Accelerating In-System Debug of High-Level Synthesis Generated Circuits on Field-Programmable Gate Arrays using Incremental Compilation Techniques

by

Pavan Kumar Bussa

B.Tech Electrical Engineering, Indian Institute of Technology Jodhpur, 2015

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

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2017

Abstract

High-Level Synthesis (HLS) has emerged as a promising technology that allows designers to create a digital hardware circuit using a high-level language like C, allowing even software developers to obtain the benefits of hardware implementation. HLS will only be successful if it is accompanied by a suitable debug ecosystem. There are existing debugging methodologies based on software simulation, however, these are not suitable for finding bugs which occur only during the actual execution of the circuit. Recent efforts have presented in-system debug techniques which allow a designer to debug an implementation, running on a Field-Programmable Gate Array (FPGA) at its actual speed, in the context of the original source code. These techniques typically add instrumentation to store a history of all user variables in a design on-chip. To maximize the effectiveness of the limited on-chip memory and to simplify the debug instrumentation logic, it is desirable to store only selected user variables. Unfortunately, this may lead to multiple debug runs. In existing frameworks, changing the variables to be stored between runs changes the debug instrumentation circuitry. This requires a complete recompilation of the design before reprogramming it on an FPGA.

In this thesis, we quantify the benefits of recording fewer variables and solve the problem of lengthy full compilations in each debug run using incremental compilation techniques present in the commercial FPGA CAD tools. We propose two promising debug flows that use this technology to reduce the debug turn-around time for an in-system debug framework. The first flow, in which the user circuit and instrumentation are co-optimized during compilation, gives the fastest debug clock speeds but suffers in user circuit performance once the debug instrumentation is removed. In the second flow, the optimization of the user circuit is sacrosanct. It is placed and routed first without having any constraints and the debug instrumentation is added later leading to the fastest user circuit clock speeds, but performance suffers slightly during debug. Using either flow, we achieve 40% reduction in debug turn-around times, on average.

Lay Summary

Designing modern digital electronic systems can be expensive and time consuming. Ensuring a design does not contain errors is especially challenging. This task, known as debugging, is often hampered by the fact that there is limited visibility into the internal operation of a circuit. Recent work has proposed methods to enhance this visibility. These methodologies often involve running the circuit many times; each run requires significant setup (compilation) time in which the design is automatically instrumented with different observation circuitry. In this thesis, we show how this compilation time can be dramatically reduced. The key insight is that the vast majority of the circuit does not change between runs; we present techniques that allow the compilation tool to focus only on the parts of the circuit that do change. This leads to significantly faster debug, potentially lowering the cost of producing working digital systems.

Preface

This thesis is related to the PhD dissertation of Dr. Jeffrey Goeders, a recent graduate from UBC who is now an Assistant Professor at Brigham Young University, Utah. He helped me in setting up the framework, answered all my technical questions and pointed me to the right place to start.

This work has been accepted as a poster paper titled "Accelerating In-System FPGA Debug of High-Level Synthesis Circuits using Incremental Compilation Techniques" at the International Conference on Field-Programmable Logic and Applications 2017 (FPL'17) and will be published in the conference proceedings. I was primarily responsible for conducting the research, performing the experiments and summarizing the results. This was done under the guidance of my advisor Dr. Steve Wilton. Dr. Wilton and Dr. Goeders also provided editorial support for all of my submitted works.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my advisor Dr. Steve Wilton for his constant motivation and support throughout my master's journey. This research would not have been possible without his guidance.

Special thanks to Jeff Goeders for helping me understand his work and providing the backbone of the framework for me to build upon. I would also like to thank other graduate students in Wilton's group for their suggestions and feedback especially during our weekly group meetings. My friends have played an important role in helping me maintain a good work life balance which was much needed at some times.

Financial support for this work has been provided by the National Sciences and Engineering Research Council of Canada, as well as from Intel Corporation.

Last but not the least, I would like to thank my parents for believing in me and providing everything I needed to reach this point.

Chapter 1

Introduction

1.1 Motivation

Recent years have seen a tremendous demand for faster computation as applications become larger and more complex. However, the performance of a processor has not been increasing fast enough to meet such computation demands mainly because of the difficulty in reducing the transistor sizes and because of the increased power dissipation. This has triggered the community to move towards parallel multi-core processing architectures where several processors/cores of same kind (*Homogeneous computing*) or different kinds (*Heterogeneous computing*) are used to gain performance or energy efficiency. Heterogeneous computing has shown promising results in situations where only a specific part of the application is to be accelerated. One way to accelerate algorithms using these heterogeneous systems is using a customized application specific processor. This is different than a homogeneous system where a general purpose processor would have to be used irrespective of the computing workload. However, manufacturing and designing an Application Specific Integrated Circuit (ASIC) like a custom processor is very time consuming and also the associated non-recurring engineering costs are high.

An alternative to using ASIC's is to use devices which are more flexible and easy to design while providing similar benefits to those of an ASIC. Field Programmable Gate Arrays (FPGAs) have emerged as reconfigurable devices with the capability of emulating any custom circuit, leading to performance gains over a wide range of applications. For every new application, an ASIC has to be manufactured from scratch while an FPGA could be transformed into any custom circuit just by reprogramming it. This makes the FPGAs very ideal for frequently changing applications.

Because of it's shorter time-to-market[72] and reasonable performance gains, many companies/academia have already started using FPGA's to accelerate their complex applications. Microsoft[64], Intel[39], IBM[27] and Qualcomm[65] have started using FPGA's to accelerate their mainstream computing. Recently, Amazon has started providing FPGA instances[2] in their cloud services which could be used on per-hour basis, making FPGA's accessible to everyone.

However, implementing a design on an FPGA is not as easy as implementing it on a processor. As with an ASIC, the design is specified in a hardware description language (HDL) like VHDL or Verilog/SystemVerilog which usually contain circuit descriptions at a much lower level (flip-flops/registers, logic blocks and the interconnections between them) known as Register-Transfer Level (RTL). This representation is later compiled and programmed to FPGA's by vendor specific Computer-Aided Design (CAD) tools. Doing this is a much more challenging and time consuming task than developing a software program for a given problem since it requires the designers to take care of the memory interfaces and the detailed scheduling of operations. As the designs become complex these approaches become tedious and are prone to errors if extreme care is not taken.

1.2 High Level Synthesis

High Level Synthesis (HLS) is an automated design process which converts a software-like program (usually written in C, C++ or Java) to a hardware/FPGA implementation. HLS raises the abstraction of the design specification to a much higher level eliminating the need for the designer to take care of the finer details like the scheduling, binding and memory interfacing which are now performed by the HLS compiler itself. A higher level of abstraction reduces the design time considerably and also makes it feasible for software developers to create a hardware implementation for their complete design or a part of their design. As software designers outnumber the hardware designers[60], in order to attract them towards FPGA's and increase their market sizes, leading FPGA companies like Xilinx and Intel have invested heavily in the HLS technology and developed commercial HLS packages like Vivado HLS[76] and SDK for OpenCL[41]. HLS not only makes it possible for software designers to use the FPGA's but also makes it easy for hardware developers to specify complex designs, improving design productivity.

Most existing HLS tools including the one considered in this work (LegUp[12]) use C as the means for design entry[57] and LLVM[47] as the C compiler. The front-end of the compiler converts the C design to an Intermediate Representation (IR) and the back-end produces an implementation specific to the target architecture. For this thesis, the target architecture is an FPGA and in this case the back-end performs allocation, scheduling and binding automatically in order to generate the HDL representation.

However, as the user is unaware of the HDL generated, he/she should have some means to verify the correctness of their implementation. In order for a HLS tool to gain widespread adoption it may need to provide a complete ecosystem like that of any software/hardware Integrated Development Environments (IDE) including the support for efficient debug.

1.3 HLS Debug

Bugs might arise at different stages of a HLS design process. Finding the root cause of a bug could be difficult without an effective debugging framework. HLS debugging techniques such as software based debug, RTL simulation and in-system debug could be used to help find the root cause of bugs that become visible at different levels of abstraction (from software program to a hardware implementation).

1.3.1 Software-like Debug

Most existing HLS tools offer the ability to debug a design by running the software code on a workstation. Standard debuggers like GDB[17] can be used to debug the software program. Logic bugs could be found easily through this method. However, in the final operating environment of the hardware circuit generated by the HLS tool, there may be other IP cores, processors or I/O devices interacting with each other and the HLS generated circuit. Many bugs related to the interfaces with these blocks that are not produced by the HLS may not be visible in a software based debug flow. Also, as this type of debugging is performed before the HLS process, it would not help the user to find any bugs arising from the HLS tool itself.

1.3.2 RTL Simulation

Some HLS tools offer debugging through RTL simulation [41][76]. This is very similar to the software-like debug technique except for the control and the dataflow which is now obtained from the simulation of the RTL generated by the HLS tool instead of the normal execution of the software code. This allows the designer to verify that the RTL circuit generated by the HLS tool matches the behavior of the original software code. However, hardware simulation is much slower than the actual hardware execution and would take a long time to simulate the design to that point where some bugs might occur. Also, as the RTL simulation is cycle-accurate, some of the transient/asynchronous events which might happen in between the cycle transitions cannot be determined.

1.3.3 In-system Debugging

For these bugs which cannot be found using the software debug or RTL simulation the only solution is to debug the circuit in the actual operating environment where it interacts with other blocks present in the system - also known as in-system debugging.

In-system FPGA debugging involves running the circuit at speed on the FPGA. At such speeds, the control and the dataflow information of the circuit are updated rapidly. Considering the throughput of these updates, it is almost impossible to show them to the user directly because of the limited I/O resources on the FPGA and also unlike the software debugging approaches it is not practical to run the circuit step by step, pausing in between to analyze the updated values. Even, if this was possible, reading the values from the FPGA each time the circuit is paused would be time consuming [68], which is not desirable. Because of these issues most in-system debugging tools adopt an alternative approach of trace-based debugging where additional circuitry (also known as debug instrumentation) is added to store the values of the signals until a set breakpoint is reached and then the circuit behavior is replayed with the help of the captured data. This instrumentation is built using the resources of the same FPGA on which the user circuit would be executing.

Embedded logic Analyzers (ELAs) such as SignalTap II [40] and Vivado's Integrated Logic Analyzer (ILA) [75] provide visibility into a hardware design by recording the circuit's execution in on-chip memories (also known as trace buffers) when a predefined trigger condition is met and then presenting the captured data in the form of waveforms. However this visibility is provided at the RTL abstraction level which makes sense only to those who can understand the underlying hardware thus not making a feasible debug option for software designers who may want to use the HLS tools.

Source-Level In-System Debugging for HLS

A software designer views the design as a set of sequential statements, while the generated hardware consists of several concurrently operating components. This mismatch between the software designer's view and the actual hardware running on the FPGA makes in-system debugging at the RTL level impractical. This becomes even worse if the HLS tool performs optimizations such as moving operations across cycle boundaries, leading to a schedule which might be unfamiliar to the user. To avoid these mismatches it is preferable to have an in-system debugging flow at the same abstraction level in which the design is implemented. This would also eliminate the need to understand the detailed implementation of their software in RTL, thus maintaining the productivity promised by HLS.

Recent work has presented debugging tools and instrumentation that allows designers to debug their hardware implementation as if it were software [10, 20, 29, 52, 55, 62]. These systems allow the user to debug code in an environment in which they are familiar, allowing them to single-step, set breakpoints, and examine variables. Critically, they allow the hardware to run at-speed, recording behavior in on-chip memory for later replay. The implementation described in [10, 23] was incorporated in the recent release of LegUp HLS which inserts custom debug instrumentation at the RTL level and maintains a database to relate the source code variables to the LLVM's IR signals to the hardware signals in the final Verilog generated in order to make the source-level in-system debugging possible.

To achieve a software-like debug experience, frameworks such as those in [20] record the history of *all* user-visible variables (or enough information that all user-visible variables can be reconstructed off-line). This provides the visibility that software designers expect, but it comes at a cost; since on-chip memory is restricted in size, only a limited portion of the circuit execution (the *trace window*) can be stored. In [21], the follow up work of [20], the authors focused on optimizing the on-chip memory utilization in order to store longer execution histories. The goal was to have longer execution histories which allows the user to find bugs easily without having to run the design multiple times (also known as debug iterations), recording a limited circuit execution each time.

1.4 Selective Variable Tracing

In this thesis we focus on a technique known as *Selective Variable Tracing* to achieve longer trace window sizes. Rather than storing a history for all user-visible variables, it is possible to only store the history of a *subset* of variables – perhaps variables in a function of interest or variables that the designer deems to be important. This could lead to much more efficient use of on-chip memory space, providing longer trace histories, possibly making it easier to determine the cause of observed incorrect behavior. In addition, recording fewer variables reduces the routing complexity and simplifies the compression logic which connects signals to trace memories, thus reducing the area of the debug instrumentation. Various automated signal selection techniques like those described in [46, 49] could be used to connect fewer interesting signals to debug instrumentation as implemented in [31]. However, in this thesis, signals were selected manually as the focus of this work is to show the impact of incremental compilation techniques in reducing the compilation time between the debug iterations and not about efficient ways of signal selection. Future work would be to incorporate these automated signal selection techniques and perform incremental debug using our proposed flows.

As debugging proceeds and the user refines his or her understanding of how the circuit is operating, the user may wish to change the subset of variables observed. In frameworks like [10, 21, 23] where the debug instrumentation is added at the RTL level, this would require a recompilation of the design (in order to reconfigure the FPGA on which the design was running) as the RTL generated would change each time the user changes the subset of variables to be observed. Recompilation is often slow, and may even lead to different place and route results, causing timing differences, possibly hiding an elusive bug (like those related to asynchronous interfaces). To be practical, we need a faster and less intrusive compilation path. To avoid the recompilation process, we propose using incremental compilation techniques.

1.5 Incremental HLS Debug

Using incremental compilation, the placement and routing of the user's design can be frozen between debug iterations, and only the instrumentation circuitry (which is added to gain observability and which depends on the set of variables to be observed) changes. This leads to faster debug iterations while at the same time maintaining the timing of the user design between debug iterations.

Incremental compilation techniques are well-supported in commercial FPGA CAD tools, and have even been applied to hardware-oriented debug infrastructures such as SignalTapII [37, 40]. However, there are at least four unique characteristics of the debugging framework considered in this thesis [23] that make a straightforward application of existing techniques sub-optimal:

1. Compared to hardware-oriented debug infrastructures such as SignalTapII [40], this framework contains much more instrumentation that needs to be recompiled in each debug iteration. When applying incremental compilation to SignalTapII, the incremental portion primarily consists of routing connections, while in this framework, the incremental portion primarily consists of a debug instrumentation which is a large design-specific custom compression circuit.

2. Although changing the set of variables to be recorded primarily impacts the instrumentation, it is also extremely intrusive to the user circuit, since a large number of "taps" are needed. Without careful consideration, this could dramatically increase the amount of logic that needs to be recompiled between iterations, limiting the effectiveness of the incremental techniques.

3. For many applications, it would be desirable to have the user circuit run as fast as possible, and not be slowed down by the presence of the instrumentation. This implies that it would be advantageous to compile the user circuit first, and only use "left over" resources to implement the instrumentation.

4. After debugging has been completed, it may be desirable to remove the instrumentation. Although it is possible to ship the design with instrumentation still present (but disabled) some security-conscious designers may worry that this creates a "back door" into the design. Therefore, our incremental flow has to support removing instrumentation without performing a complete recompilation, which would affect the timing of the circuit possibly exposing new bugs.

1.6 Contributions

In this thesis we adopt a source level in-system FPGA debug framework for HLS generated circuits [23], briefly described in Chapter 3 to implement our ideas. The following contributions have been made in this thesis to accelerate the HLS debugging flow for the considered framework.

1. We first quantify the impact of *selective instrumentation* on trace window size and debug instrumentation area by recording different 50% and 25% subsets of the variables present in the user's source code. This was done to get an idea of how much benefit could be achieved by using the selective variable tracing approach as this is complicated by the fact that a single user variable may correspond to multiple IR signals. If such variables are selected/unselected then all of it's IR signals would be recorded/ignored making the instrumentation insertion algorithm difficult to realize.

2. To avoid the recompilation process when doing selective signal tracing during each debug iteration, we propose using incremental compilation techniques. However, based on the unique characteristics of the framework considered [23] and the limitations of the incremental techniques present in commercial FPGA CAD tools as described in Section 1.5 several modifications had to be made to the RTL generated by the HLS including adding permanent taps to the user design and creating the design partitions to effectively use the incremental compilation flow.

3. We present two promising debug flows using commercial incremental compilation techniques which balance the compile time, design performance and area overhead while maximizing the amount of user data that can be stored in the on-chip trace buffer memories.

   (a) In the first flow, after the design partitions are created, the place and route for both the user and the instrumented debug partitions is performed simultaneously. This leads to the co-optimization of the user and the debug partitions resulting in the fastest clock speeds during debug. Once the debugging is done and the user decides to remove the debug instrumentation, it can be removed without performing a full compilation. However, the remaining user circuit would now run at a lower speed (when compared to the speed of the user circuit with no debug instrumentation) as it was not well optimized during the initial compilation because of the presence of the debug instrumentation circuitry.

(b) In the other promising flow, the design is first compiled with an empty debug partition which allows the user partition to be placed and routed without any restrictions, optimizing as much as possible to get better performance. Later, the debug partition is added incrementally preserving the placement and routing for the user partition from the previous compilation. This might lead to slower debug clock speeds as the debug instrumentation now uses spare resources which might be spread around the FPGA. However, when the debug instrumentation is removed after the debug is done, we could get back the actual speed for the user circuit which was obtained during the first compilation.

Using either flows, we are able to obtain an improvement of 40% in compile time per debug iteration as we now perform an incremental compilation for each iteration rather than a full compilation. We also achieve an increase of 1.6x and 2.6x in trace buffer window size when recording 50% and 25% variables respectively instead of recording *all* the variables. Together, this may lead to faster and more effective debug turns, resulting in higher productivity for designers creating complex FPGA applications.

1.7 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, the background required to understand HLS, the need for in-system debug (main focus on source level debug), the way incremental compilation works and how to use it effectively for debugging is presented. It is followed by a summary of recent research related to in-system debug, techniques to improve the trace buffer utilization in order to record longer execution histories and the use of overlays/incremental compilation in debug.

In Chapter 3, the source level in-system debugging framework used in this research is briefly explained. Different optimizations aimed at improving the trace buffer utilization are presented. Next, the advantages of selective signal tracing and the problems associated with it's practical implementation are discussed along with the proposed solution for using incremental compilation techniques.

Chapter 4 describes different incremental HLS debug flows proposed in this thesis to accelerate the debugging process by having faster and effective debug iterations. It also explains how the GUI from [23] is modified in order to automate the proposed incremental debug flows. Chapter 5 provides extensive results, comparing the different incremental flows proposed in terms of trace window size, area overhead, compile time and the user design performance both during and after debugging.

Chapter 6 concludes this thesis and suggests possible ideas for future work.

Chapter 2

Background

2.1 Overview

This thesis relies upon three major concepts: high level synthesis (HLS), in-system debug and incremental debug. In this chapter each of these is briefly discussed to provide the background required to understand this work. Also, the previous works related to these areas are presented.

This chapter is organized as follows. Section 2.2 describes the steps involved in an HLS flow followed by description of the LegUp [12] HLS framework which is used in the research described in this thesis. Section 2.3 emphasizes the need for in-system debug and summarizes some of the previous research in this field. Section 2.4 describes the incremental compilation features present in FPGA CAD/EDA tools along with their support for incremental debug and it also describes the debugging frameworks which make use of overlays/incremental compilation techniques to accelerate the debug flow. Section 2.5 describes how this thesis differs from the related works and concludes this chapter.

2.2 HLS Flow

HLS has simplified the design process of a digital system by allowing the designers to specify the design requirements at a higher abstraction level. The CAD tools associated with this process automatically converts this representation to a RTL level specification optimized for performance, area and power requirements, which could later be used as a reference for manufacturing an ASIC or configuring an FPGA to emulate the required design. Because of an FPGA's shorter time to market and other benefits when compared to an ASIC [72], several companies have either started using FPGA's to accelerate their complex workloads or started providing FPGA services for the customers to meet their needs. Amazon has recently announced their EC2 F1 instances [2] which allows the users to pay for the instances by the hour without the need for buying an FPGA. Microsoft's Catapult project to accelerate their cloud services [64] using FPGAs, Intel's acquisition of a leading FPGA manufacturer, Altera [39], IBM and Qualcomm's interest in using FPGA's to accelerate their cloud services [27, 65] hint towards the increasing demand for FPGA's in near future. A recent survey on HLS tools [57] presents and evaluates different tools that have emerged from both industry and academia [12, 36, 50, 56, 58, 61, 76] to allow both hardware and software designers (who outnumber hardware developers [60]) to use FPGA's for accelerating their workloads and also to improve the user productivity by eliminating the need to focus on timing and other interface-related details.

A typical HLS flow is shown in Figure 2.1. As discussed in Section 1.2, the majority of HLS frameworks use synthesizable subsets of C for specifying a design at an algorithmic level mainly because of its simplicity and widespread adoption, with LLVM as a C compiler. Any HLS framework can be functionally categorized into a frontend, optimizer and a backend. The frontend compiles the high level behavioral representation of a design (C source code) and converts it to an Intermediate Representation code (IR) or a Control and Data flow graph (CDFG) [13]. The optimizer then performs several code optimizations such as dead code elimination, false data dependency elimination, function in-lining and loop transformations. Typically, the tool goes through multiple passes, primarily to reduce the number of resources required on the target architecture and increase the speed of the design.

The backend consists of a series of steps: allocation, scheduling, binding and RTL generation. Together these steps convert the optimized intermediate representation code into a HDL specification. During *Allocation*, the tool determines the type and number of hardware resources (such as functional units or memory components) required to satisfy the design constraints. In *Scheduling*, the operations assigned to each of the functional units are scheduled into cycles based on dependency analysis. Each operation could be scheduled for one or several cycles depending on the functional unit to which it is mapped and also based on the complexity of the operation. *Binding* assigns the operations present in the IR code to specific functional units and also optimizes the resource utilization by allowing non-conflicting operations to share the same functional units by multiplexing them.

The order in which allocation, scheduling and binding are performed may vary depending on the algorithms used by the HLS tool and the given design constraints– area minimization or latency minimization [13]. All of these steps are highly inter-dependent and some of them may be performed simultaneously based on the tool's objectives. For example, scheduling tries to reduce the number of control steps required, subject to the number of available hardware resources which depend on the result of allocation. Once these steps are performed and the optimizations are applied, the final step is to generate RTL from the intermediate data structures which were used to store the decisions made in the previous steps.

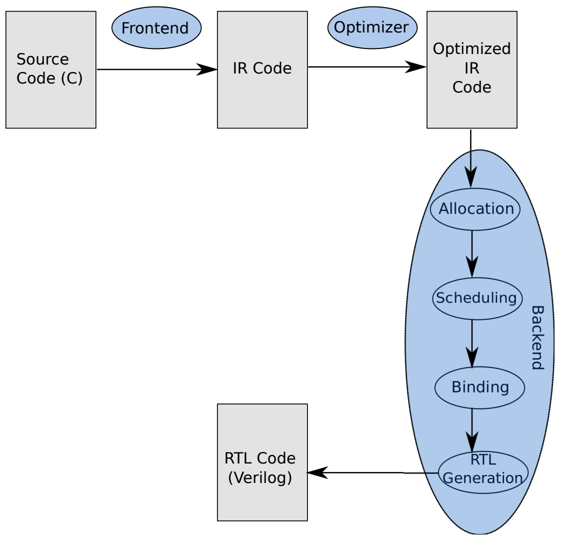


Figure 2.1: HLS Flow

LegUp HLS

LegUp is an open source HLS framework developed by the University of Toronto [12] for research/academic purposes. It takes standard C as input (without recursive functions or dynamic memory allocations) and automatically generates an RTL level implementation (Verilog) which could be synthesized for some of the Intel and Xilinx FPGA's. LegUp is not only capable of generating a pure hardware but can also generate a hybrid hardware/software system by first running the design on a FPGA based MIPS processor (Tiger MIPS [59]) and profiling the execution to determine the program segments that need to be accelerated on a hardware while running the rest of the code on processor itself.

LegUp uses the low level virtual machine (LLVM) [47] as the frontend compiler to convert the C code to IR, which is then modified by a series of optimization passes. This modified IR, with the help of newly created backend passes schedules the IR instructions into specific cycles and finally generates the RTL based on these descriptions.

LegUp uses a System of Difference Constraints (SDC) [11] approach for scheduling and the Bipartite Weighted Matching [30] technique for the binding step by default. However, the user could implement his/her own algorithms for each of these passes.

The latest release, LegUp 4.0 also includes a source level in-system debugging framework which allows the users to experience a software-like debug experience for a hardware implementation [21]. In this thesis we use the same debugging framework to prototype our ideas, which are described in Chapter 4. Some minor modifications had to be made to LegUp which are described in Appendix A.

2.3 HLS Debugging Techniques

To verify the correctness of the generated RTL, HLS tools should be able to provide a debugging platform. Only then will HLS be widely adopted. There are different types of debugging approaches for a HLS design flow:

software-like debug, RTL simulation and in-system debugging. Each of these are capable of detecting bugs arising at different level of abstractions and were discussed briefly in Section 1.3. In software-like debugging and RTL simulation techniques, it is difficult to provide the exact system inputs and replicate the final operating environment of the circuit, making some elusive bugs invisible. The most accurate and natural method of debugging would be to run the circuit generated at speed on an FPGA and then analyze it's execution with respect to the source code. Several works have been published which leverage the advantages of in-system debugging using different techniques. They can be mostly classified into scan-based or trace-based approaches. In a scan-based approach the circuit execution is paused and the circuit state is retrieved. In the trace-based approach the circuit execution is recorded in a on-chip trace buffer and the execution is replayed later for debugging purposes. The following subsections describe several of these approaches which are relevant to this thesis.

2.3.1 In-system Debugging Approaches

Using External Probes

With the help of a simple debug code, selected RTL signals in a circuit could be connected to the FPGA I/O pins and then using an external analyzer it would be possible to collect the signals from these pins and display them as waveforms for the user to analyze. However, because of the limited I/O resources on an FPGA, it is not feasible to collect the data generated by the circuit in each cycle (especially when wide signal busses are to be observed) [51], limiting the in-system debugging capabilities of such approaches. Even if it was possible to collect the required data it would not make much sense to a HLS user as the design is being developed in C and they may or may not have the knowledge of the circuit implemented on the FPGA. Therefore debugging at RTL level without being aware of the correspondence between the source code variables and the RTL signals is not practical for an HLS user.

Scan Based Debugging

In this approach, a scan chain is created by connecting the internal flip flops in a user design sequentially to a JTAG interface, allowing the user to observe the values stored in the flip flops. This is very commonly used for ASIC's where using a scan input pin, test inputs are applied and the flip flop values could be scanned out through the scan out pin, providing observability into the circuit. Some FPGA's also provide scan chains and enable reading the state of the internal flip-flops through their *readback* feature [38, 74].

Several works have used scan chains for debugging purposes [3, 43, 68–71]. The benefit of this approach is that no additional on-chip memory is required to record the circuit state. If the scan chains are not present then implementing them would take up some resources [71]. In order to read the state of the flip flops in the scan chain, the circuit must be paused which might affect the interactions among some of the blocks, potentially altering the circuit state when it is resumed. In addition, as the debugging is performed at the RTL level, it would not be beneficial to use this with a HLS design flow unless there is a way of mapping these signals back to the source code variables.

Embedded Logic Analyzers

A logic analyzer is a customized hardware unit which has the capability to capture and store selected signals from a user circuit based on predefined trigger conditions. These signals are stored continuously cycle-by-cycle in a ring-type trace buffer as long as the trigger conditions are satisfied. Later, the recorded signal values are retrieved from the buffers for further analysis. Advanced logic analyzers can store the signals using segmented buffers and multiple trigger conditions, where the signals are recorded until one trigger condition is met, the recording stops when another trigger condition is activated and is resumed again when a different trigger condition is met and so on. Trigger conditions are used to start/stop the recording of signals in order to use the trace buffers effectively.

Many commercial logic analyzers have been released by FPGA companies for their devices including SignalTap II [40] from Intel, Integrated Logic Analyzer (ILA) [75] from Xilinx and also by some third parties like Synopsys (Identify RTL Debugger [67]) which provides support for Intel, Xilinx and Microsemi devices. These logic analyzers are added automatically by the corresponding FPGA Electronic Design Automation (EDA) tools and are implemented on the same FPGA fabric as that of the user circuit. Most of the EDA tools interact with the logic analyzers through the JTAG port, in order to simplify the interface protocol. Usually, the signals are recorded cycle-by-cycle in a circular trace buffers, overwriting the previous values until a trigger condition is met. The EDA tool then reads the captured data and displays the signals in a waveform representation. As these logic analyzers have to be compatible with any circuit, the instrumentation added is simple and would record all the selected signals in each cycle, even though some of the signal values do not change.

Even in these trace-based approaches, the signals displayed would correspond to those at the RTL level; this requires the user to have an understanding of the mapping between the source code and the generated RTL in order to get the most from these waveforms. Also, as the signals are being recorded in every cycle, the on-chip trace buffers would be able to record a limited circuit execution making it difficult to find those bugs whose effects are only noticeable after some time lapse.

All of the above described in-system debugging approaches provide sufficient observability into a circuit's execution at the RTL level. However, as described in Section 1.3.3, for an HLS user it would only be meaningful if the debugging was performed at the same abstraction level where the design is specified, i.e. at the source (C code) level. The following subsections describe the related works which focus on providing source level in-system debugging opportunities for a HLS generated circuit.

2.3.2 Source Level In-system Debugging Approaches

Sea Cucumber Debugger

Sea Cucumber (SC) [34], is a circuit synthesizer which takes an input behavioral description written in its own programming model (based on Java threads) and generate a circuit implementation (JHDL [8]) that could be executed on an FPGA. The JHDL representation could be converted into a netlist that could be programmed to an FPGA through JHDL frameworks. These frameworks provide in-system debugging facilities at the JHDL level by using the readback features available in some FPGA's.

Hemmert et al. proposed a source level debugger for the SC framework in [29]. This debugger used the scan based approach as described in Section 2.3.1 for communicating with the FPGA. To be specific, it used the readback feature of an FPGA to capture a snapshot of the circuit state by freezing the clock or in other words, pausing the circuit execution. However, this information was remapped to the source code with the help of a database containing the mapping information between the source code variables, IR values and the circuit level signals. This database was created during the synthesis of the circuit and also had the information about the optimizations performed by the compiler. This allowed the user to have a software-like debug experience while having a well-optimized circuit running on an FPGA. It also provided support for single-stepping, breakpointing, watching and setting the values for variables- providing both observability and controllability. The major drawback is that the circuit had to be paused after every instruction to allow effective debugging, which is the same problem with that of any other scan based debugging approaches described in Section 2.3.1.

Event Observability Ports

In [53], Monson and Hutchings describe a trace based debugging approach and a new technique to improve the trace memory utilization in order to have longer traces of the circuit execution, which is a major concern for any of the trace-based in-system debugging approaches [10, 20, 40]. In this method top level ports were manually added to the RTL generated to obtain information about what and when the signals should be recorded unlike an ELA where the signals are recorded every cycle. These were called Event Observability Ports (EOP) and they consisted of a data and an enable signal. The enable was set high only when the corresponding signal value was updated. These ports were added to the relevant signals which were intended to be traced by the user and were connected to the trace buffer memories, thus recording the data signal only when the enable is high. This work also suggests the use of multiple trace buffers instead of a single buffer to have more flexibility and more possible optimizations.

They used Vivado HLS [76] to generate the RTL from the user specified C code, then manually instrumented the RTL and finally programmed it to a Xilinx FPGA to run the circuit at speed. They also maintained a mapping between the C code and the RTL to provide a source level debugging experience for a user. Clearly, to do selective signal tracing with this approach, the RTL would have to be modified accordingly to add new ports. This would lead to a full recompilation of the RTL code, in order to reprogram the FPGA with the modified design, which is very time consuming. Our work investigates incremental solutions to avoid these lengthy compilations.

LegUp Debugger

As mentioned in Section 2.2, the latest release of LegUp includes a trace/instrumentation based source level in-system debugger which was developed combining the ideas from the following two similar works:

Inspect Debugger In this work [10], a source level debugger with the capabilities of any other software debugger (gdb [17]) such as single stepping, breakpointing and variable inspection was proposed for the LegUp C-to-RTL synthesizer. There was no controllability provided i.e., a user could only view the values of the variables and does not have an option to update the variable values using this framework. Like the SC debugger [29], a debug database was maintained to relate the mapping of a source code variable all the way to the optimized RTL level signals.

It provided two different modes of circuit execution. First, it allowed debugging at the source level using RTL simulation (using ModelSim simulator). In this mode, single stepping through a line in the source code corresponded to simulation of the circuit for a specific number of cycles (this information is obtained from the debug database). In the other mode, it added SignalTap II [40] logic analyzer to the RTL generated from the user design to record the selected signals in on-chip memories while running the circuit at speed. Once the trigger condition is satisfied, this data is read by the GUI and displayed to the user in the context of the source code. It also provided support for SW/HW discrepancy detection by running the software or RTL simulations in tandem with the hardware execution and comparing the variable values in both the cases as they execute.

The major limitation of this approach is the use of an ELA to record a trace of the circuit's execution. As described in Section 2.3.1, this approach records selected signals in every cycle leading to poor utilization of the on-chip memories restricting the debugging ability in any given debug iteration. Moreover, in order to do selective signal tracing to have better trace lengths as described in Section 1.4, it requires a recompilation of the circuit as the RTL generated would be different for different subset of signals being recorded. This is time consuming. The major goal of this thesis is to target this problem.

Goeders's Debugger This work [20] is very much related to Inspect [10]. The major difference is that instead of using an ELA it uses its own customized debug instrumentation. This circuitry records the signals only in those cycles when they are updated (using similar techniques from [53]) and reduces the amount of control and information to be recorded by performing several optimizations. Follow-up works [21, 23] show how it leverages the scheduling information from the HLS compiler to perform dynamic signal tracing and how it uses the signal restoration techniques to improve the on-chip memory (trace buffer) utilization by a significant factor when compared to a standard ELA. However, similar to Inspect [10], changing the variables to be observed between debug iterations require a recompilation of the full design as the RTL generated would now be different– taps into the user circuit change and also the debug instrumentation is changed as it is highly customized based on the variables being recorded.

In this thesis, it is this framework which was modified to implement our proposed ideas. The optimizations used and its limitations are described in more detail in Chapter 3.

Using Source Level Instrumentations

In this approach, the debug instrumentation is added at the source level by modifying the C code. This is different than the previously described works [10, 20, 21, 29, 53] which insert the debug instrumentation after the RTL is generated by the HLS tool. Monson and Hutchings, in their recent work [55] use this approach to produce the EOP's [53] (also described in a previous subsection) automatically in the RTL generated which could later be connected to a trace buffer. This was possible by assigning the required variables/expressions in the source code to newly created top level pointers which would finally be converted into top level ports by the HLS tool. In their follow up work [54], they add support for adding EOP's for pointer variables using shadow pointers. Pinilla and Wilton's work [62] was built upon [55] and described a way to instrument even the trace buffer and the associated circuitry at the source level which might allow for better optimizations. It also proposed an Array Duplicate Minimization (ADM) technique which improves the trace buffer utilization by using the values of an array variable from the user circuit memories itself (whenever possible) while reading back the trace data removing the necessity to record them in the trace buffers.

These works provided in-system debugging capabilities for an HLS user at the source level by adding instrumentations at the source level. However, to leverage the use of selective variable tracing as described in Section 1.4, the source code had to be changed. This means that the HLS tool must compile the code after instrumentation. This requires a full compilation by the EDA tool to generate the bitstream for configuring the FPGA. If the instrumentation was added at the RTL level, then the RTL corresponding to the user circuit would never change and only the RTL for the debug instrumentation might be changed in each debug iteration. This could be compiled incrementally by some existing EDA tools by following proper techniques, which we describe in the following subsections. If the instrumentation was added at the source level, however, this would not be possible as the source code itself is changed and there is no guarantee that the RTL corresponding to the user circuit is the same as before.

In-System Assertion Based Verification

A common approach for debugging a software application is to use assertions. These assertion checks could be used for C code which is to be converted into a RTL implementation by the HLS tool, however it would not be helpful for finding those bugs which occur only during the actual execution of the circuit. Works like [14, 26] make the assertion based verification practical for an HLS application by actually synthesizing these assertion statements into assertion checker circuits (which are implemented on the same FPGA as the user circuit) and notifying the user of assertion failures by verifying them with the actual execution of the circuit running at its speed on an FPGA. Clearly, as the number of assertions increase, the area utilized by the assertion checkers would also increase which is usually not desirable. Also, if the assertions had to be changed, it would require running the HLS flow again along with a full compilation in order to reprogram the FPGA. Our work focuses on avoiding such full recompilations between successive debug iterations using incremental compilation techniques.

The majority of the in-system HLS debugging approaches described in the previous subsections [10, 20, 21, 53, 55, 62] were trace-based, i.e. they use on-chip trace buffers to store the circuit execution and later replay it to the user, providing the same debugging environment as that of a software simulation. Some of the works [21, 53] also tried to improve the trace buffer utilization by proposing different optimization techniques. However, by using selective variable tracing, the trace length could be much higher since fewer variables are recorded. This approach may require running the design several times, recording a different subset of variables each time. Doing this requires a recompilation of the RTL. A few approaches have been recently published and are described in Section 2.4 which try to reduce the recompilation time between successive debug turns.

2.4 Incremental debugging Approaches

Incremental debugging Using Commercial EDA Tools

Commercial EDA tools like Intel Quartus Prime and Xilinx Vivado have incremental compilation features [42, 73] which allow the user to preserve placement and routing for the unchanged portion of a design from previous compilation. With the help of these features, SignalTap II [40] and the Vivado ILA ELA's [75] can be used to perform incremental debugging. The following paragraphs describe the steps to be followed to achieve this.

Using Intel's SignalTap II: Incremental Debug can be performed by using SignalTap II with the help of Quartus Prime's Incremental Compilation techniques. In general, to perform an incremental compilation in Quartus Prime, the project has to be divided into design partitions [40] (by default it has one top partition which encompasses the whole design) and the preservation level for these partitions must be set to POST_FIT; this directs the tool to use the place and route results for this partition from the previous compilation. If it doesn't exist, then the tool performs a full compilation for that partition. The idea is to place and route each design partition separately in the first compilation, avoiding any cross partition optimizations. This makes it possible to make changes in one or more design partitions and just compile those partitions while preserving the details for other unchanged partitions, leading to a significant reduction in compile times. We use this incremental compilation technique in our debug flows (refer to Chapter 4) to allow in-system source level incremental debugging for HLS generated circuits. Quartus Prime also provides Rapid Recompile, in which creating the design partitions is not mandatory and the tool automatically tries to preserve the placement and routing from the previous compilation, as much as possible. However, through a series of experiments we determined that this option is optimized for minor changes and was not effective for the amount of changes that occur when we modify our debug instrumentation. Hence, we do not consider it for our work.

In a normal debugging flow using SignalTap II, the user has to first determine the signals to be observed (which are visible after running an Analysis and Elaboration step or a full compilation of the user design), define trigger conditions and other configuration details in a SignalTap file (.stp file in Quartus Prime) and add it to the project. Then, a full compilation must be done in order to insert the SignalTap II IP core into the design. The user circuit and the SignalTap II logic are treated as a single design partition and placed and routed simultaneously without any restrictions. Therefore, if the signals to be recorded are changed, the SignalTap II logic and hence the whole design partition is considered changed, requiring a full recompilation.

To perform incremental debugging using SignalTap II, Incremental Compilation for the design should be enabled by changing the preservation level to POST_FIT for the default *top* partition or by creating other partitions if required and changing the preservation settings accordingly. Next, the same steps are to be followed as described in the earlier paragraph to create and add a .stp file to the project. When a first compilation is run the tool now considers the SignalTap II logic as a separate design partition and isolates it from the user design partitions. If the signals to be observed are changed or the trigger conditions are changed, then the tool would try to incrementally route the new signals to the SignalTap II logic. If this routing was somehow not possible then it would had to recompile the user partition too. This causes only the SignalTap II partition to change while preserving the user design partitions, thus reducing the compilation time significantly.

However, when we inserted a customized debug instrumentation instead of the SignalTap II ELA core and performed an incremental compilation (using Quartus Prime) changing the signals to be observed, the user design partitions were also considered changed as the debug taps into the user circuit were changing each time. This is described in Chapter 4 along with our proposed solution to avoid such situations. When SignalTap II is used, the taps are added/changed automatically by the tool and it does this after preserving the placement and routing for the user design partitions from previous compilation, however when customized debug instrumentation is added the taps need to be changed at the RTL level itself by modifying the ports of certain design partitions (since we can't modify the internals of the tool). Moreover, the debugging would be at the RTL level as the information gathered by the EDA tool (Quartus Prime) would correspond to the RTL signals (as described in Section 2.3.1). In this thesis, these techniques are applied to a source level debugging flow [20] to make the debug process easier, efficient and faster for an HLS user.

Using Xilinx's Vivado ILA: Xilinx's Vivado design suite also provides support for incremental compilation. It does not have the concept of design partitions, but instead uses a design checkpoint file (DCP) as a reference to preserve the data for placement and routing from any of the previous compilations. After a full implementation is run for a design, a new incremental implementation can be started after making any changes to the design and providing the routed design from the previous implementation as the DCP file. This uses the placement and routing for the unchanged logic from the DCP file and incrementally places and routes the changed portion of the design.

This feature can be used along with the ILA [75] to perform incremental in-system debugging. In the first implementation, an ILA debug core can be added to the design and interesting signals can be selected for observation. In the subsequent implementations, if the set of signals selected is changed then an incremental implementation can be run by preserving the placement and routing from previous compilations using a checkpoint file. Once the implementations are run, the recorded RTL signals can be read from the debug core and interpreted as waveforms using Vivado's interface. If this is to be used with the RTL generated by the HLS tool then the debugging would be at the RTL level and additional mapping information would be required to relate these signals to the source code variables in order to make it more meaningful for an HLS user.

Using RapidSmith

RapidSmith [48] is an open-source set of tools and API's that could be used to perform any stage of a CAD flow such as placement or routing for Xilinx FPGA's. This is possible by reading the intermediate results from the Xilinx Vivado design suite after a particular stage (using XDL [7]), performing the required tasks using RapidSmith and then communicating the results back to the Xilinx Vivado design suite, which could complete the rest of the stages in a CAD flow and finally generate the bitstream for an FPGA. Using RapidSmith, an incremental place and route tool could be created based on the user requirements instead of using the commercial incremental flow from Vivado, in order to have fine-grain control over the incremental algorithms. However, implementing this is very time consuming and lacks proper documentation/support as this is known to a smaller community when compared to that of a commercial tool.

The work by Hutchings and Keeley [35] is closely related to this thesis. They use RapidSmith to incrementally add or modify the debug instrumentation after performing the place and route for user circuit using the left over resources. Their instrumentation is simple and consists of a small number of trace buffers with a small trigger circuit. When inserting this instrumentation, the required RTL signals are connected (routed) to the trace buffer and the trigger circuit inputs. The trace buffers would record the value of the signals connected to them in each cycle as there is no additional compression/scheduling logic inserted, resulting in a poor utilization of the memories. In the subsequent incremental compilations when the set of signals to be traced is changed, only a few signals have to be re-routed. If the trigger circuitry needs to be changed then a small amount of logic had to be re-placed and re-routed, leading to a significant reduction in the compile times.

In this thesis, we implement the idea of adding the instrumentation after the place and route of user circuit in one of our debug flows (in order not to disturb the user circuit) but our work is different from their's [35] in the following aspects: 1) This thesis focuses on source-level incremental debug for the HLS generated circuits rather than RTL level. 2) We use a commercial incremental flow available in Intel Quartus Prime which is mature when compared to that of an open-source tool (Rapidsmith). 3) Our instrumentation that needs to be changed during an incremental recompile is much more than just adding a few routes and the work in [35] does not quantify how effective their incremental algorithms are for recompiling such significant amount of changes incrementally.

Using Debug Overlays

A debug overlay is a virtual customized fabric which can be implemented on the same FPGA along with the user circuit during debugging. This overlay could be compiled simultaneously with the user circuit or could be compiled to the FPGA after the place and route of the user circuit (in which case the presence of the overlay would not perturb the user circuit). The idea is to create an overlay architecture which could be quickly configured to implement the required debug instrumentation circuitry during each debug iteration. This avoids recompilation of the whole design and could be faster than incremental debugging using ELA's as the amount of logic that has to be reconfigured/recompiled in case of overlays is usually much smaller when compared to recompiling the whole debug partition as in the ELA approach.

Hung and Wilton proposed an virtual overlay network for trace buffers [33] in which they used the leftover routing resources to create an overlay where all the user signals were connected to the inputs of at least one trace buffer. After compiling this overlay to an FPGA, during debugging the user can select a subset of signals for tracing (limited to number of the trace buffer inputs). In the subsequent iterations the user can change the signals he/she wishes to observe and this would just require the reconfiguration of routing multiplexers internal to the overlay. Such reconfiguration can be done using partial reconfiguration or bitstream modifications (as in [24]) within a few seconds. This work does not implement any trigger circuitry or other compression logic, which leads to poor utilization of the trace buffers. The follow up works by Eslami and Wilton [15, 16] presented an overlay architecture for inserting trigger circuits incrementally. They used the leftover logic blocks and routing resources to create this overlay which was compiled to the FPGA after the routing of the user circuit. While debugging, a trigger function is mapped to this overlay using their own routing aware placement algorithm (because of the limited routing flexibility in the overlay). If the trigger function has to be changed then it is remapped to the overlay. This is faster than the incremental recompilation of the whole debug partition directly to an FPGA without an overlay. In all of these overlay works [15, 16, 32], the debugging is done for the circuits described at RTL level and its feasibility has not yet been investigated for debugging HLS generated circuits.

However, creating an overlay architecture supporting a wide range of trigger circuits and trace signal connections would be expensive in terms of area and sometimes there may not be enough leftover resources to build the overlay. Moreover, as the size of the overlay architecture grows, the time taken to incrementally map the debug instrumentation to it would also grow. Considering the complexity and uniqueness of our debug instrumentation (customized for each subset of signals to be observed), building an overlay architecture with such flexibilities to support incremental flow would be very difficult without having significant area/performance overheads and reduced compile time benefits. To our knowledge, overlays have not been targeted for debugging HLS generated circuits at the source level.

Other Techniques

Other works used different approaches to increase the size of the circuit's execution trace stored in the trace buffers to accelerate the in-system debugging process. Most of them focus on debugging at the RTL level, but could be extended for debugging at source-level for an HLS user by maintaining additional mapping information to relate the RTL signals back to the source code variables.

Bitstream Modifications: In [24], Graham et al. modify the bitstream (which is used to configure the FPGA to implement the desired circuit) itself to instrument the debugging hardware. The debugging circuit, essentially an embedded logic analyzer was added to the design before compiling the user circuit only, however, there were no connections made between the user and the debug logic. During debugging, when the user selects/changes the signals to be traced (to gain benefits from selective signal tracing), JRoute- a run-time routing API [45] was used to determine the routing for the selected signals and the JBits API interface [25] was used to modify the bitstream accordingly. In [63], the authors use a similar approach and pre-connect (before the first compilation of the user circuit) the signals of interest to routing muxes which forward the signals to FPGA I/O pins instead of on-chip trace buffers. Then an external analyzer was used to analyze the signals coming from the FPGA.

These approaches for routing a small number of signals incrementally are very fast when compared to an incremental compilation, but if the entire debug instrumentation has to be changed then the runtime advantages are not clear. Moreover, bitstream modifications are supported by only few of the commercial FPGA's which limits the use of such approaches.

Lossy/Lossless Compression Techniques: Generic data compression techniques could be used to compress the debug data before writing it to the trace buffers in order to pack more information. However, the amount of compression achieved would vary significantly depending on the debug data as these techniques are not customized. In [5] the authors investigate the use of Bentley-Sleator-Tarjan-Wei (BSTW) [9] and a modified Lempel-Ziv based [44] lossless data compression algorithms for embedded logic analysis of a circuit running on an FPGA and propose architectures for doing so efficiently. In [4], the authors make use of lossy compression techniques in the first debug iteration to record the intervals of circuit execution in the form of a signature. Then the failing signatures were detected and only the signals from these intervals were recorded in the next debug iterations ignoring the signals from other error free intervals, increasing the observability into the circuit. These compression techniques can be used along with our incremental flows with selected tracing to further accelerate the debugging process.

Using Off-Chip Memories: External storage devices can be used to store the debug data instead of the limited on-chip memories. The debugging time would then be reduced by a greater extent since the size of the execution traces stored would be much higher. However, the amount of the debugging data generated in each cycle (bandwidth) by a circuit running at-speed is much higher compared to the bandwidth of these external memories [6], meaning fewer signals have to be traced in each debug iteration. Alternatively, it is possible to use an on-chip buffer to hold the data until it is written to the off-chip memory [1].

Recent work by Jeff Goeders [18] focused on optimization techniques to reduce bandwidth of the data generated in order to use off-chip storage devices to achieve long debug traces for HLS generated circuits.

The most common method of reducing bandwidth requirements is to observe fewer selected variables (selective variable tracing). Combining the use of off-chip memories with selective variable tracing would further ease and accelerate the debug process. However, as described in Section 1.4, this may require several debug iterations with a different subset of variables being observed each time to find a bug. This requires full recompilations between each debug iteration. This thesis focuses on solving this issue and our proposed incremental debug flows as described in Chapter 4 eliminate these full recompilations.

2.5 Summary

This chapter described previous work which focused on in-system debugging of circuits running on an FPGA. The majority of these works used a trace-based approach where debug instrumentation is added to record the circuit execution and replay it later. This has numerous advantages over other scan-based approaches. Some of these previous works were specific to circuits described at the RTL level and required the user to have an understanding of the underlying hardware to get the most out of the debug data while others were targeted for HLS generated circuits and allowed for debugging at the source level by mapping the RTL signals back to the source code variables making it more appropriate for an HLS user.

The goal of any trace-based approach is to record as much data as possible in order to provide greater visibility into a circuit's execution to speed up the debugging process. To achieve this, some of the works described in this chapter used customized instrumentation to compress the debug data that must be recorded in order to store longer circuit traces. This makes it easier to find the root cause of a bug by providing observability into a significant portion of the circuit execution. On the other hand some works made use of incremental techniques or overlays to observe a different subset of signals (selective signal tracing) in each debug iteration without the need for full recompilations. When fewer signals are recorded in an debug iteration, the trace buffers would be updated less frequently resulting in longer circuit execution traces which would help pin-point the bug. If the selected signals do not provide sufficient information to identify the bug, the selected signals can be changed and another iteration is performed. However, using incremental compilation, the turn-around time between the iterations is significantly reduced. In addition, these incremental techniques could also be used to add the debug instrumentation after the place and route of the user circuit, thus not affecting the timing for the user circuit.

In this thesis, we accelerate a source level in-system debugging approach for HLS generated circuits [23] by combining both of these approaches. Customized compression logic is added to the instrumentation to pack the debug data efficiently and the incremental compilation techniques are used to record selected variables in each debug iteration, leading to much better utilization of the trace buffers.

Chapter 3

The Debugging Framework

3.1 Overview

This chapter describes the debugging framework used to illustrate our ideas. Section 3.2 gives a brief overview of the debugging framework. Section 3.3 describes the optimizations used by the framework to improve the trace buffer utilization in order to have a larger trace window. Section 3.5 focuses on the limitations of the framework to perform selective variable tracing and our proposed methodology to overcome them. Finally Section 3.6 concludes this chapter.

3.2 Framework

The adopted debugging framework was developed by Jeffrey Goeders during his Ph.D at UBC and is referred to as Goeders's framework hereafter. References [19–23] provide the details about the framework and the optimizations used. Although it has been introduced in Section 2.3.2, in this and the following sections, we briefly describe the debugging flow and the trace buffer optimizations, which are most relevant to this thesis.

Goeders's framework has been designed to work with the LegUp [12] HLS tool; however, it could be easily extended to support other HLS tools. This framework uses a trace-based approach along with a debug database (which contains a mapping between the source code variables and the RTL signals) to provide a source level in-system debugging facility. It can be operated in two modes: *live interactive mode* and *replay mode*. In *live* mode, the user can single-step through the circuit execution and retrieve the state of the circuit from the FPGA by pausing the circuit and reading the latest entries

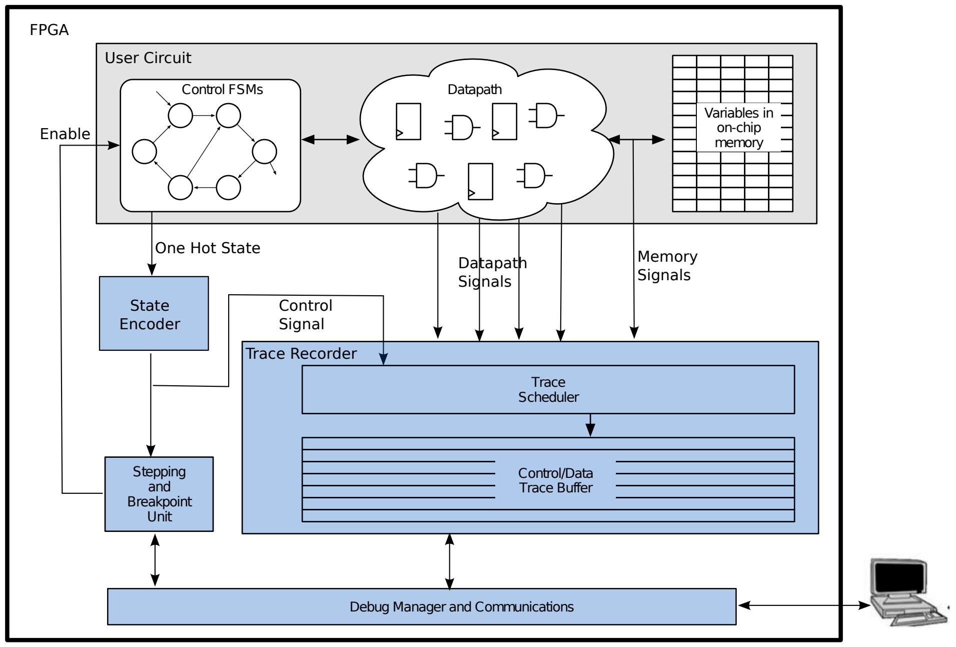


Figure 3.1: Debug Instrumentation

from the trace buffers. However, interrupting the circuit's execution may introduce additional bugs and is not always practical. In this thesis, we focus on *replay* mode in which the circuit is executed at-speed until a set breakpoint while recording the circuit's state in trace buffers. The circuit's execution is then replayed to the user using the information from these trace buffers. The debug instrumentation required to support the *replay* mode including the trace buffers are added at the RTL level. This is done by modifying the LegUp tool to automatically insert the required debug circuitry into the generated RTL.

3.2.1 Debug Instrumentation

Figure 3.1 shows the instrumentation added by the HLS tool. It consists of the following components:

Debug Manager: It acts as a communication bridge between the debugger application (which can be launched as a GUI on the user's workstation) and the instrumented debug modules. It receives all the message requests from the GUI through a serial interface and forwards them to the respective module and vice-versa. It is this module which is responsible for reading the trace buffers and sending the information to the GUI in order to display the variable values while replaying the circuit's execution.

Stepping and Breakpoint Unit: This unit controls the execution of the circuit by starting or stopping it whenever the set breakpoints are reached. The debug manager forwards the information about the added breakpoints and this module then generates an enable/disable signal to start/pause the circuit running on the FPGA when the circuit reaches the corresponding state.

State Encoder: It is necessary to record the control flow information (sequence of circuit states) in addition to the dataflow information(variable values) to replay the circuit behavior accurately. The circuit generated by LegUp uses one-hot encoding for its FSM's (Finite State Machines). These one-hot state signals are very wide and are therefore encoded/compressed by a *state encoder* module before recording them in order to improve the trace buffer utilization.

Trace Recorder: The heart of the instrumentation is an on-chip memory, known as a *trace buffer*. It is used to record the necessary information to replay the circuit execution for debugging purposes. Typically, it stores three types of data: (1) a history of variables that have been mapped to registers and logic in the datapath of the user circuit, (2) a history of variables that have been mapped to a global memory in the user circuit, and (3) a history of control flow information that describes the sequence of basic blocks executed. Each of these could be stored in a single buffer or separate buffers.

The Trace Recorder shown in Figure 3.1 encompasses this trace buffer along with the associated logic to improve its utilization. The following sections describe the architecture of the trace recorder along with the optimizations used to pack the trace buffer efficiently.

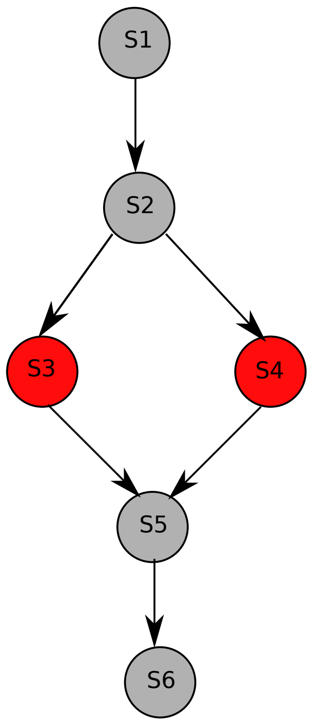


Figure 3.2: Control Flow Optimization

3.3 Trace Buffer Optimizations

Since the trace buffer is of a limited size (100Kb in [23]), we cannot store the entire run-time history of all variables and control flow information. Instead, the buffer is configured as a circular memory, so that new entries evict the oldest entries. This means that, when debugging, the user can only view the behavior of variables and control for a sliding portion of the execution (called the *"trace window"*). Clearly, the more efficiently data can be stored in the trace buffer, the longer the trace window, and the fewer debug turns that will typically be required.

We used a combined buffer to record the required control signals, memory and the datapath register signals in each cycle of the circuit execution. However, each of these signals are compressed using different optimization techniques [23] and are then packed together into a single word, which is written to the trace buffer. The effect of these optimizations are recorded in the database in order to accurately link the information back to the source code variables.

3.3.1 Control Flow Optimization

In order to trace the control flow of a circuit's execution, frameworks like [10, 40] record the FSM state value each cycle. The number of bits required to trace this information is proportional to the total number of states present in the control flow graph of the design. However, we find that instead of recording every state transition, it is enough to record in only those states previous to a state with multiple predecessors (for the rest of the states, it is obvious as it could only be reached from a single predecessor). With the help of this information and the control flow graph (which is available from the HLS compilation), the full control flow of the circuit execution can be reconstructed off-line. Therefore, as we are only recording in certain states, we can essentially number these states with a new numbering, which will require fewer bits than that required for numbering all the states. Our framework identifies the possible states that may need to be recorded and re-numbers them. We achieve a significant improvement using this optimization as we only have to record the state information (reduced number of bits) in these fewer states which are followed by a state with multiple predecessors.

Figure 3.2 shows an example control flow graph (CFG). Frameworks like [10, 40] would require 3 bits (in order to distinguish 6 possible states) of control flow information to be recorded in each cycle. However, if we look at the CFG, there are only two possible executions (S1-S2-S3-S5-S6 or S1-S2-S4-S5-S6) and therefore it is enough if we record only when the circuit is in S3 or S4 (shown in red color in the Figure 3.2), which would require only 1 bit. This significantly reduces the amount of control flow information to be recorded.

3.3.2 Dynamic Tracing of Datapath Register Signals

Unlike commercial ELA's [40, 75] which record all the datapath registers in each cycle, our framework adds custom circuitry known as a *trace scheduler* which dynamically selects what signals should be recorded in each cycle. The *trace scheduler* block uses the HLS scheduling information to record only the active datapath registers (only those which are updated) similar to the approach used in [53]. However, there are several other optimizations proposed by Goeders and Wilton in [23], to further improve the trace buffer utilization in order to have longer *trace window*.

Delay-Worst Signal-Trace Scheduling: Instead of recording the signals in the same cycle as they are generated, they could be delayed and recorded in any of the following cycles. Any such rescheduling of the signals is stored in the database, which allows for the perfect reconstruction of the trace information. Delayed recording of some signals in a worst-case state could reduce the width of the trace buffers and improve the *trace window* size significantly, especially when there are few entries with the worst case width and the rest of the entries are partially filled. Our framework recursively identifies the worst-case state and tries to delay some signals to achieve the best possible width. Figure 3.3(a) shows an example of the delay-worst scheduling. As seen, by delaying r*10* and recording it in S*6* instead of S*2*, the trace buffer width could be reduced.

Delay-All Signal-Trace Scheduling: Unlike the delay-worst scheduling, this optimization tries to delay the entry of all the signals that are updated in a state to a later state without increasing the width of the buffer. The algorithm iterates through the entries of all the states, combining them whenever possible. As a result, the rate at which the number of entries recorded in the trace buffer is reduced leading to a much larger trace window. Figure 3.3(b) shows an example of this optimization where the entry for S*1* is removed by moving those signal updates to S*7*, providing space for more updates to be recorded.

In addition to the datapath registers, LegUp HLS tool stores certain variables (like arrays or global variables) in on-chip memories driven by a memory controller logic. In order to replay the circuit execution, it is also necessary to record updates to these memory variables. There may be several on-chip memories and different memory controller signals could be driving them. It is enough to store only those memory controller signals which are updated in each cycle instead of recording all of them. This is identical to the datapath register signal tracing problem and hence the same signal-trace scheduling optimizations are used for memory updates. In fact the same trace scheduler block is used for both the datapath register and the memory signals to output the active signals in the form of a single word (can be seen in Figure 3.1).

As mentioned earlier in this section, we use a single trace buffer to record the control, memory and datapath register updates. Therefore, in a given cycle the output of the combined datapath register and memory signal-trace scheduler plus the control state signal (if it needs to be recorded in that cycle) are combined into a single entry which is written to the trace buffer.

Dual-Ported Memory Signal-Trace Scheduling: In addition to the above described signal-trace scheduling optimizations, the dual ported memory architecture offered by FPGA's can be leveraged to further improve trace buffer utilization. The single word which is to be written to the trace buffer can now be split into half and written in two entries on the same clock cycle using the available dual ports for the memory buffer. The improvements from this optimization occurs in entries where less than half of the trace buffer width is required as this corresponds to a single entry now instead of two entries. Figure 3.3(c) shows an example where S*5* requires a single entry instead of two entries.

With all these dynamic signal-tracing optimizations, the length of the circuit execution that could be traced is improved by more than 127X when compared to commercial ELAs such as [40, 75].

Linking the Control and Data flow Information

During the HLS compilation process, the information related to the mapping of the source code variables to the RTL signals, which FSM states have an entry in the trace buffer and which signals are updated in those states are saved in a debug database. While debugging, the information from the trace buffer is retrieved and the control/data flow of the circuit's execution is reconstructed off-line. When filling the last entry in the trace scheduler, it is ensured that the control state information is also recorded even though the control flow optimization may decide not to record it. With the help of this state information and the HLS scheduling information we can read the trace buffer backwards and link the signal update information to the corresponding states, which is later mapped back to the source code variable using the mapping information stored in the debug database.

3.4 Debug Flow

Figure 3.4 shows the overall debug flow for this framework. The user first compiles C code to HDL using LegUp, an open-source HLS tool (LegUp [12]). The HLS tool automatically adds instrumentation to the RTL circuit to record the behavior of *all* user-visible variables. The circuit is then compiled using a vendor-specific tool-chain and implemented on an FPGA. As the FPGA runs, the instrumentation records a history of variables and control flow information in an on-chip memory. After the run is complete or when a breakpoint is reached, the user launches a debug GUI on a workstation which connects to the FPGA and downloads the trace history. The GUI uses this history to provide a software-like debug experience. The user is able to step through the recorded execution in an attempt to understand the operation of the design and deduce the root cause of any unexpected behavior. As the user refines his or her view of the operation of the circuit, he or she may run the circuit again possibly with a different breakpoint, providing visibility into a different part of the circuit execution. Re-running this circuit in this way is called a “debug turn”; often many debug turns are required to pin-point a bug.

3.5 Selective Variable Tracing

Even though different parts of the circuit execution can be observed using the original debug flow, the size of the trace window is fixed in each debug turn. This is because of the fact that *all* user visible variables are being recorded in each debug turn. Being able to examine the value of any variable mimics the software debug experience. For some debug scenarios, however, it may be sufficient to record fewer variables. As an example, if the user has narrowed down the cause of a bug to a particular function, it may be sufficient to only trace variables within that function. Alternatively, if the user knows that certain variables are unimportant to a particular bug, they can be excluded from tracing. Recording a smaller number of variables can increase the size of the trace window in two ways. First, fewer variables may mean the trace memory can be narrower. Given a fixed trace buffer size, this means a deeper trace buffer, meaning a longer history for each variable can be recorded. Second, recording fewer variables may mean that there are more cycles in which no value is recorded, meaning the trace buffer is filled more slowly. Increasing the trace window will provide more visibility into the execution of the hardware, hopefully reducing the number of debug turns required to uncover the root cause of unexpected behavior.

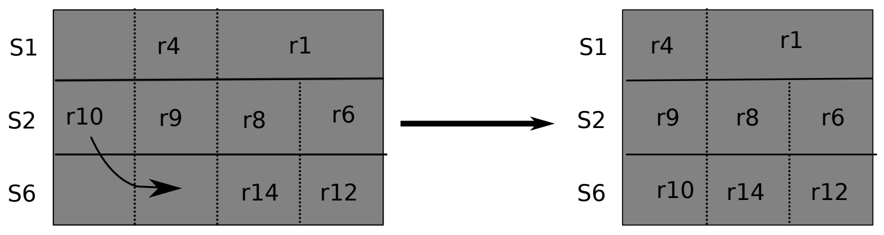
A second advantage of recording only a subset of variables is that the instrumentation logic itself will be smaller. For designs that are area constrained, there may not be sufficient unused resources to implement and route the debug logic which captures all variables. For these cases, selecting a subset of variables to observe may be the only possible way of providing the user with debugging capabilities.

For recording only a subset of user-visible variables, the original flow from Figure 3.4 can be used directly. However, as debugging proceeds and the user narrows down the cause of a bug, he or she may wish to change which variables are instrumented. In our considered framework [23], changing the instrumentation in this way requires a rerun of the HLS flow to generate the new required connections between the user and debug modules and also a different trace-scheduling logic to efficiently pack the signals to be traced in each cycle using different optimizations described in the previous section, resulting in a different RTL. This would require a complete recompile of the design, including a lengthy place-and-route to re-implement the circuit on an FPGA. This significantly impacts debug productivity.

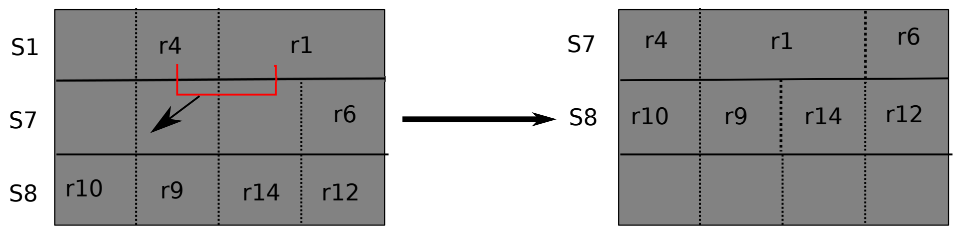
In this thesis, we address the problem of length recompilations using incremental design techniques offered by a commercial FPGA vendor tool-Intel Quartus II and reduce the turn-around time between the successive debug turns significantly. Chapter 4 describes our incremental debug flows.

3.6 Summary

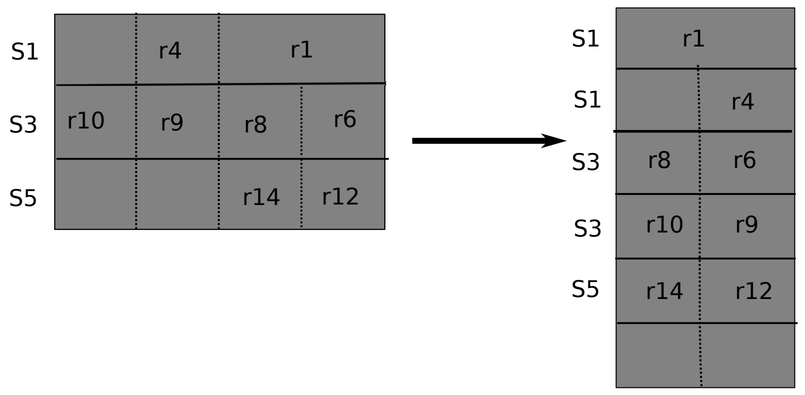
In this chapter, the debugging framework used to prototype our ideas was described along with different signal-trace optimizations used to improve the utilization of the trace buffer (trace window). In addition to these optimizations, how the selective variable tracing improves the trace window size leading to fewer debug turns is elaborated in the context of our adopted framework. Finally, the problems associated with selective variable tracing and our proposed approach to overcome them are presented.



a) Delay-Worst Scheduling



b) Delay-All Scheduling



c) Dual-Port Scheduling

Figure 3.3: Dynamic Signal Tracing Optimizations
'S*i*' indicates the state and 'r*i*' indicates the datapath/memory signals that are being recorded.

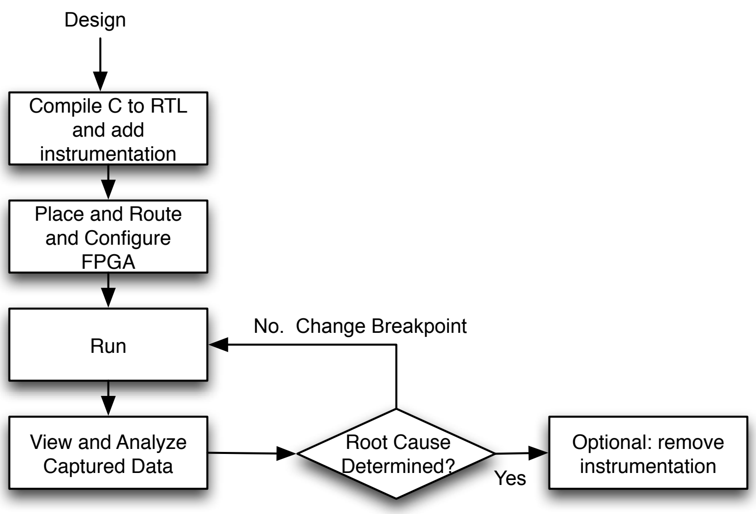

Design
Compile C to RTL and add instrumentation
Place and Route and Configure FPGA
Run
No. Change Breakpoint
View and Analyze Captured Data
Root Cause Determined?
Yes
Optional: remove instrumentation

Figure 3.4: Original Debug Flow

Chapter 4

Incremental Debug Flows

4.1 Overview

In this chapter we compare different incremental debug flows for the adopted HLS debug framework [23]. As described in Section 1.6, unlike the incremental debug flows associated with the commercial ELA's, the amount of logic that needs to be recompiled in each debug iteration for our case is significant.

FPGA vendors provide the ability to guide the place and route tools to recompile only the parts of a circuit that have changed since a previous compilation rather than recompiling the whole circuit. This promises to not only decrease debug turn-around time, but also maintain timing in the user circuit between debug iterations. However, a direct use of such incremental features does not work well; Section 4.2.1 describes this approach.

Using incremental compilation, however, requires a careful selection of the flow to balance the impact on area and delay of the user circuit, the area and delay of the instrumentation, and the compilation run-time. Further, we desire a flow which allows the instrumentation logic to be removed after debugging has completed, with as little impact on circuit timing as possible. We developed two promising flows that try to balance these metrics.

Section 4.2.2 presents an incremental debug flow in which we modify the user partition to add permanent taps to facilitate efficient incremental recompilation. Section 4.2.3 describes another incremental debug flow which is non-intrusive to the user circuit as the debug instrumentation is added after the compilation of the user partition. In Section 4.3, we introduce the GUI framework developed to automate these incremental flows. Section 4.4 concludes this chapter.

Baseline Flow

The baseline flow (Flow 1) is as described in Section 3.4 and shown in Figure 3.4. All user-visible variables are recorded and the circuit is compiled only once. However, using various breakpoints different time periods within the circuit execution can be observed. This flow does not use incremental compilation.

4.2 Incremental Flows

Commercial ELA's like SignalTap II [40] offer support for incremental RTL debug. As described in Section 2.4, since these IP blocks have to support wide range of circuits, they are very simple and consist of a generic trigger circuitry and trace buffers. While doing incremental debug, as the signals to be recorded are changed, the tool just needs to perform incremental routing in order to change the necessary connections. However, our case is more complicated because the debug instrumentation is customized based on the signals that are being recorded mainly due to the trace buffer optimizations described in Section 3.3. In addition, as the debug instrumentation is added at the RTL level, the design partitions are modified each time the variables recorded are changed.

4.2.1 A Naive Incremental Flow

In this incremental debug flow (Flow 2), we allow the user to select a subset of the user-visible variables to record. The source code is then compiled using the LegUp [12] HLS tool which is modified to insert the required debug instrumentation as in the original framework [23]. However, as shown in Figure 4.1, separate design partitions (to enable incremental compilation) for the user circuit and the instrumentation are constructed without restricting them to a specific physical region on the FPGA using fully automated scripts and the GUI interface (modified version of that from [23] and is described in Section 4.3). Based on the variables selected, the user circuit and instrumentation partitions are modified; the user circuit partition is modified to insert "taps" and the debug partition is modified to compress and record those signals in trace buffers. To obtain this debug logic, the LegUp HLS tool is run again with the selected signals. The design is then compiled using incremental techniques (from Intel's Quartus Prime 16.0). The circuit is run, and the history of selected variables are stored in the trace buffer; after the run is complete, the trace information is extracted and used with our modified debug GUI which allows the user to replay the execution in the context of the original software. The user may then choose to instrument a different set of variables, in which case the instrumentation is modified and the incremental compilation is repeated where the tool tries to incrementally place and route the changed partitions while preserving the partitions which have not been changed, as described in Section 2.4.

Intuitively, compared to Flow 1, this should result in better utilization of the trace buffer as fewer variables are being recorded, leading to longer trace windows and also reduced debug instrumentation area. However, because both the user circuit (taps) and the debug instrumentation (Trace Scheduler) change every iteration, the ability of the incremental algorithm to reduce compile time is limited and may also result in a recompilation of the user circuit which is not desirable as the timing paths in the circuit could be altered. As a result, even if the user wants to remove the instrumentation after the debugging, the user partition may have to be recompiled. In the next section, we describe another incremental flow which tries to preserve the place-and-route results for the user partition and only recompile the debug partition when the variables to be recorded are changed.

4.2.2 Incremental Flow with Permanent Taps

In this incremental flow (Flow 3), changes are localized to the instrumented circuit. The key idea is to ensure that all taps in the user circuit are maintained, even if the corresponding variables are not observed. In this flow, as shown in Figure 4.2, we once again use separate user circuit and debug instrumentation partitions. Unlike Flow 2, taps for *all* user-visible variables are added within the user partition, and these taps do not change during the entire debugging process. The taps (ports) which are not being used in the current debug turn are not optimized away as the EDA tool (Quartus Prime) treats the unused ports across the partitions as virtual pins that are temporarily mapped to logic elements (Look Up Table–LUT) which can later be connected to require ports in the subsequent debug turns without the need for full recompilation. In order to add these taps, the RTL generated by the framework in [23] was restructured and modified accordingly. We developed fully automated scripts and a GUI to perform these steps, hiding everything from the user's point of view, which we think is very important for user productivity. The GUI is briefly described in Section 4.3.

Because of these permanent taps, as the user changes which variables are observed, only the debug partition is modified and the user partition is not changed. Intuitively, this will achieve the same trace buffer utilization as Flow 2, however, it will be able to take better advantage of the incremental compilation features in the place and route tool, leading to faster debug iterations. However, the fact that taps are added within the user circuit for all variables means that the instrumented user circuit may run somewhat slower than the uninstrumented design.

As the debug instrumentation partition and the user circuit partition are compiled at the same time, the performance of the user circuit may be affected as it is not able to use all the available FPGA resources. In this flow, if the user chooses to remove the debug instrumentation after the debugging has been completed, it could be removed incrementally without the need for full compilation, however, since the first compilation is not fully optimized for the user circuit, he/she would need to be satisfied with the obtained performance.

4.2.3 Incremental Flow with Permanent Taps and Late Binding

In Flows 2 and 3, the presence of the instrumentation may negatively impact the performance obtained for the user circuit (for example, by "inflating" the user circuit if some of the instrumentation logic is placed within the boundaries of the user circuit). This is undesirable for two reasons: (1) adding the instrumentation during the first compilation changes timing paths within the user circuit, possibly hiding bugs that are being sought or exposing new ones, and (2) after debug is complete, and the instrumentation is removed, either a full recompile of the user circuit is required, possibly leading to new timing behaviors, or the designer must be satisfied with the lower performance of the design.

In Flow 4, as shown in Figure 4.3, we maintain user and debug instrumentation partitions along with the permanent taps added to the user partition as in Flow 3. However, unlike Flows 2 and 3, we perform an initial compilation with an empty debug partition (no logic). No physical region on the FPGA is reserved for this empty debug partition initially. This allows the user partition to be well optimized as it is not interfered by the debug instrumentation partition. During debugging, we then replace the empty debug partition with a partial trace scheduler corresponding to the variables that are being recorded, and debug as before. This debug partition would now be placed and routed using the leftover FPGA resources.

Intuitively, compared to Flow 3, this flow will lead to the fastest performance of the user circuit. However, when the instrumentation is added, we would expect the performance of the instrumented circuit to be lower than that in Flow 3, since the instrumentation is optimized separately from the user circuit and may now contain the timing critical paths. This means that, during debugging, it may be necessary to slow the clock slightly; whether this is acceptable depends on the design and system in which the design is being used. Once debugging has been completed, the user may choose to remove the instrumentation incrementally without the need for full recompilation and the circuit can now be run at the original (uninstrumented) clock frequency that was obtained during the first compilation.

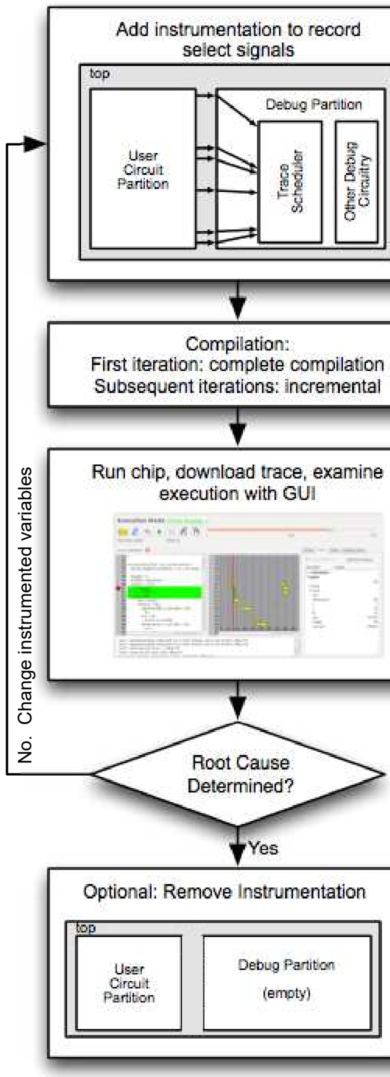

Add instrumentation to record select signals
top
Debug Partition
User Circuit Partition
Trace Scheduler
Other Debug Circuitry
Compilation:
First iteration: complete compilation
Subsequent iterations: incremental
Run chip, download trace, examine execution with GUI
Root Cause Determined?
No. Change instrumented variables
Yes
Optional: Remove Instrumentation
top
User Circuit Partition
Debug Partition
(empty)

Figure 4.1: A Naive Incremental Flow (Flow 2)

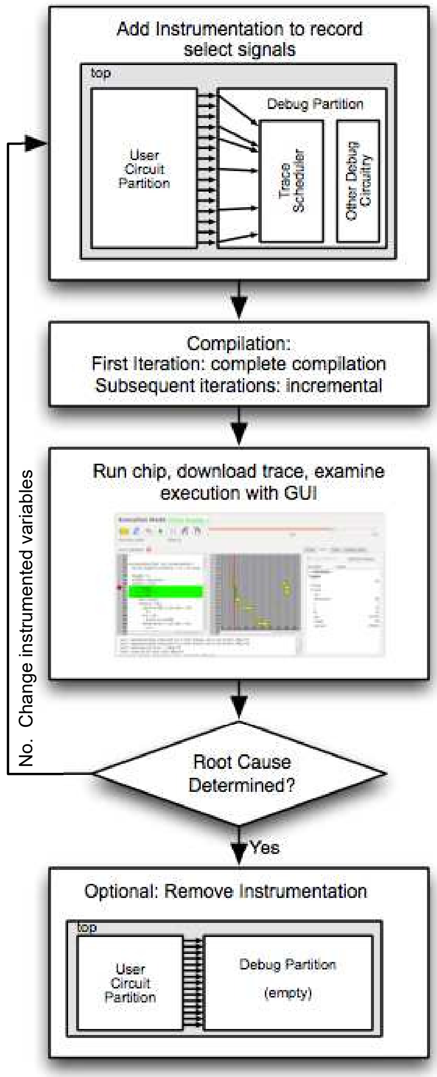

Add Instrumentation to record select signals
top
User Circuit Partition
Debug Partition
Trace Scheduler
Other Debug Circuitry
Compilation:
First Iteration: complete compilation
Subsequent iterations: incremental
Run chip, download trace, examine execution with GUI
Root Cause Determined?
No. Change instrumented variables
Yes
Optional: Remove Instrumentation
top
User Circuit Partition
Debug Partition
(empty)

Figure 4.2: Incremental Flow with Permanent Taps(Flow 3)

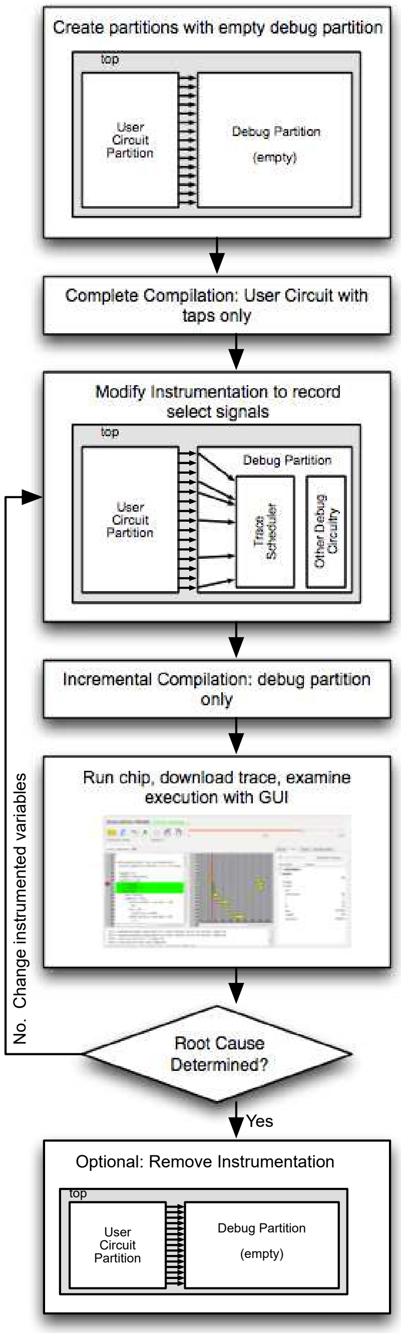
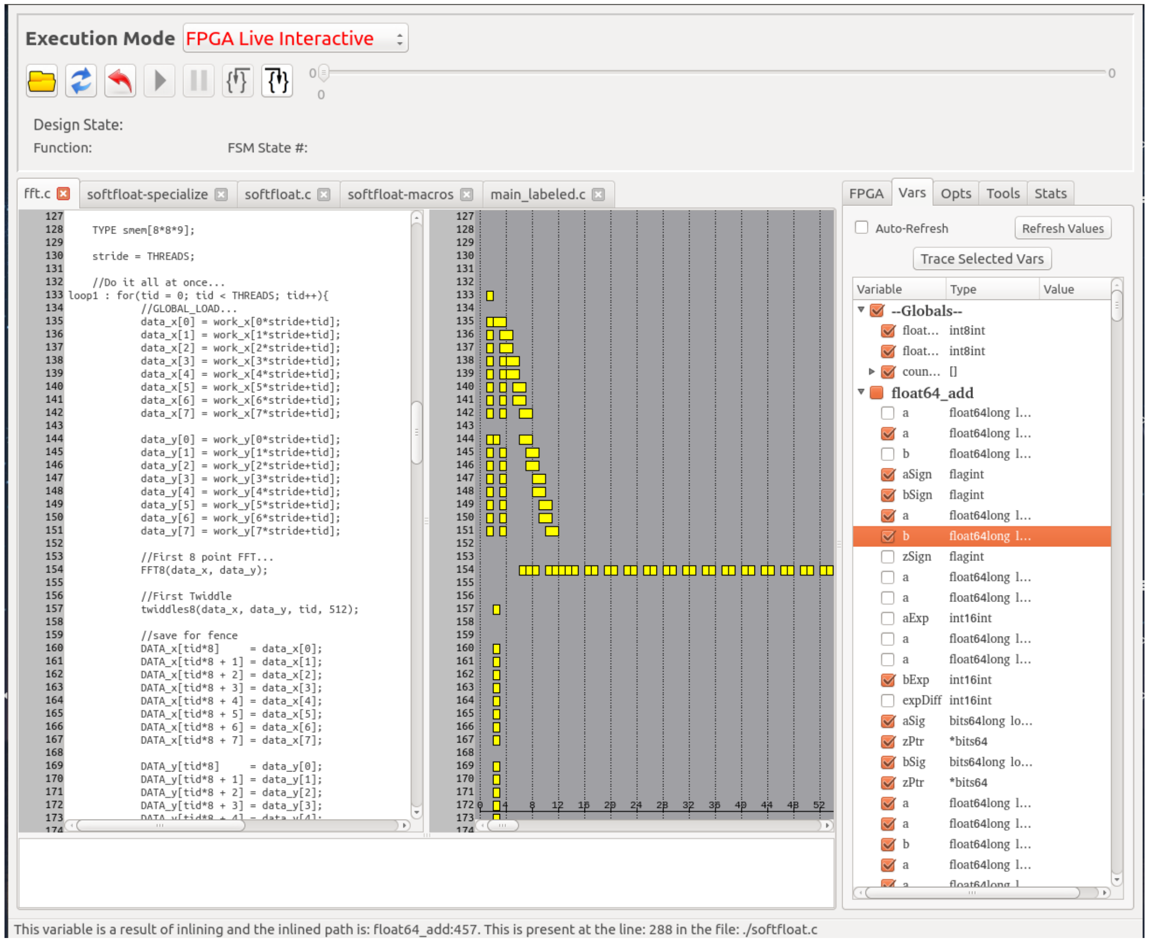

Create partitions with empty debug partition
top
User Circuit Partition
Debug Partition
(empty)
Complete Compilation: User Circuit with taps only
Modify Instrumentation to record select signals
top
User Circuit Partition
Debug Partition
Trace Scheduler
Other Debug Circuitry
Incremental Compilation: debug partition only
Run chip, download trace, examine execution with GUI
Root Cause Determined?
No. Change instrumented variables
Yes
Optional: Remove Instrumentation
top
User Circuit Partition
Debug Partition
(empty)

Figure 4.3: Incremental Flow with Permanent Taps and Late Binding (Flow 4)



Design State:

Function: FSM State #:

```
    TYPE smem[8*8*9];

    stride = THREADS;

    //Do it all at once...
loop1 : for(tid = 0; tid < THREADS; tid++){
            //GLOBAL_LOAD...
            data_x[0] = work_x[0*stride+tid];
            data_x[1] = work_x[1*stride+tid];
            data_x[2] = work_x[2*stride+tid];
            data_x[3] = work_x[3*stride+tid];
            data_x[4] = work_x[4*stride+tid];
            data_x[5] = work_x[5*stride+tid];
            data_x[6] = work_x[6*stride+tid];
            data_x[7] = work_x[7*stride+tid];

            data_y[0] = work_y[0*stride+tid];
            data_y[1] = work_y[1*stride+tid];
            data_y[2] = work_y[2*stride+tid];
            data_y[3] = work_y[3*stride+tid];
            data_y[4] = work_y[4*stride+tid];
            data_y[5] = work_y[5*stride+tid];
            data_y[6] = work_y[6*stride+tid];
            data_y[7] = work_y[7*stride+tid];

            //First 8 point FFT...
            FFT8(data_x, data_y);

            //First Twiddle
            twiddles8(data_x, data_y, tid, 512);

            //save for fence
            DATA_x[tid*8]     = data_x[0];
            DATA_x[tid*8 + 1] = data_x[1];
            DATA_x[tid*8 + 2] = data_x[2];
            DATA_x[tid*8 + 3] = data_x[3];
            DATA_x[tid*8 + 4] = data_x[4];
            DATA_x[tid*8 + 5] = data_x[5];
            DATA_x[tid*8 + 6] = data_x[6];
            DATA_x[tid*8 + 7] = data_x[7];

            DATA_y[tid*8]     = data_y[0];
            DATA_y[tid*8 + 1] = data_y[1];
            DATA_y[tid*8 + 2] = data_y[2];
            DATA_y[tid*8 + 3] = data_y[3];
```

Variable	Type	Value
--Globals--		
float...	int8int	
float...	int8int	
coun...	[]	
float64_add		
a	float64long l...	
a	float64long l...	
b	float64long l...	
aSign	flagint	
bSign	flagint	
a	float64long l...	
b	float64long l...	
zSign	flagint	
a	float64long l...	
a	float64long l...	
aExp	int16int	
a	float64long l...	
a	float64long l...	
bExp	int16int	
expDiff	int16int	
aSig	bits64long lo...	
zPtr	*bits64	
bSig	bits64long lo...	
zPtr	*bits64	
a	float64long l...	
a	float64long l...	
b	float64long l...	
a	float64long l...	

This variable is a result of inlining and the inlined path is: float64_add:457. This is present at the line: 288 in the file: ./softfloat.c

Figure 4.4: Modified GUI to support Selective Variable Recording

4.3 Automated GUI

In order for Quartus Prime to support incremental compilation features, the overall design has to be divided into partitions. Moreover, for Flow 3 and Flow 4, the user partition had to be modified to insert permanent taps into it. We have developed a GUI framework (modified from that in [23]) to do all these things automatically in the background. It has been made available on-line as an open source debugging framework at https://bitbucket.org/pavankumarbussa/.

Figure 4.4 shows a screen-shot of our GUI. A brief tutorial has been created on how to use this GUI effectively and is described later in the Appendix A. From a user's point of view, he/she just needs to open a design, optionally set the breakpoints, select the source code variables which they are interested in observing, select Flow 3 or Flow 4, click the compile button and finally program the bitstream to the FPGA once the compilation is successful. Under the hood, our GUI runs LegUp HLS to generate the RTL and create a debug database, restructures the RTL code as required (such as making the partitions and adding taps), reruns LegUp HLS to get the necessary debug instrumentation whenever the variables selected are changed and finally determines whether a full or incremental compilation has to be performed. Once the circuit has been run completely or the breakpoints have been reached, the user can enter the replay mode where the GUI reads the data serially from the trace buffer (and performs off-line analysis to reconstruct and link the control flow information with the variable updates) and replays the circuit execution by showing the updates for the selected variables as the circuit execution progresses. The user can then select a different subset of variables and repeat the same steps until he/she is satisfied with the design.

4.4 Summary

In this chapter, different incremental flows developed for the framework in [23] were described. These incremental flows enable 'Selective Variable

Tracing' without the need for full recompilation. With the help of the trace buffer optimizations (those described in Section 3.3) and the selective variable tracing, the trace buffer could be packed much more efficiently leading to larger trace window sizes. As a result, fewer debug turns might be enough for pin-pointing a bug and also the turn-around time between the successive turns is now reduced significantly as there is no need for full recompilations. Each of the proposed flows have their own advantages and disadvantages. If the user circuit is timing critical then Flow 4 would be a better option but if one desires to debug at a relatively higher clock speeds then Flow 3 would be a good option. Lastly, we introduced our GUI framework which abstracts away complexities from the user and provides a faster and efficient debug environment.

Chapter 5

Results

5.1 Overview

In this chapter, we present the results obtained for different sets of experiments conducted to evaluate our proposal. Section 5.2 describes our methodology and the benchmarks used to evaluate our flows. In Section 5.3 and Section 5.4 , we quantify the impact of reducing the number of variables that are recorded (Selective Variable Tracing) on the *trace window size* and the *debug instrumentation area* respectively. Intuitively, reducing the number of traced variables will increase the trace window size, but the amount of increase is not clear, since there is the possibility of a one-to-many relationship between the source code variables and the RTL signals.

In Section 5.5, we illustrate the impact of incremental compile on debug turn-around time for the flows described in Chapter 4. We show that by using our flows, a 40% reduction in compile times can be achieved when doing selective variable tracing when compared to the original flow proposed in [23].

The presence of debug instrumentation and permanent taps may affect the performance of the user design. The variation in the frequency of the design is described in Section 5.6 and we show how the use of an empty debug partition in the first compilation reduces this performance loss, once the debug instrumentation is removed. Finally, Section 5.7 concludes this chapter.

5.2 Methodology

As described in Chapter 3, we assume the architecture from [23]. A total trace buffer size of 100 Kilobytes is assumed. We do not split this buffer, but rather use a single buffer to store the updates to variables that are in both the user circuit's datapath and memories as well as the control flow information. We chose this configuration as it provided better trace buffer utilization by improving the packing efficiency. It should be noted that, even for any other trace buffer configuration our flows would work without any modifications. For our experiments, we used the circuits from CHStone benchmark suite [28] and compiled them using our HLS debug framework. The generated RTL was synthesized to a Stratix IV FPGA (EP4SGX530NF45C3) using Intel's Quartus Prime 16.0 on a workstation having a Quad core Intel Xeon CPU E3-1225 V2 processor.

In addition to the CHStone benchmarks that come with LegUp HLS tool, we used the largest circuit (*FFT_Transpose*) from another benchmark suite, Machsuite[66], to demonstrate the scalability of our approach. All other benchmarks from Machsuite were of similar size to those in the CHStone suite and hence were not considered. Moreover, these benchmarks (which are not provided by LegUp) were not able to compile successfully because they used an unsupported fixed point representation. We had to replace these datatypes with those that are supported by LegUp, which was time consuming.

Also in order to do 'Selective Variable Tracing' in each debug turn, we do not incorporate any special signal selection algorithms like that of [46, 49], as the focus of this thesis is on improving the debug turn-around time when a different set of variables are recorded and not on how these variables are selected. Therefore, for the purpose of our experiments, we select an unique subset of variables for each debug turn by shuffling the the list of variables (using a deterministic seed) and then selecting a required proportion.

Table 5.1: Trace Window Size(For Flow 3)

Benchmark	100% variables traced	50% variables traced	25% variables traced
	Window size (cycles)	Window size (cycles)	Window size (cycles)
adpcm	2755	3628	5460
aes	4849	10171	22834
blowfish	6586	9961	15546
dfadd	1265	1650	2342
dfdiv	4159	5640	7078
dfmul	1126	1361	1944
dfsin	2869	3547	4509
gsm	732	1413	3401
jpeg	3521	5832	7567
mips	1103	1968	3559
motion	6232	10691	15160
sha	4370	8248	13376
FFT	1127	1976	2471
Average	3130	5083 (1.6x)	8096 (2.6x)

The values in the parenthesis indicate the improvement over the trace window size corresponding to 100% variables recorded (Column 2)

5.3 Impact on the Trace Window Size

The following set of experiments were performed to analyze the advantages of 'Selective Variable Tracing'. Although reducing the number of variables to be recorded will increase the trace window size, the degree to which this occurs is not clear because of the *Static Single Assignment* (SSA) representation used for the IR code by the LLVM compiler, which is the front end of our HLS (LegUp [12]) debug framework. In SSA, each update of a source code variable is represented by an unique IR signal. A frequently updated variable would therefore result in many more IR signals than a variable that is updated less frequently. Thus, a single source code variable may correspond to multiple IR signals which may finally be converted to multiple RTL signals by the LLVM compiler. Hence, if a user chooses not to record a variable, this might result in ignoring several RTL signals. The mapping information between a source code variable and the RTL signals are stored in the database created by our framework and is used for linking the information stored in the trace buffer back to the corresponding variable. As a result, there is no direct linear relationship between the number of variables being recorded and the improvement in the trace buffer utilization (Trace Window size).

Table 5.1 shows the impact on trace window size as we vary the proportion of user-visible variables that are traced using our incremental flow with the permanent taps (Flow 3). Flow 2 and Flow 4 (incremental flow with permanent taps and late binding) would give similar trace window sizes as this quantity primarily depends on the number of variables being recorded. Column 2 shows the number of execution cycles for which the history of all the user-visible variables are stored in the trace buffer at the end of the run. To obtain this information, we used the same approach from [23] where the circuit is simulated using Modelsim to extract the required execution trace and the filling of the trace buffer cycle-by-cycle. With the help of this data, the trace buffer size in terms of number of execution cycles was obtained.

Column 3 shows the same quantity assuming 50% of the user-visible variables are recorded and Column 4 shows the same assuming 25% of the

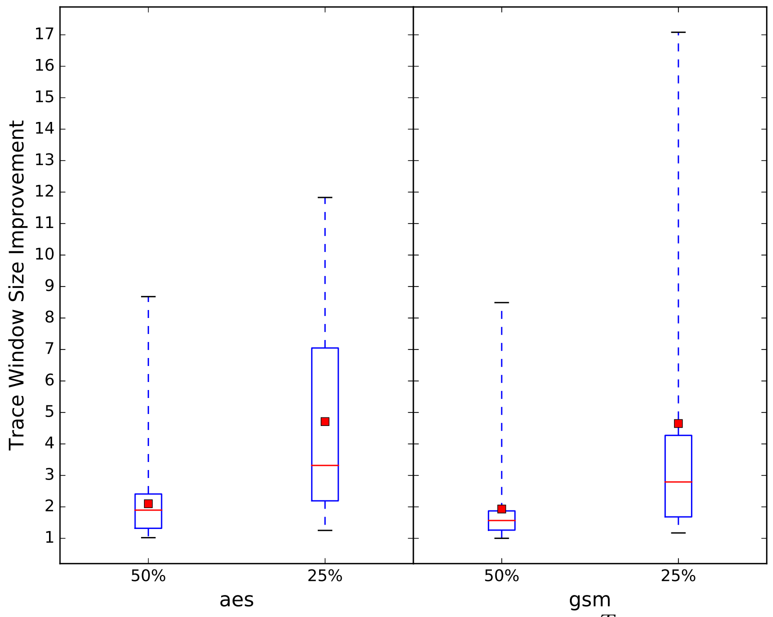


Figure 5.1: Trace Window Size Variation [$\frac{Trace_{subset}}{Trace_{full}}$]

user-visible variables are recorded. To gather these results, we ran six experiments with a different subset of the variables selected and average the results.

As the table shows, reducing the number of variables recorded increases the amount of execution history that can be stored in the trace buffers by 1.6x for the 50% case and 2.6x for the 25% case. Intuitively, the increase depends on the relative update frequencies of variables that are being recorded. If variables that are rarely updated during the execution of the program are selected, we would expect the improvement to be large because the trace buffer would now be filled slowly and thus a large portion of the circuit execution could be traced.

5.3.1 Variation in the Trace Window Size

There are two benchmarks (*aes* and *gsm*) for which the trace window size is improved by more than 4x when the number of variables recorded were reduced from 100% to 25%. In these experiments, the variables that were selected tended to be updated very infrequently. To better understand how this variability in the variable access rate affects trace window size, we took these two benchmarks and ran 100 experiments with different variable selections (unique subsets) for each of the 50% and 25% case. For each experiment, we measured the trace window size, and created the whisker plot shown in Figure 5.1. In this diagram, the box shows the second and third quartile of trace window size. The first quartile (highest trace window size) is shown as a dotted line above the box, and the fourth quartile (lowest trace window size) is shown as a dotted line below the box. The median is represented by a red line, and the average by a small red square. In all cases, these trace window sizes ($Trace_{subset}$) are normalized to that of the trace window size obtained when all the user-visible variables are recorded ($Trace_{full}$). Even though the average trace window size improvement for these benchmarks are nearly 2x for the 50% case and 4x for the 25% case (the small red dots in the Figure 5.1), there are some cases where the trace window size could go as high as 17x (for the *gsm* benchmark) if we choose a subset in which the variables are not updated frequently. However, it should be noted that in any case the trace window size when a partial subset of variables are recorded ($Trace_{subset}$) is almost the same or more than that when all the user-visible variables are recorded ($Trace_{full}$).

5.4 Impact on Debug Instrumentation Area

As stated in Section 3.5, a second advantage of 'Selective Variable Tracing' is the reduction in the debug instrumentation area as the number of variables recorded are reduced. The debug instrumentation logic in our case is the same as that of [23] which consists of a fixed-cost logic namely: communication and debug manager, trace recorder, stepping and breakpoint unit which are independent of the considered benchmark and the number of variables recorded. In addition, it also consists of a state encoder (which was described in Chapter 3 and is used for recording control information) and a trace scheduler block (for compressing and recording appropriate signals) which are dependent on the benchmark circuit as the number of states and the variables vary across the different benchmarks. However, for a given

Table 5.2: Trace Scheduler Area (For Flow 3)

	100% variables	50% variables	25% variables
Benchmark	Area	Area	Area
adpcm	2064	1231	684
aes	883	343	170
blowfish	965	577	271
dfadd	1628	1079	625
dfdiv	1441	1021	573
dfmul	1023	622	378
dfsin	4016	2837	1718
gsm	1029	539	293
jpeg	3478	2168	1315
mips	991	339	169
motion	900	501	328
sha	500	204	88
FFT	13095	7757	4759
Average	2463	1478 (1.7x)	875 (2.8x)

The values reported are the number of Stratix IV ALMs obtained after compiling the circuit using Quartus Prime 16.0. The values in the parenthesis indicate the reduction in the area when compared to the area corresponding to 100% variables recorded (Column 2)

benchmark, the number of states that are required to be encoded remain the same even if different number of variables are recorded. Therefore, it is the trace scheduler logic which changes when the number of variables recorded are changed. This is because of the change in the number of corresponding RTL signals that must be multiplexed by the trace scheduler block.

Table 5.2 shows the area of the trace scheduler block in number of Stratix IV ALMs as the number of variables recorded are reduced from 100% to 25% for Flow 3 (once again, the area reduction would be by the same factor for Flow 2 and Flow 4 as this primarily depends on the number of variables recorded). The trace scheduler area is reduced, on average, 1.7x for the 50% case and 2.8x for the 25% case. This is because fewer signals need to be time-multiplexed into the trace buffers, leading to much simpler trace scheduling logic.

Clearly, incorporating this selective variable tracing into a debug flow would have a significant impact on debug efficiency, by providing the user with an option for not recording the redundant/unimportant variables. It also reduces the amount of debug instrumentation that has to be added which is especially helpful when the user circuit itself is very large and consumes most of the resources on the FPGA. However, as described in Chapter 3, performing selective variable tracing using the framework from [23] requires a full recompilation. We have developed different incremental flows (described in Chapter 4) to address this issue. In one of these flows, Flow 2, even after making design partitions the whole design was recompiled when the variables to record were changed, as both the user partition and the debug circuit partition were changed (as described in Chapter 4). That is, the results were virtually the same as that of Flow 1 (Baseline Flow) and hence we do not include the results for Flow 2 separately. The following sections evaluate our flows (Flow 3/Flow 4 vs Flow 1) in terms of savings in the compile times between successive debug turns and the variation in frequency for the overall debug flow.

Table 5.3: Total Compile Time for Flow 3 in Seconds

Benchmark	Flow 1 100% Obs.*	Analysis overhead	Flow 3 Variables Recorded 50%*	50%	25%*	25%	0%	Flow 3 Reduction (for 25%)	Flow 3 Reduction (for 50%)
adpcm	351	31	323	176	305	179	165	49.0%	49.8%
aes	283	25	282	158	273	172	136	39.2%	44.0%
blowfish	168	19	179	110	158	108	100	35.6%	34.5%
dfadd	165	17	157	121	141	120	104	27.0%	26.5%
dfdiv	243	27	239	156	221	158	126	34.8%	36.0%
dfmul	135	14	141	106	132	105	90	22.2%	21.3%
dfsin	438	34	407	244	368	227	196	48.1%	44.4%
gsm	210	24	196	133	181	135	102	35.8%	36.9%
jpeg	939	47	897	368	848	365	317	61.1%	61.0%
mips	138	16	138	100	129	100	89	27.7%	27.4%
motion	287	24	268	140	252	136	130	52.5%	51.1%
sha	150	16	120	102	117	92	89	38.7%	32.1%
FFT	2398	231	2503	1088	2335	958	809	60.0%	54.6%
Average	454	40	450	231	420	220	189	40.9%	40.0%

** indicates that it is a full compilation and not incremental. 0% means that the debug instrumentation is removed.*

Table 5.4: Total Compile Time for Flow 4 in Seconds

	Flow 1	Analysis	Flow 4				Flow 4	Flow 4
Benchmark	100%	overhead	Variables Recorded				Reduction	Reduction
	Obs.*		0%*	50%	25%	0%	(for 25%)	(for 50%)
adpcm	351	31	314	177	174	158	50.5%	49.6%
aes	283	25	266	157	165	129	41.6%	44.7%
blowfish	168	19	153	112	111	96	34.2%	33.3%
dfadd	165	17	137	126	120	99	27.6%	23.5%
dfdiv	243	27	224	160	155	126	36.1%	34.1%
dfmul	135	14	118	110	106	89	21.5%	18.4%
dfsin	438	34	368	254	230	186	47.5%	42.0%
gsm	210	24	185	135	135	105	35.9%	35.8%
jpeg	939	47	950	369	359	309	61.8%	60.7%
mips	138	16	123	102	102	87	26.4%	25.9%
motion	287	24	247	146	143	128	50.1%	49.1%
sha	150	16	108	105	96	88	36.1%	30.3%
FFT	2398	231	2865	1258	996	767	58.5%	47.6%
Average	454	40	466	247	222	182	40.6%	38.1%

** indicates that it is a full compilation and not incremental. 0% means that the debug instrumentation is not present (removed).*

5.5 Impact on the Compile time

The following experiments were performed to demonstrate the reduction in the time taken for recompilation when an user wishes to change the debug scenario (recording different variables in our case) using our debug flows.

Table 5.3 shows the impact on total compile time of each circuit for Flow 1 (original flow) and Flow 3 (incremental flow with permanent taps). It includes the total time starting from the Analysis and Synthesis until the generation of the FPGA bitstream. Column 2 shows the overall compile time of each circuit in seconds for the baseline flow (Flow 1) which does not use incremental techniques (as in [23]). In order to create design partitions for Flow 3, we had to first run the Analysis and the Elaboration step to get the hierarchy of the design. This overhead time is shown in Column 3.

Columns 4-8 show the compile time for Flow 3 (incremental flow with permanent taps), in which the permanent taps are added to the user circuit and the changes to the RTL are localized to the debug partition. After partitioning the design, the circuit is compiled from scratch with the instrumentation added for 50% observability (we anticipate that the Flows 3 and 4 would be mostly used with reduced observability to gain faster debug turn around times). This is the first compilation and the user circuit is co-optimized along with the debug instrumentation. This is not incremental (the compile time is shown in Column 4). Then, the variables are changed to a different 50% subset and the circuit is recompiled (Incremental). Six different subsets of the variables were used for these experiments and the averaged results are shown in Column 5. Next we repeat the same experiment by starting with the instrumentation required for tracing 25% of the variables (Column 6-7). At any point the designer could decide to remove the instrumentation; this would not need a full recompile as the user partition is not modified (Column 8 shows the time taken for this step). The last two columns show the improvement in compile time for Flow 3 (25% and 50% observability) compared to the baseline flow (Flow 1).

Table 5.4 shows the same quantities for Flow 4 (incremental flow with permanent taps and late binding), in which the user circuit is placed and routed first and then the debug instrumentation is added incrementally. Column 2-3 are same as that of Table 5.3 which are the compile time for baseline flow (Flow1) and the analysis overhead respectively. Columns 4-7 show the compile time for this flow. Column 4 shows the time taken for the first compilation with no debug instrumentation. Columns 5-6 show the averaged results for six different subsets of 50% and 25% variables recorded, which are incremental compilations. Column 7 shows the time taken to remove the debug instrumentation for this flow and the last two columns show the improvement in compile time for Flow 4 (25% and 50% observability) compared to the baseline flow (Flow 1).

The overall compile time for Flow 3 (incremental flow with permanent taps) reduces by 40.9% for the 25% case and 40.0% for the 50% case. For Flow 4 (incremental flow with permanent taps and late binding) it reduces by 40.6% for the 25% case and 38.1% for the 50% case. This means that, although the initial run is somewhat longer (due to the analysis overhead) for Flows 3 and 4, each additional debug turn, in which the subset of user variables to be recorded is changed, is 38–40% faster. It should be noted that for the largest designs (*jpeg* and *FFT*), which have the largest compile times, the run-time was reduced the most (61.8%, 60.0% respectively). This suggests that this technique is scalable and will be especially helpful for large designs where users are most affected by running a full recompile with each debug iteration.

To better understand these results, we measured the size of debug instrumentation as a fraction of the overall instrumented circuit size for each circuit using our Flow 3 (incremental flow with permanent taps) implementation (would be almost the same if Flow 4 implementation was considered); these results are shown in Table 5.5. The last column indicates the percentage of the debug instrumentation in the overall circuit (column 3/(column 2 + column 3)). The ratio varied from 17% to 49%. For benchmarks such as *dfmul* where the debug instrumentation (49.1%) is as large as the user circuit, the compile time benefits observed were the least (21%) because the tool had to recompile a major portion of the design in these cases.

We also estimated the overhead in an incremental compile by running an

Table 5.5: Area Breakdown (For Flow 3)

Benchmark	User Circuit	Instrumentation (100%)		Debug Partition
		Trace Sched.	Other	
adpcm	7881	2064	700	26%
aes	7700	883	734	17.3%
blowfish	3410	965	685	32.6%
dfadd	3528	1628	654	39.2%
dfdiv	5950	1441	706	26.5%
dfmul	1702	1023	619	49.1%
dfsin	12066	4016	851	28.74%
gsm	4224	1029	946	31.8%
jpeg	21095	3478	1124	17.9%
mips	1826	991	712	48.3%
motion	7092	900	730	18.7%
sha	2167	500	622	34.1%
FFT	51703	13095	1750	22.3%
Average	10026	2463	833	30.2%

All area values are provided in number of Stratix IV ALMs. The other column represents the rest of the instrumentation logic like the Communication & Debug Manager, Trace recorder, State Encoder and Stepping & Breakpoint Unit which were described in Chapter 3.

incremental compilation twice in a row with no changes in between (Table 5.6); this run-time of the second compile averaged 160 seconds which we think is because of the analysis performed by the tool to determine changes in the design, if any. This overhead limits the benefits that are obtained using our current setup for the smaller circuits whose compile time is closer to this average overhead; clearly, as circuits grow or changes are better localized, impact of this overhead can be reduced.

5.6 Impact on the Frequency of User Circuit

The presence of debug instrumentation, design partitions and permanent taps in our flows might affect the performance of the user circuit. Because of these factors, there might still be a loss in the performance of the user circuit even after removing the debug instrumentation, once the debugging has been

Table 5.6: Incremental Compile Overhead for Flow 3 in Seconds

Benchmark	Incremental Compile Overhead
adpcm	146
aes	135
blowfish	108
dfadd	97
dfdiv	128
dfmul	86
dfsin	179
gsm	115
jpeg	287
mips	85
motion	128
sha	88
FFT	502
Average	160

finished. The following experiments quantify this loss in the frequency, if any, for our incremental debug flows (Flow 3 and Flow 4).

Table 5.7 shows the variation in the frequency of the design for Flow 3 (incremental flow with permanent taps) when compared to that of the original uninstrumented user circuit. Columns 2-3 show the f_{max} values of the design without and with the debug instrumentation (corresponding to 100% Observability) respectively for Flow 1. These f_{max} values were obtained from Quartus Prime 16.0 after compiling the design. On average there is a loss of 4.5% for the instrumented design when compared to the f_{max} of the original user circuit. This clearly shows that the instrumentation circuitry may perturb the user design, changing some of its timing paths.

Columns 4-6 show the f_{max} results for Flow 3 (incremental flow with permanent taps). First, a full compilation is run after instrumenting the logic required to trace a subset of 25% user variables (Column 4). Then we run six compilations using different subsets of 25% variables and average the results (Column 5). There is a loss of 9% in f_{max} when compared to the original circuit. When the instrumentation is removed, we get some of the performance back; Column 6 shows that there is a loss of 4.5% when compared to the original circuit. The reasons for this may be attributed to the creation of partitions, addition of permanent taps and the co-optimization of the user and the debug partitions in the initial compilation.

Similarly, Table 5.8 shows the variation in the frequency of the design for our Flow 4 (incremental flow with permanent taps and late binding) when compared to that of the original uninstrumented user circuit. Columns 2-3 show the f_{max} values of the design without and with the debug instrumentation for Flow 1 (same as that in Table 5.7). Columns 4-6 show the f_{max} results for Flow 4. In Flow 4, we start with an empty debug partition to ensure the user partition is optimized as much as possible. As the Column 4 shows, f_{max} for this initial compilation is roughly the same as the original uninstrumented user circuit (Column 2). We then replace the empty debug partition with instrumentation (required for 25% observability) using an incremental compilation. As the results in Column 5 show, this has a negative impact on the performance of the overall instrumented circuit; compared to the original circuit, adding the instrumentation lowers f_{max} by 15.7%. In some cases, the drop is larger; in *jpeg*, the overall instrumented circuit runs 41% slower than the uninstrumented version. We have observed that, in some cases, routing between the user circuit and the instrumentation becomes difficult due to congestion, causing nets to take circuitous routes. This does not occur to the same extent in Flow 3, since in that case, the user circuit and instrumentation are optimized simultaneously, meaning the user circuit can be adjusted to allow for connections within the instrumentation if necessary. Finally, when we remove the instrumentation, the value of f_{max} returns to a frequency very close to that of the original uninstrumented circuit; Column 6 shows that the frequency of the circuit after instrumentation has been removed is 1.3% slower than the uninstrumented circuit. The 1.3% loss is primarily due to the taps that are added and left in the user circuit once the instrumentation has been removed. However, it should be noted that the frequency which was obtained for the user circuit in the first compilation (with empty debug partition) is maintained throughout the debugging process and could be recovered even after the

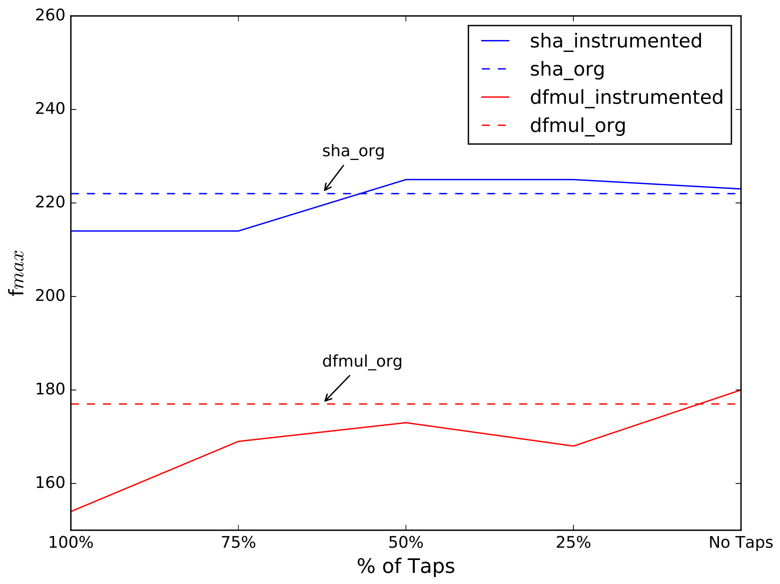


Figure 5.2: Frequency Variation with the number of taps

debug instrumentation is removed (can be seen from Columns 4 and 7 of Table 5.8).

To better understand this, we took the two benchmark circuits, namely *dfmul* and *sha*, for which the loss was very high and obtained the frequencies for these designs as the number of taps were reduced. Figure 5.2 shows how the f_{max} changes as these taps into the user partition were reduced. We randomly deleted the required number of taps for the purpose of these experiments. For both circuits, as the number of taps are reduced, f_{max} of the designs (sha_instrumented and dfmul_instrumented) reached the value of the corresponding original user circuits (sha_org and dfmul_org). This shows that the presence of such taps might slightly affect the frequency of the user circuit but however, it enables the possibility of efficient incremental recompilations. As seen from the Figure 5.2, the f_{max} values are not exactly the same as that of the original circuits when all the taps were removed. This slight variation may be attributed to the presence of partitions (even when we have no taps, we have an empty debug partition) in our design.

Table 5.7: Frequency Results: Flow 3 vs Flow 1

Benchmark	Flow 1		Flow 3		
	Original User Circuit (MHz)	With Instrumentation 100% Obs. (MHz)	First Compilation 25% Obs. (MHz)	Incremental Compilations 25% Obs. (MHz)	Instrumentation Removed 0% Obs. (MHz)
adpcm	134	134 (-0.6%)	129	129	129 (-4.1%)
aes	133	134 (0.6%)	126	125	126 (-5.3%)
blowfish	207	191 (-7.9%)	197	173	197 (-5.0%)
dfadd	190	196 (3.0%)	201	201	201 (5.6%)
dfdiv	196	184 (-6.5%)	190	190	179 (-9.1%)
dfmul	177	158 (-10.8%)	158	158	158 (-10.5%)
dfsin	179	168 (-6.4%)	173	164	173 (-3.3%)
gsm	162	171 (5.6%)	158	158	158 (-2.6%)
jpeg	98	94 (-3.4%)	95	84	98 (0.6%)
mips	173	172 (-0.6%)	154	158	163 (-5.8%)
motion	160	138 (-14.2%)	156	126	156 (-2.5%)
sha	222	205 (-7.9%)	222	200	230 (3.3%)
FFT	112	100 (-9.9%)	99	92	90 (-19.2%)
Average	165	157 (-4.5%)	158	151	158 (-4.5%)

For Columns 3 and 6 the values in the parenthesis indicates the % variation of the frequency with respect to that of the original user circuit (Column 2)

Table 5.8: Frequency Results: Flow 4 vs Flow 1

Benchmark	Flow 1		Flow 4		
	Original User Circuit (MHz)	With Instrumentation 100% Obs. (MHz)	First Compilation 0% Obs. (MHz)	Incremental Compilations 25% Obs. (MHz)	Instrumentation Removed 0% Obs. (MHz)
adpcm	134	134 (-0.6%)	135	134	135 (0.3%)
aes	133	134 (0.6%)	128	115	128 (-3.6%)
blowfish	207	191 (-7.9%)	215	173	215 (3.9%)
dfadd	190	196 (3.0%)	188	188	188 (-1.1%)
dfdiv	196	184 (-6.5%)	193	163	193 (-1.8%)
dfmul	177	158 (-10.8%)	154	154	154 (-12.9%)
dfsin	179	168 (-6.4%)	176	151	176 (-1.9%)
gsm	162	171 (5.6%)	166	166	166 (2.8%)
jpeg	98	94 (-3.4%)	100	59	100 (1.9%)
mips	173	172 (-0.6%)	172	163	170 (-1.7%)
motion	160	138 (-14.2%)	165	116	165 (3.2%)
sha	222	205 (-7.9%)	214	191	214 (-3.8%)
FFT	112	100 (-9.9%)	108	65	110 (-1.5%)
Average	165	157 (-4.5%)	163	141	163 (-1.3%)

For Columns 3 and 6 the values in the parenthesis indicates the % variation of the frequency with respect to that of the original user circuit (Column 2)

5.7 Summary

This chapter provided a detailed analysis of the results obtained for various experiments which were conducted to quantify the impact of selective variable tracing and also the impact of our incremental debug flows on compile time and the frequency of the user circuit.

As shown in Section 5.3, Selective Variable Tracing could achieve significant improvements in the trace window size and also a reduced debug instrumentation area. Our flows enable selective variable tracing to reduce the number of debug turns and also leverage the incremental compilation techniques to accelerate the debug turn around times by almost 40%, on average.

Like any other debug flow, our flows may also potentially interfere with the user circuit partition, reducing its maximum possible operating frequency. Section 5.6 shows that for Flow 3, the performance of the user circuit degrades to a much greater extent (loss of 4.5%) when compared to our Flow 4 (loss of 1.3%). However, the performance of the overall instrumented circuit is better for Flow 3 as the user and the debug partitions are co-optimized, when compared to Flow 4 where the debug partition uses the left over FPGA resources, possibly creating more critical paths. Clearly there is a trade-off between each of our flows and one has to choose them according to the application's requirements.

Chapter 6

Conclusions and Future Work

6.1 Overview

This chapter summarizes the significance of the work done in this thesis along with the contributions made and the important research findings (Section 6.2). In Section 6.3, we also present possible ideas to further explore in the direction of this work.

6.2 Summary

High Level Synthesis (HLS) simplifies the design process of a digital hardware system and makes it possible for software developers to make use of the hardware accelerators for their complex applications. However, for it to become successful, there is a need for an efficient debug infrastructure. In-system debug is becoming an important part of the HLS ecosystem because of its advantages over the usual simulation based approaches. Existing HLS debug techniques allow the user to debug a circuit at the source level as it runs on an FPGA, providing visibility into the run-time operation of the circuit. In order to do this, most such flows contain tools that automatically add additional circuitry (debug instrumentation) to record the circuit execution and then replay this information to provide a software-like debug experience. Typically, these tools record the updates to *all* the user visible variables and the control flow information.

In this thesis, we improved an existing in-system HLS debug framework by allowing the ability to selectively record only *some* user-visible variables in on-chip trace buffers. This leads to reduced instrumentation logic (by 1.7x when 50% variables are recorded and 2.8x when 25% variables are recorded) and a longer trace window (1.6x for 50% case and 2.6x for 25% case), meaning fewer debug turns may be required when searching for an elusive bug. In our original framework, if the user needs to change the variables to be recorded, it would require a full recompilation of the design, which is generally not desirable. To make this practical, incremental compilation techniques are essential to reduce the debug turn around time.

Although commercial FPGA tools contain extensive support for incremental compilation, we found that, due to several unique characteristics of the considered debug instrumentation (like the customized *trace scheduler* logic), careful application of these techniques is required. We outlined several flows to perform incremental compilation for our problem. Of the flows we examined, two were deemed promising: the first, in which the user circuit and instrumentation are co-optimized during compilation, gives the fastest debug clock speeds, but suffers in user circuit performance once the debug instrumentation is removed (a frequency loss of 4.5%). In the other promising flow, the user circuit is first placed and routed without the instrumentation logic, giving the best possible performance of the user circuit. Then, the instrumentation is added incrementally without changing the user circuit. This flow suffers somewhat in terms of debug performance, however, when the instrumentation is removed, the circuit runs almost as fast as the original uninstrumented user circuit (only a 1.3% loss in frequency). Using either flow, we achieve a significant reduction (40%, on average) in debug turn-around times, leading to more effective debug and higher productivity.

To our knowledge, ours is the first work to incorporate the incremental compilation techniques into an in-system HLS debug flow. There are many works which focus on incremental debug for the circuits implemented at the RTL level, however, the debug of HLS generated circuits is different as the instrumentation inserted is highly customized based on the variables selected (to achieve better trace window size). As a result, the amount of logic that needs to be recompiled when a user wants to record different variables is much larger.

6.3 Future Work

Short Term Goals

In order to incrementally place-and-route the changed logic between successive debug turns, we used the incremental techniques from a commercial tool (Quartus Prime 16.0) and modified our design accordingly (creating design partitions and adding taps) to get the most out of this tool.

Similar to Quartus Prime, there are several other FPGA CAD tools like Vivado which also provide support for incremental recompilations. One of our future works would be to understand the incremental flow offered by these tools and use them in the proposed debug flows to see if we could achieve any higher reductions in the compile time when compared to our current results.

Another possible work is to investigate the use of generic/customized lossless data compression schemes to compress the data generated by the trace scheduler block before writing it to the trace buffer (in order to increase the trace window size) and then evaluate the benefits obtained using our incremental flows. We anticipate that the results might slightly vary based on the amount of area overhead and also the amount of logic that would change with the change in the variables being recorded.

In this thesis we do not use any special variable selection algorithms to guide the user to select important variables for recording. We randomly select the variables in each debug turn to evaluate the effectiveness of our incremental debug flows. It would be interesting to evaluate our flows with proper variable selections in each debug turn, as a user would actually do while debugging (something like selecting variables function-by-function).

As most of the available HLS benchmarks are very small and have a compile time of less than few minutes (except one or two) we feel that our results might be suppressed. We also anticipate that the compile time reductions offered by our incremental debug flows would go up if they are used with bigger benchmarks. In future, we expect the availability of bigger HLS benchmarks or some open source practical HLS applications to evaluate our flows and get more insight.

Long Term Goals

The use of commercial incremental techniques in this work was just the first step towards our ultimate long term goal of developing a fully accelerated HLS debug framework, allowing the users to debug quickly and efficiently. The benefits achieved by using these commercial frameworks cannot be improved further as we do not have access to the internal of the tools and hence we cannot modify the algorithms used in their incremental flows. An obvious solution would be to develop a customized incremental place-and-route tool with our own requirements using open source FPGA CAD tools like RapidSmith [48].

Another possible orthogonal approach to our work is to investigate the feasibility of an overlay for HLS debug, similar to those used for incremental RTL based debug [15, 16, 32]. However, as described in Section 2.4, we feel that this is not so easy given the uniqueness of our debug instrumentation for each subset of variables that are being recorded.

Lastly, we feel that there is another direction to explore in which there could be no need for recompilation at all. For this purpose, it is necessary to develop a generic trace scheduler and a compression circuit which achieves compression ratio as close as that of a customized trace scheduler for each subset of variables being recorded. Then by using a scan register, we could mask off the respective signals that are not being recorded in the current debug turn with out any recompilation. Clearly, there is a trade-off between recompilation and the trace buffer utilization or the trace window size (which depends on the compression achieved by the trace scheduler block). This trade-off would not be clear until the generic compression circuit is developed, which we leave as future work.

Bibliography

[1] Christopher M. Abernathy, Lydia M. Do, Ronald P. Hall, and Michael L. Karm. System and Method for Streaming High Frequency Trace Data Off-Chip. `http://www.freepatentsonline.com/y2008/0016408.html`, Jan 2008. (visited on August 8, 2017).

[2] Amazon. Amazon EC2 F1 Instances with Custom FPGAs. `https://aws.amazon.com/ec2/instance-types/f1/`, 2016. (visited on August 8, 2017).

[3] Hari Angepat, Gage Eads, Christopher Craik, and Derek Chiou. Nifd: Non-intrusive fpga debugger – debugging fpga 'threads' for rapid hw/sw systems prototyping. In *FPL*, pages 356–359. IEEE Computer Society, 2010.

[4] E. Anis and N. Nicolici. Low cost debug architecture using lossy compression for silicon debug. In *Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

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

[6] ARM. Differences between On-chip and Off-chip Storage. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dgi0012d/Babhaifj.html`. (visited on August 8, 2017).

[7] Christian Beckhoff, Dirk Koch, and Jim Trresen. The xilinx design

language (xdl): Tutorial and use cases. In *ReCoSoC*, pages 1–8. IEEE, 2011.

[8] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. pages 175–184, 1998.

[9] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, April 1986.

[10] N. Calagar, S.D. Brown, and J.H. Anderson. Source-level Debugging for FPGA High-Level Synthesis. In *International Conference on Field Programmable Logic and Applications*, Sept 2014.

[11] A. Canis, S.D. Brown, and J.H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[12] Andrew Canis, Jongsok Choi, et al. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[13] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andrs Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[14] John Curreri, Greg Stitt, and Alan D. George. High-level Synthesis of In-circuit Assertions for Verification, Debugging, and Timing Analysis. *Int. J. Reconfig. Comput.*, 2011:1:1–1:17, January 2011.

[15] F. Eslami and S. J. E. Wilton. Incremental distributed trigger insertion for efficient fpga debug. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2014.

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

[17] GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`. (visited on August 8, 2017).

[18] J. Goeders. Enabling Long Debug Traces of HLS Circuits Using Bandwidth-Limited Off-Chip Storage Devices. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–143, April 2017.

[19] J. Goeders and S. J. E. Wilton. Using round-robin tracepoints to debug multithreaded hls circuits on fpgas. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 40–47, Dec 2015.

[20] J. Goeders and S.J.E. Wilton. Effective FPGA debug for high-level synthesis generated circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[21] J. Goeders and S.J.E. Wilton. Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, May 2015.

[22] Jeffrey Goeders. *Techniques for In-System Observation-based Debug of High-Level Synthesis Generated Circuits on FPGAs*. PhD thesis, The University of British Columbia (Vancouver), September 2016.

[23] Jeffrey Goeders and Steven J. E. Wilton. Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):83–96, January 2017.

[24] Paul Graham, Brent Nelson, and Brad Hutchings. Instrumenting Bitstreams for Debugging FPGA Circuits. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 41–50, 2001.

[25] Steve Guccione, Delon Levi, and Prasanna Sundararajan. Jbits: Java based interface for reconfigurable computing. In *Second Annual Military*

and Aerospace Applications of Programmable Devices and Technologies (MAPLD), Sep.

[26] M. Ben Hammouda, P. Coussy, and L. Lagadec. A Design Approach to Automatically Synthesize ANSI-C Assertions During High-Level Synthesis of Hardware Accelerators. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 165–168, June 2014.

[27] Julien Happich. Cognitive Computing Platform Unites Xilinx and IBM. `http://www.eetimes.com/document.asp?doc_id=1329377`, Apr 2016. (visited on August 8, 2017).

[28] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[29] K.S. Hemmert, J.L. Tripp, B.L. Hutchings, and P.A. Jackson. Source level debugger for the Sea Cucumber synthesizing compiler. In *Symposium on Field-Programmable Custom Computing Machines.*, pages 228–237, April 2003.

[30] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data path allocation based on bipartite weighted matching. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 499–504, New York, NY, USA, 1990. ACM.

[31] E. Hung and S. J. E. Wilton. Speculative Debug Insertion for FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 524–531, Sept 2011.

[32] Eddie Hung and Steven J. E. Wilton. Accelerating FPGA Debug: Increasing Visibility Using a Runtime Reconfigurable Observation and Triggering Network. *ACM Trans. Des. Autom. Electron. Syst.*, 19(2):14:1–14:23, March 2014.

[33] Eddie Hung and Steven J.E. Wilton. 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*, pages 19–28, 2013.

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

[35] B. L. Hutchings and J. Keeley. Rapid post-map insertion of embedded logic analyzers for xilinx fpgas. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 72–79, May 2014.

[36] Impulse Accelerated Technologies. CoDeveloper from Impulse Accelerated Technologies. `http://www.impulseaccelerated.com/ReleaseFiles/Help/iAppMan.pdf`, 2015. (visited on August 8, 2017).

[37] Intel. Increasing Productivity With Quartus II Incremental Compilation, month=May, year=2008, version=1.0, howpublished = White Paper WP-01062-1.0.

[38] Intel. Protecting the FPGA Design From Common Threats, month=June, year=2009, version=1.0, howpublished = White Paper WP-01111-1.0.

[39] Intel. Intel Completes Acquisition of Altera. `https://newsroom.intel.com/press-kits/intel-acquisition-of-altera/`, December 2015. (visited on August 8, 2017).

[40] Intel. *Quartus Prime Pro Edition Handbook*, volume 3, chapter 9: Design Debugging Using the SignalTap II Logic Analyzer. November 2015.

[41] Intel. SDK for OpenCL. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`, 2016. (visited on August 8, 2017).

[42] Intel. Quartus prime standard edition handbook: Design and synthesis. `https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf`, May 2017. (visited on August 8, 2017).

[43] Yousef Iskander, Cameron Patterson, and Stephen Craven. High-level abstractions and modular debugging for fpga design validation. *ACM Trans. Reconfigurable Technol. Syst.*, 7(1):2:1–2:22, Feb 2014.

[44] J. Jiang and S. Jones. Word-based dynamic algorithms for data compression. *IEE Proceedings I - Communications, Speech and Vision*, 139(6):582–586, Dec 1992.

[45] Eric Keller. Jroute: A run-time routing api for fpga hardware. In *Proceedings of the IPDPS Workshops on Parallel and Distributed Processing*, pages 874–881, 2000.

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

[47] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, 2004.

[48] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *International Conference on Field Programmable Logic and Applications*, pages 349–355, Sept 2011.

[49] X. Liu and Q. Xu. On signal selection for visibility enhancement in trace-based post-silicon validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(8):1263–1274, Aug 2012.

[50] Mentor. Catapult high-level synthesis. `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`, 2016. (visited on August 8, 2017).

[51] Microsemi. In-Circuit FPGA Debug: Challenges and Solutions. `https://www.microsemi.com/document-portal/doc_view/133662-in-circuit-fpga-debug-challenges-and-solutions`. (visited on August 8, 2017).

[52] J. S. Monson and B. Hutchings. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *Int'l Conf. on Field-Programmable Logic and Applications*, pages 1–6, Sept 2014.

[53] J. S. Monson and B. Hutchings. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *International Conference on Field Programmable Logic and Applications*, Sept 2014.

[54] J. S. Monson and B. Hutchings. Using shadow pointers to trace c pointer values in fpga circuits. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[55] J. S. Monson and Brad L. Hutchings. Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 5–8, 2015.

[56] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, Aug 2012.

[57] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP(99), 2016.

[58] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[59] University of Cambridge. The Tiger MIPS processor. `https://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html`, 2010. (visited on August 8, 2017).

[60] United States Bureau of Labor Statistics. Occupational Outlook Handbook, 2012.

[61] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013.

[62] J. P. Pinilla and S. J. E. Wilton. Enhanced source-level instrumentation for fpga in-system debug of high-level synthesis designs. In *International Conference on Field-Programmable Technology (FPT)*, pages 109–116, Dec 2016.

[63] Z. Poulos, Y. S. Yang, J. Anderson, A. Veneris, and B. Le. Leveraging reconfigurability to raise productivity in fpga functional debug. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 292–295, March 2012.

[64] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[65] Qualcomm. Qualcomm & Xilinx Collaborate to Deliver Industry-Leading Heterogeneous Computing Solutions for Data Centers with New Levels of Efficiency and Performance. `https://www.qualcomm.com/news/releases/2015/10/08/qualcomm-and-`

`xilinx-collaborate-deliver-industry-leading-heterogeneous`, 2015. (visited on August 8, 2017).

[66] B. Reagen, R. Adolf, Y.S. Shao, Gu-Yeon Wai, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Int'l Symposium on Workload Characterization*, pages 110–119, Oct 2014.

[67] Synopsys. Identify: Simulator-like Visibility into FPGA Hardware Operation. `https://www.synopsys.com/implementation-and-signoff/fpga-based-design/identify-rtl-debugger.html`. (visited on August 8, 2017).

[68] Anurag Tiwari and Karen A. Tomko. Scan-chain based watch-points for efficient run-time debugging and verification of fpga designs. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, Jan.

[69] K. A. Tomko and A. Tiwari. Hardware/software co-debugging for reconfigurable computing. In *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, HLDVT '00, Washington, DC, USA, 2000. IEEE Computer Society.

[70] Bart Vermeulen and Sandeep Kumar Goel. Design for debug: Catching design errors in digital chips. *IEEE Des. Test*, 19(3):37–45, May 2002.

[71] Timothy Wheeler, Paul S. Graham, Brent E. Nelson, and Brad L. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *Field-Programmable Logic and Applications, 11th International Conference, FPL*, pages 483–492, Aug 2001.

[72] Xilinx. FPGA vs. ASIC. `https://www.xilinx.com/fpga/asic.htm`. (visited on August 8, 2017).

[73] Xilinx. Vivado Design Suite User Guide: Implementation. `https://www.xilinx.com/support/documentation/sw_manuals/`

xilinx2012_4/ug904-vivado-implementation.pdf, Dec 2012. (visited on August 8, 2017).

[74] Xilinx. Virtex-6 FPGA Configuration: User Guide. https://www.xilinx.com/support/documentation/user_guides/ug360.pdf, Nov 2013. (visited on August 8, 2017).

[75] Xilinx. Integrated Logic Analyzer v6.1: LogiCORE IP Product Guide. http://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf, April 2016. (visited on August 8, 2017).

[76] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf, June 2016. (visited on August 8, 2017).

Appendix

Appendix A

A Guide to our GUI Framework

Steps to use Our Proposed Debug Flows:

1. *Open a Design:*

   – Once the GUI is loaded, click on the open file icon and select the design folder which consists of the .c, make and the config files.

   – After selecting the folder and clicking the "Open" button, LegUp is run in the background (assuming all variables are being traced) to create the database and also to have an initial design with all the taps present in the user module. This is an RTL generation step and no (Quartus Prime) compilation is necessary at this moment. At the same time, all the debug modules present in the design are isolated into separate Verilog files as it is required to have the partitions in separate files for the tool to perform the incremental compilation effectively. All these (main) files are stored in a new sub-folder "quartus_proj", so as to avoid any overwriting/deletion when LegUp is rerun for the current design.

   – Next, the information from the database is read and populated into the "Vars" tab (shown in Figure A.1). A list of all the variables present in the source code can be seen under this tab.

2. *Select the variables which are to be recorded:*

   – In the "Vars" tab, a list of all the variables are displayed along with a corresponding checkbox.

– Once the required variables are selected, click on the "Trace Selected Vars" button. In background, this triggers the creation of a file named "selectedVars.txt" and also reruns LegUp, which has been modified to make use of the information from this file and generate the RTL (Verilog file) with the instrumentation for recording the selected variables. The .v file generated by LegUp is in the design folder and has not been moved into our main "quartus_proj" sub-folder. We only need partial contents from this file, which would be copied to our main files when a Quartus Prime compilation is to be run.

– Next, as LegUp is rerun, a new database is created. Therefore, the connection to the database is refreshed and all the relevant information is populated again from the database.

3. *Creating a new Quartus Prime project (if it does not exist in the "quartus_proj" folder):*

   – After selecting the variables, go to the "FPGA" tab (shown in Figure A.2). Click on the "Create Quartus Project" button. If a project already exists, a pop up indicating this will be displayed. If not, a new project is created. Presently, the target device is Intel's CycloneV DE1-SoC.

   – After the project is created, an "Analysis and Elaboration step" is run in order to create design partitions using appropriate Tcl files.

4. *Running a full compilation (if this is the first compilation):*

   – Before running the compilation, the user has the option to start with empty debug partitions which allow the user design to be place and routed efficiently (Flow 4). If this is selected, then the partition preservation settings for the debug partitions are set to EMPTY and the user partition is set to SOURCE as this is the first compilation. If the option is not selected then all the partitions including the debug modules are set to SOURCE. If a partition is set to SOURCE, the tool recompiles it from scratch.

– Next we copy the traceScheduler logic and some other parameters which change with every run of LegUp from the .v file present in the design folder into the .v files (present in our "quartus_proj" folder) which need them. This is done to ensure that we are running the compilation for the design with instrumentation added for only the selected variables.

– After this, a full compilation for the design is run.

5. *Program the bitstream to the FPGA:*

   – After compiling the design click "Program Bitstream" button to program the DE1-SoC FPGA.

6. *Connect to the FPGA through the RS232 interface.*

7. *Run the design on FPGA and analyze the variables:*

   – Once connected to the FPGA, the design can be run (until a breakpoint or until it is completed) by clicking the "run" icon.

   – After running the design, go to "FPGA Replay Execution" mode to see how the variable values were changed as the design was running. You can use the slider present to go back and forth or use the single step/step back icons to move one step at a time.

8. *Changing the variables to be recorded and performing an incremental compilation:*

   – During debugging, if you feel some variables are not necessary then you may not want to record them (thus saving the trace memory/increasing the trace window length).

   – You can select which variables you want to record (follow the same procedure as in Step 2).

   – Next DON'T create a new Quartus Prime project, as we already have a project. Also, DON'T run a full compilation, if you want to speed up your compilation. You can use incremental compilation as you already have a previous compilation results.

9. *Incremental Compilation:*

   – When you click this button, the partition preservation settings are changed to POST_FIT using Tcl files, which directs the tool to reuse the results for the partitions which did not change from the previous compilations. In our case the user partition would not change as only the debug instrumentation is changed when the variables to be traced are changed. This preserves the results for the user partition. This saves, up to, on average, 40% of the runtime.

   – Before running the compilation, we again copy (from the design folder to our "quartus_proj" folder as in Step 4) the traceScheduler logic and other parameters which might have changed, as the LegUp would have been rerun when the variables to be recorded are changed.

   – Now a compilation is run, which would be incremental as the partition preservation settings were changed to POST_FIT.

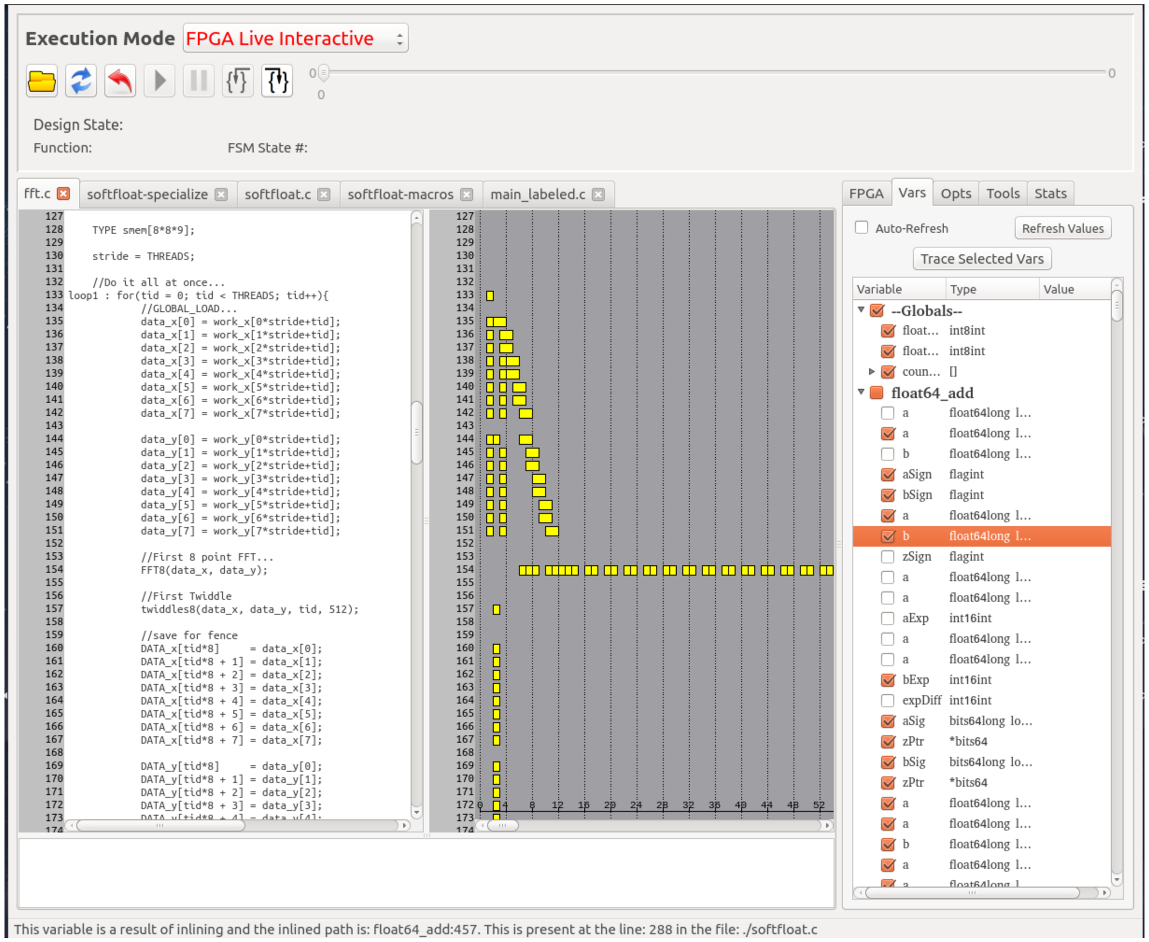
   – Follow Steps 5,6 to program and connect to FPGA. Then (as in Step 7) we can analyze the updates to the variable values during the design execution.

10. *If the bug is not found, repeat the steps starting from Step 8 until the root cause of the bug is identified.*

Note:

– For some variables, even if they are not selected for recording, you could see the values showing up for them while debugging. This is because of the pointer aliasing happening with that variable and we are being conservative in such cases and recording these variables anyway.

– For now, we are using split buffer architecture (one each for datapath registers, memory and control signals). The GUI support for single trace buffer configuration is still under development.

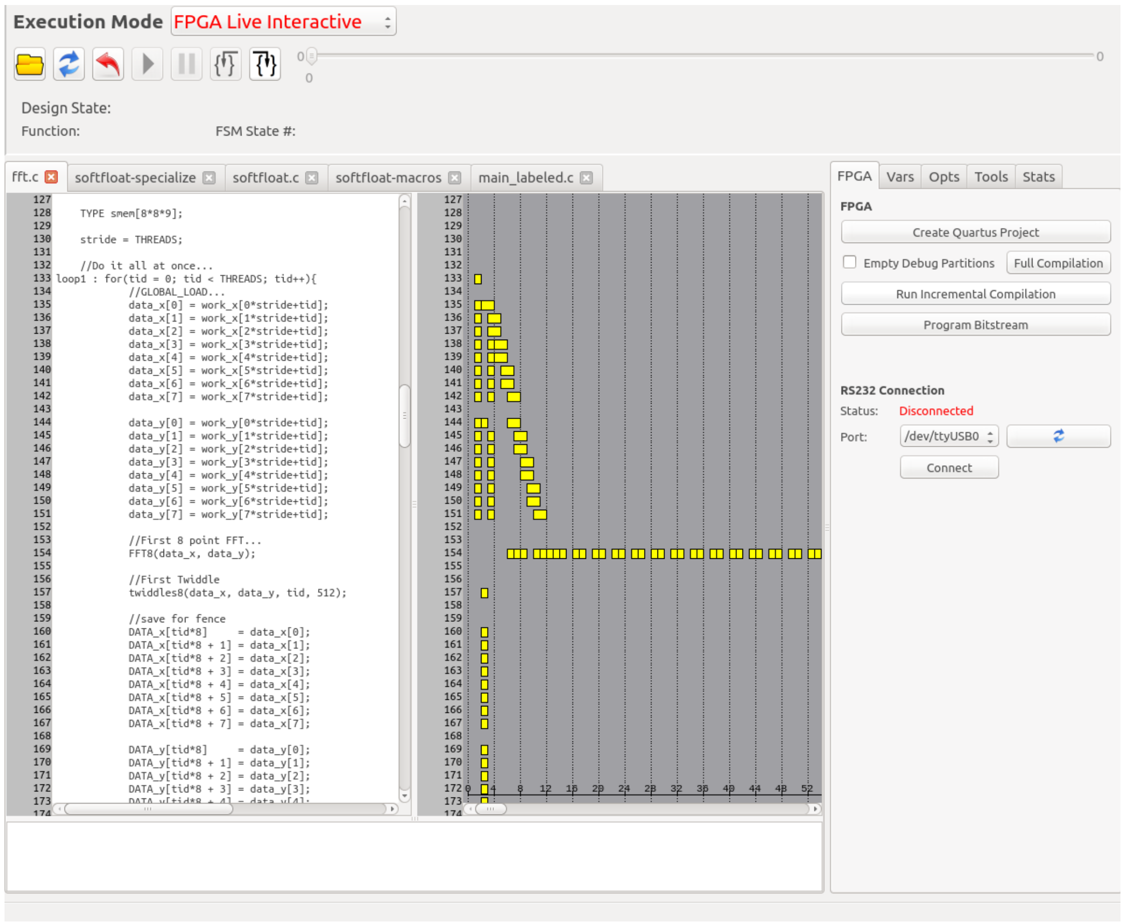


```
    TYPE smem[8*8*9];

    stride = THREADS;

    //Do it all at once...
loop1 : for(tid = 0; tid < THREADS; tid++){
            //GLOBAL_LOAD...
            data_x[0] = work_x[0*stride+tid];
            data_x[1] = work_x[1*stride+tid];
            data_x[2] = work_x[2*stride+tid];
            data_x[3] = work_x[3*stride+tid];
            data_x[4] = work_x[4*stride+tid];
            data_x[5] = work_x[5*stride+tid];
            data_x[6] = work_x[6*stride+tid];
            data_x[7] = work_x[7*stride+tid];

            data_y[0] = work_y[0*stride+tid];
            data_y[1] = work_y[1*stride+tid];
            data_y[2] = work_y[2*stride+tid];
            data_y[3] = work_y[3*stride+tid];
            data_y[4] = work_y[4*stride+tid];
            data_y[5] = work_y[5*stride+tid];
            data_y[6] = work_y[6*stride+tid];
            data_y[7] = work_y[7*stride+tid];

            //First 8 point FFT...
            FFT8(data_x, data_y);

            //First Twiddle
            twiddles8(data_x, data_y, tid, 512);

            //save for fence
            DATA_x[tid*8]     = data_x[0];
            DATA_x[tid*8 + 1] = data_x[1];
            DATA_x[tid*8 + 2] = data_x[2];
            DATA_x[tid*8 + 3] = data_x[3];
            DATA_x[tid*8 + 4] = data_x[4];
            DATA_x[tid*8 + 5] = data_x[5];
            DATA_x[tid*8 + 6] = data_x[6];
            DATA_x[tid*8 + 7] = data_x[7];

            DATA_y[tid*8]     = data_y[0];
            DATA_y[tid*8 + 1] = data_y[1];
            DATA_y[tid*8 + 2] = data_y[2];
            DATA_y[tid*8 + 3] = data_y[3];
```

Variable	Type	Value
✓ --Globals--		
✓ float…	int8int	
✓ float…	int8int	
✓ coun…	[]	
float64_add		
☐ a	float64long l…	
✓ a	float64long l…	
☐ b	float64long l…	
✓ aSign	flagint	
✓ bSign	flagint	
✓ a	float64long l…	
✓ b	float64long l…	
☐ zSign	flagint	
☐ a	float64long l…	
☐ a	float64long l…	
☐ aExp	int16int	
☐ a	float64long l…	
☐ a	float64long l…	
✓ bExp	int16int	
☐ expDiff	int16int	
✓ aSig	bits64long lo…	
✓ zPtr	*bits64	
✓ bSig	bits64long lo…	
✓ zPtr	*bits64	
✓ a	float64long l…	
✓ a	float64long l…	
✓ b	float64long l…	
✓ a	float64long l…	

This variable is a result of inlining and the inlined path is: float64_add:457. This is present at the line: 288 in the file: ./softfloat.c

Figure A.1: 'Vars' Tab



Execution Mode: FPGA Live Interactive

Design State:

Function: FSM State #:

```
    TYPE smem[8*8*9];

    stride = THREADS;

    //Do it all at once...
loop1 : for(tid = 0; tid < THREADS; tid++){
            //GLOBAL_LOAD...
            data_x[0] = work_x[0*stride+tid];
            data_x[1] = work_x[1*stride+tid];
            data_x[2] = work_x[2*stride+tid];
            data_x[3] = work_x[3*stride+tid];
            data_x[4] = work_x[4*stride+tid];
            data_x[5] = work_x[5*stride+tid];
            data_x[6] = work_x[6*stride+tid];
            data_x[7] = work_x[7*stride+tid];

            data_y[0] = work_y[0*stride+tid];
            data_y[1] = work_y[1*stride+tid];
            data_y[2] = work_y[2*stride+tid];
            data_y[3] = work_y[3*stride+tid];
            data_y[4] = work_y[4*stride+tid];
            data_y[5] = work_y[5*stride+tid];
            data_y[6] = work_y[6*stride+tid];
            data_y[7] = work_y[7*stride+tid];

            //First 8 point FFT...
            FFT8(data_x, data_y);

            //First Twiddle
            twiddles8(data_x, data_y, tid, 512);

            //save for fence
            DATA_x[tid*8]     = data_x[0];
            DATA_x[tid*8 + 1] = data_x[1];
            DATA_x[tid*8 + 2] = data_x[2];
            DATA_x[tid*8 + 3] = data_x[3];
            DATA_x[tid*8 + 4] = data_x[4];
            DATA_x[tid*8 + 5] = data_x[5];
            DATA_x[tid*8 + 6] = data_x[6];
            DATA_x[tid*8 + 7] = data_x[7];

            DATA_y[tid*8]     = data_y[0];
            DATA_y[tid*8 + 1] = data_y[1];
            DATA_y[tid*8 + 2] = data_y[2];
            DATA_y[tid*8 + 3] = data_y[3];
            DATA_y[tid*8 + 4] = data_y[4];
```

FPGA

RS232 Connection

Status: Disconnected

Port: /dev/ttyUSB0

Figure A.2: 'FPGA' Tab