An Incremental Trace-Based Debug System for Field-Programmable Gate-Arrays

Jared Matthew Keeley
Brigham Young University - Provo

An Incremental Trace-Based Debug System for Field-Programmable Gate-Arrays

Jared M. Keeley

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

November 2013

ABSTRACT

An Incremental Trace-Based Debug System for
Field-Programmable Gate-Arrays

Jared M. Keeley
Department of Electrical and Computer Engineering, BYU
Master of Science

Modern society increasingly relies upon integrated circuits (ICs). It can be very costly if ICs do not function properly, and large portions of designer effort are spent on their verification. The use of field-programmable gate arrays (FPGAs) for verification and debug of ICs is increasing. FPGAs are faster than simulation and cost less than fabricating an ASIC prototype. However, the major challenge of using FPGAs for verification and debug is observability. Designers must use special techniques to observe the values of FPGA's internal signals.

This thesis proposes a new method for increasing the observability of FPGAs and demonstrates its feasibility. The new method incrementally inserts trace buffers controlled by a trigger into already placed-and-routed FPGA designs. Incremental insertion allows several drawbacks of typical trace-based approaches to be avoided such as influencing the placing and routing of the design, large area overheads, and slow turnaround times when changes must be made to the instrumentation. It is shown that it is possible to observe every flip flop in Xilinx Virtex-5 designs using the method, given that enough trace buffer capacity is available.

We investigate factors that influence the results of the method. It is shown that making the trace buffers wide may lead to routing failures. Congested areas of the circuit must be avoided when placing the trigger or this may also lead to routing failures. A drawback of the method is that it may increase the minimum period of the design, but we show that pipelining can reduce these effects. The method proves to be a promising way to observe thousands of signals in a design, potentially allowing designers to fully reconstruct the internal values of an FPGA over multiple clock cycles to assist in verification and debug.

Keywords: FPGAs, verification, debug, incremental synthesis, observability

ACKNOWLEDGMENTS

I would like to thank Dr. Brad Hutchings for guiding me to the topic of this thesis and through its research and writing. His wisdom and patience has been greatly appreciated. I could not have done this work without him.

I am grateful for Dr. Aaron Hawkins and the IMMERSE undergraduate research program. He introduced me to the world of college research and it helped me discover the field of engineering I enjoy.

Thanks to all my colleagues in the BYU Configurable Computing Lab, past and present. They have been there to help me with random questions and problems along the way, and my work relied on tools developed by the CCL lab.

I also thank my brother Ben for helping me find spelling and grammar mistakes in this paper. I wish him success as he continues studying English and editing.

I am grateful for all assistance from my family, committee members, and friends. Many have been great examples to me, especially my father who works hard to provide and care for his family.

Finally, and most importantly, I thank my wife Katie for her love and support. She has encouraged me in this work even though it often meant spending long days and evenings without me. She is a great mother to our children and an inspiration to me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

API	Application programming interface
ASIC	Application-specific integrated circuit
BRAM	Block random access memory
CAD	Computer-aided design
CLB	Configurable logic block
DSP	Digital signal processing
ELA	Embedded logic analyzer
FIFO	First in, first out
FPGA	Field-programmable gate array
HDL	Hardware description language
IC	Integrated circuit
ICAP	Internal configuration access port
ISE	Integrated software environment
JTAG	Joint test action group
LUT	Look up table
NCD	Native circuit description
NGD	Native generic database
PAR	Place and route
PIP	Programmable interconnect point
RAM	Random-access memory
RTL	Register-transfer level
SRL	Shift register LUT
UCF	User constraint file
XDL	Xilinx design language

CHAPTER 1. INTRODUCTION

1.1 Motivation

Integrated circuits (ICs) have revolutionized the world of electronics. Today they are used in virtually all electronic equipment. Computers, mobile phones, and other digital home appliances are wide spread and an important part of modern societies and use ICs. It is critical that ICs function correctly or there can be costly or even deadly consequences. For example, the Pentium FDIV bug purportedly cost Intel $475 million to replace the flawed ICs as announced in 1995 [1]. Several people died due to flaws in the Therac-25, a radiation therapy machine. One of the Therac-25's problems was that it lacked hardware interlocks that had been present on past models of the machine [2]. These incidents highlight the importance of performing proper verification of ICs.

Verification is an important part of IC product development and FPGAs are increasingly being used to do it. A worldwide study and survey commissioned by Mentor Graphics found that over half of all designer effort is spent performing functional verification [3]. The study also found that in 2010, 55% of the industry used Field-programmable gate array (FPGA) prototyping for verification, an increase from 41% in 2007. FPGAs are an attractive platform for verification and debug because they are much faster than simulation and cost less than fabricating an ASIC prototype. IBM engineers reported that full chip-level testing using a multi-FPGA prototype is 100,000 times faster than software simulation [4]. With such a large difference in speed, designers can perform tests on FPGAs that involve a greater number of clock cycles than they could test with simulation. An ASIC prototype could achieve even greater speeds than an FPGA, but FPGAs have other advantages over ASIC prototypes. The lead time of fabricating an ASIC can be weeks or even months and can easily cost over 1 million USD [5]. An FPGA can be available immediately and costs much less than an ASIC, so many designers may find that the advantages of performing verification on an FPGA outweigh the disadvantages.

The major disadvantage of using FPGAs for verification or debug is observability, the ability to see FPGAs' internal values. Observability is key to verifying behavior and tracking down the cause of bugs. Simulators can provide full observability into all signals of a circuit, but on FPGAs and other physical prototypes only a small subset of signals can be observed through the external pins. A common solution to this problem is to record a larger subset of signals in embedded logic analyzers (ELAs) that are added to the FPGA design. ELAs provide more observability than a standard logic analyzer which can typically only observe the external pins.

ELAs improve observability but have several disadvantages. First, ELAs use resources of the FPGA. This could be a problem if there are not enough resources for the ELA. Second, the ELAs may influence the placement, routing, and timing of the circuit being instrumented. In a typical design flow the ELAs are placed and routed by the same processes as the rest of the circuit, and it was shown in [6] that this changes the results and may change the timing of paths in the circuit. The minimum period of the circuit may change if paths have greater delay than before. Finally, making changes to the ELAs, such as the signals they are observing, often requires a recompilation of the design in typical flows. Recompilation can take hours for large designs which is prohibitive of making frequent changes to the ELAs.

1.2 Preview of Approach

The purposes of this work is to present and demonstrate the feasibility of a new trace-based approach for improving FPGA observability that overcomes existing disadvantages of trace-based approaches by using incremental techniques. Here we give a brief preview of the approach which will be described in further detail in later chapters.

Similar to typical ELAs, our approach uses trace buffers and a trigger unit. Trace buffers are formed from a memory resource on the FPGA. Trace buffers record a limited-size history of the signals connected to them during regular device operation. This enables designers to run the device normally and extract the signal history observed by the trace buffers with techniques like device readback [7] for off-line analysis. Logic on the chip is used as a trigger to halt the trace buffers' recording and notify designers that the data should be extracted. The trigger allows the trace buffers to continuously record data until certain conditions are met. The trigger conditions can be determined by the designer, such as waiting for a subset of signals to be equal to some

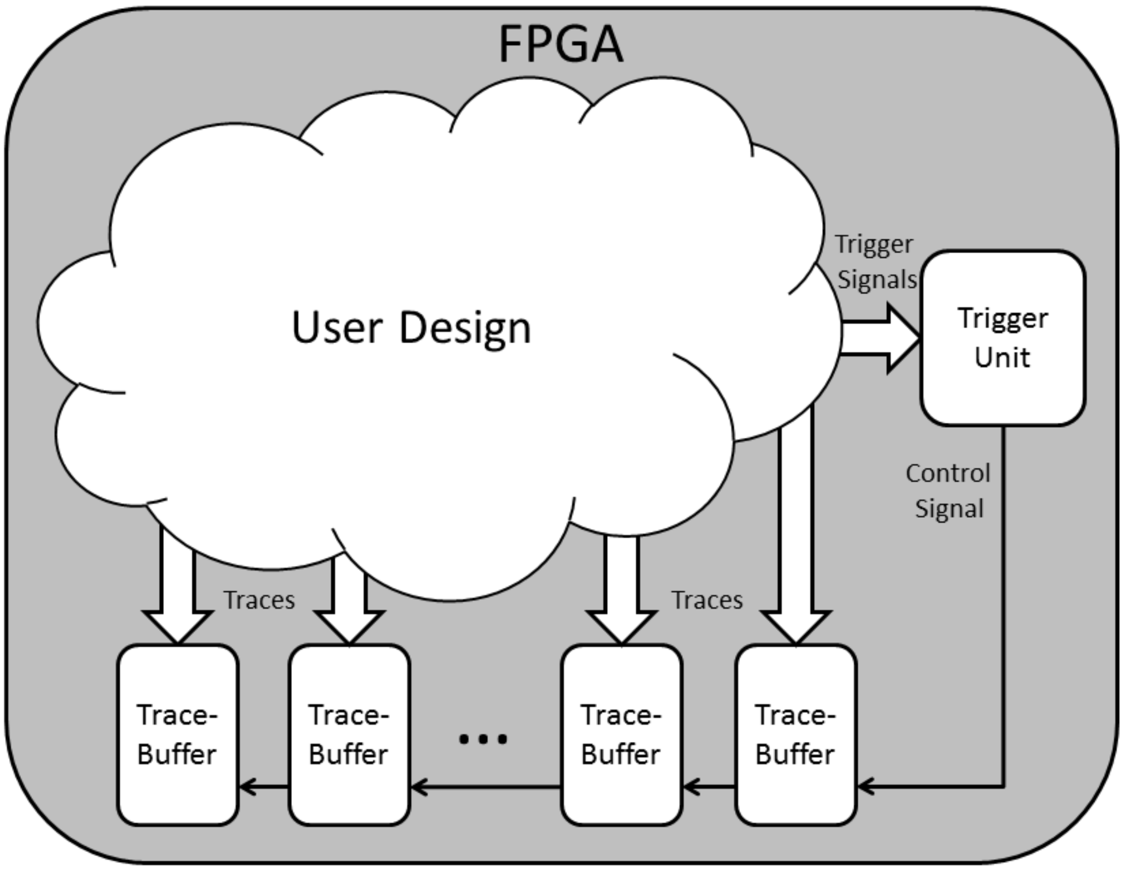

Figure 1.1: Block diagram of the proposed debug system.

value or for some signal to change within a certain interval of another signal. Designers verify functionality or hunt bugs by properly adjusting the trigger conditions and examining trace buffer data.

Figure 1.1 illustrates our proposed approach. In our approach, multiple trace buffers are distributed to observe and record nearby user signals. Distributing the trace buffers reduces the distance signals must be routed and improves circuit timing. Including more trace buffers will allow more signals to be observed. A centralized trigger unit controls the operation of all the trace buffers with a single output signal that halts their recording when necessary. The trigger unit requires a region of logic to detect conditions. We try to find a large enough region of logic as close to the center of the design as possible to improving timing performance. However, for the trigger unit size we tested, the trigger unit typically had to be placed on an edge of the user design to find enough unused logic resources. The resources that the user design requires are known because this entire system is inserted incrementally after the user design has already been placed and routed.

In this work, a Xilinx Virtex-5 FPGA [8] is used to demonstrate the approach and collect all data. We perform incremental insertion in the Virtex-5 with RapidSmith [9], an open-source set of tools and APIs that can incrementally modify Xilinx FPGAs by editing XDL files. The RapidSmith tools and the implementation details of the trace buffers and trigger presented in this thesis are specific to Xilinx FPGAs. The results and challenges of using the approach may differ between architectures. However, the approach could be used on any FPGA where it is possible to incrementally insert (place and route) additional logic and memory components in an already existing circuit.

Our proposed approach has several predicted advantages:

- It has no impact on the placement or routing of the user circuit.
- It requires no area from the perspective of the user circuit.
- It has no impact on timing if the delay of the inserted paths is less than that of the critical paths in the user circuit.
- It enables faster turn-around time for changing the observed signals or modifying the trigger unit or trace buffers compared to traditional flows.
- It increases FPGA observability by taking full advantage of all leftover trace buffer capacity.

These advantages are possible because the debug system is inserted after the user circuit has been placed and routed. We restrict our approach to only using FPGA resources left over by the user circuit so that placement, routing, and area of the original circuit is not affected. If the designer wants to make changes to the debug system they only have to rerun trace insertion and the steps that follow.

1.3 Contributions

The approach described in this thesis has two unique characteristics: (1) it is completely incremental and (2) a centralized trigger unit controls all the trace buffers. No other approach was found that possesses both of these characteristics. We show that most of the approach's advantages, discussed in the previous section, do hold. Implementation details and challenges of using the approach on a Xilinx Virtex-5 FPGA are described. We demonstrate how to create a trace buffer from a single block RAM in the Virtex-5, and show what control circuitry is required. The tools developed to do this work are the first to demonstrate the capability in a commercial FPGA to reclaim all unused memory blocks for use as trace buffers and to incrementally insert a centralized trigger unit to control all of them. We also contribute the answers to the following research questions:

- What percentage of user signals can be observed with this approach?
- How long does it take to change the observable set of signals?
- What is the timing impact of inserting a centralized trigger unit and distributed trace buffers in an existing circuit?
- How does the routing problem differ between the trace buffers and trigger unit?

Our results show that 100% of the flip-flops in a Virtex-5 circuit can be observed if there is enough trace buffer capacity. We chose to observe flip-flops because, with a knowledge of the circuit, the values of other nets could be calculated based on the values of the flip-flops. The time it takes to insert the trigger unit and trace buffers or change the observed signals is a few minutes at most. This is a faster turn-around time than recompiling the whole design which can take hours. The approach does significantly impact the minimum period of some designs, but has no impact on designs with a sufficiently large period. It also shows that placing and routing the trigger unit presents different challenges than the trace buffers. These challenges can be overcome even in circuits that utilize a large percentage of FPGA resources.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses background information and related work. Chapter 3 reveals the implementation details of the trace buffer and trigger unit. Chapter 4 presents the process for inserting the trace buffers and trigger unit into a placed and routed FPGA design. Chapter 5 presents the primary results of my approach such as runtime and impact on timing. Chapter 6 explores different design and algorithm options that influence the results. Chapter 7 presents the effects of using pipelining in an effort to improve timing. Chapter 8 concludes the thesis.

CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter describes the Xilinx Virtex-5 FPGA architecture, Xilinx design flow, and some methods of incrementally modifying Xilinx FPGA designs. These topics will help readers better understand the contributions of later chapters. Related work in FPGA debug and commercial debug tools are also described, especially those using incremental methods.

2.1 FPGA Architecture

Field-programmable gate arrays (FPGAs) are integrated circuits fabricated to be configured by a designer after manufacturing. FPGAs are composed of reconfigurable logic, memory, and routing interconnect that can be configured to perform a huge variety of tasks. FPGAs can be reconfigured an unlimited number of times such that the same FPGA can be used for one task and later be reprogrammed for a different task. They are different from application-specific integrated circuits (ASICs) which are typically hard-wired for one task. However, the flexibility of FPGAs comes with some drawbacks. An ASIC consumes less area and power and operates faster than the equivalent circuit would on an FPGA.

The specifics of a FPGA's architecture vary among vendors and product families. Many modern FPGA architectures include more than just logic, memory, and routing. Other fixed functional units are added such as DSP and block memories to boost performance or add new capabilities. This work targets the Xilinx Virtex-5 architecture and this section describes the aspects of that architecture that are important for our debug system.

2.1.1 Virtex-5 Overview

The FPGA architecture used for this study is the Xilinx Virtex-5. The Virtex-5 architecture was chosen because it was the newest architecture fully supported by the RapidSmith CAD tools used to perform routing [9]. We do not believe our results would change significantly with newer

Virtex architectures because many of the features we use have remained the same in the newer architectures such as six input look-up tables (LUTs) and 36 kilobit block RAMs (BRAMs).

The Virtex-5 is an island-style FPGA arranged in a two dimensional grid of tiles. There are multiple types of tiles in the grid. Besides their location, tiles of the same type are typically identical. The types of tiles that are important for our work are CLB, BRAM, and interconnect tiles. Each of these types of tiles is described in more detail in the sections that follow this one.

An important feature of the Virtex-5 for our work is its support of readback [7]. Readback allows users to read the current state of the FPGA's memory through JTAG, SelectMAP, or ICAP interfaces. This is done by sending a sequence of commands to the FPGA, and the FPGA will respond by dumping the contents of memory to the interface. There are two types of readback: Readback Verify and Readback Capture. Readback Verify allows the user to read the current values of all block RAM, SRL16, and LUT RAM instances in the device. Readback Capture is a superset of Readback Verify and allows the current state of the CLB and IOB registers to be read in addition to the memory elements read by Readback Verify.

2.1.2 CLB Tiles

Configurable logic blocks (CLBs) contain some of the most commonly used elements of the FPGA: look up tables (LUTs) and flip-flops. Each CLB occupies a single tile of the FPGA and is divided into two entities called slices. Each slice contains 4 LUTs, 4 storage elements, and a few other gates useful for specific functions. LUTs are the basic units of the FPGA's programmable logic. The Virtex-5 LUTs have 6 available inputs and can implement any 6-input logic function. The storage elements can be configured as flip-flops or latches, but we shall commonly refer to them as flip-flops. The output of each flip-flop is directly connected to an output pin of the slice. Most of the output and input pins of the slice are connected to an interconnect tile that is located adjacent to the CLB tile. In Chapter 3 we describe how slices are used to create a trigger unit.

2.1.3 Interconnect Tiles

Interconnect tiles are the primary location for routing resources and each is paired with a non-interconnect tile. In Xilinx FPGAs, wire segments can be connected together by programmable interconnect points (PIPs). Each interconnect tile contains hundreds of PIPs for connecting different wires. The wires span Manhattan distances ranging from 2 to 18 interconnect tiles away. Each interconnect tile also has connections to the tile it is paired with. The paired tile uses the interconnect tile to connect to resources in other tiles. For example, by properly programming the PIPs the output of the flip-flop can be connected to another FPGA resource many tiles away or it could simply be connected to one of the input pins of its own tile. Signals that must be connected to a distant resource may have to pass through multiple interconnect tiles to reach the destination.

2.1.4 BRAM Tiles

The Virtex-5 contains block random access memories (BRAMs) that can store up to 36K bits of data. Each BRAM actually occupies five Virtex-5 tiles and has five interconnect tiles associated with it for routing. The BRAMs support different configurations that offer different trade-offs of data width and depth. A wider configuration will not be able to store as many entries. Read and write operations on the BRAM require just one clock edge. The BRAMs are dual-port and both ports can be used to read or write simultaneously. The BRAM can also operate in a simple dual-port mode where one port is read-only and the other is write-only and both can still be used simultaneously.

The BRAMs have built-in FIFO support so that a single BRAM can be used to implement a FIFO [8]. Dedicated logic in the BRAM eliminates the need for other FPGA logic to track the read address, write address, and status of a FIFO. The only control signals that must be generated external to the FIFO are the read and write enable. The FIFO has full, almost full, almost empty, and empty status signals. The offsets of what to consider almost full and almost empty are parameterizable. The FIFOs can also be parameterized for different widths (number of data input bits) and depths (number of entries in the FIFO). 72 bits by 512 entries is the widest configuration and others include 36 x 1024, 18 x 2048, 9 x 4096, and 4 x 8192. The Virtex-5 FIFOs cannot be written to when they are full or read when they are empty. Attempting to do so will cause an error signal to be asserted and the contents of the FIFO will not be changed. These features of the BRAM are important for our trace buffer implementation.

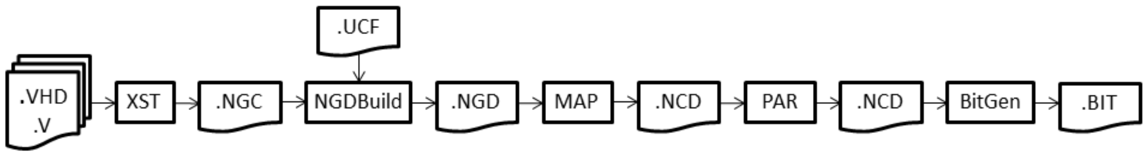

Figure 2.1: Xilinx design flow.

2.2 Xilinx Design Flow

The Xilinx design flow plays an important part in our study. We use it to compile circuits for the Virtex-5 FPGA. We also modify the flow so that we may insert our debug system after the original circuit has been placed and routed. In this section we describe each of the steps in the Xilinx design flow. Our modifications to this flow shall be described in Chapter 4.

2.2.1 Design Entry and Synthesis

Typically, a circuit to be implemented on a FPGA is initially described by the designer in a HDL such as Verilog or VHDL. When the HDL description is ready it is synthesized into a netlist form. A netlist is a collection of basic elements of logic (BELs) and a list of connections (nets) between them. When using Xilinx tools, synthesis is usually performed by Xilinx Synthesis Technology (XST) which will create the netlist in a proprietary format with the extension NGC. However, the netlist may also be synthesized using alternative means into an EDIF file and subsequent steps of the design flow will accept EDIF format as well as NGC.

2.2.2 NGDBuild

NGDBuild begins the process of technology mapping the netlist to a specific FPGA architecture. In this step, the BELs of the netlist are converted to a logical description of the design made up of Xilinx primitives. The result is output in the form of a Xilinx Native Generic Database (NGD) file. NGDBuild also adds user constraints to the design. User constraints are input to NGDBuild as a user constraint file (UCF). The UCF allow designers to specify minimum clock period, locations of I/O pads, area constraints, and other constraints.

2.2.3 MAP

Xilinx technology mapping is completed by MAP. MAP uses the NGD file from the previous step to map the logical description of the circuit to components in the target Xilinx FPGA. When MAP completes the design exists as a netlist of Xilinx primitives such as slices, BRAMs, and IOBs. For the Xilinx Virtex-5 architecture the MAP step also places the components of the design. The placer will assign each component a physical site on the device while respecting any user constraints. The placement is optimized to achieve better timing results. The result of MAP is a native circuit description (NCD) file, a fully technology mapped and placed netlist of the design.

2.2.4 PAR

After MAP places a circuit, the next step of the design flow is to route the circuit using PAR. Routing assigns the nets of the design to physical wires in the FPGA. Routing is optimized to meet timing constraints. Similar to MAP, the output of PAR is a NCD file, but this one has been updated with all the routing information.

2.2.5 BitGen

The bit generation (BitGen) process converts a placed and routed NCD file into a BIT file that can be used to configure the FPGA. After this step is complete the user may load the BIT file into the appropriate FPGA to form the circuit. Designers can now test the circuit on the FPGA to verify it functions correctly using techniques like the ones described in this paper.

2.3 Incremental Synthesis

The goal of incremental synthesis is to modify the functionality of an existing circuit with minimal changes to its current placement and routing [10]. In this section we focus on how a Xilinx design may be incrementally modified with XDL and RapidSmith. The tools and techniques described here are used to incrementally insert our debug system.

2.3.1 XDL

The Xilinx Design Language (XDL) is a human-readable ASCII format description of a circuit. Xilinx provides command line tools to convert proprietary Xilinx NCD files into XDL files and vice versa. A circuit may be modified by editing the XDL file. The edited XDL file can be converted back into an NCD file which can be used in the Xilinx tool flow. NCD files are output after several stages of the Xilinx tool flow: map, place, and route. It is possible to convert the NCD files from any of these stages to an XDL file so that incremental changes may be made. Placement, routing, and parameters like FIFO width can all be modified in the XDL file [11].

2.3.2 RapidSmith

RapidSmith is an open-source set of tools and APIs that enable CAD tool creation for Xilinx FPGAs [9]. RapidSmith is written in Java and allows designers to write Java code for their own placers and routers as well. RapidSmith performs placement and routing through modifications to XDL files. To demonstrate our method we created a placer and router that uses the RapidSmith API. This placer and router was created for the specific purpose of incrementally inserting trace buffers and a trigger unit after PAR.

2.4 Related Work

A major challenge of debugging circuits on FPGAs is observability. A limited number of signals may be observed on an FPGA's pins using an external logic analyzer but this is often inadequate and some of the pins may be used already. Methods to observe a larger number of signals can be divided into two broad categories: scan-based and trace-based. The method proposed in this paper is trace-based and incremental so that will be the focus of this section, but first scan-based debug methods shall be described.

Scan-based debug approaches capture the state of an FPGA for inspection by serially shifting it out over an external pin or an interface such as JTAG. The state of the FPGA is the values in all memory elements on the chip, such as flip-flops and embedded memory blocks. Some FPGAs have built in support for serially shifting out the FPGA's state. For example, this is possible in the Xilinx Virtex-5 FPGAs used in this paper via the readback feature [7]. In [12] it was shown how readback data can be used for debugging in a combined simulation/hardware execution environment built on JHDL [13]. If an FPGA does not have built in support it may be added by wiring up the memory elements in such a way that their data can be serially shifted out when a control signal is asserted. However, [14] showed that the average overhead for full scan is 84% additional area. A challenge with scan-based approaches is that as devices increase in size and density the time required to shift the entire FPGA state out proportionally increases [15]. Scan-based approaches cannot observe the value of all state elements every clock cycle unless the circuit is halted each clock cycle, which would significantly increase operation time.

In a typical trace-based approach a designer pre-inserts trace buffers controlled by a trigger, also called embedded logic analyzers, into the circuit before compilation. The signals to trace must also typically be selected before compilation. The trace buffers allow a window of the history of the chosen signals to be recorded as the circuit operates in real-time. For devices with readback capability the data in the trace buffers can be extracted using readback for off-line analysis. Trace-based debug does have the disadvantage of requiring FPGA resources which can influence the placement and routing of the circuit [6] and limit the number of signals that can be observed. Another disadvantage is that the circuit may have go through a time-consuming recompilation if the designer wishes to change the signals being observed or parameters of the trace buffers or trigger.

This paper proposes an incremental trace-based debug approach that enhances the observability of FPGA signals. Incremental synthesis allows us to overcome some of the disadvantages of trace-based approaches such as influencing the placing and routing of the original circuit. Instrumenting the circuit with incremental synthesis also allows us to avoid influencing the placement and routing of the original circuit because instrumentation is not included in the compilation of the original circuit. The goal of incremental synthesis is to modify the functionality of a placed-and-routed circuit while preserving as much of the original solution as possible. FPGAs are well suited for incremental synthesis because there is often unused logic and routing resources leftover after a circuit has been compiled. Another disadvantage we can avoid with incremental synthesis is recompilations of the entire circuit whenever changes are made to the instrumentation. Figure 2.2 demonstrates how incremental synthesis allows us to avoid a full compile of the circuit when adding or making changes to the instrumentation of a circuit. Incremental compile offers

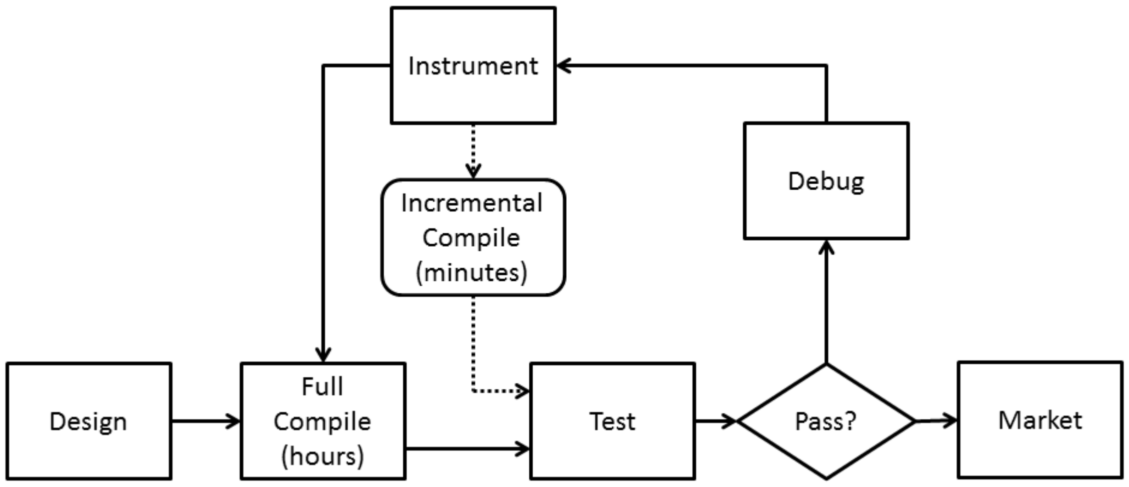

Figure 2.2: Design and debug flow demonstrating how incremental changes reduce time between debug iterations.

an alternate path out of the instrument step, and allows designers to more quickly test the circuit again.

Graham et al. [16] demonstrated the use of incremental techniques to change signals being observed without recompiling the entire design. They put unconnected embedded logic analyzers in the FPGA prior to placing and routing, and afterwards used low-level bitstream modification to connect them to the desired signals. This reduces turnaround time between debug iterations so that designers may more quickly observe different sets of signals. Graham et al. also anticipated that an updated version of their techniques could insert the entire trace buffer into the circuit with bitstream modifications. The envisioned updated techniques would be similar in concept to those in this thesis. However, their techniques relied upon the JBits API that was provided by Xilinx for Virtex-II FPGAs, but similar APIs have not been provided for other commercial FPGAs. Also, the centralized trigger unit system used in this work is different from ELAs used by Graham et al. which each had their own trigger unit. Graham's system may not scale well for observing thousands of nets, and only 128 nets were observed in their tests.

Poulos et al. [17] also used bitstream modifications to raise debug productivity. They connected signals that the designer might want to observe to multiplexer (mux) by changing the design prior to synthesis. The muxes connected the signals to FPGA pins that could be observed by an external logic analyzer. If the designer wanted to change the signals that were being observed a bitstream modification could change what is passed through the muxes. This approach does not use trace buffers like us, nor does it avoid influencing the placing and routing of the original circuit. The work of Poulus et al. has some similarities to the work done later by Hung and Wilton in [18] which did use trace buffers. At compile time, Hung and Wilton embedded an overlay network which multiplexes almost all signals of a circuit into trace buffers. The muxes in this network were formed of unused routing muxes within the FPGA fabric rather than logic resources. At debug time, the network would be configured by setting a small number of routing bits to select signals to observe. The trigger to control the trace buffers was assumed to be specified by the designer manually or driven from an external pin, so incrementally inserting the trigger was not explored as it is in this paper.

Two unique characteristics of incremental trace buffer insertion were identified by Hung and Wilton [6, 19]: (1) the trace buffers only observe and do not modify the functionality of the original circuit and (2) a trace can be observed at any trace buffer. The first characteristic is important because it means trace buffers can be inserted without changing any placement or routing of the original circuit. The trigger unit also has this characteristic because it only influences the functionality of the trace buffers and nothing in the original circuit. The second characteristic indicates that routing traces to trace buffers has a many-to-many flexibility. The many traces can be routed to any of the many free trace buffer input pins, so for each trace all trace buffer input pins are potential sinks. As long as each trace is connected to a trace buffer it shall be observed and recorded. However, this characteristic does not hold for the trigger unit because, in addition to observing signals, it is watching for these signals to meet certain conditions. We assume to detect these conditions the trigger unit requires each input signal to be routed to a specific pin/port so that it is connected to the proper logic. This assumption is a worst case scenario because some comparisons do not require a specific order for the logic, e.g. equality comparisons, but arithmetic comparisons do have this requirement. Thus, each trigger signal must be routed to a specific pin but trace buffer inputs may be routed to any appropriate pin.

This paper expands on and demonstrates an application of the work done by Hung and Wilton in [6, 19], mentioned in the previous paragraph. Hung and Wilton used VPR, an open source CAD tool that is part of the VTR project [20], to incrementally insert trace buffers into a custom FPGA architecture based upon the Altera Stratix IV. They found that postmap insertion was

98 times faster than premap trace-insertion when reclaiming 75% of the leftover memory capacity as trace buffers. In addition, postmap insertion had a less than 1% effect of the critical-path delay of the circuit. They hypothesized that there was no reason why their techniques would not work on commercial FPGA devices but could not verify this since they did not use a commercial FPGA. In this paper we demonstrate the feasibility of Hung and Wilton's incremental trace buffer insertion techniques on a commercial Xilinx Virtex-5 FPGA. We use the Xilinx ISE tools to map the original circuit onto the Virtex-5 FPGA and perform postmap trace-insertion by editing the output of these commercial tools before it is used to generate a bitstream. We also expand upon their techniques by incrementally inserting a complete debug system that includes a centralized trigger unit to control the trace buffers.

2.5 Commercial Debug Tools

Commercial debug tools support varying degrees of incremental instrumentation. All seem to support changing trigger conditions without requiring a full recompile. Beyond this, support for incremental changes is more varied. Some use other techniques besides incremental changes to avoid full recompiles. This section describes the techniques used by Tektronix Certus, Synopsys Identify, Xilinx Chipscope Pro, and Altera SignalTap II.

Tektronix Certus improves observability into multi-FPGA prototyping platforms at the RTL-level but has little support for incremental features [21]. Instead, Certus avoids the need to recompile by instrumenting such a large number of signals that it is unlikely you will need to modify them. Certus also automatically identifies influential nets that should be observed. Like Certus, our results show that our proposed system can instrument a large number of signals in some benchmarks but it is all inserted incrementally.

Synopsys Identify allows hardware to be debugged at the RTL-level [22]. Instrumentation is added into the circuit prior to compilation. Identify allows users to change the signals being probed without a full recompile. Our proposed system does not require anything to be added to the circuit before compilation like Identify does.

Xilinx Chipscope Pro requires that cores be inserted into the design before the implementation stage [23]. Some parameters of the cores can be changed incrementally. Trigger and data signals can be changed incrementally using FPGA Editor. Xilinx also supports partial reconfiguration via partitions that may allow the designer to avoid a full recompile but there are some limitations to using it in conjunction with Chipscope Pro [24]. Thus, Chipscope Pro has a lot of incremental capabilities but they may require extra design effort such as running a separate program or setting up partitions.

Altera SignalTap II has the most support for incremental changes. SignalTap II uses partitioning to allow designers to add it to the design or make change to its settings [25]. The partitions make so only portions of the design have to be recompiled rather than the entire design. This is similar to partial reconfiguration of Xilinx designs, but seems to be more integrated into the SignalTap II flow or at least better documented. Our proposed system does not require partitioning to achieve its results like SignalTap II and Chipscope Pro.

In conclusion, our debug system is unique among both past academic and commercial systems. Many do not include the capability to incrementally insert or alter the entire debug system without influencing the original circuit. Those that have included or envisioned this capability did not include a centralized trigger unit or investigate its impact. Also, some past academic work did not use a commercial FPGA for their data as we do.

CHAPTER 3. TRACE BUFFER AND TRIGGER IMPLEMENTATION

This chapter explains the two types of components that are used to create our proposed debug system: trace buffers and trigger units. The proposed system has multiple trace buffers and one trigger unit to control them and was shown earlier in Figure 1.1. This chapter focuses on describing the purpose and desired behavior of these components. It also discusses how the desired behavior is achieved in the Xilinx Virtex-5 architecture. The implementation details also reveal how the system as a whole works.

3.1 Trace Buffer Implementation

The purpose of the trace buffers is to observe signals in the FPGA and record a history of their values. Designers can access the history recorded in the trace buffers by halting the operation of the FPGA and reading back the trace buffer's data. The recorded history assists designers in debugging the circuit they have implemented on the FPGA. It allows them to identify what took place in the circuit and verify that it matches the specified functionality of the circuit.

The desired behavior for a trace buffer is that during each clock cycle the current values of the traces connected to its data input pins are stored and if the trace buffer is full the oldest value is removed. The trace buffer should do this continuously until it is halted by the trigger unit or the entire FPGA is halted. This behavior is identical to the behavior of a queue or FIFO that ejects the oldest entry when it is full and there is a new entry. As described in Chapter 2, the BRAMs in Virtex-5 FPGAs have built in support for FIFO behavior. However, the behavior of the FIFO does not match the desired trace buffer behavior. A Virtex-5 FIFO will not allow a new value to be written if the FIFO is full. It does not automatically remove the oldest entry when there is a new entry. The oldest entry may be removed by performing a read of the FIFO. Thus, some small adjustments are necessary to allow the FIFO to operate as a trace buffer that is continuously written each clock cycle.

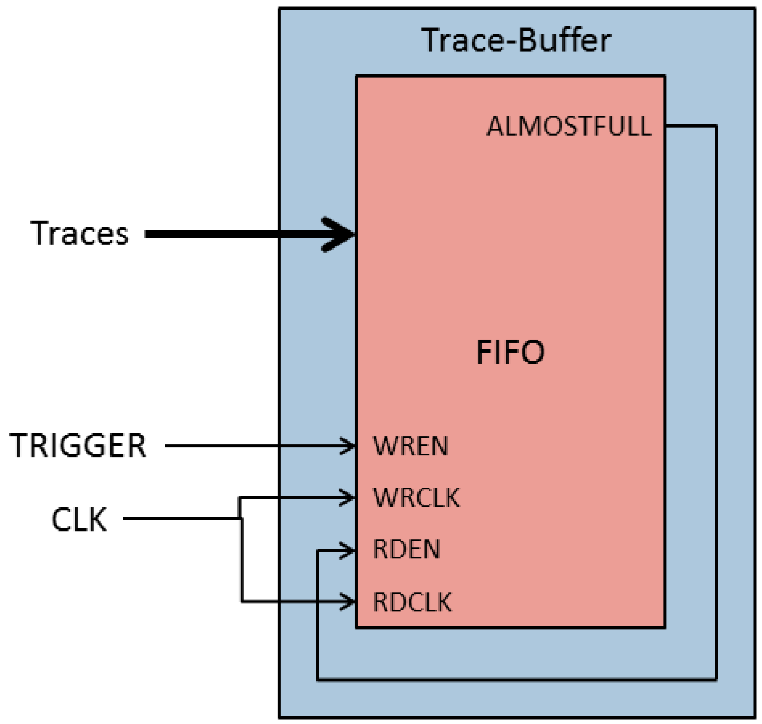

Figure 3.1: Trace buffer diagram

Figure 3.1 shows how a trace buffer can be implemented from a single Virtex-5 FIFO. Due to the widths available for Virtex-5 FIFOs, a trace buffer can have up to 72 traces, but we typically use 36 for reasons explained in Chapter 6. The FIFO's almost full signal is connected to its read enable pin. The almost full signal warns when the FIFO is almost out of space. Routing this signal to the FIFO's read enable pin makes it so each clock cycle the oldest value will be removed from the FIFO as long as it remains almost full. The FIFO can be read and written at the same time so the trace buffer can continuously write signal values as we desire. Using the almost full signal simplifies implementation because it eliminates the need for additional logic to control the read enable pin. The FIFO also requires some signals be tied off to VCC and GND but this is easily accomplished because there are many VCC and GND tieoffs available throughout Virtex-5 FPGAs.

The trigger unit controls and halts the trace buffer via the write enable pin. The trigger unit holds write enable high to allow the trace buffer to record signal values. It halts the trace buffer by driving write enable low. This will halt trace buffer writes. Reads will also stop when the FIFO is no longer almost full. We configured almost full to alert us when there is only one free entry remaining in the trace buffer. Thus, the reads will halt as soon as there are two free entries in the trace buffer. The FIFO never reaches full capacity and blocks a write because reads and writes happen at the same rate until it is halted by the trigger. The downside is that the trace buffer cannot use its full capacity but this only results in the loss of two units of depth. This is the cost of using the almost full signal to control reads, but it is worth the cost because of how much it simplifies implementation.

3.2 Trigger Implementation

In this work we investigate the use of a single global trigger unit to control all the trace buffers. This is done via a single output signal from the trigger unit that connects to the write enable pin of each trace buffer as was shown in Figure 3.1. The advantage of this approach is that it simplifies the incremental insertion and is easily scaled to any number of trace buffers. The only influence adding more trace buffers has on the trigger unit is more fanout on the output signal. The area of the trigger logic is completely independent from the number of trace buffers.

We make several assumptions of what the trigger unit will require to fulfill its purpose while also achieving good timing performance. First, we assume the trigger condition will be based on some of the circuit's user signals. Thus, some number of user signals will be inputs to the trigger unit. We assume the inputs will be connected to flip-flops to improve performance. The flip-flops will then drive logic gates that have have been arranged to detect the trigger condition, and some flip-flops may be required to improve performance or hold state values for the logic. The logic will control the output value of the trigger unit which we also assume to be driven by a flip-flop for better performance. Everything we need to implement the trigger unit is available in the slices of the Virtex 5. The slices contain LUTs that can perform the logic and they contain flip-flops. Finally, we assume that all the slices used to implement the trigger unit will be located in the same region of the FPGA as each other. Locating the slices close to each other will improve timing performance and make it simpler to route the internal signals of the trigger unit. This final assumption means that our trigger unit will be centralized rather than distributed throughout the FPGA like the trace buffers.

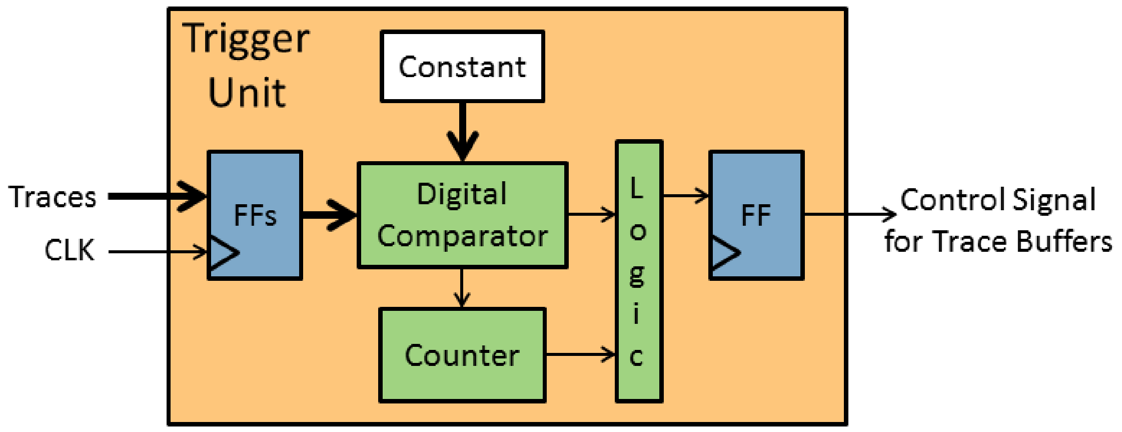

Figure 3.2: Trigger unit diagram

A sample trigger unit was created that fits all of our assumptions as shown in Figure 3.2. The sample trigger unit checks if the value of its inputs matches a parameterizable constant. Each of the inputs is connected to a flip-flop. It also contains a counter to allow a configurable delay between detecting the equality and halting the trace buffers. The counter allows a designer to configure how many clock cycles the trace buffers will continue to record after the equality is detected. Finally, some logic that is monitoring the counter and comparator controls the value of the flip-flop that drives the output signal that is connected to all the trace buffers. The sample trigger unit was compiled with Xilinx ISE 14.4 and used 91 slices on the Virtex-5 when the number of input traces was 256, which we refer to as the input width. The number of slices the sample trigger unit required is used later in the paper to determine how many slices to reserve for a trigger unit.

We do not wish to restrict our results to a single trigger unit, so we use a placeholder that could represent any number of trigger units. Formulating a single trigger unit that would work in all situations and designs is beyond the scope of this work, and may even be undesirable because one of the advantages of FPGAs is that they can be reprogrammed for any trigger event [26]. In our tests it does not matter if the trigger unit is functional or not. The important thing is that we account for the area and timing impact caused by placing the trigger unit and routing signals to it. The area impact is determined based off the area of the sample trigger unit described in the previous paragraph. An area of unused slices is found in the already placed-and-routed user circuit and the area is reserved for the placement of the trigger unit. The timing impact is investigated by routing the trigger unit's inputs and output. Routing the internal signals is not necessary because the external signals have much greater path length and delay. The inputs to the trigger unit are routed to flip-flops within the reserved area. Likewise, the output signal originates from a flip-flop source pin in the reserved area. The circuit's clock signal is also routed to the flip-flops, so that timing results can be calculated using the Xilinx TRACE tool. The placeholder trigger unit accounts for area and timing, and avoids implementing any specific internal details which will vary according to designer needs.

CHAPTER 4. INCREMENTAL TRACE INSERTION

This chapter describes our proposed method for increasing FPGA observability to simplify debug and verification. The previous chapter described how the components necessary to form our debug system are implemented. In this chapter we shall describe how those components are incrementally inserted into a design. First, an overview of our proposed method is given that describes how it fits into the Xilinx design flow. Then the two major steps of incrementally synthesizing the debug system are described: placement and routing.

4.1 Trace Insertion

To increase the observability of FPGA circuits, we propose trace buffers and a trigger unit be inserted incrementally into already placed-and-routed designs. We shall refer to the already placed-and-routed design as the original circuit or user circuit and the trace buffers and trigger that are incrementally inserted as the debug system. Nets from the the original circuit shall be incrementally connected to the trace buffers to be observed and recorded. The trigger unit shall control the trace buffers and allow them to record until trigger conditions specified by the designer are met. We propose incrementally inserting the debug system because it will reduce the impact on the original circuit's area, placement, routing, and timing.

From the perspective of the original circuit, the debug system has no area overhead. The original circuit will already be placed-and-routed and thus will already have claimed whatever area of the FPGA it needs. The debug system is inserted into whatever FPGA area has been left unused by the original circuit. Thus, the debug system has an area but it is area that is of no consequence to the original circuit because it did not need it. If the debug system were not incrementally inserted then it would increase the area of the original circuit. Incremental insertion allows us to adjust the size of the debug system to fit into whatever the original circuit does not use. The amount of trace buffers and trigger logic we can insert will be influenced by the area of the original circuit. If the original circuit used a lot of BRAMs then there will be less available for trace buffers. Likewise, if the original circuit utilizes a high percentage of slices then we may be limited in the type of trigger unit we can insert. The debug system does not influence the area of the original circuit but the reverse is not true; the original circuit influences the area of the debug system.

The goal of incremental synthesis is generally to modify the functionality of an existing circuit with minimal changes to its current placement and routing. Our goal is different than this “general-purpose incremental synthesis” because we only desire to observe signals. General-purpose incremental synthesis does not guarantee that the placement and routing of the original circuit will be preserved. However, we do make that guarantee for our method. The placement and routing of the trace buffers and trigger unit is restricted to resources unused by the existing circuit. The original circuit will be left completely intact. When the debug system is removed the circuit will be exactly the same as it was originally. This allows designers to instrument a circuit without influencing the placing or routing of the circuit.

Restricting our method to unused FPGA resources also reduces impact on timing. All the paths and timing of the original circuit are preserved. However, the minimum period may temporarily change while the debug circuitry is included. The minimum period of the circuit will change if any of the paths we add have delays greater than those of the critical paths in the original circuit. The paths we added would then become the critical paths and the minimum period would increase. The chance of this occurring increases as the minimum period of the original circuit decreases. This change in the minimum period would only apply while the debug system is included in the circuit. After the debug system is removed the circuit’s minimum period will return to its original value if inserting the system caused a change.

Figure 4.1 shows the Xilinx design flow with the addition of incremental trace insertion. The Xilinx design flow was described previously in Chapter 2. Our proposed method is inserted between the PAR and BitGen stages. When inserting instrumentation, the NCD representation of the circuit produced by the PAR process is converted to an XDL file. Trace insertion modifies the XDL file to insert the trace buffers and trigger unit and creates a new XDL that includes the modifications. This XDL file can be converted back to an NCD and the normal Xilinx flow may continue. BitGen can create a BIT file that can configure the FPGA with the circuit that includes

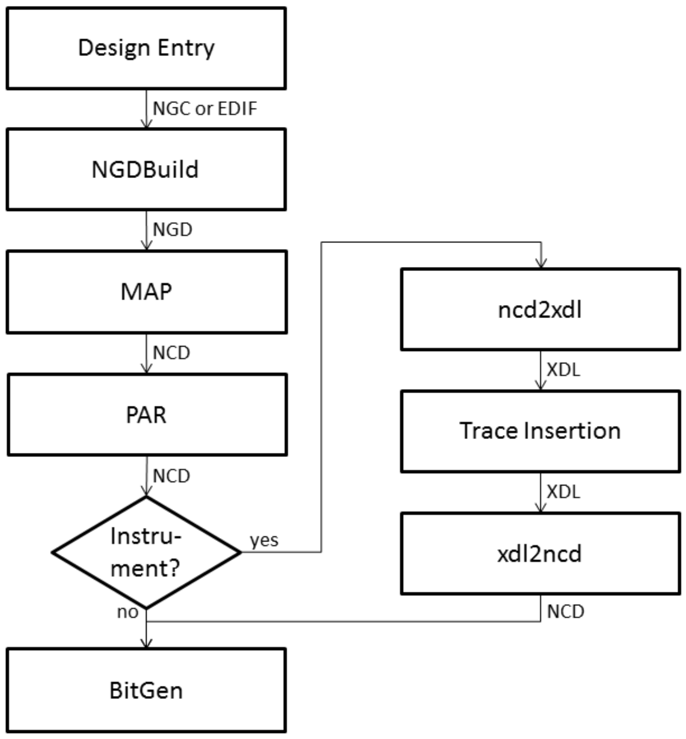

Figure 4.1: Incremental trace insertion in the Xilinx design flow.

the debug system. The ncd2xdl and the xdl2ncd conversion are done with the XDL command line tool included with Xilinx installations as described in Chapter 2.

An advantage of the incremental synthesis approach is decreased turn-around time between debug iterations. Turn-around time is reduced because the whole compilation process does not have to be rerun to make changes to the signals or logic being used for debug. This can take hours for large designs. With incremental synthesis, the signals being instrumented can be changed in minutes. Only the trace insertion, xdl2ncd, and BitGen steps need to run again. This is possible because the changes are made to the XDL file output from the ncd2xdl step. Other changes can also be made such as modifying the trigger logic or input width of the trigger unit or trace buffers. Thus, designers can more quickly perform debug iterations that observe different signals or trigger on different conditions.

The trace insertion step shown in Figure 4.1 performs the techniques that are the subject of this work. The trace buffers and trigger unit are inserted incrementally in this step for the purpose of increasing observability. As shown in the figure, an XDL file representing the circuit to instrument is one of the inputs to trace insertion. There are some additional user inputs that are not shown. Trace insertion requires two lists: a list of the nets to trace and a list of the nets to connect to the trigger unit's inputs. There are also other parameters of trace insertion that can be adjusted such as trace buffer width, trigger width, or number of trigger unit slices. The nets being traced shall be referred to as trace nets and the nets being connected to the trigger unit's inputs shall be referred to as trigger nets. The following sections describe how placement and routing are performed in trace insertion and it is shown how these steps differ for trace buffers and trigger units.

4.2 Placement

The placement step will find BRAMs and slices that the original circuit did not use and consider them for placing the trace buffers and trigger unit. The number of trace buffers to place is determined by two things: the length of the trace netlist passed to trace insertion and the number of available BRAM sites. The placer will attempt to place just enough trace buffers so that every trace net may be connected to a data input pin. If there are not enough available BRAM sites to place enough trace buffers then all the available sites will be used and nets will be trimmed off the end of the list until the number of nets is low enough to fit into the available capacity. The number of trigger nets is determined by the size of the trigger netlist. The placer must be told how many slices to reserve for the trigger unit. We do not trim the trigger netlist like the trace netlist if there is not enough capacity because if there is not enough slices to place all of the trigger unit then it will not function properly. Each of the trace buffers is independent from the others, but the slices of the trigger unit are not independent from each other and we cannot simply trim some of them. If the trigger unit could not be placed then the designer will have to create a smaller trigger unit that can fit into the available space.

After it has been determined how many trace buffers to place, the next problem is where to place them. Minimum clock period will be better if nets do not have to be routed long distances to reach a trace buffer. Thus, it is desirable that the trace buffers to be distributed through out the circuit in locations close to the sources of the nets that will be observed. To accomplish this, a random trace net is chosen for each trace buffer that needs to be placed. The trace buffer is placed in the closest available BRAM site to the source of the chosen net. This placement method ensures that the trace buffers are distributed throughout the circuit and put in locations that will be a short distance from at least one trace net.

The trigger unit requires different placement considerations than the trace buffers. An important distinction between the trigger unit and the trace buffers is that the former is centralized while the latter is distributed. Each trace buffer requires only a single BRAM site on the FPGA but the trigger unit requires multiple slices. These slices must all be in the same region of the FPGA for good performance and to simplify the routing within the trigger unit. Nets from various locations in the circuit need to be routed to the trigger unit just like the trace buffers. The trace buffers could use randomly selected nets to choose placement sites since they were distributed, but the trigger unit is a single centralized component and is unlikely to achieve good performance if its placement is based off one random net. Instead, it is desirable for the trigger unit to be placed in a location that is centrally located in comparison to the various nets that will be connected to it.

The placement algorithm of the trigger unit proceeds as follows. The center of the trigger nets is found by averaging the locations of the source of each net to be connected to the trigger unit. Available slices are then sorted by their distance from the center. Each slice is then processed in sorted order to find a region containing enough unused slices to place the trigger unit. The sorted order guarantees it will be as close to the center as possible. A breadth first search is used when processing each slice to see if enough unused slices are in the same region as it. When an appropriate region has been found for the trigger unit, each slice in the region will be configured one-by-one to form the trigger unit. As stated earlier, in our tests we only configure the flip-flops connected to the inputs and outputs of the trigger unit and do not implement the internal logic and signals. An alternative to individually configuring each slice would be to place a hard macro in the area using tools such as HMFlow [27] but this is not investigated in this paper.

A maximum congestion factor can also be specified to the placement algorithm for the trigger unit. The placer will not consider slices for trigger unit placement if the interconnect tile associated with that slice is above the specified maximum congestion. In this work, congestion is determined by the number of used routing nodes in the interconnect tile. RapidSmith uses nodes to represent routing resources of the FPGA and the connections between them are represented as edges [9]. The nodes and edges form a routing graph which routers can use to find paths between resources. An interconnect tile is highly congested if a large number of its nodes are used. The used nodes represent used routing resources that are unavailable to other nets which makes it more difficult to route through the tile. Later we show congestion can cause the router to fail to route some trigger input signals.

Examples of the placer's results may be found in Appendix A. The examples are screenshots taken from Xilinx FPGA Editor. In them the trace buffers can be seen distributed throughout the FPGA as we described. The trigger units of each example are near the edge of the original circuit where enough unused slices could be found for the trigger unit.

4.3 Routing

The routing step will route all necessary signals for the trace buffers and trigger unit to function properly. The nets that must be routed include: trace nets, trigger nets, the trigger output that controls the trace buffers, a clock for the flip-flops and trace buffers, and some local nets required for the trace buffer. The number of trace nets that must be routed will vary depending on the benchmark as discussed in the previous section. The number is dependent on how many nets need to be traced and how many unused BRAM sites there are. For most of the benchmarks the number of trace nets is over 5000. The trigger unit is usually a fixed size with 256 inputs based upon the sample trigger unit described in Chapter 3 except when we examine the effects of trigger width. The trigger output is a single net that must fanout to each trace buffer, so the number of pins it must be routed to will vary with the number of trace buffers. The clock must be routed to each trace buffer and each slice of the trigger unit that contains flip-flops. The local nets required for the trace buffer were described in Chapter 3, and include the almost full signal that connects to the read enable pin and some nets that must be tied off to VCC. In addition to routing all these nets the router must avoid the already existing routing of the circuit.

A maze router was created based upon the one developed for RapidSmith [27]. The created router uses a directed search to find a path from a net's existing connections to a given sink pin. If a net has no existing connections then it will find a path from the net's source pin to the sink. The router works by evaluating routing nodes and placing them in a priority queue so that those more likely to lead to the sink will be evaluated first. There is some special logic for the clock net so that it will use the dedicated clock routing on the FPGA. RapidSmith contains functions that process the XDL file and mark used routing resources so that the router will not use them.

The router only performs a single pass over all the nets, permanently routing each one as it goes. No rip-up and reroute is performed so once a net is routed the resources it used will be unavailable to the remaining nets. Thus, the order the nets are routed in will affect the result. Later we investigate two orders for routing the trace nets: random and sorted by distance from the nearest trace buffer. A more advanced routing algorithm that performs multiple passes could be used, but the results presented in Chapter 5 indicate that the simple maze router is sufficient and its single pass shortens runtime.

Before routing begins some routing nodes are reserved for the trace buffer and trigger sink pins, as was also done in [27]. Typically sink pins are only accessible from one routing node in an interconnect tile, so for each sink pin the routing node from which it is accessed is reserved. If no nodes were reserved we found that some routes used nodes that were required by other routes to reach sink pins. The first-come, first-served nature of the router's single pass causes this and makes it impossible to complete the routes to some sink pins. Reserving nodes prevents this problem. The reserved nodes can only be used by a net that is being routed to the pin they are reserved for. Thus, the router is prevented from using nodes that are the only way to reach sink pins that are not part of the current route.

The routing problem for the trace nets has a unique many-to-many flexibility [6, 19]. A trace net can be connected to any trace buffer data input pin to make it observable. It does not matter which trace buffer or data pin. Therefore, all the available trace buffer data input pins are potential sinks for each trace net. Given this flexibility, we choose to route traces to the nearest trace buffer with an available pin. The particular trace buffer and pin to route a trace net to is decided when it is time to route the net. This makes the routing order of the trace nets important and we investigate the effects of routing order in Chapter 6.

The trigger nets do not have the same flexibility as the trace nets. We assume the trigger nets must be routed to specific input pins on the trigger unit unlike the trace nets which can be routed to any appropriate pin on any trace buffer. This assumption is justified because the trigger logic that each input drives may be unique and not allow for nets to be interchanged with each other. There may be certain types of logic where it would be possible to interchange nets between different input pins, but these would be special cases and are not investigated. We use the worst case scenario so that our results may be representative of a wider variety of trigger units. Thus, the trigger unit requires each net in the trigger netlist be routed to a specific input pin which is determined by its postion in the list.

We skip routing the trigger unit's internal signals. It would not be necessary to route these signals if HMFlow [27] is used to insert a trigger hard macro. The hard macro would include all of the internal routing. The trigger unit in our tests is a placeholder as described in Chapter 3. The placeholder does not include the internal signals of the trigger unit. If the internal signals were routed with our router then we anticipate it would add only a small overhead compared to all the other signals that are being routed. The overhead would be small because the internal signals of the trigger unit are only routed short distances. The area allocated for the trigger unit should be sufficient for all internal routing.

CHAPTER 5. PRIMARY RESULTS

This chapter describes our methodology for testing the feasibility of our proposed incremental trace-based debug system. Routability is the primary concern but receives little attention because all routes are successfully completed in all cases presented in this chapter. The results of the tests are presented and demonstrate the proposed system is feasible and can observe thousands of signals. We present the runtimes of the incremental insertion on a set of benchmarks and the effects on minimum period.

5.1 Test Methodology

To demonstrate the feasibility of our proposed debug system we investigate the effects of using it to trace up to 100% of the flip-flops and latches in five benchmark circuits (when trace buffer capacity permits). We chose to trace the flip-flops and latches because given the values of them the other intermediate values of a circuit could be calculated using techniques like those in [28–30]. Also, the flip-flops and latches are guaranteed to be accessible because all slice registers are connected to interconnect tiles. For one of the benchmarks the number of unused BRAMs is not enough to trace all signals. In this case, and when we are not tracing 100% of the state signals, a random subset is chosen. Signal selection techniques such as those in [28–31] could be used when selecting a subset, but as in [6] we decided to take multiple random samples to gain an understanding of our techniques when applied to any register a designer may wish to observe.

Runtime and minimum period are the metrics presented. The runtime is the time it takes to place and route the trace buffers and trigger unit. The time was determined by storing the current system time between major steps in the program and then comparing the times after routing has completed. The minimum period was determined by the Xilinx ISE Trace tool, which analyzes the timing of the FPGA circuit. The number of routing failures is also monitored to verify routability, but is not presented in this chapter because it is zero in all cases for the settings used in this chapter.

Table 5.1: Uninstrumented Benchmark Summary (values in bold indicate constraining resource on signals traced)

Circuit	LUTs	FFs	Slices	BRAMs	DSPs	IOBs	Signals Traced	Max Traces
bgm	14446	**5069**	6311	0	22	289	5069	10728
LU8PEEng	18233	**5498**	7843	45	16	216	5498	9108
stereovision0	5069	**7783**	2615	0	0	366	7783	10728
stereovision1	3871	**5874**	2204	0	152	278	5874	10728
LU32PEEng	63504	18521	19095	**165**	64	216	4788	4788
Total Available	81920	81920	20480	298	320	840	-	10728

If there were routing failures this would be important to report because the debug system may not be able to function properly if certain nets are not routed. In Chapter 6 other settings are explored that do result in routing failures.

Important characteristics of the benchmarks are shown in Table 5.1. The benchmarks circuits were synthesized, placed, and routed with the Xilinx ISE tools for the Virtex-5 ML510 embedded development platform which contains a XC5VFX130T FPGA. These benchmark circuits are available as part of the VTR project [20] and represent realistic, sizable, heterogeneous designs that include a Monte Carlo simulation for a financial application, bgm, and linear system solvers, LU8PEEng and LU32PEEng. The "total available" row in the table lists the maximum number of elements available on the XC5VFX130T. This table assumes a trace buffer width of 36 is being used, so "max traces" is calculated by multiplying the number of unused BRAMs by 36. The values in bold indicate the limiting factor on the number of signals traced for each benchmark. LU32PEEng is the only benchmark where the number of unused BRAMs is insufficient to trace all flip-flops. It is also the largest circuit with over 93% slice utilization.

For the tests in this chapter there are several parameters we keep constant. The width of trace buffers is fixed at 36, meaning 36 traces can be connected to each trace buffer. At this width the trace buffers have a depth of 1024. The traces are routed in a random order. The trigger unit is fixed at 256 inputs and 100 slices. These parameters were chosen for the trigger unit based off of the sample trigger unit that was described in Chapter 3. The sample trigger unit only required 91 slices when compiled for the Virtex-5 with Xilinx ISE. We have increased this by about 10% to 100 slices to model the requirements for a trigger unit that has slightly more logic than the sample trigger unit. The trigger unit slices are placed in locations where the original circuit has not used any routing in the interconnect tile associated with the slice to avoid routing congestion. The effects of changing these parameters will be explored in the next chapter.

For most tests the results presented are the averages of multiple runs. There is variation between runs due to several parts of trace insertion being random, including: selection of trigger inputs, selection of traces (when not tracing all flip-flops), placement of trace buffers, and the order traces are routed in. Other parts of the trace insertion are also influenced by the outcomes of the random parts. For example, the placement of the trigger unit is based upon the trigger inputs so it may vary in location with different inputs. The randomization can be controlled via the random seed, so that results may be reproduced if necessary.

We assume the slices of the trigger unit must all be placed within the same region. This ensures the trigger unit has good timing performance and its internal signals only have to be routed short distances. One of the benchmarks required some special adjustments to make this possible. The placer could not find a large enough region to place the trigger unit in LU32PEEng without intervention due to the high percentage of slice utilization. To enable successful trigger unit placement in LU32PEEng, we used the Xilinx PROHIBIT constraint to reserve a region of slices on the edge of the FPGA. This violates our goal to not interfere with the placing of the original circuit but demonstrates what is necessary to use our techniques on a circuit that utilizes a large percentage of the FPGA and requires a large trigger unit. PROHIBIT does not prevent the use of routing resources in the reserved region and we did find that routes passed through the prohibited region. Thus, we could not place the trigger unit in a region with no used routing like the other benchmarks. For LU32PEEng we allow the slices of the trigger unit to be placed in locations even if there may be routing congestion. None of the other benchmarks require these extra steps to place the trigger unit so they do meet our goal of not interfering with the placement of the original circuit.

5.2 Runtime Proportions

First we examine what proportion of runtime is taken by each step of the placer and router. Here is a brief description of each step:

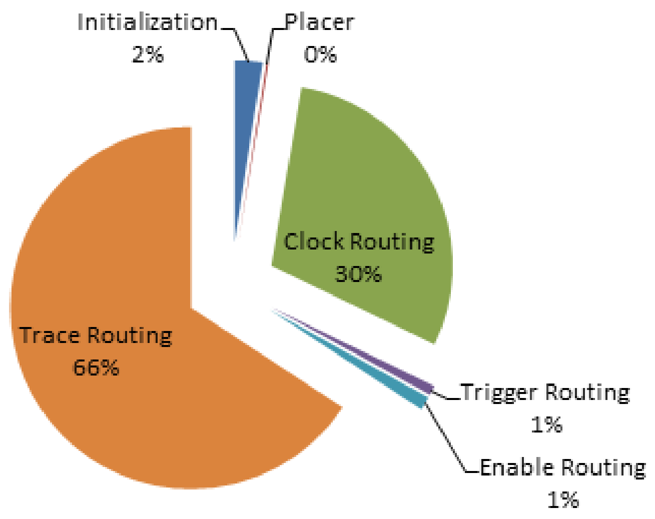

Figure 5.1: Pie chart showing proportion of total runtime for each step of trace insertion process.

- Initialization: uses the XDL input to mark used FPGA resources and reserves some resources for specific sink pins.
- Placer: places the trigger unit and trace buffers.
- Clock Routing: routes the circuit's clock signal to all inserted trace buffers and flip-flops.
- Trigger Routing: routes all of the trigger unit's input signals.
- Enable Routing: routes the trigger unit's output signal to all trace buffers and routes local trace buffer nets such as the almost full signal to read enable.
- Trace Routing: routes all of the trace buffer data inputs.

The time each step took was calculated by storing the current system time before and after each step and calculating the difference.

Figure 5.1 shows the runtime proportions based on the average runtimes from five benchmarks. The settings for these runs were described in the previous section. The runtime is dominated by the trace routing and clock routing. Trace routing was expected to take longer because the number of trace pins is an order of magnitude greater than the number of pins for the other types of nets that must be routed. However, the proportion of time spent routing the clock is unexpectedly large considering the number of pins the clock must be routed to is on the same order of magnitude as the enable or trigger pins. This is probably because of the special considerations in the router's code to make the clock use the dedicated clock routing resources of the Virtex-5. There may be ways to improve the clock routing code so that it runs faster.

Initialization, placement, trigger routing, and enable routing combined consume less than 5% of the total runtime. Initialization's runtime depends mostly on the size of the original circuit because it must process the XDL and mark used resources. For larger circuits it may take a slightly larger proportion of the runtime than the average shown here. The placement step is faster than any of the other steps even though it performs a breadth first search to find a location for the trigger unit and other operations to place all the components that are needed. With such a small runtime, optimizing the speed of placement would do little good until the runtime of the routing steps is much better. The trigger routing only takes 1% of the total runtime. This result is not unexpected because the size of the trigger unit for this test was fixed at 256 inputs which is considerably smaller than the thousands of trace inputs. If the number of trigger inputs and trace inputs were the same then we expect their proportions would be nearly equal. On the other hand, enable routing taking only 1% is unexpected when you compare it to the clock routing. The clock and enable routing are similar because both involve routing a single signal to a large number of pins. This demonstrates again that the clock routing step could be optimized to run faster.

These results indicate that the number of trace buffers and traces will have the largest influence on runtime. Reducing the number of traces will decrease the size of the trace routing problem, which took the largest proportion of the time. It can also reduce the number of locations the clock must be routed to if trace buffers are eliminated, so it also influences the second largest step. Initialization, placement, and enable routing would also be simplified by these reductions, but are of less consequence since they already take a low proportion of the runtime.

5.3 Runtime

Figure 5.2 shows the total runtime for each of the benchmarks when the maximum number of trace buffer inputs are routed. First it should be noted that all benchmarks were able to successfully route all flip-flops to trace buffers or use all trace buffer capacity if there was not enough for all flip-flops. This is an interesting result and demonstrates that the Virtex-5 architecture has

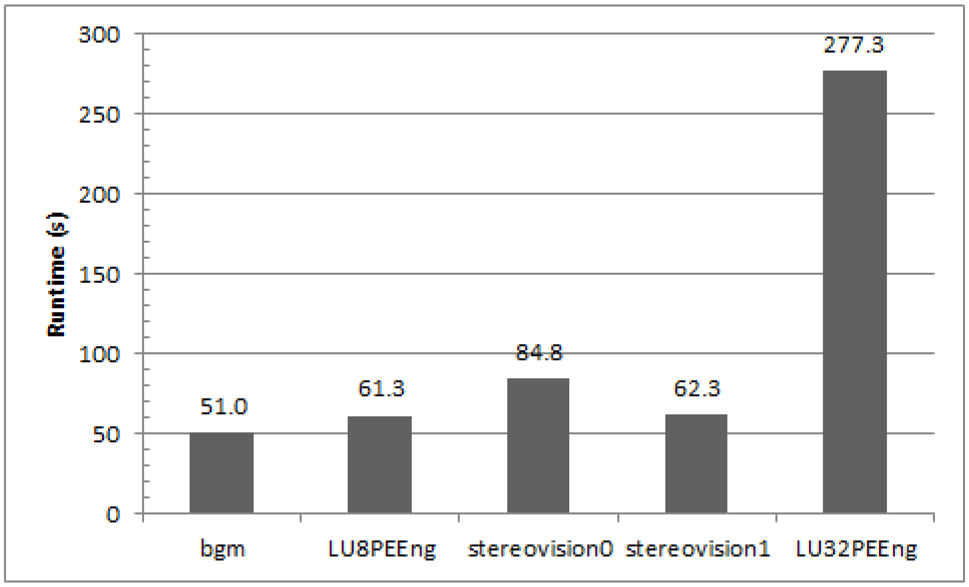

Figure 5.2: Trace insertion runtime for each of the benchmarks when the maximum number of flip-flops are traced.

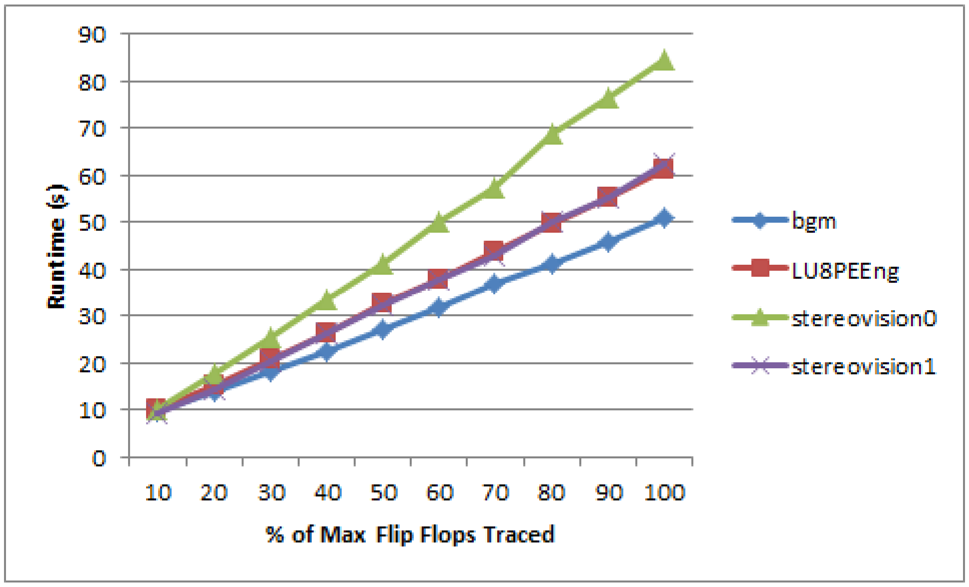

Figure 5.3: Percent of flip-flops traced vs. the trace insertion runtime.

enough routing resources to support our method. Most of the benchmarks take about a minute for the entire PAR process. LU32PEEng takes much longer although it has a lower number of traces than most the benchmarks. This must be due to the large number of resources used by

Table 5.2: Benchmark compile times in minutes and seconds.

Circuit	bgm	LU8PEEng	stereovision0	stereovision1	LU32PEEng
Compile time	11:18	29:00	10:14	25:10	54:20
Insertion runtime	0:51	1:01	1:25	1:02	4:37
Difference	10:27	27:59	8:49	24:08	49:43

LU32PEEng. The directed search of the router takes longer to find routes because so many of the routing resources are already used.

It was shown earlier in Figure 5.1 that routing the traces and the clock dominate the trace insertion runtime. Due to that dominance, changing the number of traces and trace buffers should significantly change runtime. Figure 5.3 shows how the runtime varies when different percentages of the maximum number of traces are routed for four of the benchmarks. As expected the total runtime is significantly lower when there are less traces and trace buffers. The runtime increases linearly as the number of traces is increased.

All the runtimes are less than the time it would take to recompile the entire circuit. Large and complicated circuits are known to take hours to recompile. The compile times of the benchmarks used in this work are shown in Table 5.2. The table also shows the average trace insertion runtimes from Figure 5.2 and the difference between the compile time and insertion runtime. None of compile times are in hours but even the shortest times are longer than all the runtimes for trace insertion, so incremental insertion decreases turn around time for even these circuits. Also, if incremental insertion were not used then the compile times would be higher than the values shown in the table because the Xilinx tools would have to place and route the trace buffers and trigger unit in addition to the original circuit.

5.4 Minimum Period

In this section we examine how our method affects the minimum period that the circuits can operate at. Figure 5.4 shows the minimum period of each benchmark before and after instrumentation. The period after instrumentation is the average of ten runs of trace insertion. LU8PEEng and LU32PEEng have minimum periods of about 80ns before instrumentation and they stay the same after instrumentation. The bgm benchmark has a minimum period of 15.541ns and instrumentation

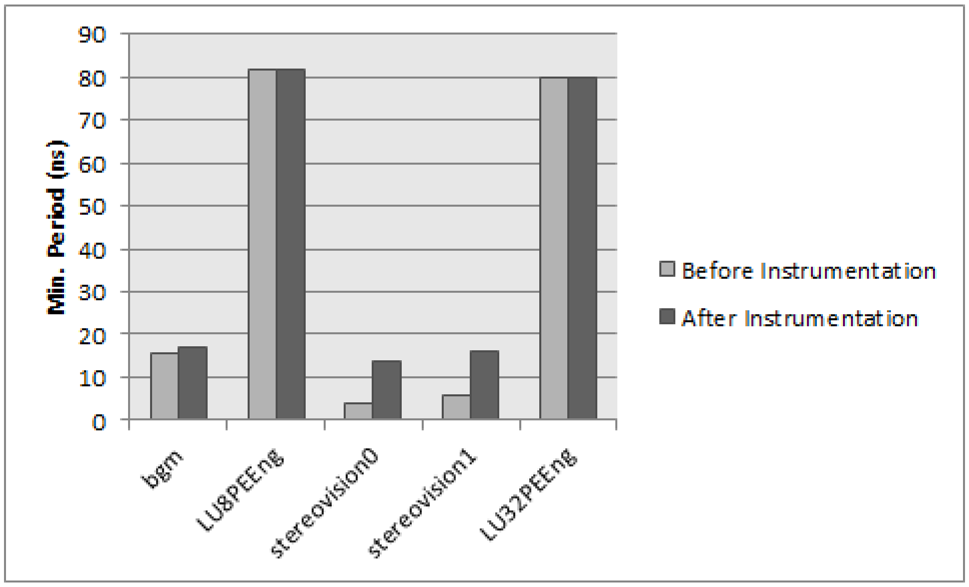

Figure 5.4: The minimum period of each benchmark before and after instrumentation.

increases this on average by about 9%. The stereovision benchmarks' minimum periods increase much more than the other benchmarks. Stereovision1 jumps from a minimum period of 5.918ns to 16.131ns on average, an increase over 272%. The percentage is even greater for stereovision0 which goes from 4.126ns to 13.549ns, over 328%. Ideally there would be no change in minimum period.

The minimum period increases if the delay of any of the paths we insert is greater than the original minimum period. In other words, if a path we insert becomes the new critical path the minimum period changes. Circuits with a higher minimum period are less likely to experience any increase because the inserted paths may be longer without becoming a critical path. Thus, there was not a change in the periods of LU8PEEng and LU32PEEng. The bgm benchmark sometimes had an increase in its minimum period and sometimes did not. The variation in periods is due to the randomization in trace buffer placement. The stereovision benchmarks have small minimum periods so new critical paths were created when trace insertion was performed.

The Xilinx TRACE tool can be used to analyze critical paths. The stereovision benchmarks were analyzed to determine the critical path. The critical path is the control signal that is an output of the trigger unit and input to the trace buffers. The trigger unit is often placed near the edge of the FPGA because closer to the center there are not enough free slices to place it. The trace buffers are spread throughout the FPGA, so some are located a long distance from the trigger unit. The critical path is between the trigger unit and one of the distant trace buffers.

We believe pipelining can be used on the critical paths to avoid increasing the minimum period of the design. Flip-flops could be incrementally added as needed in unused locations along the critical paths to reduce delay. The TRACE results show that the trigger output should be pipelined first since it is the current critical path. In Chapter 7 we investigate the use of pipelining.

CHAPTER 6. INFLUENCES ON ROUTING

In this chapter we present several investigations into parameters that may affect routing performance. We measure routing performance with three metrics: number of routing failures, routing runtime, and minimum period of the routed design. A routing failure is when the router cannot find a path for a given net and sink pin. Ideally, the number of routing failures should be zero. Likewise, the minimum period of the design should ideally have zero change compared to the minimum period of the original circuit. We also seek to reduce the runtime of the router, but we consider the other two metrics to be of greater importance. The benchmarks used in Chapter 5 are also used in this chapter. The parameters investigated in this chapter include: trace buffer width, trigger width, routing order, and congestion.

6.1 Trace Buffer Width

In this section we present the results of investigating the effects of trace buffer width on routing. The BRAMs used to implement our trace buffers in the Virtex-5 support different configurations of data width as described in Chapter 2. Wider trace buffers will allow more signals to be observed. However, wider trace buffers also require more signals to be routed to the same interconnect tiles of the FPGA. In some cases there may not be enough routing resources for all signals to be routed to the same trace buffer. Thus, routing failures are more likely when wider trace buffers are used.

We want to use the greatest trace buffer width that will result in no routing failures. The available widths include: 72, 36, 18, 9 and 4. We investigated them starting with the widest and then continuing in descending order if necessary. It is not necessary to continue after we find the greatest width that can be used without causing routing failures. We inserted the debug system into each benchmark ten times for a given width to obtain an average due to the randomness in trace insertion described in Chapter 5.

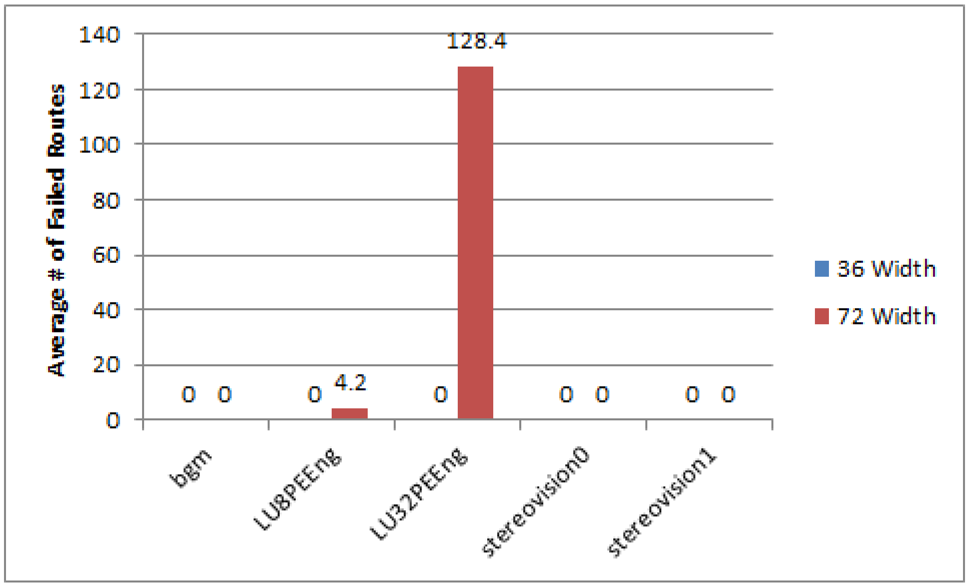

Figure 6.1: Average routing failures for trace buffer widths of 36 and 72.

Figure 6.1 shows the average routing failures for widths of 36 and 72. Where a failure is a route between a source and a single sink that the router is not able to complete. It was not necessary to tests width less than 36 because there were no routing failures when a width of 36 is used. However, when 72 width trace buffers were used there were trace routing failures for some benchmarks. We attribute these routing failures to routing congestion in the vicinity of the trace buffer. If the failures were due to routing congestion elsewhere then the failures should occur for smaller widths too but they did not.

Due to these results we use a trace buffer width of 36 in all other tests in this paper. Another option we did not explore would be to use trace buffers of different widths. For example, 36 width could be used in highly congested area and 72 in areas with lower congestion. Further tests would be necessary to determine how much congestion to allow near 72 width trace buffers. These results may differ for different architectures if there are changes in the routing channel width. In [19] it was shown that greater channel width increases routability.

6.2 Trigger Width

In Chapter 5 it was shown how varying the number of traces influenced runtime. Here we examine how varying the number of trigger signals influences runtime. We consider the number

Table 6.1: The number of trigger unit slices for different widths.

Trigger Width	0	4	8	16	32	64	128	256	512	1024	2048
Sample Trigger Slices	0	6	6	10	15	29	51	91	177	351	690
Simulated Slices	0	6	7	11	16	32	56	100	195	386	759

of trigger signals to be the trigger width. Changing the trigger width is different from changing the trace buffer width, as the previous section did. When the trace buffer width increases less trace buffers are required to observe a set number of signals. On the other hand, when the trigger width increases more slices are required to form the trigger unit because more flip-flops and logic is required.

Using Xilinx ISE we compiled the sample trigger unit with different widths. Table 6.1 shows the widths that were used and the number of slices the sample trigger unit required. We increase the sample trigger slices by approximately 10% to simulate trigger units that require more logic than our sample. The "simulated slices" row shows the number of slices used in our tests for each width. The greatest width we used was 2048 because ISE was unable to compile the sample trigger unit when a width of 4096 was attempted. ISE was able to place the sample trigger with a width of 4096, but routing failed because ISE could not route all the carry signals. A trigger width of 0 represents a single user signal being used to control the trace buffers. This may occur if the trigger unit is included in the original circuit or if all that is needed is that single signal.

The LU32PEEng benchmark was not included in these tests because it required reservation of slices to fit even the 256 wide trigger unit that is used in other tests. To fit larger trigger units in LU32PEEng would require reserving even more slices which would influence the placement and routing of the original circuit even more. Influencing or changing the placement and routing of the original circuit is something we want to avoid. Circuits that use a large percentage of the FPGA will be limited to smaller trigger units. All of the benchmarks used were able to successfully complete trace insertion without any placement or routing errors.

Figure 6.2 shows the runtimes for different trigger widths. The runtime increases exponentially with the number of trigger signals, but it is mostly flat until the widths are 1024 or higher. This results is different from what was seen in Chapter 5 for traces, which had a linear increase in runtime as the number increased. The exponential increase comes because of the additional

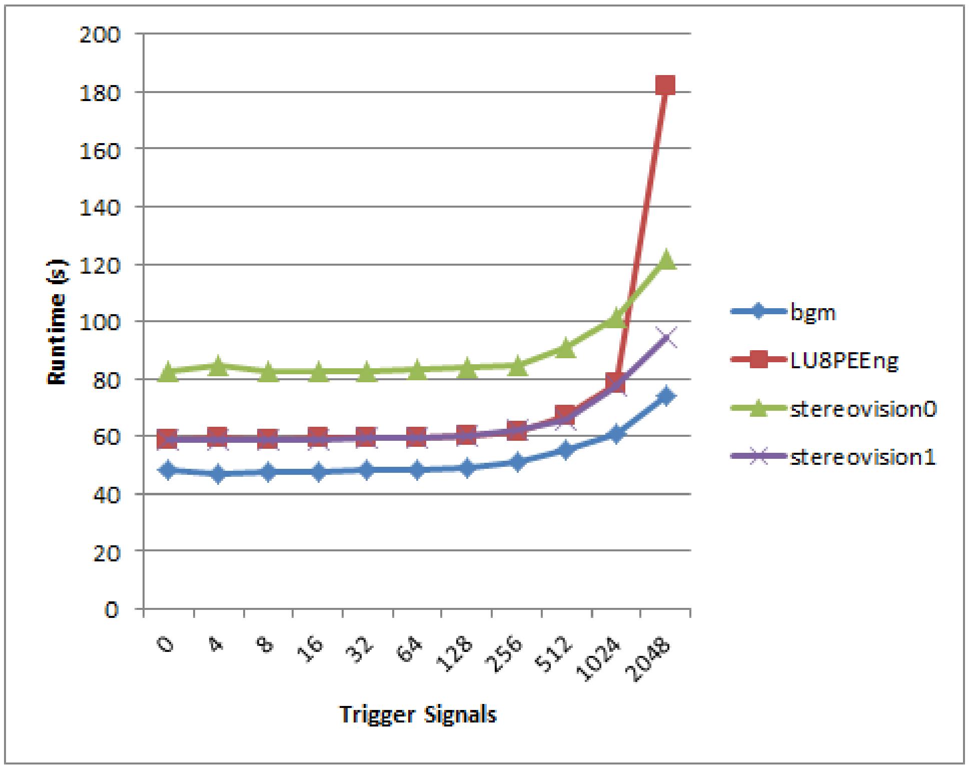

Figure 6.2: Trigger signals influence on runtime.

clock routing that is required with more slices. For example, 1024 trigger signals require at least 256 slices for 1024 flip-flops, so there are 256 more components for the clock to be routed to. 1024 traces only require 29 trace buffers which is an order of magnitude less components for the clock. In some cases the additional trigger signals also complicate the trace routing. For example, LU8PEEng has a particularly long runtime when there are 2048 signals. Closer analysis of the runtime shows that trace routing still took the largest portion of the time with an average of 74.6 seconds. This is notable because that is clearly longer than the trace runtime when there were less trigger signals, so it has become more difficult for the router to route the traces due to the increase in trigger signals. Trigger routing and clock routing took averages of 59.6 and 43.7 seconds respectively for LU8PEEng at 2048 trigger width. The other steps of trace insertion experience little change due to the number of trigger signals and only take a few seconds combined.

Designers should avoid using trigger units that require thousands of input signals if runtime is an issue. For the benchmarks used in our tests runtime did not increase much if the number of inputs signals was 512 or below. Beyond that the runtime increases quickly due to the exponential curve. However, improving the clock router may eliminate the exponential increase in runtime or at least keep the increase flat for even larger number of signals. Designers should also keep in mind the amount of slices the original design uses. None of the benchmarks used here had a problem placing trigger units of hundreds of slices, but we excluded the LU32PEEng benchmark because large trigger units cannot be placed in it without reserving slices which influences the original circuit.

6.3 Routing Order

As explained in Chapter 4, the router used in this paper does not perform any rip-up and reroute. Once a net is routed the resources it used are unavailable for other nets. Thus, if nets are routed in a different order the result may be different. Trace nets will be routed to different pins depending on the routing order because of how we have taken advantage of the flexible nature of the trace routing problem. We do not decide what particular pin to route a trace net to until it is time to route the net. For example, if net A and B are both being routed to the same trace buffer that has pin 1 and 2 available, then whichever net is routed first will be routed to pin 1 and the second will be routed to pin 2. The order the trace nets are routed in can even change the trace buffer that nets are routed to. If net A used the last available pin on the trace buffer then net B would have to be routed to a different trace buffer. Due to this, we investigated if the order of routing the nets to the trace buffers has a significant impact on the resulting minimum period or runtime. The routing order of the trigger nets does not change what pins the trigger nets are routed to because each net has to be routed to a particular pin, so the routing order of the trigger nets was not explored.

Two routing orders were examined: random and sorted by distance from the nearest trace buffer as suggested in [6]. In theory sorting the nets such that the nets that are furthest from a trace buffer are routed first should lead to a better minimum period and reduced runtime for the router. The sorted order guarantees that the most distant nets will find pins available on the nearest trace buffer. Without this guarantee the nets might have to be routed to a more distant trace buffer if other nets use all the pins on the nearest trace buffer. A longer routing distance may cause a worse minimum period due to wire delay and the runtime of the router may be longer.

Figure 6.3 and 6.4 show the average runtime and period after ten runs for four benchmarks. There is not a significant difference between sorted and random, but on average random has a

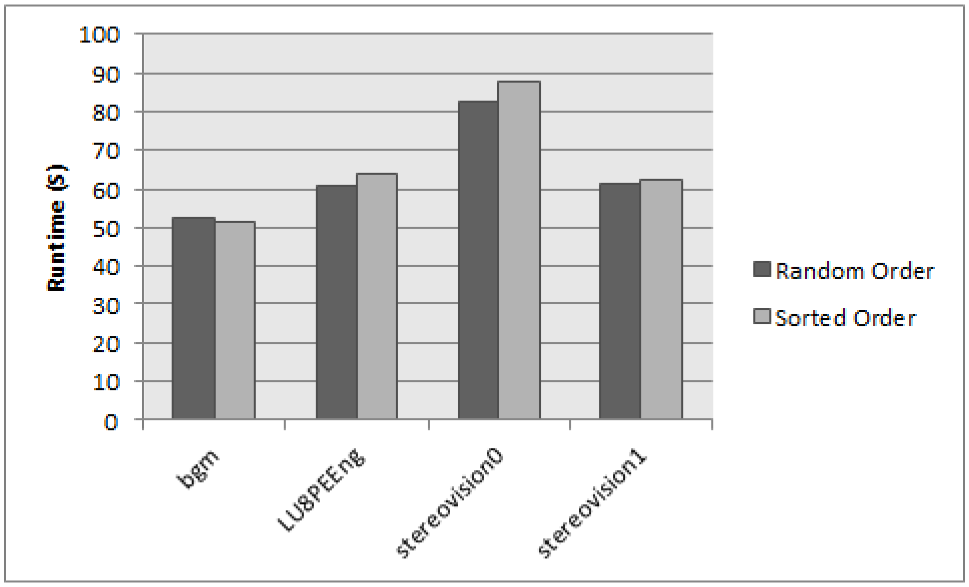

Figure 6.3: Comparison of the runtimes for random and sorted routing orders.

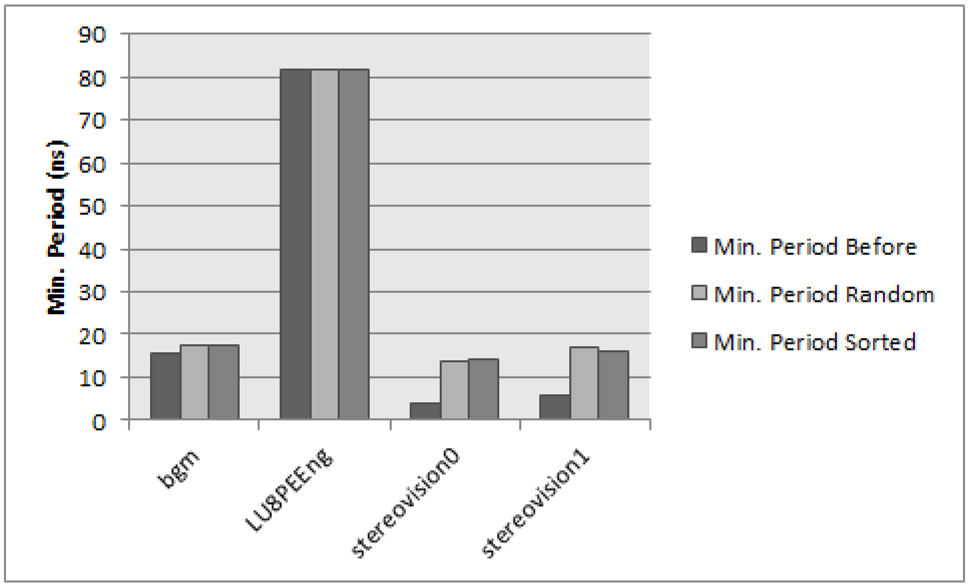

Figure 6.4: Comparison of the minimum periods for random and sorted routing orders.

slightly lower runtime and minimum period. This is surprising due to the previous discussion of how the sorted order guarantees the nets that are the furthest from any trace buffer can be routed to the nearest one. Perhaps since the trace buffers are distributed it is a rare event, even with a random order, for a net's routing distance to increase significantly even when its nearest trace buffer is unavailable. Also, the routing distance for nets that are very close to a particular trace

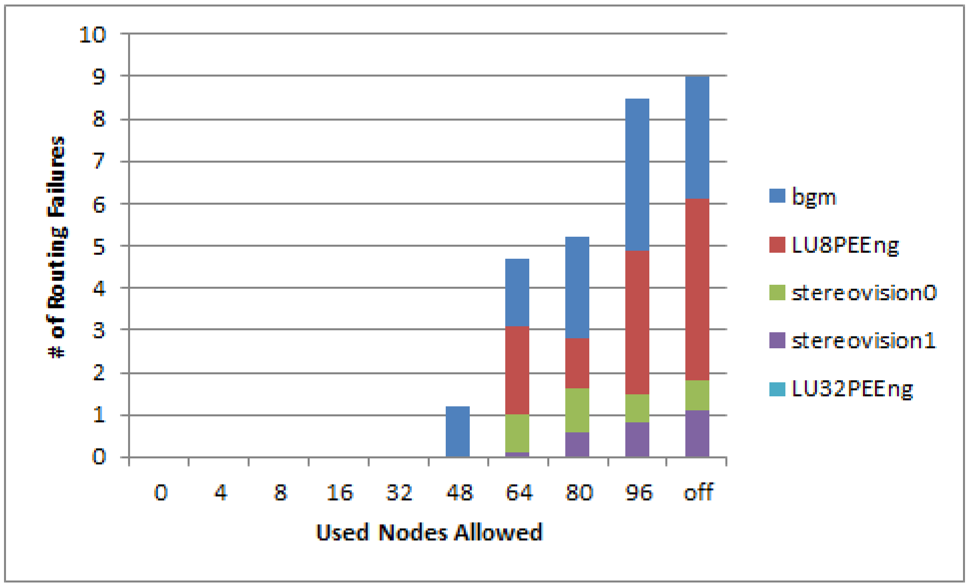

Figure 6.5: Used nodes allowed in interconnect tiles vs. the average number of routing failures across all benchmarks.

buffer may increase more than distant nets when that trace buffer is unavailable. Due to these results all other tests in this paper route in a random order unless stated otherwise.

6.4 Congestion

We investigated how routing congestion affects the routability of the trigger unit. We define congestion based on the number of used routing nodes. RapidSmith uses nodes to represent the routing resources of the FPGA and the connections between them are represented as edges [9]. These nodes and edges form a routing graph which routers can use to find paths between resources. An interconnect tile of the FPGA is highly congested if a large number of its nodes are used. The used nodes cannot be used by other nets and make it more difficult to route additional nets through the tile.

Figure 6.5 plots the used routing nodes allowed when placing the trigger unit against the number of failed routes. In each column the total number of routing failures across all the benchmarks is shown for the amount of "used nodes allowed" shown below the column. If 0 used nodes are allowed this indicates that a slice will not be used for trigger unit placement if its accompanying interconnect tile has any used nodes. As the number of used nodes increases this means we allow the trigger unit slices to be placed in locations where the already placed-and-routed circuit has used some of the routing nodes in the accompanying interconnect tile. The final column, off, means congestion is not considered during placement so the interconnect tile may have any number of used nodes. The benchmarks have no routing failures until the number of used nodes allowed is 48 or 64. At this point the routing congestion starts to be too much for all signals to route successfully on most the benchmarks.

LU32PEEng is an exception to this trend. As noted in Chapter 5, LU32PEEng must reserve some slice locations so that the trigger unit can be placed. We did not have to do this for any of the other benchmarks. However, reserving the slices does not prevent the Xilinx ISE tools from routing through the interconnect tiles that neighbor them. The placer is not able to find a placement for the trigger unit in LU32PEEng until 80 used nodes are allowed. This means the original circuit has already used a number of routing resources where the trigger unit is placed. Even when any number of used nodes are allowed by turning off congestion detection, the router is able to successfully route LU32PEEng. Due to these results, for all other tests in this paper LU32PEEng is placed with congestion detection off but all the other benchmarks are placed with 0 used nodes allowed.

CHAPTER 7. IMPROVING TIMING WITH PIPELINING

In this chapter we demonstrate that pipelining may be used to reduce our method's impact on minimum period. Ideally trace insertion should not change the minimum period of the circuit. If minimum period increases then the circuit must run at slower speeds. This chapter only proves the feasibility of using pipelining to reduce the delay of the paths we insert. We do not achieve the ideal but we believe further pipelining could allow trace insertion to avoid increasing minimum period in the majority of circuits.

7.1 Pipelining Methodology

In Chapter 5 we saw that some circuits will have a large increase in minimum period due to trace insertion. In particular, the minimum period of the stereovision0 and stereovision1 benchmarks increased by over 3x in some cases. An increase in minimum period is undesirable because the circuit will have to run at slower speeds and verification will take longer. To address this problem we explore the use of pipelining.

Pipelining is a commonly used method to reduce the delay of a path. Figure 7.1 shows a path before and after it has been pipelined. The path begins at a source flip-flop and ends at a sink flip-flop and initially it has a long delay. We are primarily only concerned by delay that is the result of path length, but path delay is also caused by including logic elements on the path. Paths begin and end at memory elements; in the figure flip-flops are used but BRAMs and other memory elements could also be sources or sinks on paths. The bottom path of the figure shows the same path with a pipeline flip-flop in the middle. The path has now been broken into two shorter delays rather than one long delay. The disadvantage is that an additional clock cycle will be required for signals to travel from the source to the sink. However, it may not matter that an additional cycle is required and the clock may be able to run faster as a result of pipelining.

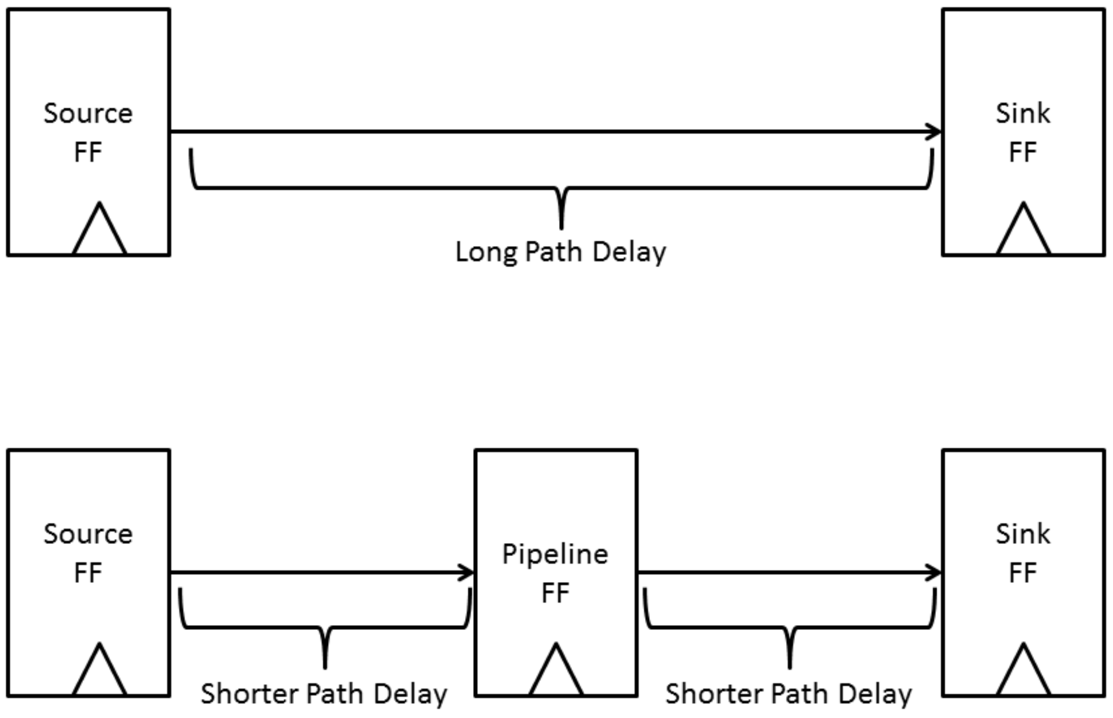

Figure 7.1: A path before and after a pipeline flip-flop has been added to it.

Our trace insertion method increases the minimum period of a circuit when it creates new critical paths. A critical path is a path whose delay exceeds the delay of all other paths. The critical path determines the minimum period at which the circuit can operate. Values will not be transferred over the path correctly if its delay is not respected. The stereovision benchmarks experienced large increases in minimum period because the existing period was not long enough to support the path delays we needed for trace insertion. Some of the paths we inserted became new critical paths for the circuit.

Analysis with the Xilinx Trace tool showed that the critical path in the stereovision benchmarks after trace insertion was the control signal that travels from the trigger unit to all the trace buffers. We shall refer to this signal as the control signal in this chapter. There is no logic on the path of the control signal, so most of its delay comes from path length. It was not surprising that the control signal became a critical path because some trace buffers are located significant distances from the trigger unit. We demonstrate the feasibility of pipelining by doing it on the path

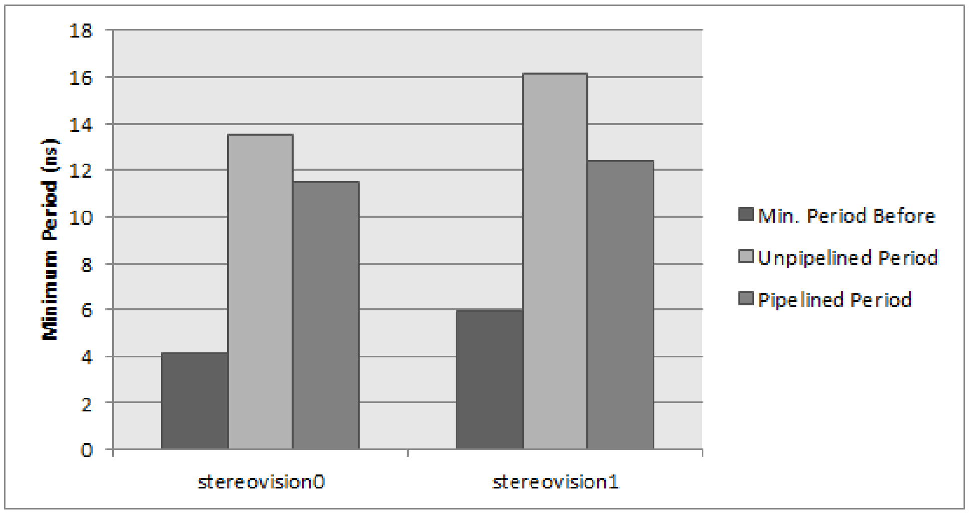

Figure 7.2: Minimum period before trace insertion and after with and without pipelining.

of the control signal. If pipelining is feasible and successful then we should see a reduction in the minimum period.

We propose pipelining the control signal with two stages on every path. Where a path is the route between the the control signals source pin and one of its sink pins. The example shown in Figure 7.1 was a one stage pipeline. A two stage pipeline would have two pipeline flip-flops between the source and sink. The distributed nature of the trace buffers helps us plan in advance a pipelining strategy. We take an average of the trace buffer's locations to determine the point that could be considered their center. A flip-flop shall be placed as close to this center as possible to form the first stage of the pipeline. After the first stage the control signal shall fanout to four flip-flops placed in the four quarters of the FPGA. The output of each of the four second stage flip-flops shall be routed to the trace buffers in its quarter of the FPGA. This is a simple yet effective way to pipeline the control signal to the trace buffers distributed throughout the FPGA. It may be possible to get better results by using even more stages or dividing the second stage into more than four, but this pipeline method should be sufficient to see if pipelining can improve the minimum period.

7.2 Pipelining Results

Figure 7.2 shows the several minimum periods of the stereovision benchmarks. The first bar is the original minimum period of the uninstrumented circuit. The second is the minimum period after trace insertion with no pipelining as was done in Chapter 5. The third bar is the minimum period after trace insertion with pipelining averaged over 20 runs. The minimum period is observed to improve by pipelining, so pipelining is a feasible way to reduce the delay of inserted paths. Stereovision0's average period dropped from 13.549ns to 11.476ns, an improvement of over 15%. Likewise, stereovision1's average period dropped from 16.131ns to 12.411ns, an improvement of over 23%. These circuits' could benefit from further pipelining because they are still more than 2x the original period.

The period with our minimal pipelining is still significantly greater than the original because other inserted paths have now become the critical path. We again used the Xilinx Trace tool to identify the critical paths on the pipelined circuits. For stereovision0 the critical path is now a path of the pipeline, typically a path from a second stage pipeline register to a trace buffer. This indicates that the control signal will benefit from further pipelining. For stereovision1 the critical path is now typically a trace signal path. Thus, to further improve the minimum period of stereovision1 some of the traces should be the next paths pipelined.

Pipelining some paths of the debug system may require additional considerations. The control signal was simple to pipeline because adding a few cycles to the path of this signal does not matter. The only effect of these additional cycles is the trace buffers halting a few cycles later, but the trace buffer depth is sufficient that the cycles when the trigger condition occurred will still be captured. Pipelining trace paths would be more complicated. If only some trace paths are pipelined then their data will be some number of cycles off the data from other signals. One way to solve this might be to pipeline all traces with an same number of stages. However, it would be difficult to pipeline all traces in this manner since there are so many of them and there are limited flip-flops available in useful locations for pipelining. A better way to solve this problem would be to realign the data during off-line analysis. This could be done by having the trace insertion step create a log of traces that were pipelined and the number of stages in their pipelines. This log could be used during analysis to line up the extracted trace buffer data to the correct clock cycles.

CHAPTER 8. CONCLUSION

8.1 Summary

FPGAs are a increasingly being used for IC verification. They run faster than simulation and cost less than fabricating ASIC prototypes. However, the major disadvantage of performing verification on FPGAs is a lack of signal observability.

In this thesis we have investigated a new incremental trace-based method for increasing the observability of FPGAs. The method incrementally inserts trace buffers and a trigger unit in an already placed-and-routed FPGA circuit. A unique characteristic of the method is the centralized trigger unit that controls all the distributed trace buffers. This simplifies incremental insertion and allows the method to easily scale to any number of trace buffers. Advantages of this incremental method include not affecting the placing and routing of the user's circuit, taking full advantage of leftover BRAMs to observe more signals, and decreasing the turn-around time when changes are made to the debug system.

We demonstrated the method could instrument 100% of the flip-flops given that enough trace buffer capacity exists. This was done on a commercial Xilinx Virtex-5 FPGA, further distinguishing this work from others. Some pitfalls had to be avoided to achieve this result and avoid routing errors. First, we found that some benchmarks could not route all traces when using the widest trace buffer configuration. We also found that in most circuits the trigger unit should not be placed in locations with high routing congestion. When the trace buffers are configured properly and the trigger unit placement accounts for congestion no routing failures occur even when thousands of signals are observed by trace buffers.

The time it takes to perform the method was less than five minutes for all benchmarks. Most of the benchmarks only took about one minute. This means that a designer could insert or change circuit instrumentation for debug relatively fast. These runtimes are much smaller than the hours it can take to recompile large designs in flows where changing the signals being observed requires a complete recompile.

One drawback of our method is that it can increase minimum period for some circuits. This occurs if the delay of any of the paths we insert is greater than the current minimum period. The two benchmarks with the smallest periods experienced large increases in their periods. Those benchmarks with a minimum period of 20 ns or greater had little or no change. We demonstrated that pipelining can be used to improve the minimum period if needed.

In this thesis we also noted how the trigger unit and trace buffers differ in placement and routing. Each trace buffer only requires a single BRAM which simplifies placement. The routing problem for the trace buffers has a many-to-many flexibility because nets that must be observed can be routed to any available trace buffer pin. On the other hand, the trigger unit is more difficult to place because it requires a region of unused slices. The trigger unit also does not have routing flexibility like the trace buffers because inputs must be connected to specific pins to create the correct logic. Due to the more complicated placement and routing of the trigger unit it may require reserved slices in designs that use a large percentage of the FPGA as one of our benchmarks did.

8.2 Future Work

We believe there are many opportunities to build upon the methods presented in this thesis. For example, we showed that pipelining could reduce impact on minimum period. More work on incremental pipelining is needed so that we and others doing similar work can avoid increasing minimum period.

CAD tools could be created that simplify the effort required for designers to use our methods. The process of interpreting the trace buffer data readback from the FPGA could be greatly simplified by tools. Waveforms could be constructed from the data and signals could be more easily associated with their data. Tools could also be made that simplify the process of inserting trace buffers and a trigger unit for designers.

Another area for possible future work is in reconstructing the values of signals that were not observed by the trace buffers based off of the values of signals that were. With a gate-level knowledge of the circuit this may be straight forward for many signals. For example, if it is known that the value of a signal is the OR of two signals we are observing then it could be reconstructed during analysis based off of the other two signals. The reconstruction effort would need to know what optimizations had occurred during synthesis in order to properly reconstruct signals. Ways to enable collaboration between synthesis and reconstruction would need to be explored. Reconstruction would be particularly useful when all signals in a design cannot be observed or the designer wants to eliminate the need to observe all signals. If all the state values within an FPGA can be reconstructed we predict it could enable time-saving features for debugging, such as the ability to restore the FPGA to a desired state to quickly reproduce errors.

REFERENCES

[1] Nicely, T. R., 2011. Pentium fdiv flaw faq. 1

[2] Leveson, N. G., and Turner, C. S., 1993. "An investigation of the therac-25 accidents." *Computer,* **26**(7), pp. 18–41 ID: 1. 1

[3] Foster, H., 2011. Challenges of design and verification in the soc era. 1

[4] Asaad, S., Bellofatto, R., Brezzo, B., Haymes, C., Kapur, M., Parker, B., Roewer, T., Saha, P., Takken, T., and Tierno, J., 2012. "A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation." In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, ACM, pp. 153–162. 1

[5] Kottolli, A., 2006. "The economics of structured-and standard-cell-asic designs." *Technical Solutions Engineer, Open-Silicon*. 1

[6] Hung, E., and Wilton, S. J. "Incremental trace-buffer insertion for fpga debug.". 2, 13, 15, 29, 31, 44

[7] Xilinx, 2012. Virtex-5 fpga configuration user guide, Oct. 19. 2, 8, 12

[8] Xilinx, 2012. Virtex-5 fpga user guide. 4, 9

[9] Lavin, C., Padilla, M., Lamprecht, J., Lundrigan, P., Nelson, B., and Hutchings, B., 2011. "Rapidsmith: Do-it-yourself cad tools for xilinx fpgas." In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 349–355 ID: 1. 4, 7, 12, 27, 46

[10] Brand, D., Drumm, A., Kundu, S., and Narain, P., 1994. "Incremental synthesis." In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, IEEE Computer Society Press, pp. 14–18. 11

[11] Beckhoff, C., Koch, D., and Torresen, J., 2011. "The xilinx design language (xdl): Tutorial and use cases." In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pp. 1–8 ID: 1. 12

[12] Hutchings, B. L., and Nelson, B. E., 2001. "Unifying simulation and execution in a design environment for fpga systems." *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* **9**(1), pp. 201–205 ID: 1. 12

[13] Hutchings, B., Bellows, P., Hawkins, J., Hemmert, S., Nelson, B., and Rytting, M., 1999. "A cad suite for high-performance fpga design." In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, pp. 12–24 ID: 1. 13

[14] Wheeler, T., Graham, P., Nelson, B. E., and Hutchings, B., 2001. "Using design-level scan to improve fpga design observability and controllability for functional verification." In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, FPL '01, Springer-Verlag, pp. 483–492. 13

[15] Iskander, Y. S., Patterson, C. D., and Craven, S. D., 2011. "Improved abstractions and turnaround time for fpga design validation and debug." In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 518–523 ID: 1. 13

[16] Graham, P., Nelson, B., and Hutchings, B., 2001. "Instrumenting bitstreams for debugging fpga circuits." In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pp. 41–50 ID: 1. 14

[17] Poulos, Z., Yang, Y., Anderson, J., Veneris, A., and Le, B., 2012. "Leveraging reconfigurability to raise productivity in fpga functional debug." In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pp. 292–295 ID: 1. 14

[18] Hung, E., and Wilton, S. J. E., 2013. "Towards simulator-like observability for fpgas: a virtual overlay network for trace-buffers." In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, ACM, pp. 19–28. 15

[19] Hung, E., and Wilton, S. J. E., 2012. "Limitations of incremental signal-tracing for fpga debug." In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 49–56 ID: 1. 15, 29, 41

[20] Rose, J., Luu, J., Yu, C. W., Densmore, O., Goeders, J., Somerville, A., Kent, K. B., Jamieson, P., and Anderson, J., 2012. "The vtr project: architecture and cad for fpgas from verilog to routing." In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, ACM, pp. 77–86. 15, 32

[21] Synopsys, 2011. Identify: Simulator-like visibility into hardware debug. 16

[22] Tektronix, 2012. Certus asic prototyping debug solution, Sep. 16

[23] Xilinx, 2012. Chipscope pro software and cores user guide, Oct. 16

[24] Xilinx, 2013. Partial reconfiguration user guide, April 26. 17

[25] Altera, 2013. Quartus ii handbook version 13.0, May. 17

[26] Ko, H. F., and Nicolici, N., 2009. "Resource-efficient programmable trigger units for post-silicon validation." In *Test Symposium, 2009 14th IEEE European*, pp. 17–22 ID: 1. 21

[27] Lavin, C., Padilla, M., Lamprecht, J., Lundrigan, P., Nelson, B., and Hutchings, B., 2011. "Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping." In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 117–124 ID: 1. 27, 28, 29, 30

[28] Ko, H. F., and Nicolici, N., 2009. "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* **28**(2), pp. 285–297 ID: 1. 31

[29] Prabhakar, S., and Hsiao, M., 2009. “Using non-trivial logic implications for trace buffer-based silicon debug.” In *Asian Test Symposium, 2009. ATS ’09.*, pp. 131–136 ID: 1. 31

[30] Cheng, W., Chuang, C., and Liu, C. J., 2006. “An efficient mechanism to provide full visibility for hardware debugging.” In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pp. 4 pp.–814 ID: 1. 31

[31] Hung, E., and Wilton, S. J. E., 2011. “Speculative debug insertion for fpgas.” In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 524–531 ID: 1. 31

APPENDIX A. PLACEMENT EXAMPLES

A.1 Placement Screenshots

The figures shown in this appendix are screenshots from Xilinx FPGA editor. They are examples of what circuits look like after we have placed trace-buffers and a trigger in them. The BRAMs used as trace-buffers are highlighted in green and the trigger slices are highlighted in red. The components in blue are the original circuit.

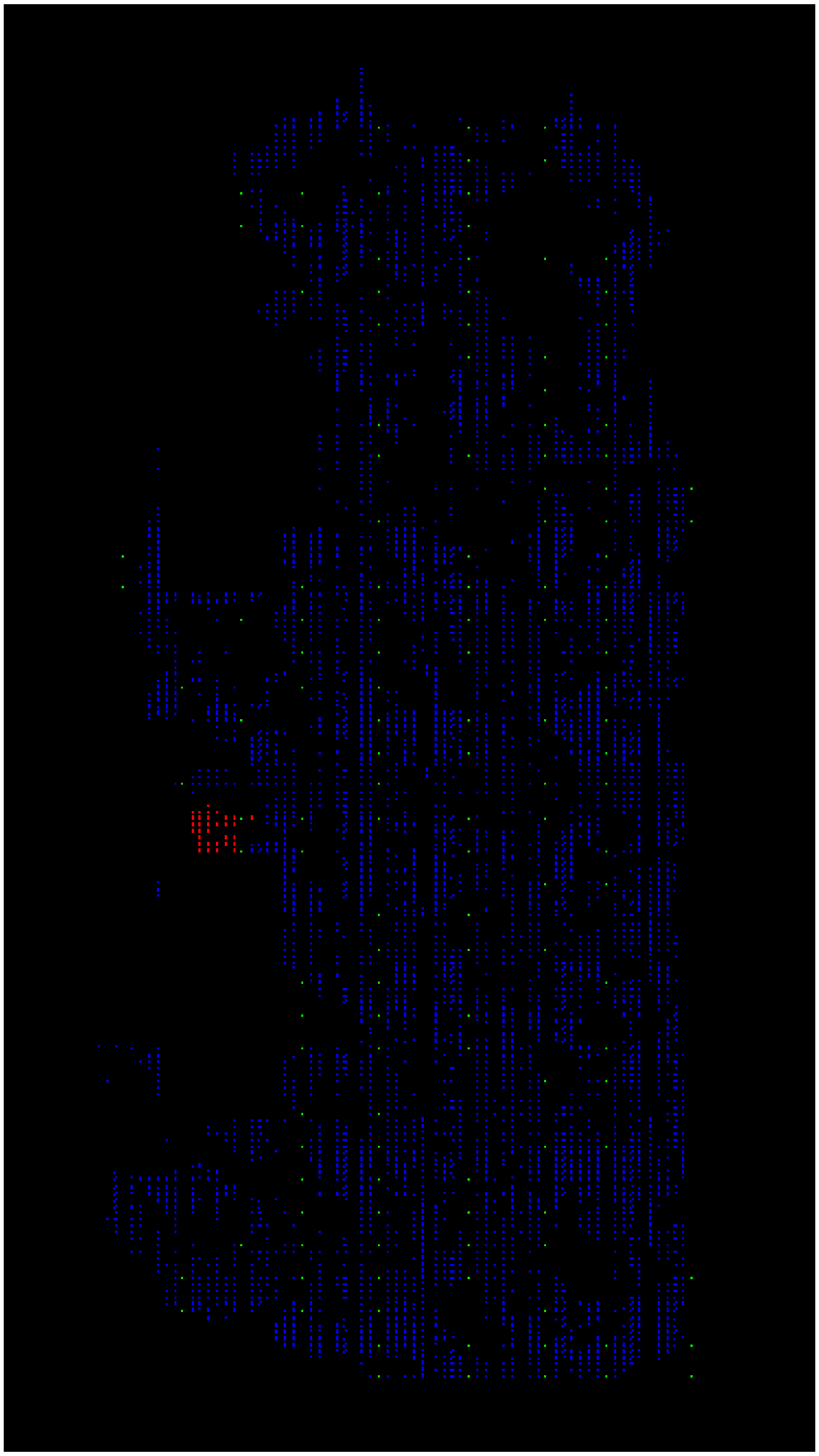

Figure A.1: FPGA Editor screenshot of the placement of bgm

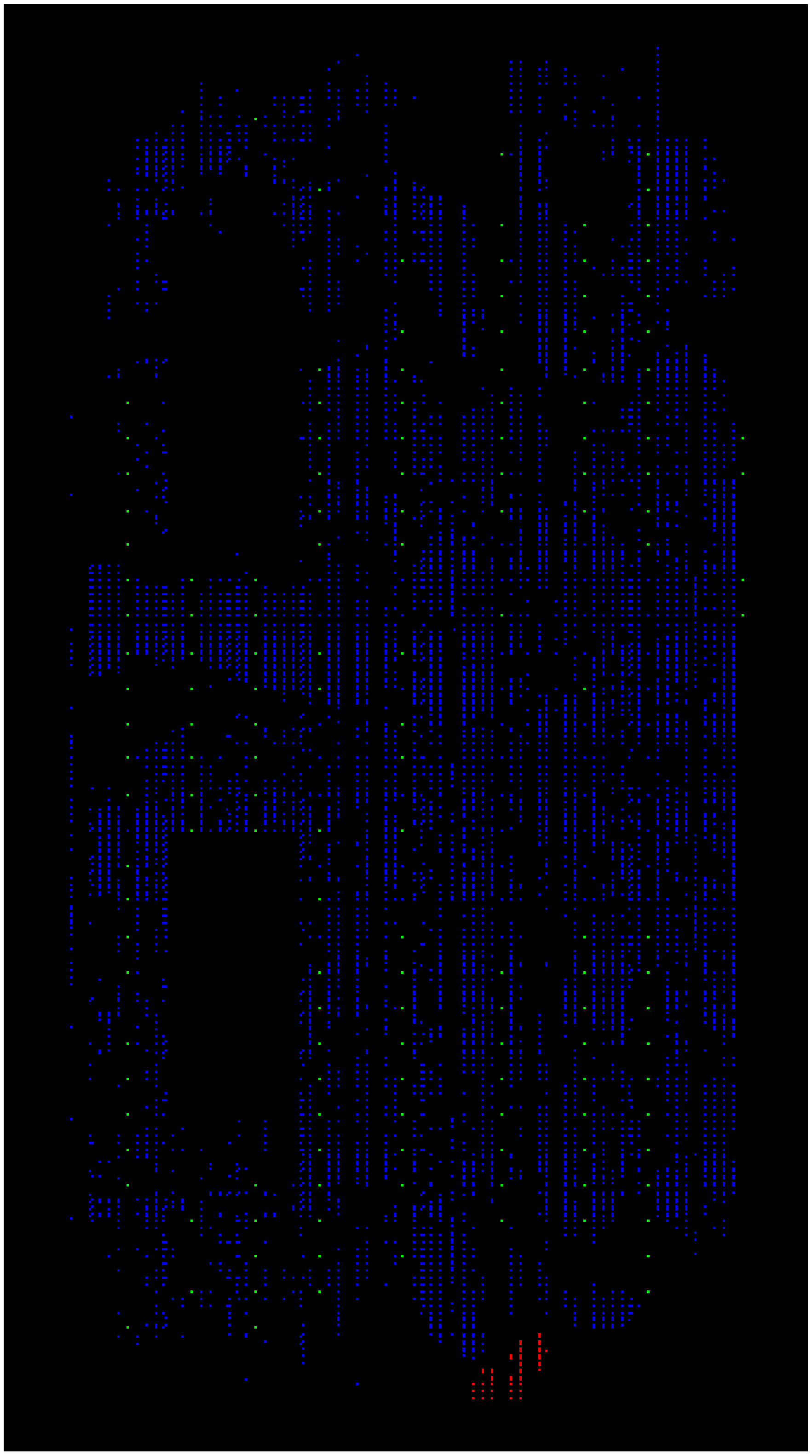

Figure A.2: FPGA Editor screenshot of the placement of LU8PEEng

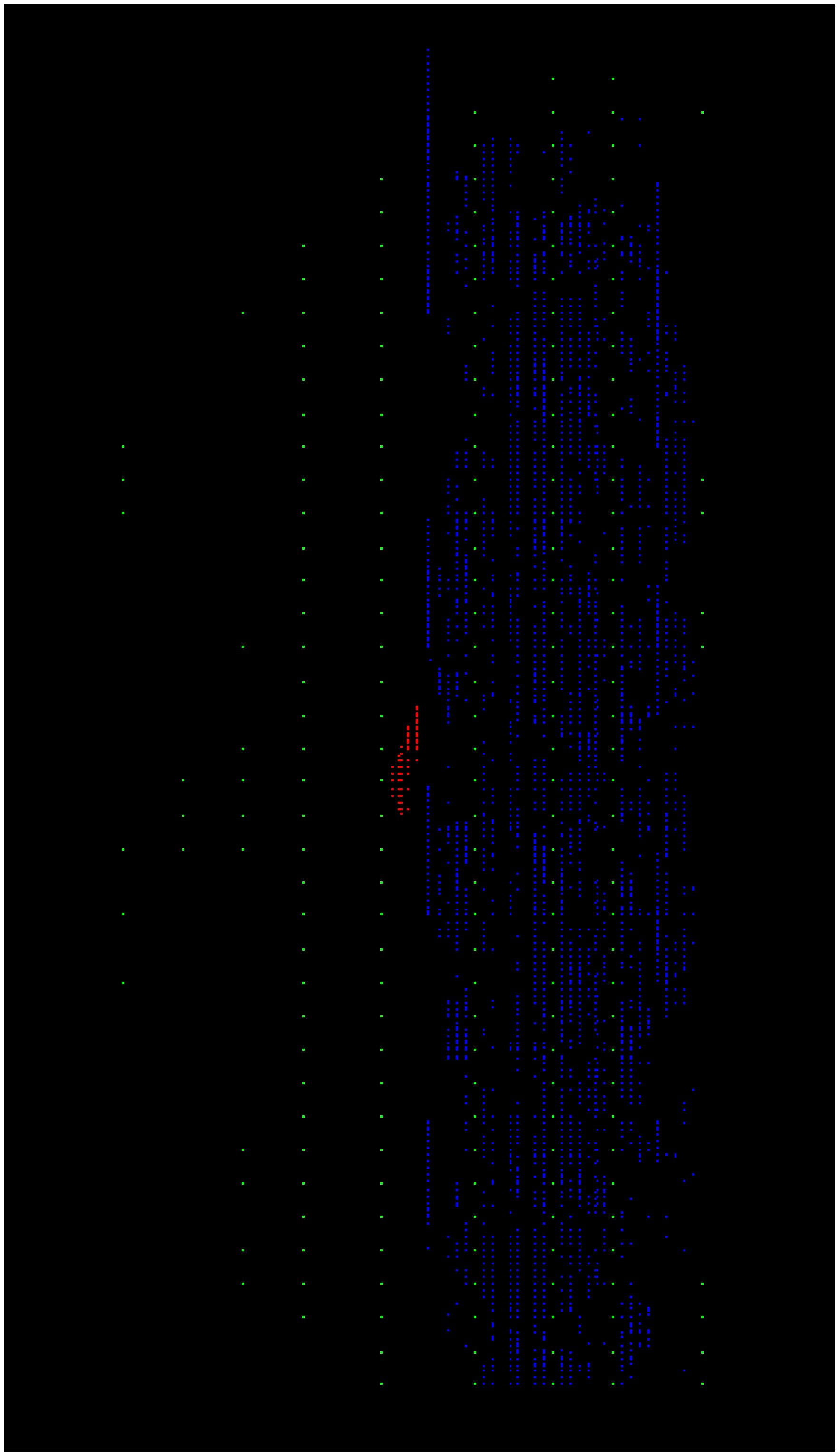

Figure A.3: FPGA Editor screenshot of the placement of stereovision0

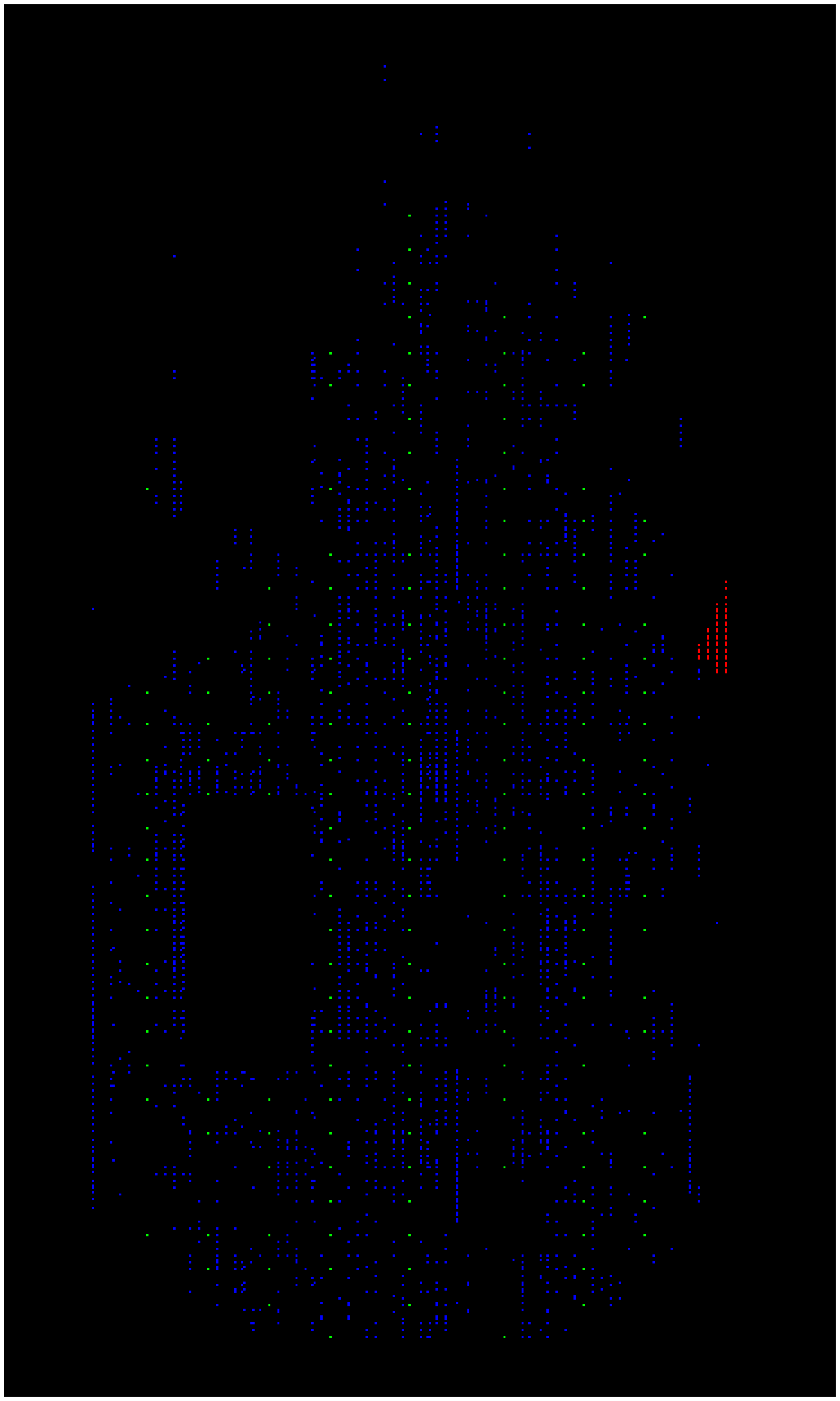

Figure A.4: FPGA Editor screenshot of the placement of stereovision1

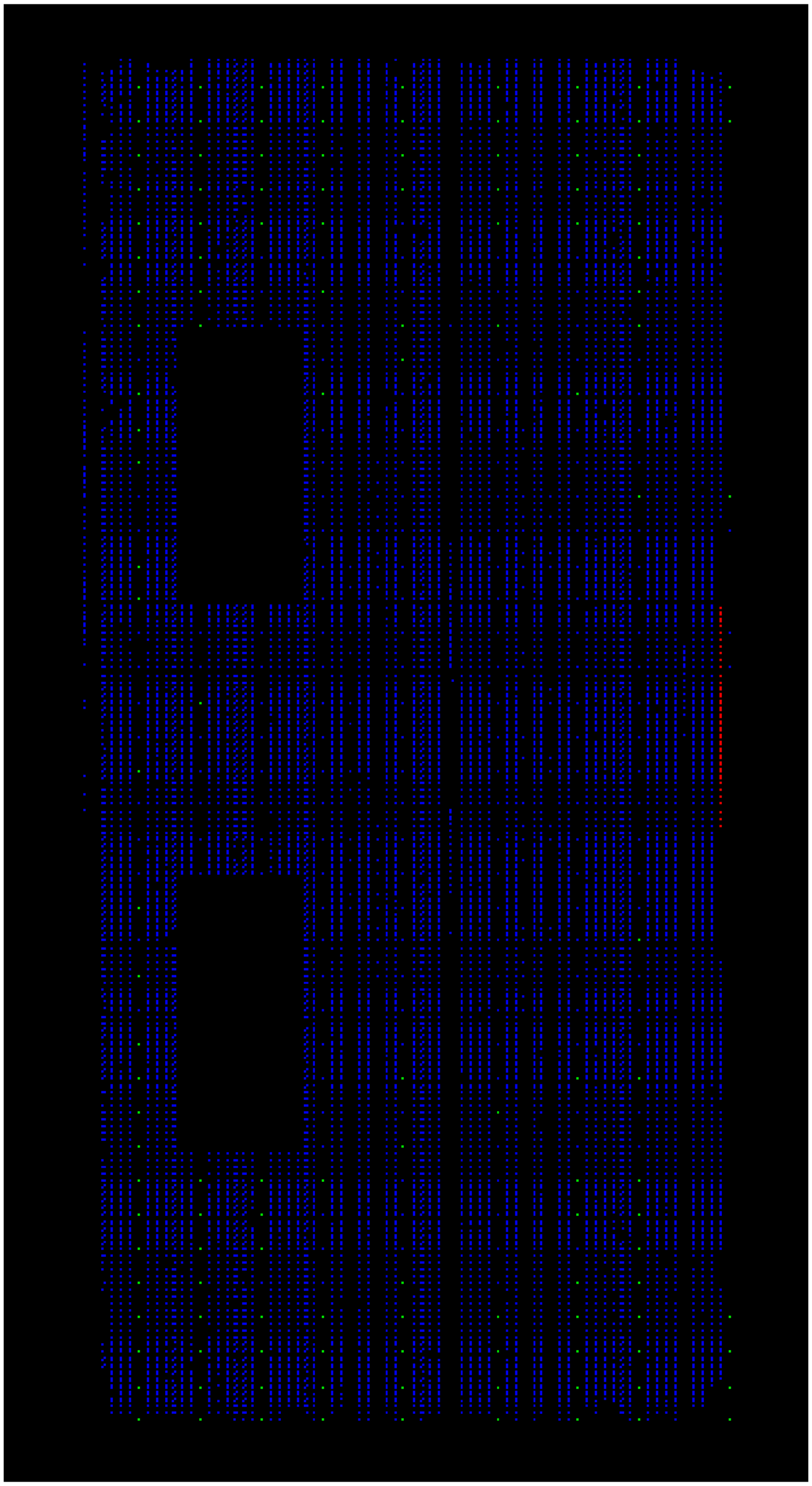

Figure A.5: FPGA Editor screenshot of the placement of LU32PEEng