 4.4 constructs a mapping solution that does not use these missing connections. We use an n-bit comparator comparing two sets of signals or one set of signals and a constant as well as an n-bit range comparator as our trigger circuit; such a trigger circuit might be used when a user wishes to stop/start recording data when the value of an n-bit bus is equal to another n-bit bus or a user defined constant, where n is 16, 32, 64, 128, 256. The trigger signals are randomly selected from the registered signals of the benchmark circuit (as in [70]); we use multiple runs for each trigger circuit and choose a different set of trigger signals for each run, averaging the results over multiple runs.

Figure 4.10 shows the runtime of our trigger mapping algorithm (this includes connecting the trigger logic to the trace buffers and user circuit). As the results show, for all benchmarks except for *mkDelayWorker32B*, inserting a new trigger

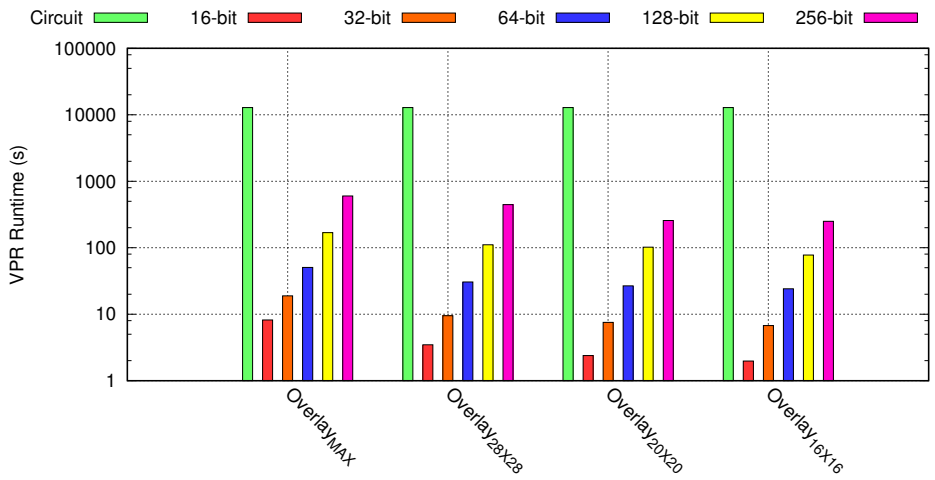


Figure 4.11: Average trigger mapping runtime as a function of overlay size.

circuitry is an order of magnitude faster than a recompilation. Benchmark *mkDelayWorker32B* has mapping times similar to the compile time of the original user circuit for large triggers (256-bit comparator), primarily because this benchmark is relatively small, and the original compile time is short.

Figure 4.11 shows the average runtime of our trigger mapping algorithm as a function of overlay size. As the results show, it is faster to map a trigger circuitry into a smaller overlay.

Table 4.4 shows the critical path delay of each circuit before and after inserting different size comparators into the design for different overlay sizes. As the table shows, inserting a large trigger circuit can introduce a new long critical path to the circuits with a short critical path. For example, the critical path delay of *mkDelayWorker32B* which is only 6.5ns increases by 2.13ns when inserting a 256-bit comparator. In the worst case, the delay of *stereovision2* increases by 6.13ns. However, it should be noted that this increase in the circuit critical path delay is temporary during debugging; the operating frequency of the circuit can be set as

Table 4.4: Effect of Trigger Insertion on Critical Path Delay (ns).

Overlay Size	$Overlay_{max}$			$Overlay_{28X28}$		$Overlay_{20X20}$		$Overlay_{16X16}$	
Circuit	Original	16 to 128	256	128	256	128	256	128	256
bgm	19.7	19.7	19.7	19.7	19.7	19.7	19.7	19.7	19.7
LU8PE	89.6	89.6	89.6	-	-	89.6	89.6	89.6	89.6
LU32PE	91.9	91.9	91.9	91.9	91.9	91.9	91.9	91.9	91.9
LU64PE	89.2	89.2	89.2	89.2	89.2	89.2	89.2	89.2	89.2
mcml	66.5	66.5	66.5	-	-	66.5	66.5	66.5	66.5
mkDW	6.46	6.46	8.5	6.46	8.5	6.46	8.5	6.46	7.1
stereo2	11.9	11.9	18.0	11.9	12.7	11.9	12.4	11.9	13.1

normal when the debug instrumentation is not required.

4.6.3 Comparison to Incremental Triggering without using Overlays

In Chapter 3, an incremental routing technique (without an overlay to guide the routing) was used to route the required nets of the trigger netlist. That work was unable to route several benchmarks due to congestion during the incremental routing phase. In contrast, using the approach presented in this chapter, the pre-synthesized overlay fabric enables us to use a simulated-annealing based placement algorithm in order to increase the chance of finding a routable placement solution without requiring a long runtime.

Using overlay for trigger insertion, we were able to successfully insert the same trigger circuitry (i.e. bitwise AND) from Chapter 3 for *stereovision0* and *stereovision2* assuming a channel width $1.2W_{min}$. For circuit *mkPktMerge*, we created an $Overlay_{21X21}$ and were able to successfully insert a trigger function with 256, and 512 inputs, again assuming a channel width of $1.2W_{min}$. It should be noted that this benchmark has only 515 signals, while the trigger circuitry with 512 inputs has 655 nets; this shows the flexibility of the overlay architecture. The overlay size

of *raygentop* (Overlay_{10X10}), and *or1200* (Overlay_{14X14}) is too small to support a trigger circuitry with more than 512 and 1024 inputs, respectively. Another difference between this work and the previous work is that the entire circuit needs to be loaded into the memory of a CAD tool to construct an incremental placement and routing solution for the trigger circuitry at debug time; while in this work, the overlay architecture is only required to map a desired trigger circuitry at debug time.

4.7 Summary

This chapter demonstrated a non-intrusive framework based on overlays that can be used for instrumentation without requiring circuit recompilation during FPGA debug. At compile time, the overlay architecture is incrementally established on top of the user circuit using leftover resources after circuit compilation. This mapping is done using a best-effort adaptive CAD technique that preserves the underlying circuit mapping. At debug time, the desired trigger circuitry can be rapidly mapped to the overlay architecture. This is done using a routing-aware placement algorithm to increase the routability of the placement solution.

The key novelties of this work are in: (1) a non-intrusive triggering framework that separates the user circuit from the debug instrumentation eliminating the need for recompilation during debug iterations, (2) an adaptive, and flexible overlay architecture that provides enough flexibility for arbitrary combinational trigger circuits, (3) efficient methodology for overlay construction using incremental CAD algorithms that exploit the spare logic and routing resources unused by the user circuit resulting in zero area overhead, and (4) CAD algorithms to rapidly reconfigure the overlay for a new trigger circuitry at debug iterations.

We have shown that the overlay architecture provides enough flexibility to implement combinational triggering scenarios an order of magnitude faster than a full recompilation accelerating debug productivity. Our experiments have shown that our triggering framework has a negligible impact on the critical path delay.

To extend our solution of non-intrusive instrumentation framework used for rapid implementation of combinational trigger circuits, the contributions in Chapter 5 are concerned with addressing the more challenging problem of implementing sequential (state-based) trigger circuits.

Chapter 5

Customized FPGA Overlay and Mapping Tools for Rapid Triggering Capabilities

Chapter 4 detailed our non-intrusive instrumentation framework for rapid trigger insertion during debugging. As part of our framework, we introduced an overlay architecture (a 2D torus) and mapping tool (simulated-annealing based placement algorithm) suitable for implementation of combinational functions; we did not consider implementing state-based trigger circuits. Yet, supporting such complex trigger circuits are essential to ensure that the collected data is useful for the designer. An example of a sequential trigger might be a state machine that determines *nth* occurrence of a bus value. Hence, in this chapter we introduce a specialized overlay architecture and tailored mapping algorithms that provide the ability of efficient implementation of complex trigger circuits, suitable for an in-

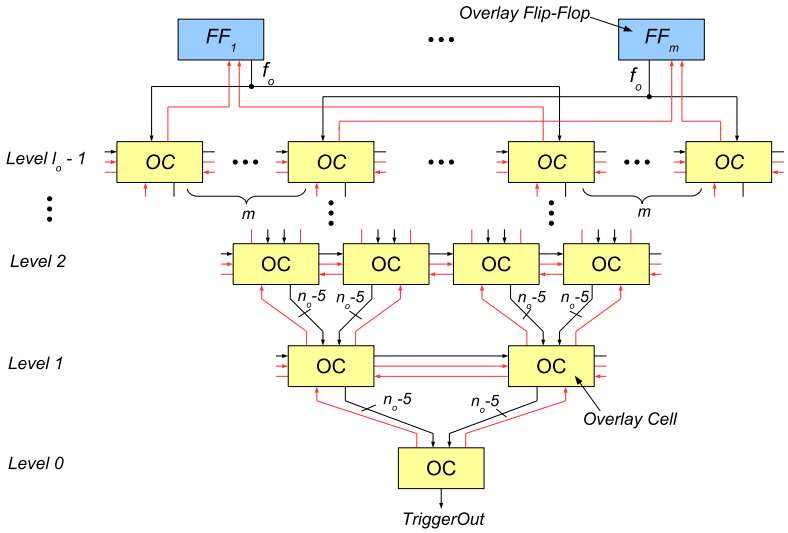
system debug ecosystem. Section 5.1 describes our optimized overlay architecture and the method by which the overlay architecture is implemented. Since we are limited to only resources not used by the user circuit, it is necessary to make a reasonable trade-off between area and flexibility. To achieve this, we customize the overlay architecture to provide adequate flexibility for implementing complex trigger circuits, including arbitrary combinational and state-based trigger circuits. Section 5.2 describes our specialized algorithms that take advantage of the characteristics of the customized overlay architecture to rapidly map trigger circuits. Section 5.3 details the experimental methodology that was used to evaluate our techniques. Following this, in Section 5.4, we discuss results from these experiments and compare our technique to Intel's SignalTap II tool in terms of runtime, area overhead, and circuit delay. Finally Section 5.5 summarizes this chapter. An early version of this contribution was published in [43], and an extended version was submitted to ACM Transactions on Design Automation of Electronic Systems (TODAES) with minor revisions.

5.1 Overlay Architecture

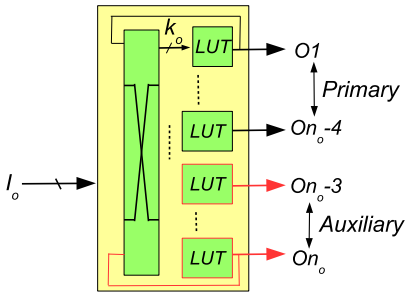
In this section, we describe a parameterized overlay architecture family that is optimized to implement combinational and sequential trigger functions. We first describe a logical view of the architecture, and then show how it is implemented on an FPGA in an efficient manner.

5.1.1 Parameterized Overlay Architecture Family

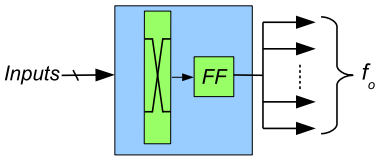
Our overlay is specialized to implement trigger circuits. To implement the logic part of the trigger circuit, our overlay contains a number of *cells*, each contain-



(a) The overall structure of the multi-level overlay architecture.



(b) Overlay Cell (OC) details



(c) Overlay flip-flop details

Figure 5.1: (a) An overview of the parameterized overlay architecture that is comprised of cells connected via primary (shown in black) and auxiliary (shown in red) connections, and a set of flip-flops connected to cells in level $l_o - 1$ (b) *O*verlay cell design with k_o-input LUTs, local routing, and outputs to other cells. (c) Overlay flip-flop design with a flip-flop, local routing, and fanout to other cells.

ing n_o look-up tables (LUTs). Each look-up table has k_o inputs. As discussed in Chapter 4, building an overlay around LUTs allows for an efficient mapping of the overlay to the underlying FPGA.

Figure 5.1a shows an overview of the overlay architecture that is comprised of

cells and flip-flops. Within an Overlay Cell (OC), n_o look-up tables are arranged as shown in Figure 5.1b. As will be discussed below, four of the LUTs are designated as *auxiliary* LUTs, and the remaining $n_o - 4$ are designated as *primary* LUTs. Each LUT drives one output of the cell. The inputs to each LUT are connected to I_o inputs and the n_o feedback paths through a fully-connected crossbar. As we will show below, our interconnect pattern limits I_o to be $2n_o - 6$ (this is less than is common for standard FPGA architectures, but we will show it is sufficient for the triggers we implement).

As shown in Figure 5.1a, cells are connected in a triangular reduction-network pattern, motivated by work in [71, 89, 124]. The use of the triangular pattern was motivated by work in [71] in which the authors profiled a large number of circuits with this property, measuring their 'shape', and properties such as average fanin, fanout, and convergence. In that paper, the authors concluded that such circuits often have triangular shapes, which motivated our architecture. In Figure 5.1a, the levels are labeled $0..l_o - 1$ starting from the output; level i contains 2^i cells. The overlay has a single output from level 0 which is used to control the trace buffers. To support sequential triggers, the overlay also contains a bank of m flip-flops shown at the top of Figure 5.1a.

The interconnect between cells is also shown in Figure 5.1a. We distinguish between two categories of interconnects: Each cell output is labeled as *primary* or *auxiliary*. There are $n_o - 4$ primary outputs; all but one are connected to a cell in the previous level. The other primary output is connected to a neighbouring cell in the same level. Two of the four auxiliary outputs are connected to cells in the next level; the other two are connected to neighbouring cells in the same level. Non-neighbouring cells are connected by making a series of *route through*, i.e. by

Table 5.1: Architectural Parameters Describing Logical Overlay.

Overlay Architecture Parameter	Meaning
n_o	Look-up tables per cell
k_o	Inputs per look-up table
l_o	Number of levels of cells
m	Number of flip-flops
f_o	Flip-Flop feedback fanout

passing through intermediate cells using unused inputs, outputs, and LUTs as route through.

The cells in level $l_o - 1$ are connected to the m flip-flops. Each overlay flip-flop is accompanied by a multiplexer as shown in Figure 5.1c. Each input to the multiplexer is driven by one of the auxiliary outputs of a cell in level $l_o - 1$. Since there are 2^{l_o-1} cells in level $l_o - 1$ and m flip-flops, each flip-flop multiplexer has $\frac{2^{l_o-1}}{m}$ inputs. The output of each flip-flop drives one of the $2n_o - 10$ primary inputs to a cell in level $l_o - 1$. Rather than connect each flip-flop output to each of the 2^{l_o-1} cells, we have found that we can obtain sufficient flexibility by connecting each flip-flop output to a smaller number of cells which we denote f_o. In our experiments in Section 5.4, $f_o = 32$. Table 5.1 presents a summary of architectural parameters describing overlay fabric. A family of overlay fabrics can be described using these parameters.

The inputs to the overlay fabric are the trigger signals from the user circuit. As described in Section 5.2.4, any of the primary inputs to each cell can be used as an input pin of the overlay fabric.

5.1.2 Overlay Construction

The overlay architecture is constructed on a per-circuit basis. The overlay fabric is added to the user circuit at compile time, using only those logic and routing resource that were not used by the user circuit. Rather than compiling an RTL description of the overlay fabric using normal CAD tools, we map the overlay directly to the underlying FPGA primitives (e.g. LUTs, and routing segments). This leads to a far more efficient overlay implementation. In this chapter, we assume an FPGA architecture similar to Intel Stratix IV device described in Section 5.3. We modified the overlay mapping approach described in Section 4.3 to implement our multi-level overlay architecture into FPGA available resources. Our algorithm consists of cell selection, overlay placement, and *best-effort* routing. An overview of the algorithm is provided below.

Selection and Adaptive Allocation. Given the mapping of the user circuit, the algorithm first collects all logic clusters (LCs) inside the FPGA which (1) contain at least n_o logic elements (LEs) unused by the user circuit, and (2) have at least $2n_o - 6$ unused inputs. Each such LC has sufficient unused resources to implement a single cell of the combinational part of the overlay architecture. We also collect LCs which contain (1) at least one unused LE, and (2) at least $\frac{2^{l_o-1}}{m}$ unused inputs; each such LC can be used to implement one of the flip-flops in the overlay fabric.

Based on the number of LCs selected, we then choose the number of levels of the overlay, l_o. In this way, the size of the overlay *adapts* to the size of the user circuit; if a circuit uses most of the available FPGA, few such LCs will be found, and the overlay will be small, limiting the complexity of trigger functions that can be supported. If there are many unused resources, l_c will be large, leading to an

overlay that can implement more complex trigger functionality.

Overlay Placement. We then construct a logical view of the overlay architecture, and assign potential sites on the FPGA for each overlay cell. We employ the simulated annealing based placement algorithm presented in Chapter 4 to position the cells in these sites.

Overlay Routing. After overlay placement, we attempt to create all connections between the cells. We use the *best-effort* routing heuristic presented in Chapter 4. This heuristic iteratively discards illegal connections (beginning with auxiliary connections) until a legal routing solution is achieved. This approach ensures that the primary connections have higher priority for using routing resources than the auxiliary connections. In this way, our approach can be used for user circuits that already stress the FPGA routing.

5.2 CAD Algorithm for Overlay Personalization

At debug time, a mapping algorithm is required to implement a trigger netlist onto the overlay fabric. At this stage, there are three challenges: (1) the mapping algorithm should be fast to provide rapid debug iterations; (2) the pre-synthesized overlay has limited flexibility compared to a standard FPGA due to its limited routing topology and missing connections as described in Section 5.1.2. If we perform placement and routing as separate phases, a poor placement can easily lead to an un-routable solution, and (3) the overlay architecture is highly constrained since it is customized for trigger-type circuits.

Our overall trigger mapping flow is presented in Figure 5.2. It has two major phases: Pre-processing and Placement/Routing. In this section, we describe the details of each phase; for clarity, we first describe the mapping algorithm for com-

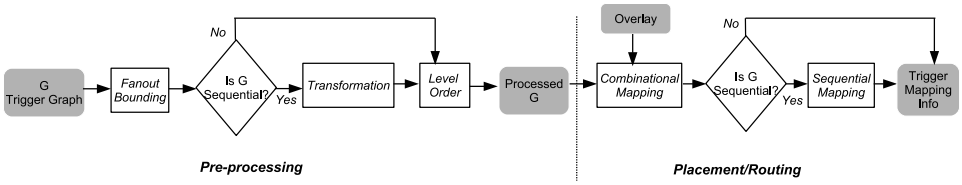


Figure 5.2: Overlay reconfiguration flow that takes a graph representation of a trigger netlist and outputs a placed and routed netlist onto the overlay fabric.

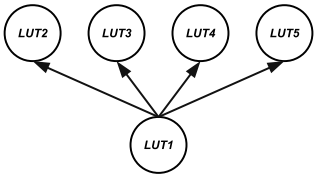
binational triggers and then generalize the algorithm for sequential trigger circuits.

5.2.1 Trigger Circuit Modeling

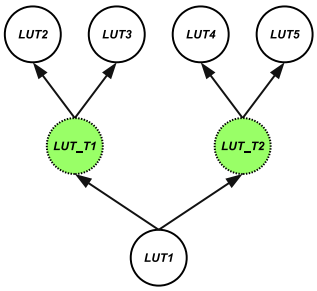
We make the same assumptions as in Chapter 4 that the trigger circuit has been previously mapped to lookup tables and flip-flops using a technology-mapping algorithm such as [102]. A trigger circuit netlist can be represented as a directed graph *G(V, E)*, where each node in *V* represents either a LUT or a flip-flop. Each edge in *E* is a connection between LUTs and/or flip-flops in the trigger circuit; we refer to these nets as *trigger nets*. Each trigger net has one source node, but may have multiple sink nodes. The graph may have multiple input edges, each representing an input signal to the trigger circuit, and has a single output edge, representing the trigger output.

5.2.2 Mapping Combinational Triggers

Our overall mapping flow is shown in Figure 5.2. In this section, we describe this flow assuming the trigger circuit is combinational; in Section 5.2.3 we generalize this for a sequential trigger circuit.



(a) A multi-sink net before fanout bounding.



(b) Two-sink nets after fanout bounding.

Figure 5.3: Fanout bounding takes a multi-sink trigger net (a) and decomposes it to multiple two-sink nets (b).

Pre-processing

Pre-processing modifies G to suit the overlay architecture by taking the following steps:

1. Fanout bounding. Since each output pin of each overlay cell is only connected to one cell, we bound the fanout of the trigger nets to be as small as possible to better match these single output connections of the overlay. Hence, this step decomposes any multi-sink nets into multiple two-sink nets. This is performed by replacing a multi-sink net with a binary tree with the source node as the root, the sink nodes as the leaves, and replicator LUTs as intermediate nodes as illustrated in Figure 5.3. Clearly, the result is a functionally equivalent circuit. Fanout bounding techniques such as [60, 114] can be used to minimize the number of intermediate nodes and depth; we do not consider this optimization in this chapter.

2. Level Order. In this step, a *Level-Order Traversal* algorithm is used to traverse the trigger netlist graph. The output is a list of LUTs sorted based on

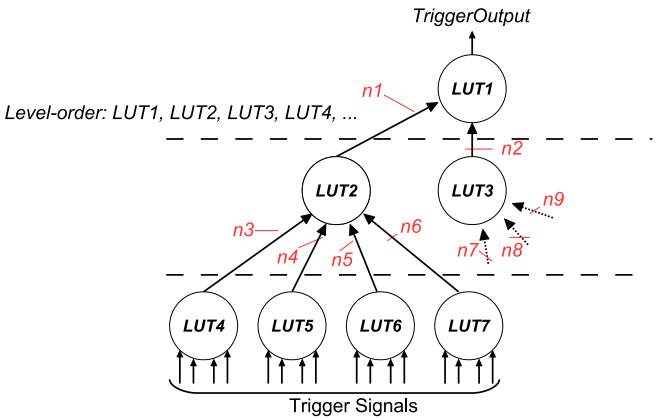


Figure 5.4: A simple single-sink combinational trigger netlist.

level-order.

Placement and Routing

Placement and routing finds a mapping of each LUT in the trigger circuit (which we refer to as a *trigger LUT* to a LUT in the overlay fabric (which we refer to as a *LUT slot*). We process the graph in Level Order, meaning each trigger LUT is placed after its sink LUTs to ensure there is a connection between the source and the sink LUTs. In this section, we first describe the mapping algorithm assuming the trigger circuit contains only single-sink nets, using the example in Figure 5.4. We then generalize the algorithm for circuits which contain at least one net with two sinks. Due to the pre-processing step in Section 5.2.2 , we are assured that no net has more than two sinks.

Mapping trigger LUTs using overlay primary resources. Figure 5.5 illustrates the details of the connections between overlay cells described in Section 5.1.1. Associated with each output pin for each cell is a 4-input LUT driving the output pin; as described in Section 5.1, these LUTs are categorized as primary and auxiliary

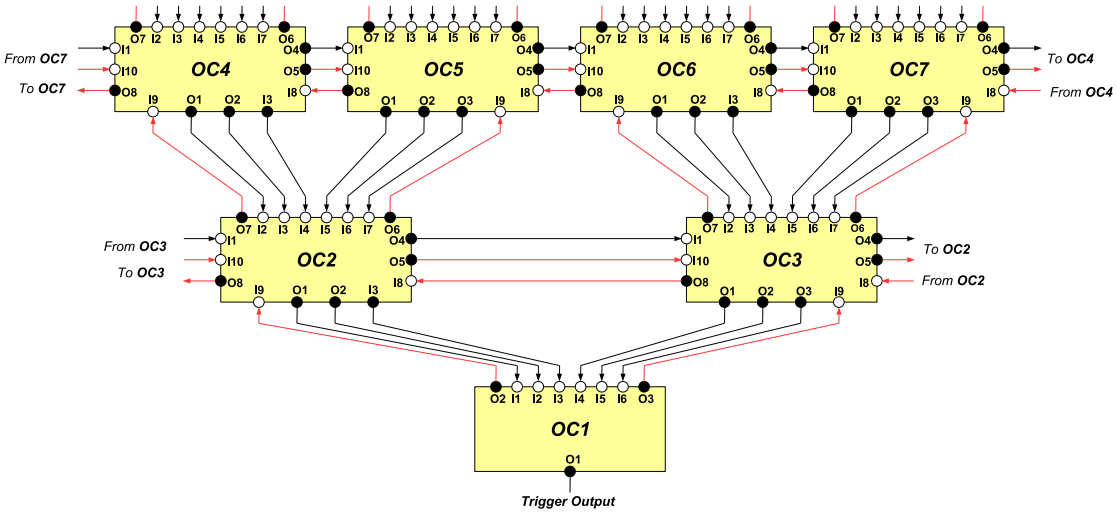


Figure 5.5: Detailed view of connections between overlay cells assuming n_o = 8, k_o = 4 and l_o = 3. Each cell (except OC1) has ten input pins. Seven (I1-I7) are driven by primary (shown in black) output pins. Three (I8-I10) are driven by auxiliary (shown in red) output pins. Each output pin is driven by a 4-input LUTs. LUTs and routing resources inside each cell are not shown for simplicity.

LUTs. Each primary 4-input LUT can be used to implement a 4-input function or it can be used as a route-through to pass signal to neighbouring cells. Auxiliary LUTs are only used as route-through.

The algorithm starts by assigning the first element in the Level Order list to the LUT-slot that drives the output pin of the overlay fabric. In our example, *LUT1* in Figure 5.4 is assigned to LUT-slot inside *OC1* that drives *O1* in Figure 5.5. Once *LUT1* is placed into *OC1*, the overlay output pin is assigned to *LUT1*'s output net, and the primary input pins of the overlay cell are assigned to the input signals of *LUT1*. Since the primary inputs of OC1 are driven by two cells (i.e. *OC2* and *OC3*), input pins are distributed between OC2 and OC3 evenly. Hence, *n1* is assigned to *I1*, and *n2* to *I4*.

The rest of the trigger LUTs in the Level Order list are considered sequentially; each trigger LUT is mapped to a LUT-slot using Algorithm 2. Note that since we are assuming only single-sink nets in our example, the trigger LUT has only one predecessor. This algorithm is given the input pin of the trigger LUT's predecessor; because the trigger LUTs are processed in level order, we are assured the predecessor has already been assigned a LUT-slot. The algorithm determines the LUT-slot that drives this input pin and determines whether this candidate LUT-slot is a legal location for the trigger LUT being placed (lines 3-8 in Algorithm 2). In our example, when placing *LUT2*, the LUT-slot that drives *O1* of *OC2* is considered as a candidate LUT-slot since *n1* was assigned to *I1* of *OC1*.

To determine whether a candidate LUT-slot is a legal location, the *Validate* routine checks for two conditions: (1) the overlay cell that contains the candidate output pin must have enough unassigned input pins to accommodate the input signals of the trigger LUT, and (2) the overlay cell has at least as many unassigned input pins as its unassigned output pins. This latter condition is to ensure that the cell can later be used as a route-through cell; as we will discuss, this is crucial to prevent blocked routing situations when placing future trigger LUTs. If the validation passes, the algorithm returns the specific LUT-slot and a path from the output pin of this LUT-slot to the assigned input pin of the predecessor (lines 8-13 in Algorithm 2). In our example, cell *OC2* is free and the validation step passes; the algorithm places *LUT2* onto the LUT-slot that drives output pin *O1* inside *OC2*. As before, primary inputs pins are distributed among the driving cells evenly; in our example, the input pins are distributed among *OC3*, *OC4*, and *OC5*. Therefore, a valid input assignment is *n3* to *I1*, *n4* to *I2*, *n5* to *I3* and *n6* to *I5*.

If the validation for a candidate LUT-slot fails, the algorithm considers LUT-

Algorithm 2: Mapping a trigger LUT onto a LUT-slot inside overlay cells

```
Input:  overlay /* Overlay Fabric */
        trigger_LUT /* To be mapped trigger LUT */
        assigned_input_pin_of_predecessor1 /* Input pin that was
assigned to trigger_LUT's first predecessor */
        assigned_input_pin_of_predecessor2 /* Input pin that was
assigned to trigger_LUT's second predecessor */
Output: found_LUT_slot
        routing_path1 /* Routing path from found_LUT_slot to
assigned_input_pin_of_predecessor1 */
        routing_path2 /* Routing path from found_LUT_slot to
assigned_input_pin_of_predecessor2 */
1  /* a set that keeps the candidate LUT-slots in the current level */
2  current_level_candidate_LUT_slots ← ∅
3  driver_LUT_slot =
     GetDriverLUT(overlay, assigned_input_pin_of_predecessor1)
4  current_level_candidate_LUT_slots.insert(driver_LUT_slot)
5  while (current_level_candidate_LUT_slots ≠ ∅) do
6      next_level_candidate_LUT_slots ← ∅
7      foreach (LUT_slot ∈ current_level_candidate_LUT_slots) do
8          if Validate(overlay, trigger_LUT, LUT_slot) == true then
9              found_LUT_slot = LUT_slot
10             LUT_slot_output_pin = GetOutputPin(overlay, LUT_slot)
11             routing_path1 =
                 GetPath(LUT_slot_output_pin, assigned_input_pin_of_predecessor1)
12             if assigned_input_pin_of_predecessor2 == ∅ then
13                 return found_LUT_slot, routing_path1
14             routing_path2 =
                 SignalRoute(overlay, LUT_slot_output_pin, assigned_input_pin_of_predecessor2)
15             if routing_path2 ≠ ∅ then
16                 return found_LUT_slot, routing_path1, routing_path2
17         /* use the LUT-slot as a route-through to find other candidates */
18         foreach (free_primary_input_pin ∈ cell containing LUT_slot) do
19             driver_LUT_slot = GetDriverLUT(overlay, free_input_pin)
20             next_level_candidate_LUT_slots.insert(driver_LUT)
21     current_level_candidate_LUT_slots = next_level_candidate_LUT_slots
22 return ∅ /* return empty indicating that no LUT-slot was found */
```

Algorithm 3: SignalRoute

```
Input:  overlay /* Overlay Fabric */
        source_output_pin /* Source output pin */
        sink_input_pin /* Sink input pin */
Output: routing_path /* Routing path from source to sink */
1  /* a set that keeps the candidate output pins in the current level */
2  current_level_output_pins ← ∅
3  /* returns the output pin that drives sink_input_pin */
4  driver_pin = GetDriverPin(overlay, sink_input_pin)
5  current_level_output_pins.insert(driver_pin)
6  while (current_level_output_pins ≠ ∅) do
7      next_level_output_pins ← ∅
8      foreach (output_pin ∈ current_level_output_pins) do
9          if source_output_pin == output_pin then
10             routing_path = GetPath(source_output_pin, sink_input_pin)
11             return routing_path
12         /* go to the neighbours*/
13         foreach (free_input_pin ∈ the cell containing output_pin) do
14             driver_pin = GetDriverPin(overlay, free_input_pin)
15             next_level_output_pins.insert(driver_pin)
16     current_level_output_pins = next_level_output_pins
17 return ∅ /* return empty indicating that no path exists */
```

slots in cells that fan-in to the cell containing the candidate LUT-slot. The candidate LUT-slot is then used as a route-through (lines 18-20 in Algorithm 2). As mentioned earlier, condition (2) of the validation ensures the algorithm is able to use the candidate LUT-slot in this way. The algorithm performs a validation for each new candidate LUT-slot until a legal location is found. If the algorithm cannot find a legal location (i.e. the algorithm reaches the last level), the mapping fails (line 22 in Algorithm 2).

Mapping two-sink trigger nets. We do not have to deal with trigger nets with more than two sinks because of the fanout bounding performed in pre-processing

phase as described in Section 5.2.2. For trigger circuits that contain at least one net with two sinks, additional processing is required. When placing a trigger LUT with two sinks, both sink LUTs have already been placed due to the fact that trigger LUTs are placed in a level order. For such a trigger LUT, we first find a candidate LUT-slot based on its first sink using primary resources as explained above. We then determine whether there is a path between the trigger LUT and its second sink (line 14 of Algorithm 2). To determine whether there is such path, we use Algorithm 3 which is based on a breadth-first search. Both Primary and Auxiliary resources are used to enhance the routing flexibility when finding such a path. If a path is found (Lines 15-16 in Algorithm 2), the candidate LUT-slot is accepted, and the path is returned to the calling routine so that it can map the routing resources that mark up the path as used. If there is no path between the candidate LUT-slot and the second sink, the candidate LUT-slot is rejected and Algorithm 2 considers other LUT-slots as described earlier.

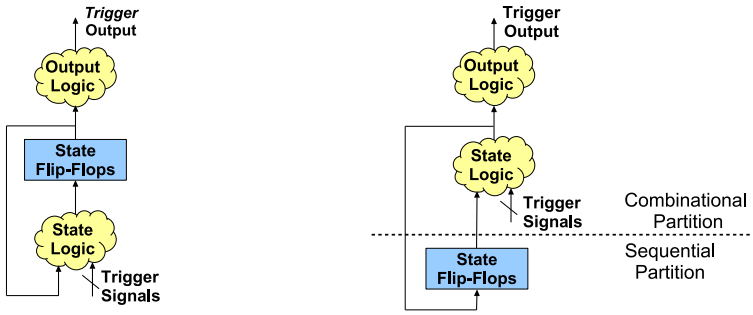
5.2.3 Mapping Sequential Triggers

In this section, we describe how sequential trigger circuits are mapped.

Pre-processing

For a sequential circuit, the pre-processing phase is modified in three ways:

1. Fanout Bounding. As described in Section 5.2.2, fanout bounding on G is performed by replacing each multi-sink net with a *k-way* tree. As in Section 5.2.2, for each combinational node, we use $k = 2$. For each sequential node (a flip-flop), we use $k = f_o$. Recall from Section 5.1.1 that f_o was defined as the fanout of each overlay flip-flop. This ensures that the solution



(a) Original sequential trigger.

(b) Transformed sequential trigger.

Figure 5.6: Transformation takes a sequential trigger (a) and restructures it to sequential trigger (b) where trigger netlist can be cut into two partitions: combinational and sequential.

matches the topological constraints of overlay flip-flops.

2. Transformation. As shown in Figure 5.2, if G is sequential, a transformation is performed to restructure a sequential trigger state machine. As shown in Figure 5.6, the state machine is restructured so that the trigger output logic is fed directly from the state logic. This allows us to separate the combinational and sequential parts of the trigger to better match the overlay architecture. Although this transformation means that trigger output is produced one cycle earlier. This can be tolerated by pipelining the connection between the trigger overlay and the trace buffer, as we described in Section 5.2.4.

3. Level Order. The Level Order algorithm performs a level-order traversal of the combinational part of G.

Algorithm 4: Mapping a trigger flip-flop onto a flipflop-slot inside overlay fabric

Input: *overlay* /* Overlay Fabric */
all_assigned_input_pins_of_sinks /* Set of input pins that were assigned to trigger flip-flop's sinks */
output_pin_of_driver /* Output pin that was assigned to trigger flip-flop's driver LUT */

Output:
found_flipflop_slot /*Flipflop_slot of the overlay that trigger flip-flop can be assigned to it */
all_routing_paths /* Routing paths between the found_flipflop_slot and the combinational part */

```
/* set of unassigned flip-flop slots inside overlay fabric */
unassigned_flipflop_slots = GetFlipFlops(overlay)
foreach (candidate_flipflop_slot ∈ unassigned_flipflop_slots) do
    flipflop_slot_output_pin =
      GetOutputPin(overlay, candidate_flipflop_slot)
    all_routing_paths ← ∅
    success = true
    foreach (assigned_input_pin ∈all_assigned_input_pins_of_sinks) do
        routing_path =
          SignalRoute(overlay, flipflop_slot_output_pin, assigned_input_pin)
        if routing_path == ∅ then
            success = false
            break
        all_routing_path.insert(routing_path)
    if success == true then
        flipflop_slot_input_pins =
          GetInputPins(overlay, candidate_flipflop_slot)
        foreach (flipflop_slot_input_pin ∈flipflop_slot_input_pins) do
            routing_path =
              SignalRoute(overlay,output_pin_of_driver, flipflop_slot_input_pin)
            if routing_path ≠ ∅ then
                all_routing_paths.insert(routing_path)
                return found_flipflop_slot, all_routing_path
return ∅ /* return empty indicating that no flipflop-slot was found */
```

Placement and Routing

Placement and routing of a sequential trigger circuit is performed in two steps. First, the combinational portion is mapped as in Section 5.2.2. Then, the flip-flops in the trigger circuit (which we refer to as trigger flip-flops) are placed into the overlay and connect to the combinational portion of the circuit. Each trigger flip-flop is considered sequentially, and mapped to a flip-flop in the overlay fabric (which we refer to as a *flip-flop slot*) using Algorithm 4. This algorithm considers candidate slots sequentially (line 3 of Algorithm 4). Each candidate slot is validated using two invocations of breadth-first routing in Algorithm 3. The first invocation determines whether it is possible to find a path between the flip-flop slot output and each sink LUT (lines 7-12 in Algorithm 4). Since the output pin of each overlay flip-flop is connected to f_o input pins of cells in level $l_o - 1$ as described in Section 5.1.1, finding a path from a sink LUT to any of those cells is sufficient. The second invocation of the breadth-first routing algorithm determines whether it is possible to find a path between the output pin of the driving LUT and the input pins of the candidate overlay flip-flop slot (lines 15-19 in Algorithm 4). Since the inputs of each overlay flip-flop are driven by a fraction of cells in level $l_o - 1$, finding a path to any of those cells is sufficient. If both invocations are successful, the candidate is selected; if not, another candidate flip-flop slot is considered (the algorithm reiterates from line 3 in Algorithm 4). If no legal flip-flop slot is found, the algorithm fails.

5.2.4 Connecting Trigger Circuit to the User Circuit

After mapping the trigger logic to the overlay fabric, we connect the trigger inputs from the user circuit to the trigger logic. Rather than having dedicated trigger fabric

input pins, every connection between cells in the overlay fabric can be replaced with a connection to the user circuit. For example, in Figure 5.5, if pin I2 of OC2 is mapped to a trigger input signal, the connection between OC4 and pin I2 of OC2 can be replaced by a connection between the user circuit and pin I2 of OC2. This re-route can be done using an incremental routing algorithm which reroutes specific signals using free resources while leaving the rest of the design unchanged.

The last step to enable triggering is to connect the single trigger output to all trace buffers across the chip, which can introduce a long critical path. Hence, we assume the trigger output is registered to minimize impact on the user circuit's critical-path delay.

5.3 Experimental Methodology

In Section 5.4, we experimentally map synthetic trigger circuits to an overlay architecture, implemented on top of a set of user circuits. In this section, we describe how we obtain and map the user circuits, the overlay architecture assumptions, and the synthetic trigger circuits.

5.3.1 Underlying User Circuits

We evaluate our techniques using VPR, which is part of the academic Verilog-To-Routing project [96]. An open-source tool such as VPR provides us to demonstrating our approach since it requires low-level resource manipulation, whereas device information is proprietary in commercial tools.

Using VPR, we packed, placed, and routed a set of benchmark circuits onto the smallest FPGA array that can fit the circuit using an architecture based on Intel Stratix IV, characterized by logic cluster size $N = 10$, look-up table size $K = 6$,

Table 5.2: Benchmark Summary.

Circuit	6-Input LUTs	FFs	FPGA Size	W_{min}	Logic Cluster Used	Logic Cluster All	RAM Used	RAM All	Free (%) LCs	Free (%) LEs
bgm	32384	5362	75x75	80	4111	4200	0	120	2	25
LU8PE	22634	6630	61x61	92	2667	2745	45	80	3	18
LU32PE	76211	20898	111x111	128	9105	9213	150	252	1	17
LU64PE	147556	39552	153x153	156	17591	17595	293	475	0	17
mcml	101858	53736	100x100	86	7350	7400	38	208	0	9
mkDW	5590	2491	42x42	76	916	1302	41	42	30	47
stereo2	29943	18416	89x89	92	5889	5963	0	154	1	57

Table 5.3: Values of the Overlay Architectural Parameters, Used for the Experiments.

Overlay Architecture Parameter	Value
n_o	8
k_o	4
l_o	8
m	8
f_o	32

cluster input and output flexibilities of $F_{c-in} = 0.15$ and $F_{c-out} = 0.10$, respectively, channel segment length $L = 4$, and inputs per cluster $I = 33$. Table 5.2 summarizes the benchmark circuits, resources available in the smallest FPGA array, and resources occupied by the circuit. W_{min} presents the minimum number of routing tracks required to route each circuit on the minimum-sized FPGA array. We inflate this value by 30% since this is thought to best reflect the situation in a real FPGA [8, 96]. We make the same assumptions as in Chapters 3 and 4 that any unused RAM block in the FPGA can be reclaimed as a trace buffer. Furthermore, only related logic elements are packed into a cluster to better reflect industrial tools [67]. The column labelled *Free* shows the percentage of free LCs and LEs after user circuit compilation.

5.3.2 Overlay Architecture Assumptions

Table 5.3 presents the parameters that describes the overlay architecture and their values. Importantly, n_o is less than N in the target FPGA and k_o is less than K in the target FPGA; this ensures that partially filled logic clusters in the FPGA can be used to implement overlay cells. We assume that the number of overlay cell levels l_o = 8, which is sufficient to implement many of the type of trigger circuits we describe later in this section.

5.3.3 Synthetic Trigger Circuit Assumptions

In order to adequately evaluate our technique on a wide range of trigger circuits, we generate 1940 synthetic triggers, as described below. We consider three categories of trigger circuits: combinational triggers, state-based triggers, and complex triggers.

Combinational Triggers. We consider three types of combinational trigger circuits. First, we consider trigger circuits which consist of a single n-bit comparator comparing two sets of n-bit signals in the user circuit. We vary n from 16 to 256 (in powers of two), and for each value of n, we generate 100 circuits with different sets of input signals. Second, we consider trigger circuits that compare an n-bit signal to a constant; again, we vary n from 16 to 256, and for each value of n, we generate 100 circuits with different values of the constant and different input signals. Finally, we consider circuits containing a range comparator to determine if an n-bit signal in the user circuit is within a given range. As before, we vary n from 16 to 256, and for each value of n, we generate 100 circuits with different ranges and different signals in the user circuit. All together, this represents 1500 synthetic combinational circuits.

State-based Triggers. We created a program which generates synthetic state machines according to two parameters: the number of states and the maximum number of transitions from each state. Using this program, we generated synthetic state machines by sweeping the number of states through $\{8,16\}$ and the maximum number of transitions from each state through $\{2,3,4,5,6\}$. For each combination, we generated 20 state machines, giving a total of 200 state machines.

Complex Sequential Triggers. Finally, we consider trigger circuits which contain a counter, providing the ability to trigger after a specified number of occurrences or absences of a specified condition. We created a program which generates a synthetic trigger circuit that compares n-bit quantities in the user circuit with a constant, and contains an m-bit counter, along with circuitry to assert the output signal when the count reaches a specified value (up to $2^m - 1$). Using this program, we generated synthetic triggers by sweeping n from 16 to 128 (in powers of two) and m in the range $\{4,5,6\}$. For each combination, we generated 20 triggers, giving a total of 240 complex sequential triggers.

5.4 Experimental Results

5.4.1 Overlay Implementation Runtime

Table 5.4 presents the compile time overhead required to place and route the overlay fabric (characterized in Table 5.3) on top of each benchmark circuit. Adding our proposed overlay to each benchmark circuit adds less than 2.5% to the circuit compile time. This low overhead is due to two main reasons: (1) instead of creating the largest possible overlay out of unused logic resources, we assumed that the number of overlay cell levels l_o = 8; this results in fast placement. (2) the simple

Table 5.4: Overlay Compile-Time Overhead.

Circuit	Uninstrumented Circuit Compile-time (s)	Overlay Compile-time (s)
bgm	2989	21
LU8PEEng	2071	48
LU32PEEng	8773	54
LU64PEng	35615	18
mcml	13074	19
mkDW	756	14
stereo2	3574	28

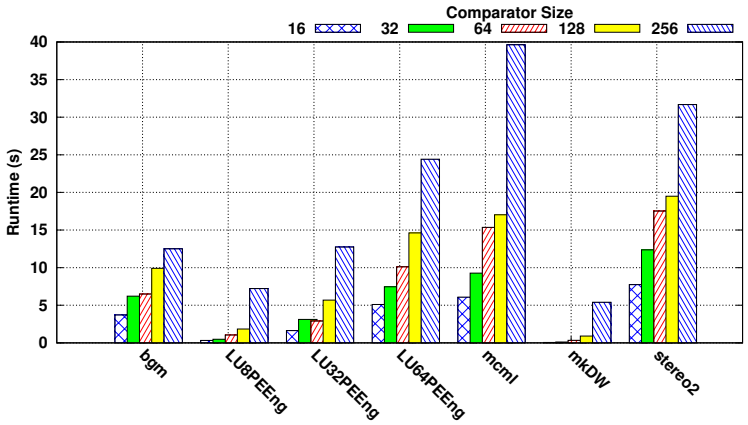
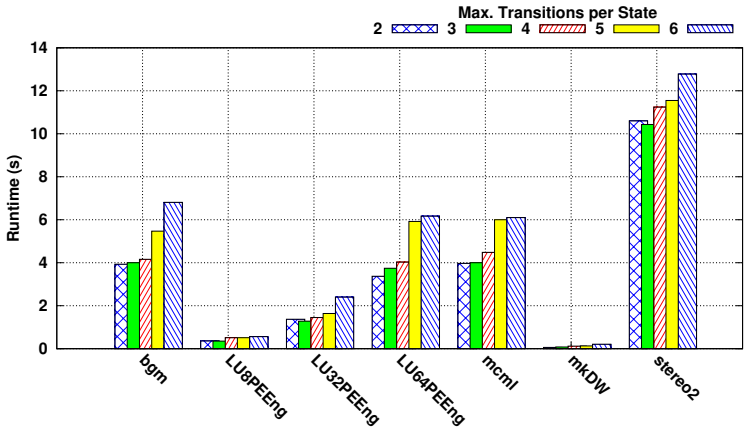


Figure 5.7: Debug-time: combinational trigger mapping runtime.

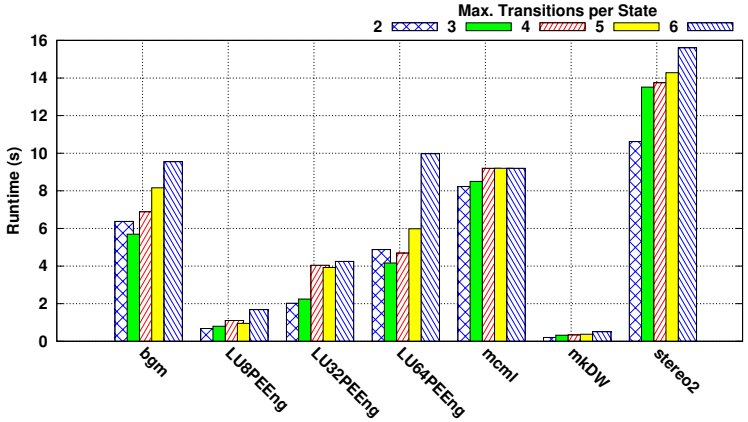
interconnect topology of our overlay architecture results in fast routing. The overlay compile time is a one-time overhead since the overlay is created only once after the user circuit and is used multiple times during debug iterations with the same circuit.

5.4.2 Trigger Mapping Runtime

Figure 5.7 shows the runtime of our trigger mapping algorithm for combinational triggers (including connecting the trigger signals to the fabric). The vertical axis



(a) 8 states.



(b) 16 states.

Figure 5.8: Debug-time: state-based trigger mapping runtime.

represents the mapping time, averaged for all trigger circuits with the given comparator size. The mapping was successful for all 1500 combinational triggers. As shown in Figure 5.7, the mapping time for each trigger circuit was less than 40 seconds. This runtime includes both the time to map the trigger functionality to the fabric, as well as the time to connect the trigger signals in the user circuit to the fabric. In all cases, time to connect the trigger signals to the fabric was dominant and the mapping runtime increases as the size of comparators increases.

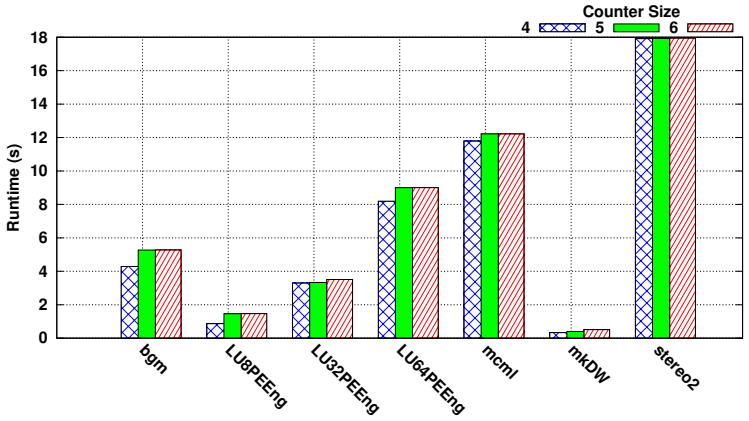


Figure 5.9: Debug-time: mapping runtime of complex sequential trigger circuits with 128-bit comparator.

Figure 5.8a shows the runtime of our trigger mapping algorithm when mapping state-based trigger circuits with 8 states as a function of the maximum number of transitions per state. The runtime depended heavily on the underlying user circuit. Circuits such as *stereovision2* had more routing congestion around memory blocks, meaning the runtime to find a legal path between the trigger output and all available memory blocks is larger.

For each of our circuits, the average compile time was less than 14 seconds. Figure 5.8b shows the same results for state machines with 16 states. In this case, not all mappings were successful. For the case with a maximum of 6 transitions per state, we found that 16 of the 20 trigger circuits could be mapped to the fabric. Of those that were successful, each trigger could be mapped in less than 16 seconds.

Figure 5.9 shows the runtime of our trigger mapping algorithm for mapping complex trigger circuits with 128-bit comparators as a function of counter size. We were able to map all the trigger circuits; on average, the runtime for the largest circuit was less than 18 seconds. As mentioned before, the runtime to connect the

Table 5.5: Effect of Trigger Insertion on Circuit Critical Path Delay (ns).

Circuit	Critical Path Delay (ns)								
	Original	16 States					128-bit Comparator		
		Max Transitions per State					Counter Size		
		2	3	4	5	6	4	5	6
bgm	19.72	19.72	20.7	20.09	21.35	21.3	19.72	19.72	19.72
LU8PE	89.64	89.64	89.64	89.64	89.64	89.64	89.64	89.64	89.64
LU32PE	91.93	91.93	91.93	91.93	91.93	91.93	91.93	91.93	91.93
LU64PE	89.23	89.23	89.23	89.23	89.23	89.23	89.23	89.23	89.23
mcml	66.59	66.59	66.59	66.59	66.59	66.59	66.59	66.59	66.59
mkDW	6.47	10.86	11.19	10.98	11.45	12.19	8.83	8.76	10.11
stereo2	11.93	12.2	12.59	12.76	12.9	13.52	11.93	11.93	11.97

trigger signals to the fabric was dominant and increasing the counter size has little effect on the runtime.

As the results show, implementing a new trigger circuitry using our techniques is significantly faster than performing a lengthy circuit recompilation at each debug iteration.

5.4.3 Circuit Critical Path Delay

Using our techniques for trigger insertion, the critical path delay of all benchmarks remained the same except *bgm*, *mkDelayWorker32B*, and *stereovision2*. Table 5.5 shows the critical path delay of the circuits before (column labeled *Uninstrumented*) and after inserting different trigger circuits averaged over all the trigger circuits. The critical path delay increases no more than an additional 1.6ns for *bgm*, 5ns for *mkDelayWorker32B*, and 1.6ns for *stereovision2*; in all cases, the worst case is when inserting trigger circuits with 16 states and a maximum of 6 transitions per state. The increase in delay in these circuits is because these circuits

have a shorter critical path than the other circuits; circuits with a short critical path are more sensitive to trigger insertion as a large trigger circuit can introduce a new long critical path to the circuit. However, this delay increase is temporary during debugging and the circuit can operate at its normal frequency when the debug instrumentation is not required.

Comparison with Circuit Recompilation. As shown in Table 5.5, our techniques had no impact on the critical path delay of the benchmark circuits in our experiments, except for *bgm*, *mkDelayWorker32B*, and *stereovision2*, which are circuits with short critical path. We also evaluate the critical path delay of these affected benchmarks when the trigger circuit is compiled with the user circuit in order to compare with our techniques when trigger circuits with 16 states and a maximum of 6 transitions per state are inserted (worst case). Our experimental results show that after compiling the user circuit with these trigger circuits, the critical path delay increases to 19.84ns, 6.53ns, and 12.25ns for *bgm*, *mkDelayWorker32B*, and *stereovision2*, respectively. As expected, recompiling the user circuit with the trigger circuit has less effect on the circuit delay compared to our incremental techniques using an overlay. However, using our techniques enable the designers to rapidly implement the desirable trigger functions without requiring a recompilation.

5.4.4 Comparison with Intel Quartus Prime

In this section, we compare our techniques with incremental compilation feature of Intel Quartus Prime, which can be used to modify the reconfiguration of SignalTap II Logic Analyzer.

Incremental compilation is enabled by setting the user circuit partition to *post-*

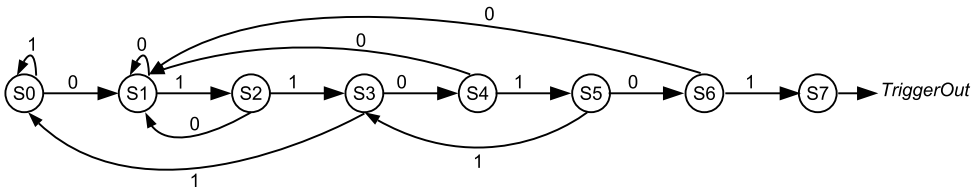


Figure 5.10: An example of a state-based trigger to detect when the pattern *0110101* has been received on a signal in consecutive cycles.

Table 5.6: Comparison between This work and Quartus Prime on Intel Stratix IV (EP4SGX180) Architecture for *mcml*.

mcml	This Work	Quartus Prime
Circuit full compilation (s)	13074	2880
Circuit Fmax (MHz)	15.01	28.82
Runtime at each debug iteration (s)	3	460

fit; the tool will attempt to preserve the placement and routing of the user circuit.

To compare our techniques to Intel's approach, we created a benchmark state-based trigger with a single user circuit input and a single trigger output; the circuit asserts its trigger output when the pattern *0110101* has been received on the input in consecutive cycles. A state diagram showing this behaviour is in Figure 5.10.

To instrument the circuit, we employ SignalTap II. We selected one registered signal at random from the circuit output as the trigger input.

The degree to which the trigger circuit can be changed without recompiling the design is limited in SignalTap II. Changing the number of states, trigger input signals, or transition conditions in the trigger circuit would require a recompilation of the trigger circuit (using Intel Quartus Prime's incremental compilation flow, the recompilation can be limited to the trigger circuit). The tool provides an option that allows the target state of each transition to be changed without recompiling the design, however, this flexibility comes at a cost in terms of resource usage

depending on the size of the trigger circuit. In our example, the trigger circuit requires 461 LEs when this option was disabled. Enabling this option increases the resource usage to 721 LEs, an increase of 56%.

Table 5.6 shows the full compilation time and maximum operating frequency (Fmax) for a large, 100,000 LUT, benchmark circuit *mcml* without any instrumentation, when targeting a Stratix IV architecture in version 15.1 of Quartus Prime and an architecture similar to Stratix IV using our flow.

To understand the impact of making a trigger circuit change on the runtime required to perform a debug iteration, we modified our example trigger to detect a different pattern, *0111101*. Table 5.6 shows the runtime to recompile the trigger circuit using our techniques and Quartus Prime.

Unsurprisingly, the academic CAD tool takes significantly longer time to compile the uninstrumented circuit. However, the runtime required to perform a debug iteration is two orders of magnitude faster using the trigger insertion flow presented in this chapter. This is due to the ability to rapidly map the trigger circuit onto the overlay fabric rather than performing a recompilation.

5.5 Summary

This chapter presented a multi-level overlay architecture and new CAD techniques that are specialized for for small combinational and sequential circuits with a single output; such circuits are typical of common trigger functions. Such specialization provides enough flexibility to enable complex triggering capabilities suitable for debugging. We have shown that the overlay fabric can be reconfigured to map various combinational and sequential triggering scenarios in less than 40 seconds, enabling rapid debug iterations. Our experiments have shown that our customized

overlay architecture and trigger mapping algorithms have only a small impact on the critical path delay.

Chapter 6

Post-Silicon Coverage using Overlays

As discussed in Chapter 2, coverage estimation during post-silicon is a challenging task and there is no standardized coverage metric and methodology for coverage analysis. The key challenge is the poor observability into the behaviour of a running chip. Although this visibility problem can be addressed by designing circuit-specific monitors and implementing them on-chip along with the user circuit, this can introduce significant area overhead. On the other hand, implementing a small subset of important coverage monitors negatively impacts a complete coverage analysis.

In this chapter, we make the observation that we can re-purpose existing FPGA-based on-chip debug cores from earlier in this thesis to facilitate on-chip coverage monitoring. More specifically, we show that the instrumentation frameworks presented in Chapter 4 and Chapter 5, which were designed to implement trigger

circuits, can be used to support the time-multiplexed implementation of coverage monitors. Instead of employing this infrastructure – a virtual overlay fabric that can be rapidly reconfigured (without compilation) during the verification process – to record a limited window of trace data, we instead use this overlay to measure coverage over the entire circuit execution. Since this overlay is implemented using spare FPGA resources, the area overhead of instrumentation is essentially zero and its performance overhead is negligible.

In this chapter, we describe our proposed instrumentation framework to efficiently implement on-chip coverage monitors in Section 6.1. In Section 6.2.1 we will revisit the overlay architecture presented in Chapter 5, and show how it can be modified to provide enough flexibility for implementing coverage monitoring circuits. Section6.3 describes coverage monitor circuits that are mapped to this overlay fabric to gather branch coverage data. Section 6.4 describes our mapping algorithm to rapidly map these branch coverage monitors to a set of configuration bits. Section 6.5 details the experimental methodology and steps that were performed to evaluate our techniques. In Section 6.6, we evaluate our techniques in terms of area overhead, overlay flexibility, and runtime. Finally in Section 6.7 we present our conclusions. A condensed version of this work was published in [46].

6.1 Framework

Figure 6.1 illustrates our overall framework. First the user circuit is compiled onto the FPGA as normal, without any coverage monitor instrumentation. Then, the user circuit is frozen, and instrumentation is added to the circuit using only resources (logic blocks, memories, and routing tracks) that are not used by the user circuit, without disturbing the packing, placement, and routing of the user circuit.

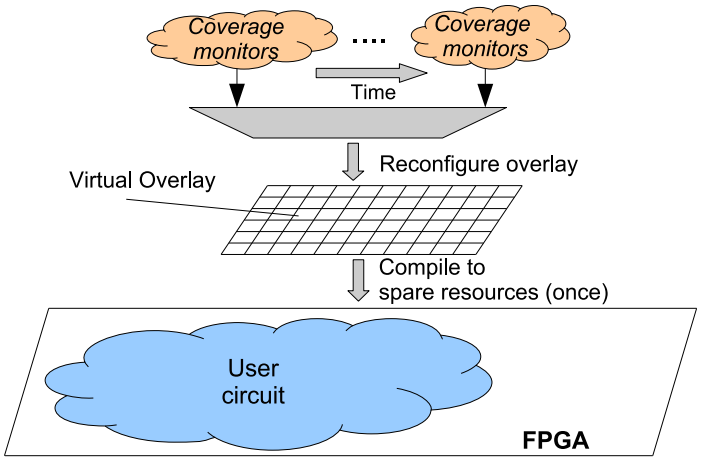


Figure 6.1: Overview of our framework.

Rather than instrumenting the user circuit with fixed coverage monitors, our instrumentation consists of a flexible overlay fabric that is flexible enough so that it can be configured, at runtime, to implement one or more coverage monitor functions. The flexibility of the overlay to implement coverage monitor functions comes from a set of configuration bits. Since the overlay is specialized for coverage monitor circuits, the overlay can be configured by setting a small number of configuration bits. Importantly, mapping the coverage monitor functions to this set of configuration bits is much faster than a full recompilation of the user circuit with coverage monitors.

After the user circuit has been mapped to the FPGA, and the overlay fabric mapped to the available resources, testing of the circuit proceeds. Before running the circuit under all test cases, coverage monitor functions applicable to this user circuit are identified, and a subset of the set of these coverage monitor functions are mapped to the overlay, as described above. The design is then run and the monitor circuits capture coverage data. The coverage data is stored in on-chip

memory while the design is running at-speed under all test cases. After the design execution, this overlay fabric is then reconfigured to implement another subset of coverage monitors and the design is re-executed under all test cases and the coverage data is stored in on-chip memory. This process is repeated and the overlay fabric is frequently reconfigured until all coverage monitors are implemented and all coverage data is gathered. At the end, the coverage information can be extracted from the instrumentation (using device read-back techniques) for off-line coverage analysis and an overall coverage measurement. Importantly, overlay reconfiguration and coverage instrumentation does not change the user circuit characteristics during the validation process since the overlay is separated from the user circuit.

The size of the overlay fabric can be adjusted based on the proportion of the FPGA that is unused by the user circuit; if the user circuit uses most of the FPGA, the overlay fabric will be small, and can only implement a small number of coverage monitor functions at at time, while if there are many unused FPGA resources after mapping the user circuit, a larger fabric, capable of implementing many simultaneous monitor functions can be implemented.

6.2 Coverage Overlay Architecture

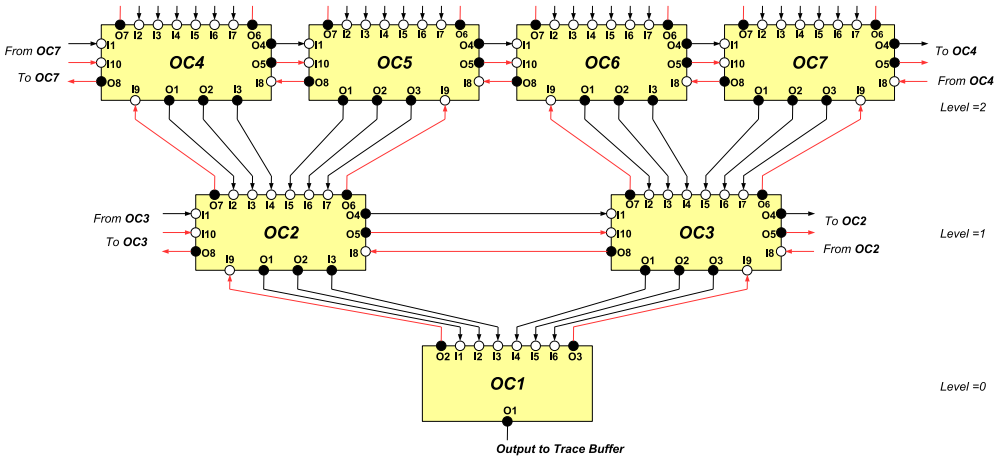
The design of a suitable overlay fabric is critical for our method. The overlay must be flexible enough to implement a variety of coverage monitor functions, with as little overhead as possible. A key observation in this work is that the overlay needed to gather coverage information is very similar to that previously proposed to enhance visibility during post-silicon debug. This section presents the overlay architecture optimized to implement coverage monitor circuits. This section shows how the debug-oriented overlay presented in Chapter 5 can be enhanced to better

implement coverage monitor functions.

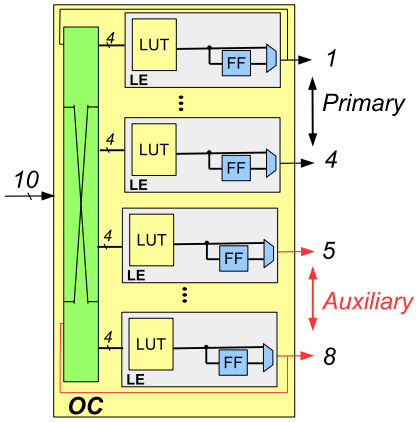
6.2.1 Overlay Architecture

Figure 6.2 illustrates our overlay architecture. As shown, the overlay is comprised of a number of cells connected in a triangular reduction-network pattern. Levels are labeled $0..l_o - 1$; level i contains 2^i cells similar to the debug overlay architecture presented in Chapter 5 (Figure 5.1a). To better implement coverage monitor functions, we revise this debug overlay fabric; instead of including a bank of flip-flops separated from overlay cells, we enhance the overlay cell design where each overlay cell consists of logic elements (LEs). The design of an overlay cell is shown in Figure 6.2b. Each overlay cell contains eight LEs. Each LE contains a 4-input LUT and a flip-flop. The LEs within a cell are connected using a fully-connected crossbar network. Each overlay cell has ten input and eight output pins (except the cell in level 0). The interconnect between cells is also shown in figure 6.2a. The trigger output is used to control trace buffers. As described in Chapter 6, we distinguish between two categories of interconnects: (1) primary connections that are used to flow signal in forward direction, (2) auxiliary connections that are added to enable each overlay cell to flow data in backward direction for supporting flip-flop feedbacks of coverage monitor circuits as will be described in Section 6.4.

Overlay cells are designed to be similar to FPGA logic clusters for efficient overlay compilation. We used the same incremental approach used in Chapter 5 (Section 5.1.2) to incrementally compile this overlay architecture onto FPGA spare logic clusters and routing tracks after user circuit compilation while preserving the mapping of the underlying user circuit.



(a) The overall structure of the coverage overlay architecture.



(b) Overlay Cell (OC) details

Figure 6.2: (a) Coverage overlay architecture that is comprised of cells connected via primary (shown in black) and auxiliary (shown in red) connections, inputs to the overlay are control signals from the user circuit and its output controls trace buffers. (b) Overlay cell design with Logic Elements (LE), local routing, and outputs to other cells, each LE contains a 4-input LUT and a flip-flop.

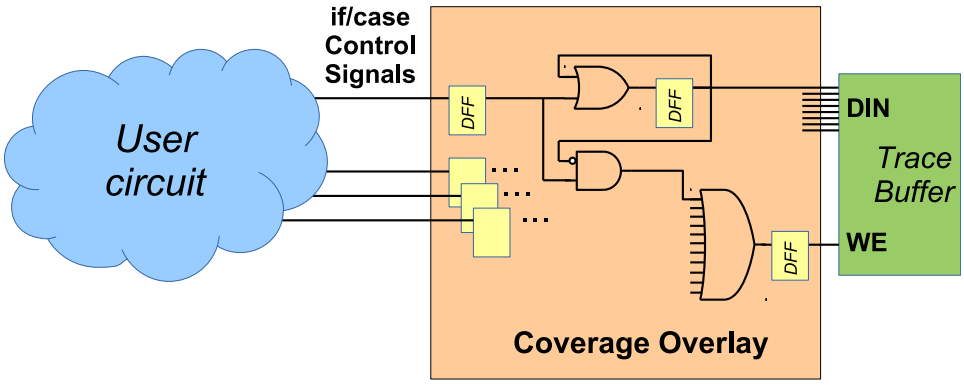


Figure 6.3: Branch coverage instrumentation that is mapped onto the overlay fabric.

6.3 Branch Coverage Instrumentation

Key to the flow described in Section 6.1 is the ability to map coverage monitor functions onto the pre-synthesized overlay fabric. In this work, we consider branch coverage, where instruments are attached to the control signals generated as part of if–else-if–else, case, and ternary operator (?:) statements.

Coverage Monitor Circuitry: Figure 6.3 shows our coverage instrumentation. Control signals extracted from the user circuit are first captured into a flip-flop, implemented inside the trigger overlay, in order to limit the effect of this instrumentation on the timing performance of the circuit. The output of this initial flip-flop is then fed into a second flip-flop that captures the value of the control signal in the previous clock cycle. The upper OR gate causes this second flip-flop to latch when a high value is encountered, and to remain at that value, while the lower AND gate (with inverted input) detects the cycle at which a rising transition occurred. Once this transition is detected, it passes through a reduction OR gate (and another pipelining flip-flop) that causes the trace-buffer to write the state of

all coverage monitors upon any rising transition. Since each coverage monitor can only transition once, the size requirements of this trace buffer scale linearly with the number of monitors. Due to the nature of FPGA technology, all flip-flops can be initialized with a value of '0'.

Control Signal Visibility: The ability to attach coverage instrumentation onto if/case control signals is dependent on whether this exact signal exists; due to synthesis optimizations, signals that exist in the original HDL may not be present in the uninstrumented result. Attaching monitors to such signals will cause a different optimization trajectory, and it would be expected that doing so would incur an area or delay overhead. This is an issue that exists regardless of whether coverage instruments are inserted at compile-time, as with the baseline case, or whether they are time-multiplexed at runtime as we propose. The overhead for the runtime case, however, is compounded by the necessity to preserve all control signals to allow coverage monitors to be attached at runtime, whereas in the compile-time case only those control signals attached to pre-selected monitors would be affected. We quantify this effect in our experiments; however, for future work we believe this overhead can be mitigated by employing spare FPGA logic to also 'reconstruct' the value of optimized signals.

6.4 Coverage to Overlay Mapping

During validation, a mapping algorithm is required to map coverage monitor functions onto the pre-synthesized overlay fabric. We use the simultaneous placement-and-routing algorithm 2 presented in Chapter 5 to find a mapping of LUTs of coverage monitor functions to a LUT-slot inside the overlay cells. For mapping sequential part of coverage monitor functions, we revised this algorithm to also

place each flip-flop of the coverage monitors onto a flip-flop slot in an overlay cell. Both forward and backward connections in the overlay routing network are used to enhance the routing flexibility when connecting flip-flop feedback paths.

6.5 Methodology

In order to evaluate our approach, we implemented our techniques using the FPGA CAD tool VPR, which is part of the open-source academic Verilog-To-Routing (VTR) project [96]. As discussed in Chapter 3, using an open-source tool was necessary because implementing our approach requires low-level resource manipulation, whereas device information is proprietary in commercial tools.

Similar to our methodology in Chapter 3, we used circuits that are supplied with the VTR project as underlying user circuits. Using VPR, we packed, placed, and routed each circuit onto the smallest FPGA array that can accommodate the circuit using the default VPR architecture based on the Intel Stratix IV. As is commonplace in FPGA research, we first map benchmark circuits to the minimum sized FPGA that it fits onto, and using the minimum number of routing tracks. We then inflate this minimum value by 30% to reflect realistic routing demand. For each circuit, unused RAM blocks in the FPGA are reclaimed for recording coverage information. In these experiments, we constructed an overlay fabric with the number of overlay cell levels $l_o = 8$ using incremental techniques presented in Chapter 4.

6.6 Experimental Results

In this section, we evaluate our runtime instrumentation techniques for inserting on-chip monitors for branch coverage at runtime in terms of (1) runtime and (2) the impact on circuit critical path delay versus compile-time instrumentation in which

Table 6.1: Benchmark circuits, total number of branch control signals, maximum number of on-chip monitors that fit on FPGA device at each instrumentation iteration, number of total instrumentation iterations for branch coverage analysis.

Benchmark	# of Control Signals	Compile-time Instr.		Run-time Instr.	
		# of monitors per iteration	# of iterations	# of monitors per iteration	# of iterations
LU8PEEng	2335	291	9	60	39
LU32PEEng	7639	238	33	60	128
mcml	1213	303	5	60	21
or1200	647	647	1	60	11
stereovision2	211	105	3	60	4

a number of pre-selected monitors are inserted into the design before compilation. Additionally, we evaluate the area overhead of preserving all control signals required for runtime instrumentation as was discussed in Section 6.3.

Coverage Instrumentation: Table 6.1 presents the total number of branch control signals for each benchmark circuit. The compile-time column in the table presents the maximum number of on-chip coverage monitors that fit into the FPGA array that accommodates the circuit. As expected, it was not possible to insert all the monitors at once (except for *or1200*) as they did not fit into the device. Hence, it requires multiple compilation iterations to instrument the design for coverage analysis. The number of recompilation iterations is also presented in Table 6.1 for each benchmark.

For our runtime instrumentation approach, we are able to map a subset of 60 monitors (attached to 60 control signals) onto the overlay fabric. Selecting subsets of monitors to map to the overlay, we group monitors related to if/else or case statements into subsets comprising of 60 monitors. For each benchmark, we map

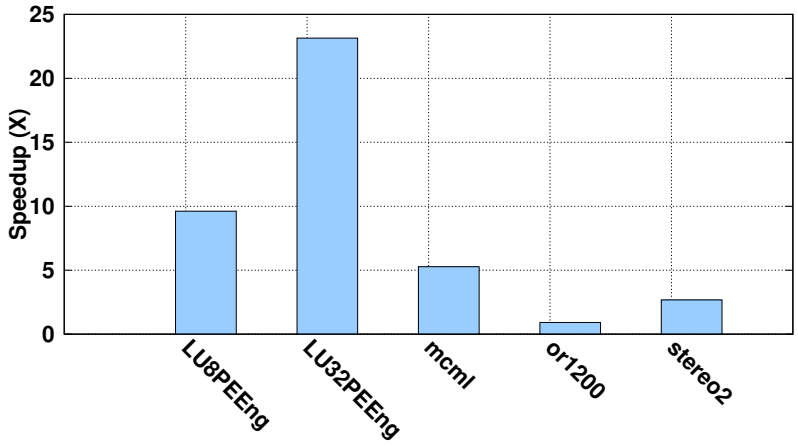


Figure 6.4: Speedup of runtime branch coverage instrumentation in comparison to compile-time instrumentation over multiple instrumentation iterations.

each subset onto the overlay and average the mapping runtime over all subsets. For our benchmarks, the time to map a subset of 60 monitors to the overlay fabric is less than 10 seconds on average. Even thought our approach requires more iterations to implement all monitors to gather all branch coverage data, it only takes 10 seconds to reconfigure the overlay to map a new subset of monitors. In contrast, the compile-time approach suffers a full recompile time of the user circuit with the instruments at each iteration. Figure 6.4 shows the total speedup of our techniques (this includes the circuit compilation time, overlay construction time, and multiple iterations of overlay reconfiguration time) when compared with the compile-time instrumentation which includes multiple iterations of full recompilation. As the results show, large circuits, and in particular, circuits with many branch control signals, significantly benefit from our runtime instrumentation. For instance, our approach provides the speedup of 23X for *LU32PEEng* benchmark that has a large number (thousands) of control signals. This speedup is achieved by

Table 6.2: Circuit critical path delay (ns) of uninstrumented circuit, instrumented circuit at compile-time and runtime instrumentation.

Benchmark	Uninstrumented	Compile-time Instr.	Runtime Instr.
LU8PEEng	89.6	90	90
LU32PEEng	91.9	90.2	91.9
mcml	66.6	63.5	67.31
or1200	11.6	11.6	11.87
stereovision2	11.9	11.8	12

eliminating a need for recompilation at each instrumentation iteration. For small circuits with a relatively small number of control signals, the benefit of our approach is smaller. For *or1200* benchmark, although it was possible to insert all monitors into the FPGA at compile time, as expected, this lead to a more complex and difficult mapping problem for the FPGA CAD tools which increases compilation time compared to our approach where the instrumentation is separated from the user circuit.

Table 6.2 presents the critical path delay of the user circuit without any instrumentation, after compile-time instrumentation and using our approach. For compile-time instrumentation, the critical path delay of the benchmark circuits changes as the instrumented circuit is recompiled from scratch. For some benchmarks, the critical path delay slightly decreases which can attributed to the algorithmic noise of CAD algorithms. Using our approach for runtime instrumentation, we had no impact or little impact on the circuit critical path delay; the critical path delay increases no worse than an additional 2.3% for *or1200*. As explained earlier, this delay increase is partly due to preserving all branch control signals. Additionally, small circuits with short critical path (such as *or1200* and *stereovision2*) are more sensitive as our instrumentation can introduce a new long critical path to the

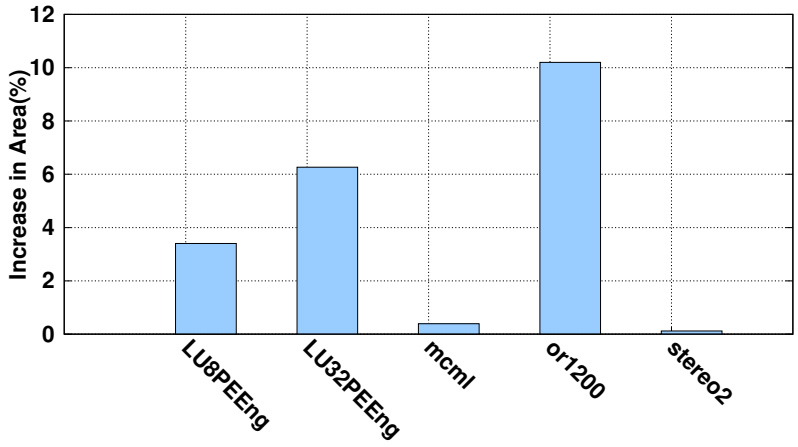


Figure 6.5: Impact of preserving all control signals on area in comparison to the uninstrumented circuit where these signals can be optimized away if possible.

circuit.

Impact of Control Signal Visibility on Area: Figure 6.5 shows the area overhead of preserving all branch control signals in comparison to the same circuit where the synthesis tool is free to optimize these signals. As discussed in Section 6.3, this area increase (average 6.4%) is the price we have to pay to enable runtime instrumentation.

6.7 Summary

This chapter demonstrates an instrumentation framework that enables on-chip coverage observability in a way that does not require large area overhead of monitors during FPGA-based validation. To achieve this, we utilize existing FPGA debug cores and techniques presented in Chapter 6 that employ an overlay fabric for debug instrumentation in order to implement monitors and gather branch coverage data. Because the overlay is customized for coverage monitoring, the time to com-

pile monitors to the overlay fabric is fast and the overlay can be reconfigured at runtime to implement monitors in a time-multiplexed fashion to gather coverage data when the design is running at speed under all test cases. Since the overlay is separated form the user circuit, the user circuit characteristics remain the same during post-silicon validation process.

The key novelties of this work are in: (1) a scalable and area-efficient coverage instrumentation framework that enables coverage monitoring at post-silicon, (2) specialized overlay architecture and mapping tools supporting runtime implementation of monitors, (3) branch coverage instrumentation to enable runtime evaluation of branch coverage during post-silicon validation.

The significance of this work is that it enables designers to gather coverage data with low overhead during FPGA-based validation while the circuit is operating many orders of magnitude faster under more realistic test cases than simulation.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The past several decades have seen tremendous growth in the capacity and capability of FPGAs. FPGA platforms are commonly used for fast prototyping to evaluate and validate the functionality of complex designs. Debugging a design while it is running at execution speed on a real hardware enables the designers to verify the design functionality in system-level with long real-world stimulus and higher functional coverage compared to software simulators.

However, post-silicon debug and verification is challenging due to the poor visibility into the design behaviour. As described in Chapter 2, observability can be added by including commercial or academic trace-based instrumentation. This instrumentation often records the runtime behaviour of selected signals in the chip, allowing it to be played back later using debug tools. Most of these debug flows, however, require the design to be recompiled every time the instrumentation is changed. For very large designs, this can be prohibitive which can severely limit

debug productivity.

Moreover, as discussed in Chapter 2, coverage analysis during post-silicon validation is very difficult task due to lack of observability. On-chip circuit-specific coverage monitors have not been widely used in practice since implementing a large number of coverage monitors imposes a high area overhead and is not affordable in silicon. Hence, in this thesis, we have proposed area-efficient methods for instrumentation of FPGA designs for debug and coverage monitoring in a way that does not require frequent recompilation during FPGA debug and verification process. The rest of this chapter is as follows. Section 7.2 provides a summary of the contribution of this thesis and Section 7.3 discusses their limitations and possible research directions based on the findings from this thesis.

7.2 Summary of Contributions

In Chapter 3, we used incremental compilation techniques to insert the trigger circuitry into a design without requiring a full recompilation; trigger logic elements are distributed throughout the unused logic resources after user circuit compilation and spare routing resources are used to make the required connections between these logic elements and to the user circuit using incremental routing techniques. Our results show that it is feasible to distribute trigger logic over spare logic resources after user circuit compilation. However, making connections between these distributed trigger logics depends on the routing congestion in the user circuit.

In Chapter 4, we enhanced the ability of rapid trigger implementation by introducing an instrumentation framework through the use of overlay architectures, which are compiled once, and configurable between debug iterations. We then presented an overlay architecture based on a 2D torus that provides adequate flexibility

for implementing combinational trigger circuits.

We then present non-intrusive, adaptive, and best-effort CAD techniques for mapping the overlay architecture on top of the user circuit. Our approach is area-efficient because we only use those resources left unused by the user circuit and hence the area overhead is potentially zero. The construction algorithm includes three major parts: (1) overlay cell selection, in which partially used or empty logic clusters are selected as the overlay cells to create a logical overlay fabric; (2) adaptive overlay placement, in which an adaptive simulated annealing based placement algorithm is used to place the logical overlay into those FPGA resource left unused by the user circuit while minimizing the total wire length; and (3) best-effort overlay routing in which the routability-driven PathFinder algorithm is used to iteratively resolve routing congestion. If congestion cannot be resolved, illegal connections that are overusing routing resources are iteratively removed from the overlay until congestion is resolved and a legal routing solution is achieved.

At debug time, a routing-aware simulated-annealing based placement algorithm was proposed to place a trigger circuitry onto the overlay; swaps that could result in routing failure were heavily penalized to enhance the routability of the placement solution.

Our experiments have shown that for the benchmarks that were investigated, we were able to build overlay architectures with different sizes depending on the available spare resources while increasing the circuit compile time by an average of 22% assuming a maximally-sized overlay fabric. However, this compile time is only required once to build the overlay architecture. During debugging, this overlay fabric can be reconfigure multiple times to implement different trigger circuits. Our experiments have shown that the overlay architecture provides enough flexibility

to implement different combinational trigger circuits at least an order of magnitude faster than a full recompilation, enabling rapid debug iterations.

In Chapter 5, we extend our instrumentation framework in Chapter 4 by introducing a new overlay architecture and mapping tools optimized for implementing sequential (state-based) trigger circuits. We proposed a parameterized overlay architecture family that is specialized to implement sequential trigger functions. To implement the logic part of the trigger circuit, the overlay contains a number of cells connected in a triangular reduction-network pattern. To support sequential triggers, the overlay also contains a bank of flip-flops. We then modified the overlay construction strategy presented in Chapter 4 to construct this multi-level overlay fabric on top of user circuit. The overlay fabric increases compile time no more than 2.5% for the benchmark circuits that were investigated. As mentioned before, the overlay compile time is a one-time overhead.

Also in Chapter 5, we developed a trigger mapping flow to implement trigger circuits onto the overlay fabric between debug iterations. To take advantage of the specialized overlay architecture, we developed a mapping flow to perform a gradual and simultaneous placement and routing to ensure a legal mapping solution, which includes two major phases: (1) Pre-processing, in which the trigger circuit graph is modified to suit the optimized overlay architecture; (2) Placement/Routing, in which the logic elements of the trigger circuit are placed and routed onto the overlay cells. We first described a mapping algorithm for combinational triggers and then generalized the algorithm for sequential trigger circuits. Our experiments have shown that the overlay fabric can be rapidly reconfigured to implement different trigger circuits. For our benchmarks, we were able to reconfigure the overlay fabric in less than 40 seconds to implement different combinational and sequential

triggering scenarios. This results suggest that our triggering framework is a viable approach for enabling rapid triggering capabilities during FPGA debug.

In Chapter 6, we utilized trace-based debug infrastructure to provide visibility for coverage evaluation during post-silicon verification; instead of using trace buffers to record trace data for a limited number of clock cycles, we used trace buffers to record coverage data during the entire design execution.

In this chapter, we first proposed a framework for implementing on-chip coverage monitors with low area overhead and in a way that does not require recompilation of the user circuit. Our proposed solution is through the use of an overlay architecture which is compiled once, and is configurable at runtime to implement coverage monitors in a time-multiplexed fashion to gather coverage data. We revisited the multi-level overlay architecture and mapping algorithms presented in Chapter 5, and made them suitable to support the time-multiplexed integration of coverage monitors.

We evaluated our techniques for runtime implementation of on-chip branch coverage monitors versus compile-time instrumentation, where the design is instrumented with a fixed number of coverage monitors before compilation. Our experiments have shown that using our approach to gather branch coverage data is up to 23X faster compared to compile-time instrumentation. To allow runtime branch coverage monitoring, all control signals for if/case in the original HDL were preserved that resulted in an area overhead of 6.4% on average compared to uninstrumented design where these signals are optimized if possible. The significance of this contribution is that it enables designers to find functional coverage during FPGA-based validation with a low area overhead and without performing lengthy recompilations.

7.3 Limitations and Future Research Directions

This section discusses limitations of this work and outlines a number of possible extensions which can be explored in future work.

Using our techniques for incremental instrumentation may increase the delay of the instrumented design. This may limit the use of our techniques in debug and verification of applications that require strict timing constraints. One example would be video encoding/decoding engine in media streaming applications, where frames are expected to be processed in a certain frame rate. To minimized the effect of instrumentation on circuit timing, future work can apply pipelining techniques to the overlay connections in order to reduce the effect of debug or coverage instruments on circuit performance. One challenge would be to map the trigger or coverage monitor circuits to this pipelined overlay without changing the functionality. Typically, the drawback of doing this is an increase in signal latency as the signal will require more clock cycles to reach its destination, resulting in a delay to stop/start capturing data. However, this will not be an overhead as stopping/starting recording data a few cycles after an interesting event is tolerable.

In Chapter 6, in order to enable runtime branch coverage monitoring, we identified branch control signals from the original HDL and prevented the synthesis tool from optimizing these signals, so that these signals exist for runtime coverage monitoring at post-silicon. This preservation prevented the synthesis tool to fully optimize the design. This could result in an area or delay overhead in the user design. One possible solution to avoid these overheads is to reclaim FPGA spare logic resources to duplicate logics required to reconstruct optimized signals using incremental techniques. This way, the tool is free to fully optimize the user circuit.

Another possible direction is to use our instrumentation framework for implementing compression schemes. To make the most efficient use of trace buffers, it is often useful to compress data before it is stored. In general, trace data is extremely compressible. This compression may be general purpose as described in [11] or optimized for a specific application as in [52]. A general purpose compression engine may be too large to implement in an overlay, however, simple application-specific compression schemes may be very suitable.

In this thesis we assumed the user circuit is not implemented on an overlay (but rather compiled to the native FPGA resources). As discussed in Chapter 2, several researchers have described how overlays can accelerate the design and compile time of FPGA designs in recent years. If the user circuit is implemented on an overlay, another interesting architectural possibility exists: co-optimizing the overlay architecture for both the user circuit and the debug instrumentation. This would eliminate the need for a separate debug overlay and would provide the ability to rapidly change the allocation of space between the user circuit and the debug instrumentation. Implementing such an overlay also needs to employ custom compilation tools. Therefore, an interesting direction of research could be to find a way to create a single overlay architecture that works efficiently for the user circuit and implements desirable instruments for post-silicon debug.

We showed that our techniques perform better in terms of overlay construction time and routability when less circuit routing congestion exists. [67] showed that commercial tools do not necessarily pack and place as densely as VPR as long as the design fits onto the device in order to provide better circuit routability. Consequently, we expect that our techniques to be more effective in commercial tools than in VPR. Additionally, in this thesis debug instrumentation is limited to us-

ing only the resources that were not used by the user circuit. Although this has some advantages such as allowing designers to debug the instrumented design that is close to the uninstrumented circuit, and preserving low-level optimizations, we would like to investigate the effect of relaxing this limitation in future. For example, it may be beneficial to enable the tool to pre-reserve some logic blocks and routing wires specifically for constructing the overlay on top of a less compact packed and placed design.

Finally, in this thesis we used overlays to implement triggering circuitry that is used to monitor for a specific event to control recording data. Alternatively, instead of simply controlling when to record data, it may also be interesting to explore the possibility of using virtual debug overlays to also change signal values in the circuit, that can be used to implement bug fixes or evaluate error resilience (e.g bug injection) during post-silicon debug.

Bibliography

[1] IEEE Standard for property specification language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010. doi:10.1109/IEEESTD.2010.5446004. → page 27

[2] Design and verification in the SoC era. Mentor Graphics, 2014. URL https://verificationacademy.com/seminars/design-verification-soc-era. → page 1

[3] Modelsim: ASIC and FPGA design. Mentor Graphics, 2014. URL http://www.mentor.com/products/fv/modelsim/. → page 1

[4] M. Abramovici. In-system silicon validation and debug. *IEEE Design Test of Computers*, 25(3):216–223, May 2008. → page 18

[5] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proceedings of the Design Automation Conference*, pages 7–12. ACM, 2006. → page 19

[6] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *Proceedings of the International Conference on Hardware and Software: Verification and Testing*, HVC'10, pages 60–75, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19582-2. → page 28

[7] R. Agalya and S. Saravanan. Simultaneous signal selection for silicon debug through mixed-integer linear programming. In *International Conference on Emerging Trends in Engineering, Technology and Science (ICETETS)*, pages 1–3, Feb 2016. → page 19

[8] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on*

Very Large Scale Integration Systems, 12(3):288–298, 2004. → pages 66, 94

[9] G. Al-Hayek and C. Robach. From design validation to hardware testing: A unified approach. *Journal of Electronic Testing*, 14(1):133–140, Feb 1999. ISSN 1573-0727. doi:10.1023/A:1008317826940. URL https://doi.org/10.1023/A:1008317826940. → page 27

[10] Amazon. Amazon ec2 f1 instances: Run customizable FPGAs in the AWS cloud, Dec. 2016. URL https://aws.amazon.com/ec2/instance-types/f1/. → page 2

[11] E. Anis and N. Nicolici. On using lossless compression of debug data in embedded logic analysis. In *International Test Conference*, pages 1–10. IEEE, 2007. → page 125

[12] E. Anis and N. Nicolici. Interactive presentation: Low cost debug architecture using lossy compression for silicon debug. In *Proceedings of the conference on Design, automation and test in Europe*, pages 225–230. EDA Consortium, 2007. → page 19

[13] E. Anis Daoud and N. Nicolici. On using lossy compression for repeatable experiments during silicon debug. *IEEE Transactions on Computers*, 60 (7):937–950, 2011. → page 19

[14] 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 *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 110–117, Nov 2012. → page 28

[15] K. Balston, M. Karimibiuki, A. J. Hu, A. Ivanov, and S. J. E. Wilton. Post-silicon code coverage for multiprocessor system-on-chip designs. *Transactions on Computers*, 62(2):242–246, Feb 2013. → page 28

[16] K. Basu and P. Mishra. Efficient trace data compression using statically selected dictionary. In *VLSI Test Symposium (VTS)*, pages 14–19. IEEE, 2011. → page 19

[17] K. Basu and P. Mishra. Rats: restoration-aware trace signal selection for post-silicon validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(4):605–613, 2013. → page 19

[18] K. Basu, P. Mishra, and P. Patra. Efficient combination of trace and scan signals for post silicon validation and debug. In *International Test Conference*, pages 1–8. IEEE, Sept 2011. → page 19

[19] S. BeigMohammadi and B. Alizadeh. Combinational trace signal selection with improved state restoration for post-silicon debug. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1369–1374, March 2016. → page 19

[20] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, MA, USA, 1999. ISBN 0792384601. → pages 37, 38, 59

[21] S. Birk, J. G. Steffan, and J. H. Anderson. Parallelizing FPGA placement using transactional memory. In *International Conference on Field-Programmable Technology (FPT)*, pages 61–69. IEEE, 2010. → page 59

[22] T. Bojan, M. A. Arreola, E. Shlomo, and T. Shachar. Functional coverage measurements and results in post-silicon validation of core 2 duo family. In *International High Level Design Validation and Test Workshop*, pages 145–150. IEEE, Nov 2007. → page 28

[23] M. Boul and Z. Zilic. *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, 2008. → page 28

[24] M. Boule and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 221–228. IEEE, 2005.

[25] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *International Symposium on Quality Electronic Design (ISQED)*, pages 613–620. IEEE, 2007. → page 28

[26] A. Brant and G. G. Lemieux. ZUMA: An open FPGA overlay architecture. In *Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE, 2012. → pages 6, 30

[27] P. K. Bussa, J. Goeders, and S. J. E. Wilton. Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation

techniques. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, Sept 2017. → page 24

[28] D. Bustan, D. Korchemny, E. Seligman, and J. Yang. Systemverilog assertions: Past, present, and future SVA standardization experience. *Design & Test of Computers, IEEE*, 29(2):23–31, 2012. → page 27

[29] N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debugging for FPGA high-level synthesis. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014. → pages 4, 17, 23, 24

[30] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, Sept. 2013. URL http://doi.acm.org/10.1145/2514740. → page 23

[31] D. Capalija and T. Abdelrahman. Tile-based bottom-up compilation of custom mesh-of-FUs FPGA overlays. In *International Conference on Field Programmable Logic and Applications*. IEEE, 2014. → pages 6, 31

[32] D. Capalija and T. S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *International Conference on Field programmable Logic and Applications*, pages 1–8. IEEE, Sept 2013. → page 31

[33] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. iDEA: A DSP block based FPGA soft processor. In *International Conference on Field-Programmable Technology*, pages 151–158, Dec 2012. → page 31

[34] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. C. Wang. Challenges and trends in modern SoC design verification. *IEEE Design Test*, 34(5):7–22, Oct 2017. → pages 2, 15

[35] J. Cong, J. Peck, and Y. Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. In *Proceedings of the international symposium on Field-programmable gate arrays*, pages 137–143. ACM, 1996. → page 53

[36] J. Cong, H. Huang, and X. Yuan. Technology mapping and architecture evalution for k/m-macrocell-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 10(1):3–23, 2005. → page 53

[37] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. → page 2

[38] P. Cooke, L. Hao, and G. Stitt. Finite-state-machine overlay architectures for fast FPGA compilation and application portability. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):54, 2015. → page 31

[39] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the international conference on Hardware/software codesign and system synthesis*, pages 13–22. ACM, 2010. → page 6

[40] H. M. Cristinel Ababei and K. Bazargan. Three-dimensional place and route for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1132–1140, 2006. → page 59

[41] E. A. Daoud and N. Nicolici. Real-time lossless compression for silicon debug. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(9):1387–1400, Sept 2009. ISSN 0278-0070. → page 19

[42] F. Eslami and S. J. Wilton. Incremental distributed trigger insertion for efficient fpga debug. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014. → pages vi, 9, 36

[43] F. Eslami and S. J. Wilton. An improved overlay and mapping algorithm supporting rapid triggering for FPGA debug. *ACM SIGARCH Computer Architecture News*, 44(4):20–25, 2017. → pages vii, 11, 76

[44] F. Eslami and S. J. E. Wilton. An adaptive virtual overlay for fast trigger insertion for FPGA debug. In *International Conference on Field Programmable Technology (FPT)*, pages 32–39. IEEE, Dec 2015. → pages vi, 10, 51

[45] F. Farahmandi, R. Morad, A. Ziv, Z. Nevo, and P. Mishra. Cost-effective analysis of post-silicon functional coverage events. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 392–397, March 2017. → page 29

[46] E. H. Fatemeh Eslami and S. J. E. Wilton. Extending post-silicon coverage measurement using time-multiplexed FPGA overlays. In *European Test Symposium (ETS)*. IEEE, May 2018. → pages vii, 12, 106

[47] H. Foster, A. Krolnik, and D. Lacey. *Assertion-based design*. Kluwer Academic Publishers, Boston, USA, 2004. ISBN 1402080271;9781402080272. → page 27

[48] H. D. Foster. Trends in functional verification: a 2014 industry study. In *Proceedings of the Annual Design Automation Conference*, page 48. ACM, 2015. → pages 2, 15

[49] R. O. Gallardo, A. J. Huy, A. Ivanov, and M. S. Mirian. Reducing post-silicon coverage monitoring overhead with emulation and bayesian feature selection. In *International Conference on Computer-Aided Design (ICCAD)*, pages 816–823. IEEE/ACM, Nov 2015. → page 28

[50] M. Gao and K. T. Cheng. A case study of time-multiplexed assertion checking for post-silicon debugging. In *International High Level Design Validation and Test Workshop (HLDVT)*, pages 90–96. IEEE, June 2010. → page 29

[51] J. Goeders and S. Wilton. Effective FPGA debug for high-level synthesis generated circuits. In *International Conference on Field Programmable Logic and Applications*, pages 1–8. IEEE, 2014. → pages 23, 24

[52] J. Goeders and S. J. Wilton. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134. IEEE, 2015. → pages 19, 125

[53] J. Goeders and S. J. Wilton. Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):83–96, 2017. → pages 4, 17, 23, 24

[54] F. Golshan. Test and on-line debug capabilities of IEEE Std 1149.1 in UltraSPARC TM-III microprocessor. In *Proceedings of the International Test Conference*, pages 141–150. IEEE, 2000. → page 16

[55] J. Goodenough and R. Aitken. Post-silicon is too late avoiding the $50 million paperweight starts with validated designs. In *Proceedings of the Design Automation Conference*, pages 8–11, New York, NY, USA, 2010. ACM. → page 15

[56] M. Gort, F. M. D. Paula, J. J. W. Kuan, T. M. Aamodt, A. J. Hu, S. J. E. Wilton, and J. Yang. Formal-analysis-based trace computation for

post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(11):1997–2010, Nov 2012. → page 19

[57] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32 (5):38–51, Sept 2012. → page 31

[58] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. In *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 41–50. IEEE, 2001. → page 21

[59] J. Havlicek and Y. Wolfsthal. PSL and SVA: Two standard assertion languages addressing complementary engineering needs. *Design and Verification Conference and Exhibition (DVCon)*, pages 1–7, 2005. → page 27

[60] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fan-out in logical networks. *Journal of the ACM (JACM)*, 31(1):13–18, 1984. → page 83

[61] A. Hopkins and K. McDonald-Maier. Debug support for complex systems on-chip: A review. *IET Proceedings-Computers and Digital Techniques*, 153(4):197–207, 2006. → page 16

[62] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker. A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 143–149. IEEE, May 2011. → page 30

[63] E. Hung and S. J. Wilton. Scalable signal selection for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6): 1103–1115, 2013. → page 19

[64] E. Hung and S. J. Wilton. Towards simulator-like observability for FPGAs: a virtual overlay network for trace-buffers. In *Proceedings of the international symposium on Field programmable gate arrays*, pages 19–28. ACM, 2013. → pages 22, 34, 42

[65] E. Hung and S. J. Wilton. Accelerating FPGA debug: Increasing visibility using a runtime reconfigurable observation and triggering network. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 19 (2):14, 2014. → page 23

[66] E. Hung and S. J. Wilton. Incremental trace-buffer insertion for FPGA debug. *IEEE Transactions on Very Large Scale Integration Systems*, 22(4): 850–863, 2014. → pages 22, 34, 41, 45

[67] E. Hung, F. Eslami, and S. J. Wilton. Escaping the academic sandbox: Realizing vpr circuits on xilinx devices. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52. IEEE, 2013. → pages 45, 64, 94, 125

[68] E. Hung, A. S. Jamal, and S. J. E. Wilton. Maximum flow algorithms for maximum observability during FPGA debug. In *International Conference on Field-Programmable Technology (FPT)*, pages 20–27. IEEE, Dec 2013. → page 22

[69] E. Hung, T. Todman, and W. Luk. Transparent insertion of latency-oblivious logic onto FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 1–8. IEEE, 2014. → page 22

[70] B. L. Hutchings and J. Keeley. Rapid post-map insertion of embedded logic analyzers for xilinx FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 72–79. IEEE, 2014. → pages 22, 70

[71] M. D. Hutton, J. Rose, J. P. Grossman, and D. G. Corneil. Characterization and parameterized generation of synthetic combinational benchmark circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):985–996, 1998. → page 78

[72] Intel. Intel FPGA SDK for OpenCL, Programming Guide UG-OCL002 (17.1). December 2017. → page 23

[73] Intel Corporation. Intel completes acquisition of altera, Dec. 2015. URL https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/. → page 2

[74] Intel Corporation. Quartus Prime Standard Edition Handbook Volume 3: Verification; 14. Design Debugging with the SignalTap II Logic Analyzer. San Jose, CA, USA, Nov. 2017. → pages 4, 11, 17, 21, 23, 24

[75] Y. Iskander, C. Patterson, and S. Craven. High-level abstractions and modular debugging for FPGA design validation. *ACM Transactions on Reconfigurable Technol. Syst.*, 7(1):2:1–2:22, Feb. 2014. ISSN 1936-7406. → page 16

[76] Y. S. Iskander, C. D. Patterson, and S. D. Craven. Improved abstractions and turnaround time for FPGA design validation and debug. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 518–523. IEEE, 2011. → page 16

[77] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell. Adapting the dyser architecture with DSP blocks as an overlay for the xilinx zynq. *SIGARCH Comput. Archit. News*, 43(4):28–33, Apr. 2016. ISSN 0163-5964. → pages 6, 31

[78] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8. IEEE, May 2016. → pages 6, 31

[79] A. Jamal and S. S. Wilton. Architecture exploration for HLS-oriented FPGA debug overlays. In *International Symposium on Field-Programmable Gate Arrays*, pages 1–10, 2018. → page 25

[80] D. Josephson. The good, the bad, and the ugly of silicon debug. In *Proceedings of the annual Design Automation Conference*, pages 3–6. ACM, 2006. → page 3

[81] A. B. Kahng and S. Mantik. Measurement of inherent noise in eda tools. In *Proceedings International Symposium on Quality Electronic Design*, pages 206–211, 2002. → page 5

[82] N. Kapre, N. Mehta, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, A. DeHon, et al. Packet switched vs. time multiplexed FPGA overlay networks. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 205–216. IEEE, 2006. → page 6

[83] H. F. Ko and N. Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(2): 285–297, 2009. → pages 4, 17

[84] H. F. Ko and N. Nicolici. Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging. In *European Test Symposium*, pages 62–67. IEEE, May 2010. → page 19

[85] H. F. Ko and N. Nicolici. Mapping trigger conditions onto trigger units during post-silicon validation and debugging. *IEEE Transactions on Computers*, 61(11):1563–1575, Nov 2012. → page 17

[86] D. Koch, C. Beckhoff, and G. G. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2013. → pages 6, 30

[87] A. Kourfali and D. Stroobandt. Efficient hardware debugging using parameterized FPGA reconfiguration. In *International Symposium Workshops on Parallel and Distributed Processing*, pages 277–282. IEEE, 2016. → page 22

[88] B. Kumar, K. Basu, M. Fujita, and V. Singh. Rtl level trace signal selection and coverage estimation during post-silicon validation. In *International High Level Design Validation and Test Workshop (HLDVT)*, pages 59–66. IEEE, Oct 2017. → page 29

[89] P. D. Kundarewich and J. Rose. Synthetic circuit generation using clustering and iteration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):869–887, 2004. → page 78

[90] A. Landy and G. Stitt. A low-overhead interconnect architecture for virtual reconfigurable fabrics. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, pages 111–120. ACM, 2012. → pages 6, 31

[91] M. Li and A. Davoodi. Multi-mode trace signal selection for post-silicon debug. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 640–645, 2014. → page 19

[92] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer. Beyond traditional microprocessors for geoscience high-performance computing applications. *IEEE Micro*, 31(2):41–49, 2011. → page 2

[93] C. Liu, H.-C. Ng, and H.-H. So. Quickdough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *International Conference on Field-Programmable Technology*. IEEE, 2015. → page 31

[94] X. Liu and Q. Xu. Interconnection fabric design for tracing signals in post-silicon validation. In *Proceedings of the Annual Design Automation Conference*, pages 352–357, New York, NY, USA, 2009. ACM. → page 19

[95] X. Liu and Q. Xu. On efficient silicon debug with flexible trace interconnection fabric. In *International Test Conference*, pages 1–9. IEEE, Nov 2012. → page 19

[96] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014. → pages 35, 42, 63, 66, 93, 94, 113

[97] R. Lysecky, K. Miller, F. Vahid, and K. Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *Proceedings of international symposium on Field-programmable gate arrays*, pages 271–271. ACM/SIGDA, 2005. → pages 6, 30

[98] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan. Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8, Piscataway, NJ, USA, 2015. IEEE Press. → page 19

[99] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *Proceedings of the international symposium on Field Programmable Gate Arrays (FPGA)*, pages 203–213. ACM, 2000. → page 59

[100] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the international symposium on Field-programmable gate arrays (FPGA)*, pages 111–117. ACM, 1995. → pages 58, 63

[101] Mentor Graphics. Certus silicon debug, May 2017. URL https://www.mentor.com/products/fv/certus-silicon-debug. → pages 17, 18

[102] Mishchenko et al. ABC: A system for sequential synthesis and verification, Mar. 2012. URL http://www.eecs.berkeley.edu/~alanmi/abc/. → pages 37, 53, 82

[103] P. Mishra, R. Morad, A. Ziv, and S. Ray. Post-silicon validation in the SoC era: A tutorial introduction. *IEEE Design Test*, 34(3):68–92, June 2017. → pages 2, 15

[104] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Proceedings of the Design Automation Conference*, pages 12–17. ACM, 2010. → pages 2, 3, 6, 15

[105] J. S. Monson and B. Hutchings. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, Sept 2014. → pages 23, 24, 25

[106] J. S. Monson and B. L. Hutchings. Using source-level transformations to improve high-level synthesis debug and validation on FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 5–8. ACM, 2015. → pages 23, 24

[107] M. H. Neishaburi and Z. Zilic. An infrastructure for debug using clusters of assertion-checkers. *Microelectronics Reliability*, 52(11):2781–2798, 2012. → page 28

[108] S. Prabhakar, R. Sethuram, and M. S. Hsiao. Trace buffer-based silicon debug with lossless compression. In *International Conference on VLSI Design (VLSI Design)*, pages 358–363. IEEE, 2011. → page 19

[109] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE/ACM, 2014. → page 2

[110] B. Quinton and S. Wilton. Concentrator access networks for programmable logic cores on SoCs. In *International symposium on Circuits and Systems*, pages 45–48, 2005. → page 18

[111] B. Quinton and S. Wilton. Post-silicon debug using programmable logic cores. In *International Conference on Field-Programmable Technology*, pages 241–247. IEEE, Dec 2005. → page 18

[112] B. R. Quinton, A. M. Hughes, and S. J. Wilton. Post-silicon debug of complex multi clock and power domain ICs. In *International Workshop on Silicon Debug and Diagnosis*. IEEE, 2010. → page 19

[113] K. Rahmani, S. Proch, and P. Mishra. Efficient selection of trace and scan signals for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(1):313–323, Jan 2016. → page 19

[114] J. E. Savage. *The Complexity of Computing*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1987. ISBN 089874833X. → page 83

[115] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2013. → page 31

[116] W. Shum and J. H. Anderson. Analyzing and predicting the impact of CAD algorithm noise on FPGA speed performance and power. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 107–110, New York, NY, USA, 2012. ACM. → page 5

[117] S. Sirowy and A. Forin. Wheres the beef? why FPGAs are so fast. Technical Report MSR-TR-2008-130, Microsoft Research, Microsoft Corp., Redmond, WA, 2008. → page 2

[118] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3(3): 81–84, Sept 2011. → page 31

[119] Synopsys. Identify: Simulator-like visibility into hardware debug, May 2017. URL https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/identify-rtl-debugger-ds.pdf. → pages 4, 17, 21, 23

[120] P. Taatizadeh and N. Nicolici. Automated selection of assertions for bit-flip detection during post-silicon validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12): 2118–2130, 2016. → page 29

[121] S. Thakur and D. Wong. On designing ULM-based FPGA logic modules. In *Proceedings of the international symposium on Field-programmable gate arrays*, pages 3–9. ACM, 1995. → page 53

[122] B. Vermeulen and S. K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design & Test of Computers*, 19(3):37–45, 2002. → pages 16, 18

[123] T. Wheeler, P. Graham, B. E. Nelson, and B. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 483–492. Springer-Verlag, 2001. → page 16

[124] S. J. Wilton, N. Kafafi, J. C. Wu, K. A. Bozman, V. O. Aken'Ova, and R. Saleh. Design considerations for soft embedded programmable logic cores. *IEEE Journal of Solid-State Circuits*, 40(2):485–497, 2005. → page 78

[125] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings on Software Testing, Verification, and Analysis*, pages 152–158, Jul 1988. → page 27

[126] Xilinx. High-Level Synthesis, Vivado Design Suite User Guide UG902 (v2016.1). San Jose, CA, USA, Jun 2016. → page 23

[127] Xilinx. Programming and Debugging, Vivado Design Suite User Guide UG908 (v2017.1). San Jose, CA, USA, April 2017. → pages 4, 17, 21, 23

[128] J. S. Yang and N. A. Touba. Enhancing silicon debug via periodic monitoring. In *International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 125–133. IEEE, Oct 2008. → page 16

[129] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *International Conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70, 2008. → page 31