Improved Abstractions and Turnaround Time for FPGA Design Validation and Debug

Yousef Shafik Iskander

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Cameron D. Patterson, Chair
Thomas L. Martin
Paul D. Plassmann
Sedki M. Riad
Ricardo A. Burdisso

August 6, 2012
Blacksburg, Virginia

Keywords: FPGA, reconfigurable computing, debug, design validation, partial reconfiguration, development

Improved Abstractions and Turnaround Time for FPGA Design Validation and Debug

Yousef Shafik Iskander

(ABSTRACT)

Design validation is the most time-consuming task in the FPGA design cycle. Although manufacturers and third-party vendors offer a range of tools that provide different perspectives of a design, many require that the design be fully re-implemented for even simple parameter modifications or do not allow the design to be run at full speed. Designs are typically first modeled using a high-level language then later rewritten in a hardware description language, first for simulation and then later modified for synthesis. IP and third-party cores may differ during these final two stages complicating development and validation. The developed approach provides two means of directly validating synthesized hardware designs. The first allows the original high-level model written in C or C++ to be directly coupled to the synthesized hardware, abstracting away the traditional gate-level view of designs. A high-level programmatic interface allows the synthesized design to be validated with the same arbitrary test data on the same framework as the hardware. The second approach provides an alternative view to FPGAs within the scope of a traditional software debugger. This debug framework leverages partially reconfigurable regions to accelerate the modification of dynamic, software-like breakpoints for low-level analysis and provides a automatable, scriptable, command-line interface directly to a running design on an FPGA.

This work is supported by DARPA and the United States Army under Contract Number W31P4Q-08-C-0314. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Approved for Public Release, Distribution Unlimited.

Dedication

To my unconditionally loving and supportive parents, Shafik and Horia Iskander, to whom I owe so much in teaching me the value of education and making sacrifices for me to have one.

To my wife Tere and son Alex, who have brought me untold joy with their wonderful distractions. I wrote much of my preliminary exam while holding Alex as an infant so he could sleep, and took many breaks to play blocks while writing this dissertation.

Acknowledgments

I am permanently indebted to many people for the completion of this dissertation. I have surely forgotten to mention someone, and am unable to properly express my gratitude to those I have listed.

My adviser, Cameron Patterson provided me with an opportunity when I had all but given up and provided exceptional support and meaningful insight which has been the real education. I am grateful for his mentoring. Thanks are also due to Dr. Jaime de la Ree for helping get past some particularly difficult obstacles. Thanks are of course due to all who served on my committee, Tom Martin, Sedki Riad, Paul Plassmann, and Ricardo Burdisso who held me to very high standards to which I can be proud of.

I'm indebted to my longtime friend Stephen Craven who not only encouraged me to continue when I was about to give up, but provided the motivation to return to school to pursue my doctorate, provided an opportunity to do so, and pushed me to do even better. He also provided invaluable support and advice throughout.

I'm grateful to the many special friends I made during my once in a lifetime stay in Los Alamos, New Mexico at the Los Alamos National Laboratory. Thanks are due to Kei Davis, Heather Quinn, Zack Baker, Paul Graham, Sara del Valle, Hugh Greenberg, and Tony Salazar for selflessly sharing their knowledge, friendship, and experience; to M'hamed "Jebb" Jebbanema for his friendship and hiking trips through White Rock Canyon; and to

Jim and Gabriela Stiner for memorable evenings at the “Stiner Diner” at their lovely home in White Rock.

I’m also grateful to Dr. Linda Wills of Georgia Tech, a patient and insightful woman to whom I owe so much for forming my first graduate school experience. She was instrumental in my desire to return to complete my doctorate and my success as graduate student. Her husband Scott was also an influential professor. Sadly, he passed away while I was writing this dissertation.

Thanks to the members of the Configurable Computing Machinery (CCM) Lab at Virginia Tech, including Neil Steiner, Lee Lerner, Tingting Meng, and the students who worked on this project with me: Athira Chandrasekharan, Suresh Raja Gopalan, Tony Frangieh, and Guruprasad Subbarayan. Thanks also to Brian Knight for his invaluable support on this project and his friendship.

Thanks is due to the always friendly staff of Sub Station II in Blacksburg, Va. I was sitting in that restaurant with my good friend Stephen Craven when I decided to attend Virginia Tech and it is also there I spent many enjoyable lunches thinking about my work. I highly recommend the #9.

I, like so many other in this field, are indebted to the authors of open-source software which is frequently used without credit or thanks to the many anonymous authors. Please see the Appendix for a listing of the open-source software packages that were instrumental in this research.

This work was performed at Virginia Tech in Blacksburg, Va. with support provided through Luna Innovations Incorporated in Roanoke, Va.

Contents

List of Figures

List of Tables

Listings

Acronyms

ABV	Assertion-Based Verification
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
BSCAN	Boundary Scan
CFFT	Complex Fast Fourier Transform
DRC	Design Rule Checks
DSP	Digital Signal Processor
DUT	Device Under Test
EDA	Electronic Design Automation
EDIF	Electronic Data Interchange Format
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GSR	Global Set/Reset
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HLL	High-Level Language
HLS	High-Level Synthesis
HLV	High-Level Validation
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IDE	Integrated Development Environment
I/O	Input/Output
IP	Intellectual Property
JTAG	Joint Test Action Group

LED	Light Emitting Diode
LLD	Low-Level Debug
MAC	Media Access Control
NoC	Network-on-Chip
OS	Operating System
PAL	Programmable Array Logic
PAR	Place-and-Route
PATIS	Partial module-producing, Automatic, Timing-aware, Incremental, Speculative floorplanner
PDC	Programmable Debug Controller
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level
RTR	Runtime Reconfiguration
SHA-1	Secure Hash Algorithm
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

Chapter 1

Introduction

Digital system validation has reached a critical turning point. With the rate of new productions at an all time high due to a rapidly increasing consumer and commercial demand, the importance of an efficient and productive debug methodology has reached a tipping point. Time-to-market is a critical metric directly tied to production cost, however the increasing complexity and density of these systems works directly against the goal of improving efficiency and productivity.

In the last 20 years Field Programmable Gate Arrays (FPGA), a type of programmable logic device, have dramatically reduced design and validation times for developing complex digital systems. FPGAs give developers a near instantaneous means of prototyping and physically validating a digital design in real world conditions, when compared to the weeks-long wait required to manufacture an Application Specific Integrated Circuit (ASIC). However, even this level of rapid prototyping environment is not as flexible as needed. Build times for FPGAs may take days making even small changes a lengthy and unpleasant process.

In addition to long build times, FPGAs are notorious for the lack of visibility and control into the design, possibly the two most important attributes of validating and understanding any

system. By their very nature, FPGAs are blank slates allowing any arbitrary, custom design to created. The excellent reputation they have earned in high-performance, small-area, and low-power industry sectors such as telecommunications, defense, and high-performance computing, as well as an effective prototyping technology for practically any sector highlights the necessity for increasing productivity and efficiency during validation and debug.

Several debug approaches are currently in widespread use. FPGA vendors, who are able to leverage their own knowledge of these proprietary devices overwhelmingly favor recreating the look and feel of an actual logic analyzer. To do so, special cores need to be configured and built into the design. Embedded logic analyzers internally record signal activity based on preselected criteria defined during the build phase and are physically integrated into the design. Third-party vendors, at a disadvantage by not being able to seamlessly integrate their products into the normal design flow, rely on add-on modules also integrated into the design, but also rely on aging interfaces that reduce the effectiveness, speed, and interactivity required for validation and debug.

FPGAs are unique in that they can reprogrammed an unlimited number of times, while some even allow one or more sub-regions of the configurable fabric to reconfigured independently from the rest of the device. This Runtime Reconfiguration (RTR) or Partial Reconfiguration (PR) capability has, to our knowledge, never been been leveraged for development-time methodologies. In addition, the ability to validate against the original software reference model, also suffers. Once hardware development begins, the reference model is physically isolated from the rest of the design process, and must be repeatedly and manually checked.

The wealth of external interfaces and cores available on FPGAs allows some unique approaches for debug. Software allows a great amount of flexibility since it can be easily changed within a matter of minutes, and the availability of small, unobtrusive processors that can be embedded into a design afford interesting new approaches to design validation.

If the perspective and context of debugging is changed from outside the FPGA to within, design validation and exploration are no longer limited in visibility, and the entire scope of all design registers are easily accessible. Software control of a design allows interesting approaches to exploration and inspection, and if coupled with a means of controlling execution, unprecedented perspective and insight into a working design is possible.

1.1 Motivation

Once the design is implemented and programmed into the FPGA, debugging or analyzing the design is difficult. There are no general purpose input/output facilities such as a keyboard or console as with a general-purpose computer, and no industry-wide standard targeted specifically for FPGAs. Current solutions are often borrowed from the ASIC industry, such as Joint Test Action Group (JTAG), which addresses a limited portion of the design. Manufacturer solutions, as well as the multitude of ad-hoc solutions also only allow a limited view of the design either spatially or temporally. The trade-off being that only a limited portion of the design can be viewed or a limited segment of the execution can be observed. Changing the perspective of this view is tightly coupled to the actual implementation and difficult to modify. The typical debugging solution is typically to instantiate cores within the design and record signal activity for later inspection. Changes to the perspective are treated like any change to the design and often require the same time-consuming implementations. In contrast, software development and debug environments allow complete visibility and control of the memory and execution space.

The frequent comparisons in productivity studies to software development environments are no coincidence. FPGA development environments and flows have been harshly judged against their software counterparts [1,2]. Software development environments are frequently

cited as the gold standard for hardware development because of the speed and efficiency that they bring. This research aims to adopt several of these facilities for FPGAs, all based around the concepts of *controllability*, *agility*, and *visibility* of a design loaded and running on hardware.

1.2 Problem Statement

Typically, a hardware design is first modeled using a conventional high-level programming language, such as C/C++, Java or even a formal modeling framework such as Matlab. The development then proceeds by reconstructing a more concise design, this time specifically oriented towards the actual hardware. This is achieved by filling in implementation details unique to the platform such as specialized resources and device constraints. The high-level reference model is algorithmic in nature, rather than cycle-accurate and contains no references to timing, which is elemental to hardware implementation. The transition from reference model to hardware implementation creates both a logical and physical gap in the designers understanding of the design space and the continuity of the design from model to reality. There are no failsafe automated solutions to converting the reference model to RTL, rather this is a manual process and the effort in validating the model is then repeated anew for the hardware design. The process of validating the original, vetted reference to hardware is a manual, unreliable, and error-prone approach. These separate flow processes are shown in Figure 1.1. It is noteworthy that not only is each phase of the development cycle almost entirely independent of the other, but also that no validation or debug techniques are applicable across any of the stages.

Furthermore, once implemented onto the FPGA, little meaningful and interactive interaction with the design is possible. There exists little to no abstraction between the physical

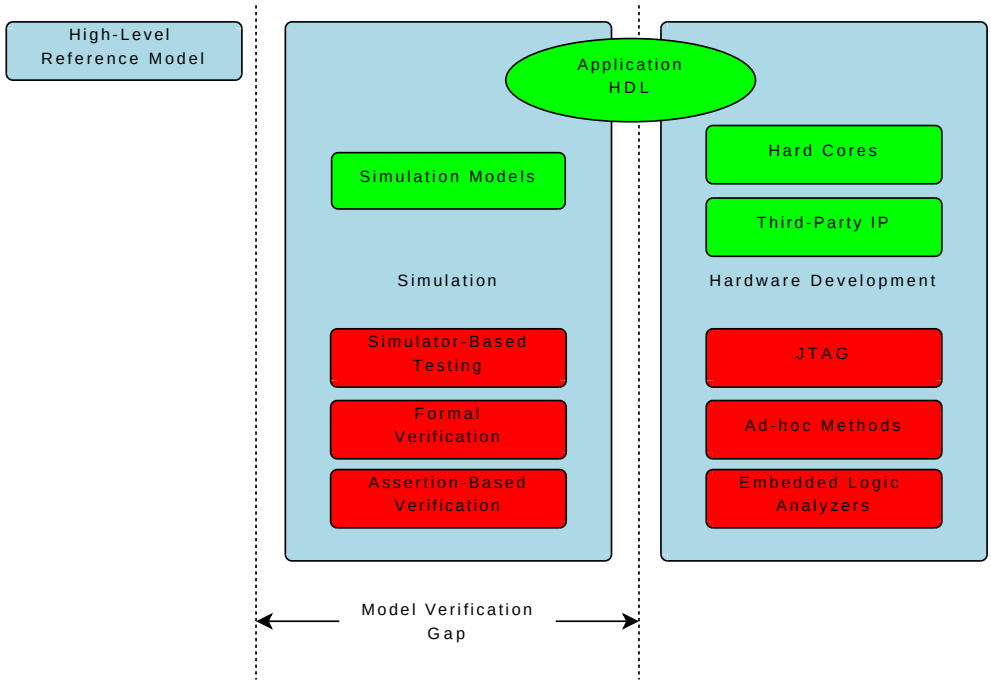

Figure 1.1: Hardware development flow.

implementation and the debugger's perspective of the design. Much like the development language, debugging is mostly a low-level task. Lengthy, error-prone implementations hamper FPGA productivity, as does limited visibility into the designs. Other than the primary inputs and outputs, the design is effectively sealed off from inspection. A cohesive and interactive control mechanism of a design is needed that allows both rapid and unlimited access to all aspects of the design.

1.3 Contributions

The contributions of this work are as follows:

- Development of a tool that addresses the *model verification gap* by raising the abstraction level for FPGA validation and debug.

Debugging is traditionally a frustrating process since developers are responsible for closing the loop when validating synthesized hardware against earlier reference models. The customary view of hardware validation is at the RTL level, which when validating high-level functionality, is much too primitive. The developed system allows synthesized hardware and the implementation to be concurrently executed and compared at the target design speed on the same platform.

- Development of a tool that improves visibility, control, and agility for FPGA debug by introducing software development environment facilities.

  Unlike software development, FPGAs have few means to inspect or debug a running design. The internal state elements of the design have limited or no visibility, and execution occurs at millions of clock cycles per second. An error may occur and by the time it manifests itself, the state of the device may have changed so much that its diagnosis is impossible. The developed framework brings some of the useful facilities found in software development environments to FPGAs. The system presented here allows a running, implemented design to be controlled and inspected at its target speed with the interface and agility of software development tools. While most FPGA development tools place the debugger's viewpoint outside the design, the developed framework places it within the FPGA. The presented work moves the designer's perspective inside the FPGA where its execution can be manipulated and its state explored from within, treating the FPGA itself as a random access memory.

1.4 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 Background

This chapter provides background material on both hardware and software development

to frame the perspective of the challenges faced in developing for FPGAs. Development and debug life cycles for FPGAs are explored, beginning from modeling and ending with simulation and hardware-based verification and validation. Commercial solutions, including the high-level language development techniques meant to mirror software development flows are also reviewed.

Chapter 3 Related Work

A survey of current debug-related work, both commercial and academic, is provided in this chapter. FPGAs are proprietary and the means of gaining insight into a design are limited and generally restricted to two methods: signal capture and JTAG. High-level synthesis is also discussed in this section since raising the abstraction of design entry to an algorithmic level also alleviates the need to do low-level analysis. Finally, a review of shortcomings of many of these methods looks to frame the presented research's contributions.

Chapter 4 Improving Abstraction and Turnaround Time

The model verification gap is analyzed to define the High-Level Validation framework. This framework aims to map untimed software models to implemented hardware designs by leveraging the wealth of information that can be extracted during hardware development through simulation. Rather than relying on simulation to validate, the proposed framework aims to move validation to hardware and tie its execution directly to the software model. Visibility, agility, and control for hardware-level debugging are then addressed by proposing a software-style debugging environment for FPGAs. A method for implementing breakpoints as well as rapidly enabling, disabling, and altering them is also presented through the use of an on-board microprocessor.

Chapter 5 Implementation

In this chapter, the two debug and validation frameworks are implemented and benchmarked. First, the High-Level Validation framework is developed to non-invasively instrument and

map a C-language reference model of an encryption core. The framework requires the development of a custom hardware peripheral to interface to the module to be tested. The system is shown to be capable of testing a hardware design executing at its intended target design speed. Finally, the Low-Level Debug framework is constructed, leveraging the Xilinx Runtime Reconfiguration flow. With this framework, debug scenarios can be modified and re-implemented into the hardware within minutes by isolating the debug breakpoint and top-level modules each into their own separate reconfigurable region. Capabilities such as those found in software debugging environments, including breakpoints, stepping capabilities, and full immediate access to all design registers are demonstrated. The Low-Level Debug framework is demonstrated on three large designs where implementation time is prohibitively long.

Chapter 2

Background

In this chapter, an overview of programmable logic and how it evolved into modern day Field Programmable Gate Arrays (FPGA) is presented, followed by a step-by-step discussion of the stages of development. The various methods of FPGA debugging and some of the challenges of each method are then presented. Next, high-level synthesis, an alternative to traditional hardware development languages that seeks to retarget conventional software programming languages for FPGAs, is presented. Finally, runtime and partial reconfiguration is explained.

2.1 An Overview of Programmable Logic

In the dawn of digital development, designers were required to physically build systems in order to test and validate them. Each wire of a design was individually connected to discrete logic chips, transistors, and other components. Often to improve visibility, colorful wires were used, each wrapped around small posts, lending this style of development its descriptive name *wire wraps*. The resulting rat's nest of wiring revealed the fragility and complexity of even a modestly-sized design. Frequently, finished products were just neater

implementations of these prototypes, now committed to a manufactured component board with embedded traces. It would be some time before electronic design automation (EDA) tools established a more rapid and reliable means of prototyping designs while simultaneously introducing a new set of challenges.

FPGAs derive their name from their ability to be repeatedly programmed *in the field* or wherever they are installed, without the need to physically replace the device. Today, FPGAs are even more flexible and support the means of remotely re-programming the devices, further reducing the cost of ownership and maintenance. The *gate array* portion of the name refers to a historical and now purely logical organization of the array of logic gates that could be programmed to implement an arbitrary logic function, as shown in Figure 2.1. The gate arrays have since been replaced by a more efficient and compact mechanism. Modern FPGAs are still organized as arrays as seen in Figure 2.2, however specialized resources such as arithmetic blocks, memories, and processors are now interspersed among a vast sea of programmable logic elements.

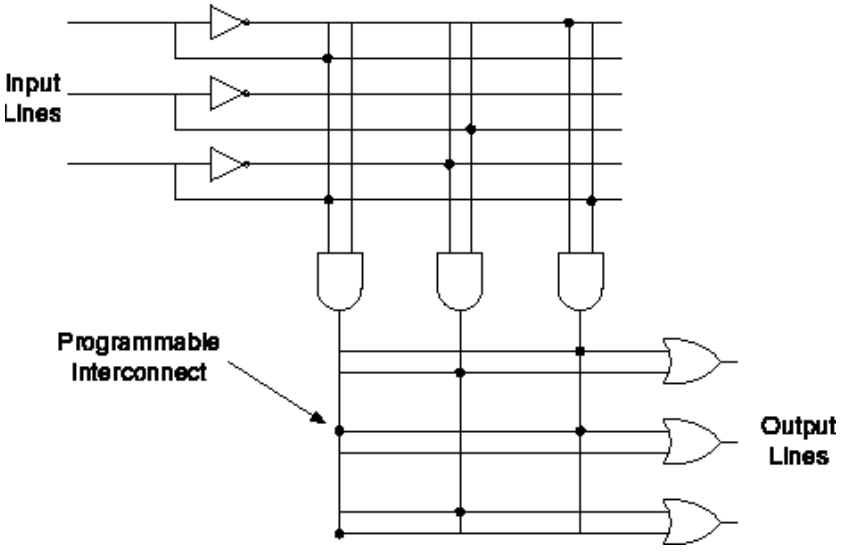

Figure 2.1: Early programmable logic devices with discrete gates connected by programmable interconnects.

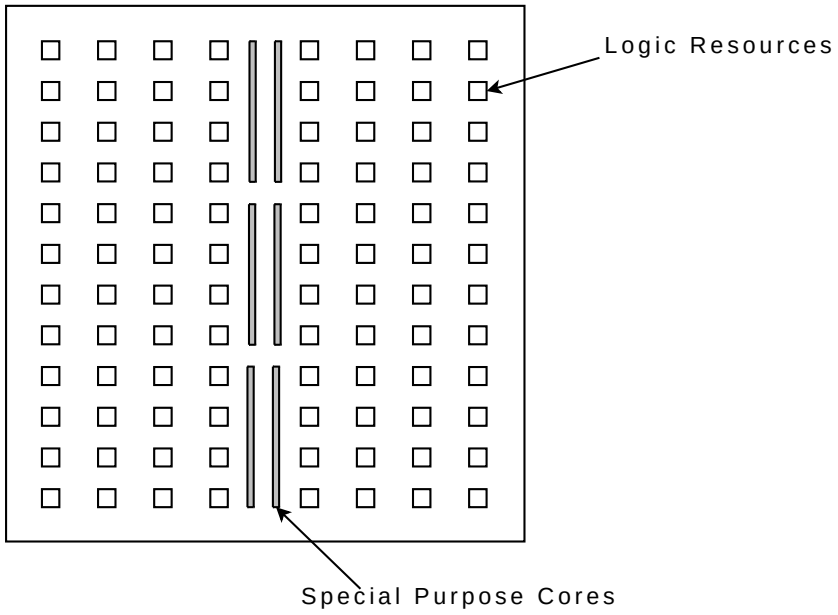

Figure 2.2: Modern FPGAs organized as a regular array of resources.

As Integrated Circuit (IC) technology advanced, multiple discrete components were combined into chips defining fundamental building blocks of digital systems, thereby allowing more complexity to be embedded into a smaller area. The need for a practical, repeatable, and efficient means of developing and prototyping complex designs became evident. Early FPGAs made their debut as precisely such an integrating technology allowing the interconnections and necessary logic (also referred to as *glue logic*) between these chips to be reconfigurable through software programming rather than physical wires. Manufacturers could now quickly deliver a product, placing the components on a board, all connected to an FPGA. The necessary connections and required logic could be experimented with and repeatedly revised, deferring the final design to even after units were installed. This also allowed revisions to be issued to previously sold and shipped products, which could then be updated on-site. Vendors who might not otherwise have had the financial means or number of forecasted products to justify an ASIC could still produce a viable and compelling

product to compete in what was previously an exclusive market open only to the wealthiest companies.

Rather than be relegated to the ranks of a prototyping technology, FPGAs began to mature into their own distinct product line and became the focal point in the realm of high-performance industries once reserved for ASICs. Input/Output (I/O) speeds of FPGAs became competitively fast for real-world applications and the logic density—the equivalent number of physical logic gates—skyrocketed when innovative means of implementing and packing logic functionality into smaller areas was developed. One such innovation was the Lookup Table (LUT) [3] that eliminated the need to physically implement an array of logic gates and the means to programmatically connect them to realize an arbitrary logic function. Instead, the inputs of a logic function are replaced with address lines to a programmable read-only memory that returns the expected result of the original logic function when the corresponding lines are activated. As a result, the functionality of a LUT resembles that of the logic function's truth table. A conceptualization of this is shown in Figure 2.3. In reality, the actual implementation of a LUT is less intuitive but more compact. The vendor's EDA tools determine which signals to group as address lines, and compute the resulting output to be stored in the LUT's memory. This is far more space- and speed-efficient than trying to map a logic function to a statically built array of gates and determining the connections necessary in order to realize that function.

Advancing EDA tools and increasingly more powerful workstations advanced the state-of-the-art, and as a result design processes became more productive. The advent of Hardware Description Languages (HDL) made the representation, modeling, simulation, and validation of hardware systems even more efficient. Whereas previously a digital system could not be validated without physically building and prototyping it with discrete components, validation could now be done within a simulation, quickly finding and fixing logic errors before building the design.

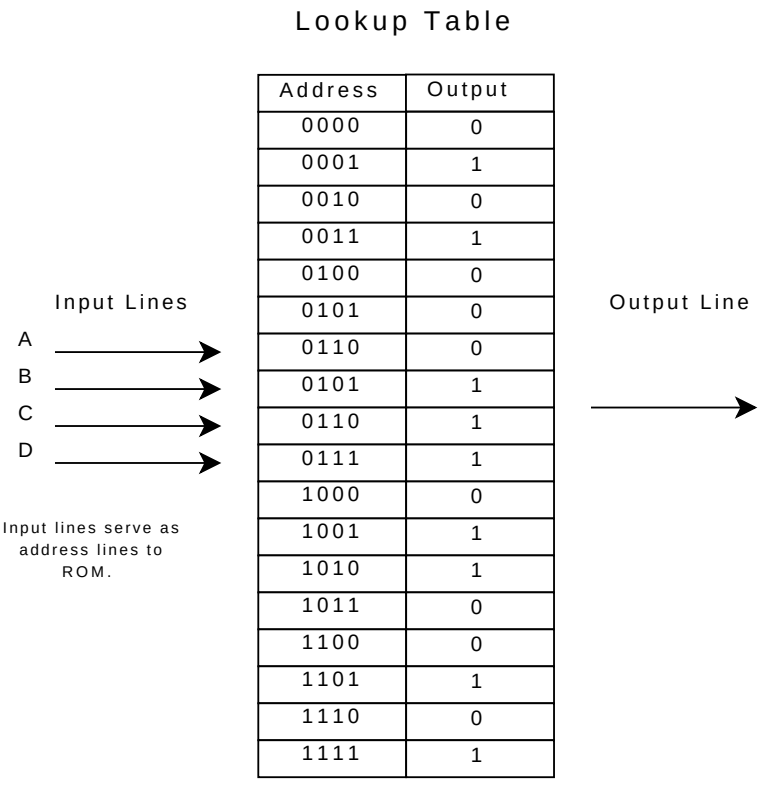

Figure 2.3: LUTs replace arbitrary logic with simple memories.

Figure 2.1, shows a simplified representation of one early implementation of a Programmable Array Logic (PAL) device, and the predecessor to modern FPGAs. The logic gates in this case—AND, OR, and inverter gates—are placed in a regular grid pattern and connected by programmable interconnects. The device is programmed by selectively enabling the programmable interconnects where crossing wires intersect, which when set create a route between a source and one or more sinks. Innovations such as LUTs obsoleted this architecture allowing more logic to be packed into an equivalent area. At the same time, smaller manufacturing processes also contributed to the increasing density and speed. A timeline of the densities of LUT-based architectures is shown in Table 2.1.

The data in Table 2.1 was compiled from Xilinx data sheets [4–12] of major FPGA releases between 1998 and 2010. For each release, the largest part was selected. When two companion families existed (typically manufacturers will build a more modest companion part that is

Table 2.1: Timeline of logic density of Xilinx FPGAs.

ReleaseYear	**Device**	**Number of Logic Cells/LUTs**
1997	XC5215	1936
1999	XC40250XV	20,102
2000	Virtex-II XC2V8000	93,184
2002	Virtex-II-Pro XC2VP100	99,216
2004	Virtex-4 XC4VLX200	200,448
2006	Virtex-5 XC5VLX330	207,360
2009	Virtex-6 XC6VLX760	758,784
2010	Virtex-7 XC7V2000T	1.955 M

smaller in size, consumes less power, and subsequently less expensive), the larger family was selected. Logic cells were chosen as the comparison metric because it appeared across all devices, albeit under different names. One logic cell equates to one LUT (the word LUT does not appear until the release of the 4000-series in 1999). Logic cells were also chosen because this is the smallest, most primitive user-programmable structure in the device and from where FPGAs derive their core functionality. The number of inputs to the LUT also increased during this time period, further compressing the amount of logic that could be packed into a single logic cell. Logic capabilities increased rapidly yet as shown in Chapter 3, methods of working with the devices has remained relatively unchanged.

It should be noted that there is no one clear metric by which to truly measure FPGA density. Vendors frequently change terminology, restructure the architectures, and emphasize different parameters in the product literature. While earlier literature focused on the number of "equivalent logic gates"—the equivalent number of primitive Boolean logic gates that the device could implement—this terminology quickly fell out of favor. Later releases would also cite of the "maximum number of gates"—another ambiguous and short-lived parameter—which also similarly increased over the same period. The effectiveness of attempting to measure logic density is further confused when *hard macros* or hard cores are

considered. Macros are discrete components physically built into the FPGA, typically complex or frequently utilized cores that are useful across a large number of designs. Requiring designers to implement their own cores promotes poor design practices, reduces re-use, forces high levels of design specialization for a particular architecture which makes it difficult to port to future releases while at the same time wasting valuable real estate in the reconfigurable fabric. Common hard macros include memories, signal processing cores, multipliers, communications cores such as Ethernet MACs, and even microprocessors (dual PowerPCs were available on the Xilinx Virtex II-Pro). The impact of including such cores is dramatic when the number of resources freed and the potential for increased performance and power efficiency are considered.

FPGAs are not only expensive, but difficult to program as well. Unlike conventional computers, programmable logic design has not had its renaissance. Whereas higher level programming languages such as C or FORTRAN eventually obsoleted the use of assembly-level languages ushering in an era of innovation, progress, and accessibility [13], the complexity of FPGAs and their chief purpose—to accelerate computation well beyond the capabilities of conventional architectures—make these obstacles unlikely to be resolved for some time. FPGAs are programmed using low-level, primitive languages which describe the transfer and transformation of signals between registers. Higher levels of abstraction are difficult to obtain.

Despite the many drawbacks, FPGAs are popular in the communications, industrial, and military sectors. FPGAs are the primary processing power for Internet router giant Cisco's flagship products. FPGAs are frequently deployed in aircraft, satellites, and other vehicles where lots of processing power is needed but space and power are limited. FPGAs are particularly attractive for satellites since the application can be remotely changed, allowing the satellite to be dynamically retasked while in orbit to changing mission requirements.

Due to the cost and difficulty in development, FPGAs are not typically found in consumer electronics, however vendors are now producing a family targeted specifically towards the automotive industry's information and entertainment ("infotainment") consoles found in high-end vehicles. German automakers BMW and Mercedes have outfitted their high-end sedans with FPGA-powered video processing equipment [14] such as night-time, low-light, or rear-facing assistive technologies; other auto manufacturers are following suit by replacing certain on-board systems with FPGAs.

FPGAs currently operate at a comparatively lethargic clock rate, yet can outperform conventional systems running at an order of magnitude higher clock speed. In the last decade, operating frequencies have risen from sub-100 MHz to around 500 MHz clock rates in the premium speed-grades. In comparison, commodity processors such as those from Intel and AMD achieved the 1 GHz milestone in 2000 [15]. Yet, FPGAs remain as one of the few viable means of accelerating streaming applications, and performance may exceed that of even high-performance commodity processors. The explanation for this discrepancy is in the flexibility of the FPGA's unique architecture.

FPGAs do not carry the customary operational overhead of conventional computers; they are not general-purpose processors that have a single or limited thread of execution and a fixed instruction set. Yet neither are FPGAs natively multi-process capable as are most modern microprocessors. Multi-processing is a clever slight-of-hand that gives the appearance of simultaneously processing several unrelated tasks, when in fact the processor is only physically capable of servicing one process at a time. Multi-processing capability allows the processor to quickly and transparently change processes either after a predetermined time interval or when a higher priority task has preempted the current task. The general-purpose, fixed-instruction set architecture and multi-process capabilities of microprocessors make them difficult if not impossible to target to specialized, high-performance applications,

particularly those that process continuous streams of data. In addition, operating systems (OS), specifically user-friendly consumer OSes, are busy servicing device driver requests, controlling peripherals such as network interfaces or hard drives, and managing memory allocation to improve user experience.

Dramatic performance gains achieved with FPGAs are possible through the flexible, reconfigurable architecture. Unlike conventional processors and associated software tools that serialize a set of predefined instructions to perform a task, FPGAs have no such restrictions. Instead, FPGA developers can custom design an accelerator to include as many specialized pipelines or processors as desired. FPGA applications are referred to as a *custom processors*, *hardware accelerators*, *co-processors*, or any number of similar names. With a custom design such as this, performance can dramatically improve. The inclusion of high-performance, specialized cores has increased the capabilities of FPGAs to enter many more industry sectors rather than simply serving as glue logic.

Despite the great potential of FPGAs, the development process is not as streamlined as it is for software development. In contrast, the growing prominence of software engineering has heightened structured software architecture and reuse, while the same is rarely true or even possible with hardware development. The primary culprit is the uniqueness and variations of subsequent architectures and how tightly-coupled a design becomes to the target device, and as explained next, the complex development process.

2.2 FPGA Development

FPGA development typically begins with a functional, high-level language (HLL) model that is used throughout the design cycle as a reference for both simulation and hardware development. HLLs provide access to elegant data structures, complex control constructs,

and resources such as file systems and debuggers that enable rapid algorithmic development and extensive testing, yet are not burdened with timing and synchronization details.

Once the high-level model has been validated against the original specification, HDLs such as VHDL or Verilog are used to develop cycle-accurate simulation models targeting a specific hardware platform. HDLs are low-level languages used to describe the transfer of signals between registers and characterize the concurrent nature of a hardware design. Simulation models specify the register-level transfer and processing of data, bus-widths and control signals that were not defined in the functional reference model. Most device families contain specialized, high-performance cores, such as Digital Signal Processing (DSP) cores and memories that are common in designs. These can be instantiated into a user design like any other component, however their interfaces, behavior, or configuration are frequently specialized for the specific architecture and are subject to change in subsequent architectures.

With some effort, simulation models may be validated against the reference design, but this is often of limited usefulness. Simulation models for Intellectual Property (IP) and hardware modules are often themselves non-synthesizable behavioral HDL models optimized for simulation or obfuscated to protect the IP. These may inadvertently differ from the actual implementations. As a result, the model being simulated will not accurately reflect the synthesized hardware version. Additionally, simulators silently accept legal but physically impossible coding practices for FPGAs. Optimizing synthesizers may misinterpret a developer's intent and alter the design producing unexpected results, whereas synthesis pragmas are intended to produce alternate versions in synthesis [16]. There are a handful of solutions available to debug the resulting physical design, none of which offer the rapid turnaround and full visibility found in software development.

On the other hand, software developers have created generations of tools to rapidly validate and debug their applications. Build tools such as `Make` [17] analyze source code dependencies

and selectively rebuild only the necessary dependencies. Linkers and library archives preserve object independence until the final static linkage or, as in the case of dynamic linkage, until runtime. Small revisions can quickly be made to an application and tested directly on the target platform, usually in a matter of minutes for even complex applications. Debuggers and profilers offer extensive visibility into an executing application and allow complex breakpoints to be specified and altered without recompiling. Figure 2.4 shows the software compilation flow. Software compilation units remain as independent intermediate objects even after being compiled and integrated into a library archive until linkage or runtime, and as such are easily revised. No such development model is readily available for FPGAs.

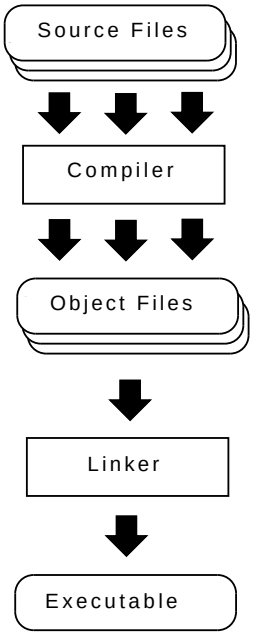

Figure 2.4: Overview of software development flow.

This flow is in stark contrast to the traditional FPGA flow. In software compilation, a relatively short time is required and high level of parallelism is possible in producing intermediate files. The intermediate object files remain independent even within their destination library archive, and are easily individually replaced. However, in FPGA design flows the separate components of the design are combined into a monolithic format during the early stages of

compilation, typically following synthesis. A basic depiction of the FPGA flow is shown in Figure 2.5.

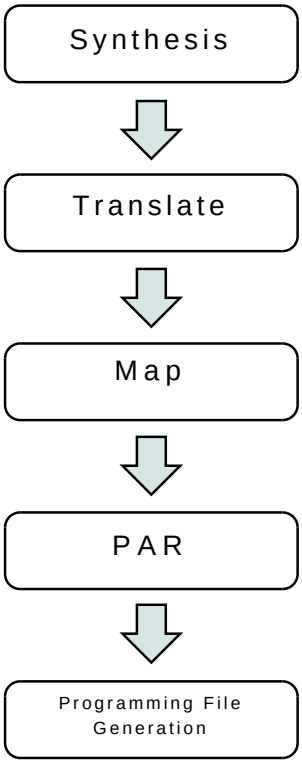

Figure 2.5: Simplified view of FPGA development flow.

Unlike software, FPGA development flow is serial. While individual modules may be synthesized in parallel, the rest of the flow aggregates all the individual components into a singular monolith and operates on the entire design globally. Unlike software, many operations can have a global impact on the rest of the design. For instance, logic optimizations or reductions can have a ripple-effect that emanate beyond module boundaries and throughout the entire design. Place-and-Route must find an optimum solution that locates components where they can be successfully connected using the FPGA's complex routing architectures. This is a challenging problem in that there may be a limited number of routes with acceptable latencies. At that stage, large designs are fragile and changes to accommodate routing or

timing can be disruptive causing a series of changes. A thorough discussion of each of the stages is given in Section 2.2.1.

Vendors are aware of the time burden associated with FPGA development and have attempted to address these issues. Modular and incremental flows for FPGAs, ideas borrowed directly from ASIC development, attempt to compartmentalize designs and avoid lengthy re-implementations [18, 19]. However these flows require that floorplans and interfaces be stable. This is a difficult prerequisite to meet especially in the early stages of development where the design may be in a constant state of flux. Even with modular and incremental flows, a seemingly small change may still require a full re-implementation. The PATIS project [20] discussed in Section 4.3.1 addresses this problem.

2.2.1 Design Flow Overview

The FPGA development flow is an elaborate multi-stage process. Vendors attempt to mask the complexity by providing applications that automatically generate scripts or interfaces for specialized cores, or graphical, user-friendly applications to control and monitor the entire build process. Yet, there remains a great deal of device- and design-specific information that must be known and applied at each step. Originally, the tool flow was developed to be a definitively layered sequence, with each stage performing a distinct task while deferring the low-level, device-specific implementations for later stages. However for optimization purposes, device-specific information is now included at every stage and the layers have lost their well-defined boundaries. Typically, an application will provide suggestions or attempt a preliminary effort for subsequent stages enabling iterative executions between the two without user intervention. Proprietary file formats also have evolved to include much of the meta-information produced in earlier stages, reducing the number and complexity of

command-line switches that needed to be passed along to different stages. These are subtle, yet welcome improvements.

FPGA flow consists of five steps: *synthesis*, *translation*, *mapping*, *place-and-route*, and finally *programming file generation*. Vendors have not adopted a universal terminology, but the purpose of each stage is consistent across different vendor's flows. A general diagram outlining the flow between stages is shown in Figure 2.6 with an overview of each of the stages following.

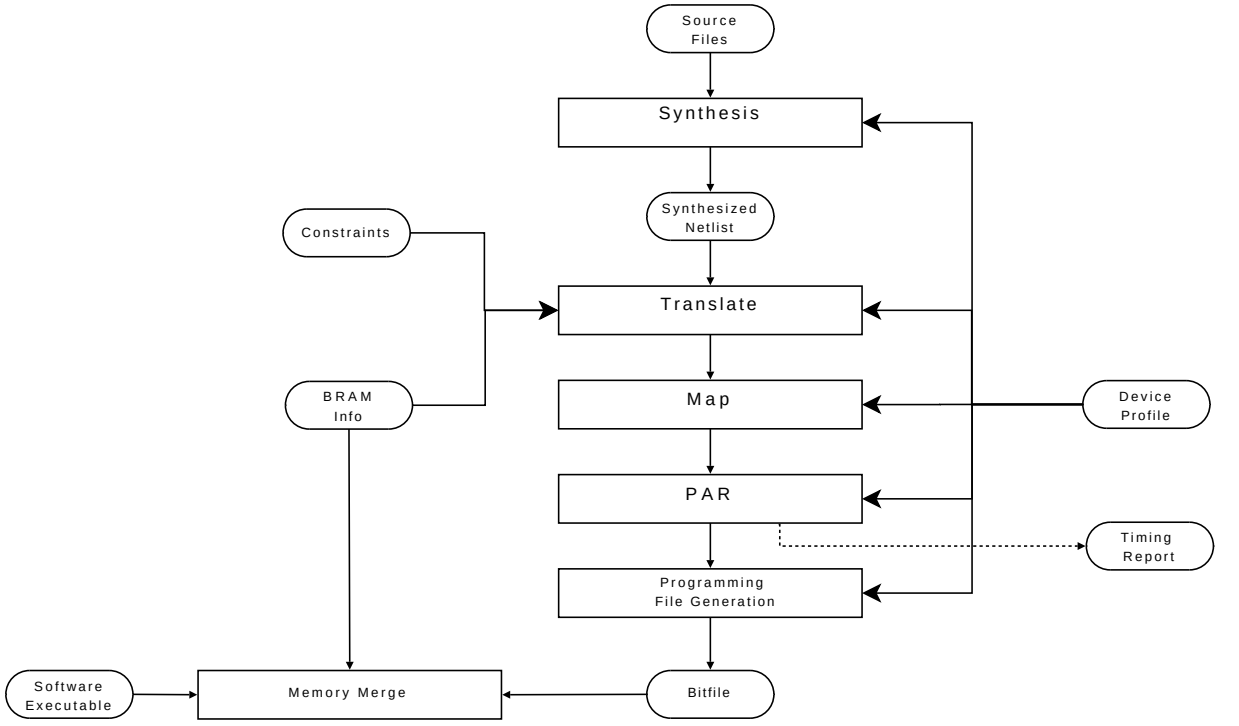

Figure 2.6: Detailed overview of FPGA development flow.

Synthesis

During synthesis, the original HDL is converted to a *structural netlist*. The structural netlist, also referred to as simply a *netlist*, consists of elements in the design and their connections. HDL can be written as a behavioral specification, which raises the layer of abstraction

to higher-level constructs (including control flow, clock events, and some basic arithmetic operations) or as structural HDL which defines a more primitive, precise but less intuitive definition of the design. Netlists are purely structural in makeup, defining the elements and their interconnects of the design without any high-level elements or language representations. Rather, only *primitives*, the most basic elements of digital design such as Boolean gates, or physical structures found on the FPGA are found in netlists. The synthesizer is one of the few applications for FPGA development that can be written by independent third parties. However, users are rewarded for using the vendor's synthesis tool since these are typically tightly integrated with the rest of the tool flow.

In the past, synthesis was device-independent. However the increasing number of high-performance and device-specific cores found on FPGAs which can be explicitly instantiated into the design forces the synthesizer to possess some knowledge of the device. Also, being the first stage of the development process, the synthesizer is in the unique position to analyze the original source including pragmas written by the original author and perform preliminary mapping efforts based on design patterns found in the original HDL. Synthesizers rely heavily on hard-coded HDL design patterns that it uses to recognize and build optimized code or instantiate hard macros for the target architecture. The process of identifying and producing specific design elements is referred to as *inference*. A core is *inferred* when the synthesizer recognizes the predefined HDL pattern, such as a memory, state machine, shifter, or arithmetic core and then instantiates an optimized interface for that pattern. If the synthesizer is not able to infer the pattern correctly such as from a non-standard implementation, the component will not be built in the most optimized manner and placed into one of the FPGA's physical cores, or may not operate as expected. For example, memory inference fails when specified in capacities other than powers-of-two. The list of inferences and their corresponding HDL patterns are listed in the synthesizer's documentation. An example of one

```
module v_shift_registers_1 (CLK, SI, SO);
  input CLK,SI;
  output SO;
  reg [7:0] tmp;

  always @(posedge CLK)
  begin
    tmp <= tmp << 1;
    tmp[0] <= SI;
  end

  assign SO = tmp[7];
endmodule
```

Listing 2.1: Verilog shift register definition from XST synthesizer documentation.

such pattern for a Verilog shift register, taken directly from the documentation of Xilinx's XST synthesizer [21], is shown in Listing 2.1.

There are several advantages to inference. First, common components of a design can be created in a structured, repeatable manner without the need to explicitly instantiate device-specific components. This promotes some level of platform-independence as well as reusability. Second, the user or the tools can allocate inferred instances across available resources for the device or if necessary implement them in logic if there are not a sufficient number of cores available. For optimization purposes, the synthesizer needs to know architectural details of the target device to determine if there are a sufficient number of cores available, either through instantiation or inference. However, the synthesizer does not definitively allocate or assign resources. The final assignment is done later once the entire design's resource requirements have been resolved. The design can also be entered schematically and processed by the synthesizer, however in practice this is not as common as HDL. The synthesizer produces a structural netlist, either in the industry-compliant Electronic Data Interchange Format (EDIF) or the vendor's proprietary binary format. The advantage of

using the vendor-specific format is that additional information may be passed within the files to subsequent tools of the flow, reducing the burden on the developer. Synthesis occurs relatively quickly (on an order of seconds at best) depending on the size and complexity of the module. Modules may also be separately synthesized, providing some separation from the rest of the build process.

Translation

Once the synthesizer has produced a netlist, the design must then be *translated*. During translation, also referred to as *building*, netlist components produced during synthesis are further reduced to primitive logic gates. Device-specific cores instantiated in the original HDL are also not processed during this phase. This is the last stage of the flow where non-proprietary or industry-wide input files are accepted. The rest of the flow occurs entirely within proprietary file formats and must be handled using the vendor's tools. Along with the device information, user constraints are also provided to the build tool. *User constraints* are specifications such as physical pin assignments, timing constraints, and memory parameters for the proper operation of the design that the tools cannot infer on their own. Pin assignments map the device's physical external pins to the design's top-level module input and output ports. Timing constraints specify timing and signal-related parameters, typically for interfaces with strict timing requirements such as Ethernet MACs, high-speed I/O interfaces, or the design's clocks. Timing is not significant for translation and is principally used once the design's connections have been routed (during place-and-route) to ensure that the design will meet timing. Since the tools are proprietary, it is not definitively known to what extent timing is addressed at this stage, but it is most likely that they are integrated into the design databases, analyzed for preliminary design rule enforcement, and referenced for mapping optimizations.

Mapping

During the next phase, the mapper will logically map components produced during translation onto the target device. The mapper does not physically map elements to a specific core, but logically identifies candidate target core types. It is during this phase that the abstraction benefits of inference are realized. The mapper can now definitively determine which, if any instantiations must be implemented as logic, rather than the preferred core assignment strategy. For example, a design with more RAMs than the available physical memory cores will require some of those RAMs to be implemented as distributed RAMs. The *distributed* specifier indicates that the component will built using the FPGA's logic blocks rather than be instantiated to a physical, special-purpose core. Decisions during this and later stages have the potential to create a ripple-effect of complications. If an instance must be implemented as distributed, more logic is now required than originally assumed. Tightly interconnected cores such as memories require that all the corresponding logic be placed in close proximity to one another in order to meet timing requirements. As the design components of the design have no physical locations assigned at this time, the mapper has no way of determining if the final design will fit on the part, if there are sufficient routing resources available, or if the final routing will meet timing. These issues are addressed during the next Place-and-Route phase. Mapping handles optimizations such as *constant folding* where a constant signal makes logic in its path unnecessary. For instance, a constant zero signal feeding an AND gate will make the gate unnecessary as its output will in turn always be a constant zero. This gate can be removed without affecting the operation of the design. Optimizations such as these may free up resources, making it possible to pack even more logic into a smaller area which in turn may make routing and timing closure easier. Timing closure, which is addressed during Place-and-Route, is achieved if components are both placed and connected ("routed") in such a way that the signal's latency does not violate the

timing constraints. For example, a signal whose latency along a path exceeds that of of the clock period violates timing or *does not meet timing*.

Place-and-Route

In Place-and-Route (PAR), the components that were previously logically mapped onto the FPGA's resources are now physically placed and the wires routed between them. Conventionally, the Map and PAR stages were distinct, however current tool implementations will perform a preliminary PAR during the map stage and reiterate mapping if required. Without this optimization, mapping would need to be rerun manually if PAR fails. Placement and more specifically routing are very computationally and memory intense. At this stage, precise models of both the FPGA and the design must be built and stored in memory. The algorithms used during PAR rely heavily on traversing these structures, and as such require many non-adjacent memory accesses which quickly overrun caches. PAR is the most time-consuming stage of the build process, and is the stage most likely to fail for complex designs with high device utilization.

Meeting timing requirements are also the responsibility of PAR. A poorly placed design stresses the router which must find an efficient route—out of a vast number of possibilities—to meet timing. Further complicating PAR is that routing is not a simple distance-dependent metric. That is, placing two components physically close to one another does not guarantee that a low-latency route exists between the two. FPGAs have complex, segment-based routing architectures where wires are categorized as local or long-distance, spanning one or multiple segments before reaching a programmable interconnect where it is patched to another segment. The individual segments, along with the programmable interconnects which add latency, do not have simple timing models. Furthermore, routes may be congested resulting in a circuitous path to connect two components. The best placement of a component

for timing purposes may be non-intuitive, such as physically distant location where there are fewer, but physically longer routing segments to the destination. Counter-intuitively, this can improve timing. As such, PAR is a lengthy and iterative process. For large designs utilizing a high percentage of the FPGA, PAR times exceeding a day are not uncommon. Timing report generation is a separate procedure from PAR and is useful to determine details of which parts of the design are causing timing to fail. Often, a different design technique such as pipelining an operation alleviate timing failures. A successfully placed and routed design is ready to be committed to a programming file and loaded onto an FPGA.

Programming File Generation

Finally, the actual file that will be loaded into the FPGA is produced during programming file generation. The *bitfile*, *programming file*, *configuration file*, or *bitstream* is produced by interpreting the final placed and routed design into a binary file comprised of a stream of bits which configure the FPGA's resources. It is common in this stage for the generation application to perform a final design check ensuring there are no conflicting programming options, such as a conflict that might physically damage the device. A header is prepended to the programming file that contains special instructions for the loader and may activate special features on the FPGA, such as security. The FPGA is ready to be programmed with a special programming cable and loader application which either directly programs the FPGA or commits the design to an on-board FLASH memory or other external storage.

If the design contains RAM blocks, the contents can now be initialized. If RAMs were specified early in the design and build process, a logical mapping of design elements to memory elements would have been produced. This mapping is initially used during the translate phase, but is again referenced to prepopulate the RAMs before configuration so that memories are not empty at power up. This is particularly relevant if the design contains a

microprocessor. Without this capability, processors would need temporary boot up sequences to hold them in a legitimate state until the RAMs are populated or require firmware to load programs from another source, such as FLASH or external disks. ROMs on the other hand, are typically initialized during synthesis by specifying their contents either in the HDL or as an external file that the synthesizer processes and incorporates into the netlist.

2.3 Debug Methodologies

For FPGA design debugging, there are a limited number of strategies. FPGAs are proprietary devices, which limits third party products in this area. Unlike conventional, commodity computers, FPGAs have no standardized architecture or discernible platform like an operating system or even basic input/output by which to interact. Debugging generally requires intimate knowledge of the device and the design. Improvised solutions are common, such as activating diagnostic LEDs when certain events occur. This requires an intimate knowledge of the design and is a lengthy process of progressively interpolating to the cause and re-implementing the design each time. External logic analyzers can be effective, but require signals of interest to be routed to external physical pins. This can interfere with design pin assignments and affect timing closure. If the signal is within several layers of modules, each enclosing module will need its interface modified in order to route the signal to the external pin. Once finished, the interfaces must be reverted and the design re-implemented anew. Lengthy implementation times, especially during place-and-route make these ineffective strategies.

In the creation of large or complex designs, a high-level language reference model is often first created to develop and validate the algorithm. This reference model becomes isolated from the remaining development cycles due to language and technological incompatibilities.

Subsequent development in HDLs re-implements the algorithm anew, focusing on data flow, extracting parallelism, timing, and control signal interaction—aspects not captured in the reference model. After simulation, the HDL design may be adapted for synthesis. Though the same HDL may be used for both simulation and synthesis, this is not always the case. Third-party IP may be implemented differently for simulation and synthesis for faster simulation and optimized for the target platform. Simulation models may be obfuscated or encrypted to protect intellectual property. Unintentional variations between the models may exist or an optimizing synthesizer may misinterpret designer intentions, providing unreliable or unpredictable results during development [16]. The resulting gap between HLL models and synthesizable hardware is the *model verification gap*, illustrated in Figure 2.7.

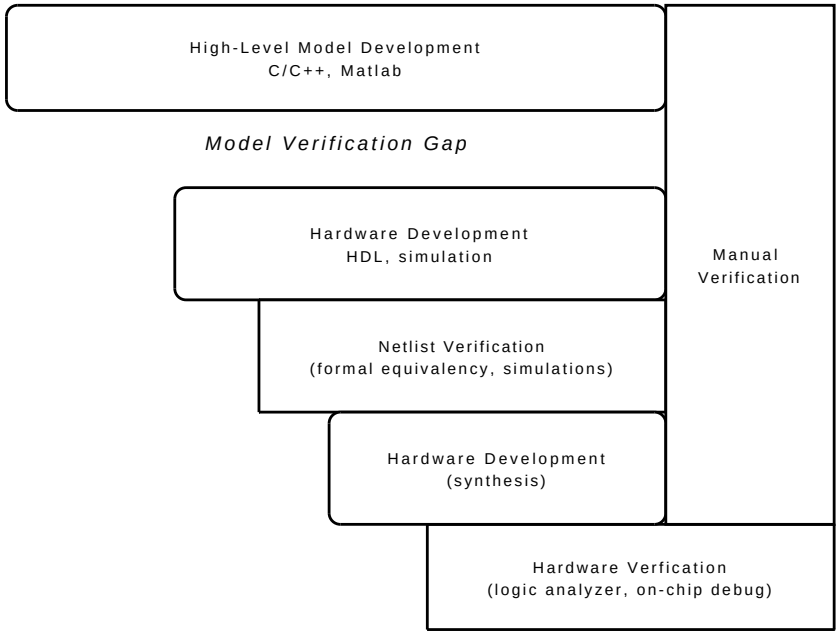

Figure 2.7: The *model verification gap* in hardware development.

The model verification gap is a significant shortcoming in hardware development. The design validation team must develop ad-hoc solutions to bridging it, or bridge it manually. Either way, the effectiveness and productivity of validating against the original reference model is limited and largely targeted towards the simulation model. No tools are known to exist

that allow the original HLL reference model to be directly linked to the synthesized and implemented hardware.

Designers have long relied on many of the same debugging methods, many of which are complex and difficult to adapt to different scenarios. As designs have increased in size and complexity, tools have lagged in the ability to adequately handle these demands. The size of FPGAs has increased so dramatically in recent years; 32-bit processor based workstations are now incapable of targeting these larger parts while 64-bit processors are now the norm [22]. Visibility into a design programmed onto an FPGA is limited, and some types of design errors such as timing errors, can be elusive across design turns. Even worse, the synthesized version of a design can differ from what was prototyped in simulation. Unlike software designers who can frequently develop and test on their target environment, hardware developers do not have the same visibility.

Third-party IP, which addresses the growing interest in modular design and reuse long practiced in software development, also presents its own challenges. A vendor may develop several models of an IP core: one optimized for fast simulation using high-level, non-synthesizable behavioral constructs and another device-optimized version for synthesis. Often these models differ. Pragmas (suggestions that the designer can provide to the tools) placed in the code to guide the synthesizer or enable special conditions during simulation may also complicate the crossover from simulation to synthesis. The result is a design that performs differently in simulation than it does on the hardware [16].

2.3.1 Custom Methods

Developers frequently employ improvised solutions while debugging. One such prevalent practice is to activate diagnostic LEDs commonly found on development boards when certain conditions or events occur. Developers will iteratively place triggers throughout the design to

illuminate an LED when certain areas of logic are active or desired conditions have been met. This technique is carried out by iteratively placing activating logic around areas believed to be problematic and interpolating to a likely cause of failure. An intimate knowledge of the circuit and often the device is required. If a shared memory or on-board processor is present, a developer may develop custom programs to inspect those memory locations. External logic analyzers are also used, but require signals of interest to be routed to external physical pins. This sometimes interferes with design pin assignments and can affect timing closure.

The drawback to these techniques is that any change triggers a full and lengthy rebuild of the entire design, which in some cases can be an entire day. The worst and all too common scenario is that several days may pass with no appreciable insight into the cause of the error.

2.3.2 Simulator-Based Development

Simulators are a standard tool available to developers, but have a rapidly diminishing rate of return of usefulness if too much time or effort is invested in the simulation model. Simulators are useful in developing custom logic, yet integrating with third-party IP or proprietary vendor cores can be problematic. To do so, developers rely on vendors to provide an accurate simulation model of their core, which is not always the case [16]. In some scenarios, such as with I/O or device configuration cores, no simulation model is feasible. An example particularly relevant to this research is the Internal Configuration Access Port (ICAP) core available on Xilinx FPGAs. The ICAP allows direct internal access to the configuration registers of the FPGA and enables a form of introspection on the configuration registers or more commonly allows the design logic to trigger and perform a reconfiguration of a portion of the FPGA while it is running. Since the ICAP relies on a configured FPGA and its fabric, a simulation model is unfeasible. This makes the ICAP a very difficult target for developers and requires multiple development turns.

Simulators make all signals and registers in the design readily available and have intuitive user interfaces where waveforms can be arranged and compared. Signal transitions or values can be searched for, breakpoints can be set, and results saved for later analysis or regression testing. In contrast, the same level of flexibility is available to software developers directly on the target platform.

Newcomers to digital design quickly learn that even though a design appears to work well in simulation, it may not behave as expected once implemented on the FPGA. While it is common for software developers to debug directly on the target hardware, to do the same is difficult in hardware development. Simulators are a convenient development environment, but one that can be overused since they provide no guarantee of accuracy. This mismatch is further compounded by hardware languages which allow practices that are legal in simulation, but impossible to implement in hardware. Optimizing synthesizers may misinterpret a designer's intention and implement the logic differently or even strip it from the design altogether [16]. Additionally, fine-grain simulation can be very costly in terms of time and resource requirements. Extremely large and complex designs, such as processors, may take days to accurately simulate even a single minute of operation.

2.3.3 Commercial Debug Solutions

The proprietary nature of FPGAs places the burden of creating the highly specialized tools necessary to debug them primarily on the manufacturer. As such, there are a limited number of third-party commercial tools that enable debugging directly on FPGAs, with the vendor's own tools being the most frequently used. Of those, embedded logic analyzers (ELA) are the most common class of debug tool, most often implemented as cores that are instantiated as part of the design with the results viewed through a software application with the look and

feel of an actual physical logic analyzer. Third party debug tools are often external state analysis applications.

When using embedded logic analyzers, designs may eventually accommodate or adapt to their presence and when removed create timing or routing anomalies that are difficult to trace and necessitate a new series of fixes. Frequently, this requires full re-implementation of the design. Routing and logic resources become a concern as wires are physically routed for each monitored signal and the cores consume on-chip resources. The addition of monitoring signals creates an increased burden for the router which must now incorporate the additional signals while continuing to meet timing constraints. These tools primarily rely on the limited on-chip memory for data storage and may compete with the user's design for these resources. Graphical user interfaces restrict interaction with the design since they can not be scripted as part of an automated testbench or integrated with other tools. Unlike software debuggers or test environments, these tools have no programmatic or script-engine capability of accessing or manipulating the design despite interest in such features [23].

These techniques suffer from long turnaround times since they are integrated into the design. The entirety of the design must be run through the tool flow, regardless of which portion or even how little of it has changed. In a worst-case and all too frequent scenario, a developer may only be able to test one revision per day, whereas in software development it is common practice to unit test or debug code directly on the target machine upwards of tens of times per day. Debugging may require multiple revisions in order to fully understand the design flaw.

To demonstrate the often lengthy time requirements for design turnaround, Table 2.2 presents Xilinx ISE 9.2i place-and-route CPU times for five non-trivial designs performed on a 2.66 GHz Intel Core i7-920 processor, targeting an XC4VFX100 part [20]. Blank table entries represent implementations that did not produce a result after 30 hours and that were terminated. The Xilinx MicroBlaze processor [24] is a configurable, pipelined, soft-core processor,

Table 2.2: PAR times for non-trivial designs with typical timing goals.

Design	f_{clk} **(MHz)**	**Flat design implementation (minutes)**	**Normal floorplan implementation (minutes)**
5 Microblazes	127.4	330	80
10 Microblazes	127.4	-	270
20 Microblazes	125	-	-
3 CFFTs	256.4	-	35
6 CFFTs	250	75	-

whereas Complex Fast Fourier Transforms (CFFT) are commonly found in DSP applications. These results make evident that even a few design iterations might result in week's worth of work, provided that the designs achieve closure at all. The time required for place-and-route is a common complaint amongst designers in surveys [25] and is responsible for lengthy verification cycles.

2.3.4 Assertion-Based Verification

Assertion-Based Verification (ABV) or simply *assertions* provide designers a means of verifying simulation designs by specifying conditions that must or must not occur. This requires an extensive understanding of the design in order to formally capture and characterize the assertions and how they must be formulated. There are two classes of assertions: *static assertions* are mathematically proven against the design, while *dynamic assertions* are performed during an extensive and exhaustive simulation. Neither method can provide complete coverage of all possible execution branches, but are widely accepted as necessary for verifying designs prior to production.

Assertions can be specified through a wide variety of different languages and environments. Verilog and VHDL both have language facilities to implement assertions and are defined

in IEEE Standard 1364 [26] for Verilog and IEEE Standard 1076 [27] for VHDL, as does SystemC [28] and System Verilog [29]. Cadence's *e* [30, 31] (formerly known as SpecMan) and Bluespec [32] are two examples of commercially supported languages that can be used for assertion-based verification.

2.4 High-Level Language Synthesis

The efficiencies of high-level programming languages are an attractive goal for hardware design. High-level language synthesis (HLS) aims to provide the same level of abstraction to hardware that is provided to software, isolating the source code from the underlying architecture which allows the program to be built for different platforms. HLLs allow the design to be layered, separating algorithms from low-level implementation details, thereby simplifying maintenance. HLS on the other hand, is still in its infancy and is not yet the equivalent of a HLL for hardware development.

Designs are typically first prototyped in an HLL, such as C, C++, or Matlab, where functional and algorithmic aspects of the design are specified using complex control constructs [33]. However, HLLs are untimed and do not capture the necessary architectural details for the complete specification of a hardware implementation. High-level models serve a critical purpose in providing a reference or "golden" design for development and are referenced throughout the development cycle to determine functional correctness. These comparisons are often manually performed and limited by the restricted access to the physical design.

High-level programming languages were developed to provide a level of abstraction above the underlying hardware architectures. As software design methods evolved—from the earliest programs being coded directly in machine code, then later in assembly languages—the need

for platform independence quickly became evident. The advent of compilers and high-level languages makes this abstraction possible, changing the programmer's view of hardware from physical registers and architecture-specific opcodes to meaningful, symbolic variables, complex control constructs, data types, and universal operators. Pursuit of a similar paradigm for hardware design was inevitable.

HDLs, such as Verilog and VHDL, provide a similar, but lower form of abstraction than those of software languages. HDLs are Register Transfer Level (RTL) languages, describing the flow of signals between registers and the operations on those signals. RTLs can further be divided into two variants: behavioral and structural RTL. Behavioral RTL describes a design by its behavior in terms of high-level operations such as `if-else` or other control statements and operations such as addition or subtraction.

Structural RTL provides the lowest-level of specification, instead declaring the structures of the circuit and the connections between them. Structures can be primitive operations such as Boolean or bitwise operations or even the macros provided by the architecture. Structural RTL is also used to design high-performance cores since more control is possible to reduce space requirements or ensure timing will be met. Designers using structural RTL have precise control over each signal, yet the implementation of such a design can be laborious, confusing, and error-prone.

Behavioral representations allow the design to be described using more abstract, higher-level expressions, such as defining an addition operation to occur on two signals rather than having to explicitly specify the implementation of an addition across the entire signal. Most notably, behavioral languages can specify events for signal transitions, such as clock edges, and have control constructs similar to those in high-level programming languages. These two features alone can be used to describe complex and fundamental design elements, such as state machines. Behavioral language constructs are further subdivided into synthesizable and

non-synthesizable, where the latter provides even more complexity and abstraction suitable for design testing.

HLS is an extension of these abstractions for FPGAs where a subset of a high-level language is processed to produce synthesizable RTL. HLS requires that the design entry be critically analyzed for structure and flow in order for the translators to correctly interpret the intent. Important constraints in hardware design, such as size and power, are no longer within the precise control of the designer. The abstraction performs its intended function, in part. Designs entered using HLS resolve to functional correctness far quicker than when using HDLs. Yet, long verification cycles still remain. Designs must be precise in their implementation because integration with traditional verification tools, such as formal equivalency, can be difficult or even impossible. Despite the front-end language portability of HLS, designs tend to evolve around the specific compiler being used, defeating the objective of portability. HLS designs also perform poorly in coverage tests because of the additional overhead that these tools introduce. The translation of HLL to RTL may produce artifacts, such as unreachable states or logic that can never be activated, known as *dead logic*. Debugging at the RTL level or integrating third-party IP are hampered by the unintelligible machine-generated RTL. Additionally, the untimed nature and various transformations from HLL to RTL make co-simulation and debug difficult, if not impossible. Often, the only solution is to attempt to debug the generated RTL which is frequently unresolvable to the original design [34]. An example design of a counter implemented in Accelerated Technologies ImpulseC is shown in Listing 2.2, while an abbreviated synthesizable netlist generated from the code is shown in Listing 2.3.

The listing shows a few of the disadvantages of using HLS. While a counter is a trivial function, the HLS implementation is not. The required overhead code is compiler-specific, requiring numerous special calls in order to set up and handle input and output. These

```
void counter(co_stream input_stream, co_stream output_stream)
{
  int counter;

  do {
    co_stream_open(output_stream, O_WRONLY, INT_TYPE(32));
    co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));

    counter = 1;
    co_stream_write(output_stream, &counter, sizeof(int));

    while ((co_stream_read(input_stream, &counter, sizeof(int))
            == co_err_none) {
      counter++;
      co_stream_write(output_stream, &counter, sizeof(int));
    }

    co_stream_close(output_stream);

  } while (1);
}
```

Listing 2.2: Implementation of a counter with HLS.

sections are not portable, as is a high-level language. The example uses blocking First In, First Out (FIFO) queues to control execution without the need for flags or signals. A read or write request will block until the FIFO is ready. The use of FIFOs in this manner simplifies a design, making it a streaming operation without additional overhead or checks.

The resulting synthesizable HDL exhibits more complexity than a hand-written counter. Nowhere in the HDL can the actual counter—a simple addition—be discerned. In a more complex module, it would be even more difficult to try and locate signals of interest to debug or integrate additional IP.

```
process (clk,reset)
  begin
    if (reset='1') then
     thisState <= init;
    elsif (clk'event and clk='1') then
      if (stateEn = '1') then
        thisState <= nextState;
      end if;
    end if;
  end process;

  stateEn <=
    '0' when thisState = b0s1
      and p_output_stream_rdy = '0' else
    '0' when thisState = b0s2
      and p_output_stream_rdy = '0' else
    '0' when thisState = b0s3
      and p_output_stream_rdy = '0' else
    '0' when thisState = b1s0
      and p_input_stream_rdy = '0' else
    '0' when thisState = b1s1
      and p_output_stream_rdy = '0' else
    '0' when thisState = b2s0
      and p_output_stream_rdy = '0' else
    '1';

// **** Several lines cut ****

process (ni113_suif_tmp,thisState)
  begin
    case thisState is
    when init =>
      nextState <= b0s0;
    when b0s0 =>
      nextState <= b0s1;
    when b0s1 =>
      nextState <= b0s2;
    when b0s2 =>
      nextState <= b0s3;
    when b0s3 =>
      nextState <= b1s0;
    when b1s0 =>
      if ((not ni113_suif_tmp(0)) = '1') then
        nextState <= b2s0;
      else
        nextState <= b1s1;
      end if;
    when b1s1 =>
      nextState <= b1s0;
    when b2s0 =>
      nextState <= b0s0;
    when b3s0 =>
      nextState <= finished;
    when finished =>
      nextState <= finished;
    when others =>
      nextState <= init;
    end case;
  end process;
```

Listing 2.3: Synthesizable RTL generated from Listing 2.2.

2.5 Dynamic Runtime Reconfiguration

Xilinx's Partial Runtime Reconfiguration (RTR) [35] allows preselected regions of an FPGA to be reconfigured while the rest of the device continues normal, uninterrupted operation. There is tremendous potential for design size reduction with RTR as one or more modules that are not simultaneously required can be swapped in and out temporally, saving space. Reconfiguration takes less than a second, making it practical for many applications. This concept is illustrated in Figure 2.8. The reduced space requirements can be significant as opposed to the traditional methodology of allocating static, permanent space to the entire design. By convention, most FPGAs designs are static without regard to the actual execution profile [36]. However, the dynamic methodology of RTR has yet to achieve acceptance.

Implementing RTR is a manual and imprecise art form. The RTR tool set has never been part of the mainstream application suite, rather it is distributed as a patch. Recently, the RTR patch requires an additional license fee. The I/O interfaces to reconfigurable regions, referred to as *bus macros*, are restricted to only one edge of the region. Until recently bus

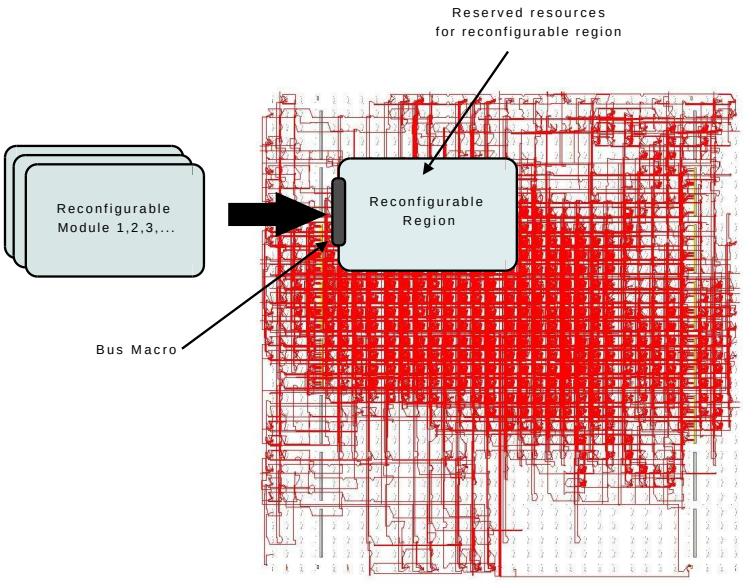

Figure 2.8: Partial reconfiguration for multiple logic modules in the region.

macro creation and placement was a manual task. Automated bus macro placement was the subject of a recent research topic at Virginia Tech [37] as part of the PATIS project, and was eventually eliminated from the vendor's build process altogether in favor of another method. The interface between the reconfigurable region and the rest of the design is critical to reliable, error-free operation. Alignment between the two is analogous to constructing two physical, interlocking parts. During reconfiguration, I/O between the reconfigurable region and the rest of the design must be suspended. Furthermore, the I/O ports must be properly physically aligned on all reconfigurable modules. Reconfigurable regions are restricted to only rectangular shapes (as opposed to ASICs where rectangular shapes are merely a convention) and developers are responsible for placing and sizing the region within the FPGA so that it has sufficient resources for all of the candidate modules. If a module with different resources or interface is later added to the set of reconfigurable modules, this may disturb the floorplan of the static region. Resource estimation floorplanning was the subject of another area of research in the PATIS project [38].

Besides the increased burden of development, designers are reluctant to adopt RTR because of its dynamic nature. In RTR, the act of swapping modules into and out of the reconfigurable region turns a single design into multiple, distinct designs, increasing the test space. Unlike most components of a manufacturer's portfolio, RTR cannot be simulated and developers must develop directly on the hardware where visibility is limited. This uncertainty and the resulting difficulty in verifying such a design are a few reasons that commercial designers avoid RTR.

2.6 Summary

FPGAs contain not only programmable logic, but many are available with an assortment of specialized processing cores such as microprocessors, memories, DSP cores, and specialized I/O cores. Combined with the capability to realize and implement an arbitrary function while also extracting parallelism, this flexible architecture provides a performance-competitive platform on which to produce high-speed, space-efficient, low-power streaming applications. FPGAs are capable of processing high-speed streams of continuous data, rather than the non-deterministic read-buffer-process-write cycle required with conventional computer architectures.

The great flexibility and power of FPGAs comes at a price: the design process is complex, time-consuming, and prone to subtle problems. Unlike software which has a widely accepted and open process with a wealth of independently contributed tools, FPGA tools are proprietary and vendor-specific. The complexity of FPGAs is instrumental in the difficulties encountered when trying to debug and diagnose problems. The lengthy implementation cycle, as well as the lack of visibility into a design are the most significant obstacles. Alternative design methods, such as high-level languages which were elemental to the rise of

computers and software, has not made FPGAs more accessible in the same manner. Rather, HLS faces other obstacles to acceptance.

A unique capability of FPGAs is the ability to dynamically reconfigure a select region of the device. This capability was previously commercially inaccessible or undesirable because of its complexity and uncertainty in production environments, however it holds significant promise in development and debug environments. A review of debug-related research and products reveals partial reconfiguration as an unexplored and potentially useful tool in accelerating the debug process.

Chapter 3

Related Work

The FPGA debugging field has amassed a large body work. The goal of developing an intuitive and productive means of debugging is ambitious given the closed nature of FPGAs and their prominent role in electronic system design. The earliest implementations have their roots in the ASIC market, many based around long-standing industry standards such as JTAG which were developed to verify individual components as well as entire systems. FPGA-specific products are firmly rooted in leveraging on-chip components, such as JTAG cores or embedding specialized debug cores into the design.

3.1 Categorization of Debug Approaches

A review of commercial and research FPGA debugging products reveals two prominent approaches: embedded logic analyzers and JTAG-based analysis tools. Embedded logic analyzers recreate the familiar interface of physical logic analyzers but are implemented as part of the FPGA design by inserting special cores wired directly to signals of interest. These cores rely on a *capture methodology* where the selected signal's activity is internally recorded

beginning when predetermined conditions are met. While some limited changes may be made at runtime, the trigger conditions that initiate the capture in addition to the selected wires are implemented as part of the design's logic and specified at design time. As part of the physical design, these cores compete for system resources, specifically on-chip RAM. Captured signal traces are stored on-chip until read out via JTAG and displayed in a graphical user interface resembling a simulator's waveform viewer. Interaction with the design is limited: neither debug scenarios nor tasks can be scripted into a larger comprehensive or automated test framework. Capture conditions cannot be appreciably modified without re-implementing the entire design, and different parts of the design cannot be inspected other than those whose signals were initially specified. However, designs can typically run at or close to their target operating frequency.

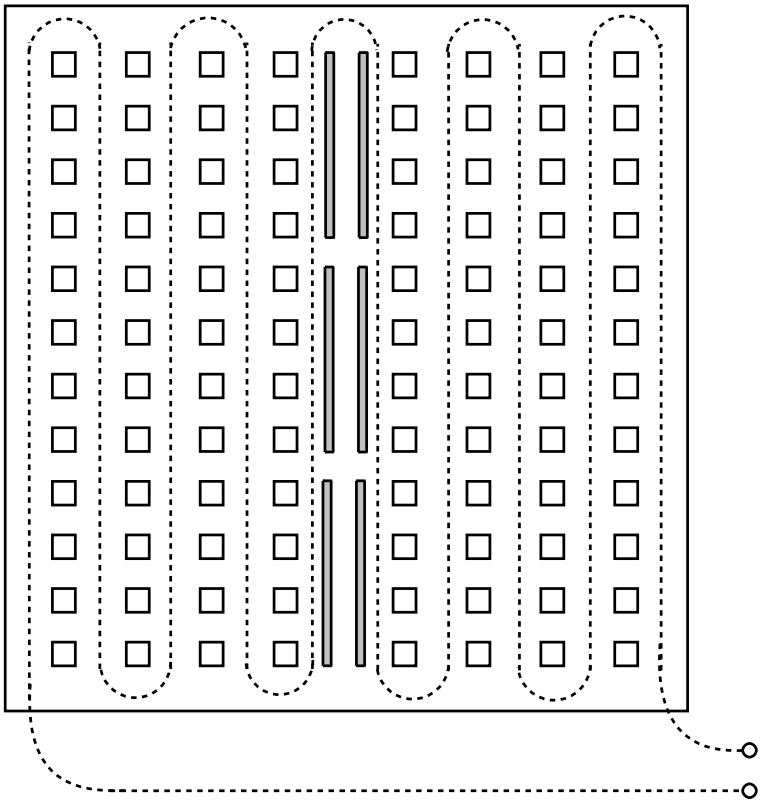

Figure 3.1: JTAG chains visit every resource in the FPGA.

Table 3.1: Timeline of Xilinx Boundary Scan chain length.

Release Year	Device	Approximate Size of Configuration Chain (bits)
2002	Virtex-II Pro XC2VP100	34,272,000
2004	Virtex-4 XC4VLX200	51,367,424
2006	Virtex-5 XC5VLX330T	82,696,192
2009	Virtex-6 XC6VLX760	184,823,072
2011	7 Series XC7V2000T	447,337,216

The other prominent debug approach utilizes JTAG for state inspection. JTAG is an acronym for the Joint Test Action Group, an industry board assembled to define the standard now found in IEEE Standard 1149.1 [39]. JTAG is a generic, non-proprietary means of debugging and controlling electronic components, embedded in a range of products from simple components to sophisticated processors. While JTAG tools do not require the invasive instantiation of cores and routing of signals as found in embedded logic analyzers, it is slower to operate. A snapshot of the device state is captured internally and must be serially shifted out in its entirety for inspection. Figure 3.1 shows a conceptualized JTAG chain winding throughout the entire FPGA, passing by each resource. The snapshot typically requires that the design's execution be halted. Once retrieved, any state register of the design can be inspected.

JTAG-based tools offer complete visibility without significantly altering the design as is the case with embedded logic analyzers, however they are limited in how much control can be exerted over the design. JTAG-based tools do not offer the ability to control an FPGA design's execution, nor can the debug scenario be altered. JTAG operates at a significantly slower clock rate than the average FPGA design and as devices increase in size, density, and complexity, the time required to shift the entire chain out proportionally increases. A simplified schematic representation of a JTAG chain is shown in Figure 3.2 and Table 3.1 shows the size of the configuration bitstream for the largest of the major device releases since

2002 [40–44]. The configuration bitstream is a good approximation of the boundary-scan chain length.

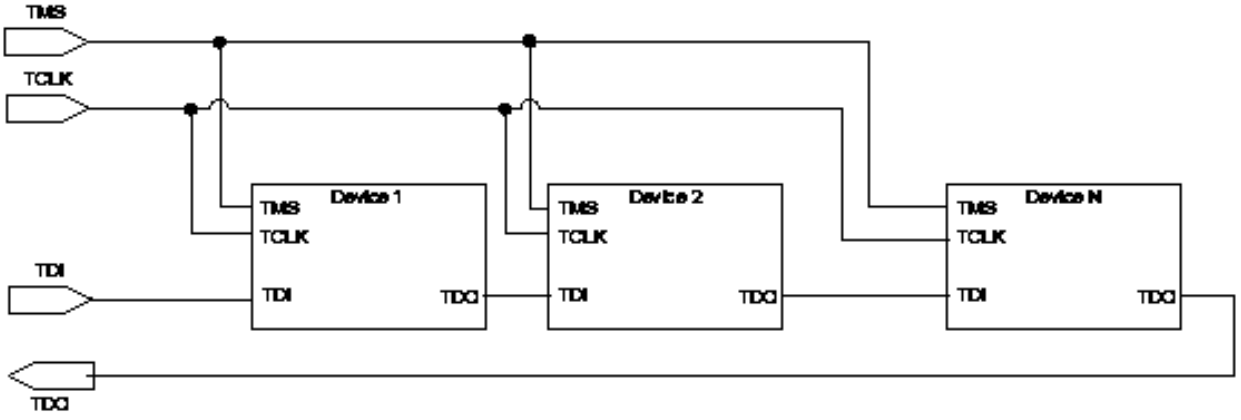

Figure 3.2: JTAG daisy chains require few connections.

JTAG requires few connections but is flexible enough to allow devices to be daisy-chained. Each JTAG-enabled device is manufactured with vendor- and model-specific identifiers so that multiple devices can be placed into the same chain and still be uniquely identified. JTAG requires only small number of input and output pins making it easy to place on the smallest of component boards. As such, it has become the defacto standard for debug and testing on virtually all electronic devices. It is found on virtually all FPGA devices.

As shown in the Figure 3.2, the header's external input is wired to the first device's TDI pin; in turn, its TDO output is connected to the next device's TDI input and so on until the final device's TDO pin is wired back to the header. A single master clock and device select are globally wired to all devices of the chain. One of JTAG's weaknesses stems from its simplicity. The sole data line restricts data transfer to a serial model throughout the entire chain. JTAG's modest clock rate makes this model unsustainable as devices and likewise chain length increases. Rapidly increasing FPGA densities make JTAG less practical for future devices.

3.2 Commercial Debug Products

3.2.1 Vendor Products

The proprietary nature of FPGAs restricts development of debug products mostly to the original manufacturer. However there have been a few notable third-party contributions. By far, vendors prefer developing embedded logic analyzers (ELA), extending the waveform viewer window found on simulators into the debug environment. Xilinx's ChipScope Pro [45] and Altera's SignalTap II [46] are two examples of the embedded logic analyzer class of tools. ELAs require that specialized cores be instantiated as part of the design. Signals of interest are manually selected through a user interface. The entire design is then run through the standard tool flow. Figure 3.3 shows how different components of Xilinx's ChipScope Pro interact with and become deeply integrated into the design. Wires for the framework and signals of interest are routed globally while the debug cores compete with the primary design for resources. The most obvious and limited resource is RAM, which are plentifully required by debug cores since signal traces are stored on-chip until read out and displayed. Modifications to the debug configuration, such as changing which signals are to observed can be disruptive, create routing contention, which in turn may cause the design to fail subsequent timing analyses. ELA tools are often restricted to the top-level design signals and are not able to inspect arbitrary internal signals without the need to modify intermediate interfaces.

3.2.2 Third-Party Products

Despite the closed nature of FPGAs, there are several notable commercial products. Gate-Rocket's Device Native Verification [47] system allowed designs to be run directly on an

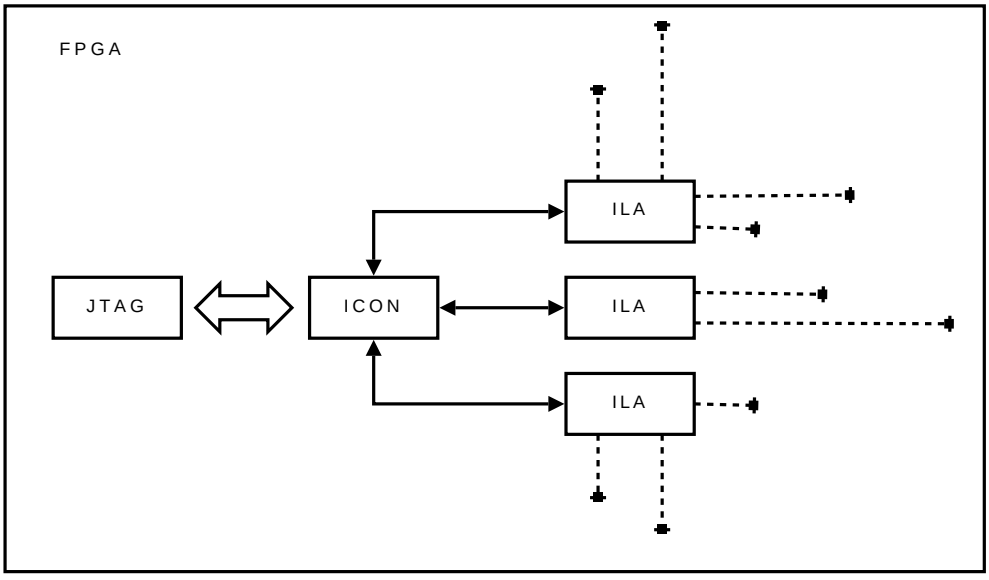

Figure 3.3: ChipScope gathers data through physically routed nets.

FPGA. The system comprised of an external Rocket Drive housing an FPGA which connected directly to a workstation. The system tightly integrated with several simulators, utilizing their user interface and waveform viewers. Individual waveforms could be displayed in the waveform viewer directly from the hardware while bypassing traditional and slower software simulation. Technical details were sparse in public literature and documentation was restricted to customers so the system's architecture was not widely understood. Despite widespread marketing efforts and enthusiastic praise as an innovative approach, GateRocket ceased operations in July of 2011 due to lackluster sales [48].

Sandbyte's FPGAXpose [49] framework provides visibility into all design storage elements by reading their state through a JTAG connection. While eliminating the need to instantiate cores into the design, this exposes the application to the disadvantages of JTAG. JTAG is a comparatively slower interface more suitable for interactive work, rather than full-speed analysis. Like other third-party products reviewed, details of this architecture are not widely publicized.

Synopsys' Identifty RTL debugger [50] bears a close resemblance to conventional software integrated development environments. Debugging is performed in a source code viewer on the original RTL, or through a waveform viewer. Like embedded logic analyzers, a specialized Instrumentor core is instantiated into the design but rather than recording signal traces, annotations are displayed on the original source RTL. Identify combines traditional integrated software development environment features such as syntax highlighting and context pop-ups for symbolic register values with hardware debugging. The incremental development flow is leveraged which attempts to preserve previous implementations, thereby reducing build times. JTAG is used to shift the FPGA's state out to the Identify user application for inspection.

The Cadence Palladium toolset, part of the Incisive XE suite, is a stand-alone emulator/ accelerator allowing rapid hardware/software co-verification. Product literature claims from 100 up to 1,000,000 times speedup over conventional software-based RTL simulation. Debugging capabilities include embedded logic analyzer access with immediate test point modification without the need for re-implementation, global visibility without the use of physical or routed wire probes, and limitless trace capture [51]. It is unclear what architecture supports these claims.

3.3 Debug-Related Research

The most ambitious debug products have been the product of research projects. Research-related debug products are often very innovative, but tend to be restricted to specific architectures. Unfortunately, the most exciting debug products are no longer supported.

During development, the NPU processor [52] incorporated custom logic that automatically halted execution and allowed automated, full inspection of the state. NPU is a 32-bit,

binary compatible processor based on the Intel 486 architecture. Developed on an FPGA, the developers were motivated by some common FPGA obstacles to productivity. Long turnaround times and limited visibility into the design were responsible for slow progress in development. Incorporating custom breakpoint logic activated by the program counter, the breakpoint logic automatically suspended execution and triggered an automated state dump over JTAG. The technique relies on JTAG's ability to push custom instructions into the design and later shift out the device state. The breakpoint address was set by JTAG, and when activated triggered a JTAG shift making the device's internal state available. This research presented several novel approaches to debugging including generously segmenting the design into 11 distinct and manageable JTAG chains to accelerate the otherwise lengthy shift operation, and the automated approach to shifting out state and automatically resuming execution. This domain-specific technique is not suitable for arbitrary designs and aspects of it required re-implementation for modifications to the debug framework.

Researchers at the Lappeenranta University of Technology developing an embedded active magnetic-bearing controller on a Virtex-II Pro FPGA used the device's PowerPC processors to read internal signal data and to debug the design. The authors make note of the difficulty in achieving target clock rates and the large amount of memory required for their debug activities [53]. Approaches such as these tend to be domain-specific and require repeated modification and re-implementation of the design.

An in-situ debugging system developed at the Los Alamos National Laboratory employed JTAG for state readback [54] in a unique way. Based on a Xilinx reference design [55], the researchers extended the capabilities of an external Microchip PIC processor to provide access to both transactions and internal state. The system was primarily developed to aid in debugging otherwise inaccessible installations, such as those deployed into orbiting satellites. The microcontroller directly drives a JTAG chain, through the second of the FPGA's two

Boundary Scan (BSCAN) modules, leaving the first available for vendor tools. The debug core included a programmable triggering mechanism which enabled system traces to be recorded directly into internal Block RAMs (BRAM). The framework allowed both reading and writing to arbitrary design registers and was accessible from a Linux command-line application.

JHDL [56] developed at Brigham Young University, was first conceived as a high-level hardware description language for RTR applications. JHDL evolved into one of the earliest full-blown design environment supporting high-level design entry, floorplanning, netlist generation, schematic generation, and an integrated debug and simulation environment. Despite the use of Java, JHDL is a structural design environment, enabling rapid design entry using a popular high-level language that appears to have effectively separated the design from the platform, a limitation that plagues conventional HDLs. The target device is specified as part of the design and is easily re-targeted to another supported platform. Low-level, device-specific instantiations are avoided through the use of abstracted component libraries. JHDL also provided several unique perspectives to hardware development. First, both hardware and software could be co-developed in the same environment using the same language. While hardware development occurred by extending preexisting JHDL libraries, software development occurred in the conventional sense. Another unique aspect of JHDL was the combined hardware and simulation environment that could be executed in parallel, effortlessly switching between the two while annotating different views of the design, such as a schematic or waveform view. Development appears to have halted years ago, with the last supported devices being the Xilinx XC4000, Virtex, and Virtex II [57].

Like JHDL, JBits [58] was also Java-based circuit design language. Developed by Xilinx and first targeted towards the Virtex line of FPGAs, JBits provided a hierarchical, object-oriented approach to design entry. Using Xilinx's Runtime Parameterizable (RTP) Core library, a

parameterizable library interface to Xilinx's core device components, runtime reconfiguration was accessible from a high-level language as were floorplanning capabilities using JPlace. These unique high-level capabilities have not reappeared since JBits was discontinued. JBits was unique in that it allowed direct manipulation of the low-level bitstream implementation from a high-level language. However this direct access made JBits complex requiring the tool to contain architecture-specific details for each of the supported devices. JBits' bitstream manipulation capabilities made it an ideal candidate for hardware-based debugging tools. A project at Brigham Young University used JBits to instrument a JHDL design for debug by directly manipulating the final bitstream and was demonstrated to be effective [59].

The BoardScope [60] debugging suite provided graphical and command-line user interfaces to the design's data flow. BoardScope was integrated with the JBits [59] suite and the DDTScript language which enabled instantiation of a run-time core abstraction that could be precisely placed and manipulated on the FPGA fabric. Inputs to cores could be altered, state read, and bitstreams exported for later use, allowing for interactive debugging [61]. BoardScope never appears to have been offered as a commercial product and is not available for architectures beyond the Virtex-II Pro.

3.4 High-Level Synthesis

HLS is in of itself not a debug methodology, but a means of raising the abstraction layer away from the low-level hardware implementation, thereby eliminating the need to debug at that level. HLS offers a direct path to creating synthesizable HDL from an HLL. While HLLs are the fastest, most precise, and abstract means of specifying and validating algorithmic correctness, HLS has yet to gain widespread acceptance for reasons such as poor coverage, large overhead, and dead logic. The automatically machine-generated RTL is difficult to integrate with third-party IP cores and to process with formal verification tools [34].

Accelerated Technologies ImpulseC [62] high-level synthesis framework is based off a subset of the C programming language. The compiler is based off the StreamsC compiler [63] developed at Los Alamos National Lab, which in turn relies on the Stanford University Intermediate Format (SUIF) [64] to generate synthesizable netlists from high-level language. The user interface is similar to software development environments, with features such as code completion and syntax highlighting. Like many HLS environments, the machine generated RTL is difficult to reconcile to the original source code, making it difficult if not impossible to manually debug or verify. A complete reference on programming for FPGAs with ImpulseC can be found in [65].

ImpulseC was the basis of at least two notable research projects investigating improving developer productivity. Research done at the University of Florida [66] developed a means of embedding assertions authored from the high-level programming environment into the synthesized netlist, providing more visibility of the synthesized design. The synchronous assertions provided immediate feedback to the developer, increasing confidence and verifiability. An overview of assertion-based verification can be found in Section 2.3.4.

A Virginia Tech doctoral dissertation [36] focused on abstracting away low-level implementation details for partial Runtime Reconfiguration (RTR) into ImpulseC's high-level language. This work was significant in several regards. First, RTR is a low-level, device-dependent technique which requires significant knowledge of both the device and the design. RTR is an unlikely candidate to be transparently integrated into a high-level language. Second, RTR enables a smaller, less-expensive, more power-efficient device to have a region designated for temporally allocating modules for on-demand execution rather than statically allocating all the modules at design time. This satisfies one of the criticisms for HLS that machine-generated RTL designs are less power-efficient than custom RTL designs.

3.5 Deficiencies in Existing Approaches

The previous overview of commercial and academic debug environments betrays a pattern: vendors provide embedded logic analyzers while third-party vendors favor JTAG access methods. This is not surprising. Vendors have the distinct advantage of leveraging the knowledge of their own proprietary architectures and in some cases embedding on-chip debug peripherals to assist. Third-party vendors are left with few options other than the industry-standard JTAG to access the internals of the FPGA or to develop their own interfaces. The primary shortcoming of all the approaches is the lack of visibility into the FPGA.

Third-party providers depend largely on JTAG since it is one of the only standardized interfaces found on all platforms. There is the option using the abundance of I/O that the FPGA has to offer, however this is not as attractive as option as it may seem. The GateRocket and Cadence products sell their own development platforms to leverage this opportunity, providing a higher level of service. There is a drawback to this approach, notably that these platforms are more than likely not at all similar to the final target platform. They most certainly will lack the peripherals, connectors, and configuration of the final platform. Porting to another device, even to a different sized part within the same family, can be problematic particularly with regards to placing and routing the design.

The benefits of the vendor's embedded logic analyzers are clear, however there are drawbacks to the approach. These tools are sealed within their user interface which makes integration with other tools difficult. The primary disadvantage of ELA-style approaches is the semi-permanence of the chosen context. That is, the use of these tools is akin to a design decision. The debug facilities become deeply integrated into the design, particularly in regard to wire routing. Memory requirements are also a difficult trade-off. Onboard memory is hardly ever sufficient for the design and with ELA it must be shared. Memory use must be delicately balanced between the number of signals to monitor and the number of samples to record.

Table 3.2: Summary of debug products.

Source	**Product**	**Class**	**Comments**	**Status**
Vendor	Xilinx ChipScope Pro	ELA	Instantiated cores, on-chip trace recording, graphical user interface	Available
Third-Party	GateRocket	Proprietary	External platform; waveform viewer integration	Defunct July 2011
	Sandbyte FPGAXpose	JTAG	No embedded cores, complete state visibility	Available
	Synopsys Identify RTL	JTAG	Embedded core; annotations on waveform viewer, source code viewer	Available
	Cadence Palladium	Proprietary	External, stand-alone accelerator; software co-verification; embedded logic analyzer; rapid test point modification; very expensive; targeted toward ASIC development	Available
Research	JHDL	Unknown	High-level language with integrated debugging and simulation support; rapid perspective change between simulation and on-hardware debugging	No longer supported
	JBits	Unknown	Direct manipulation of bitstream; rapid debugging; limited to select devices	No longer supported
	BoardScope	Unknown	Graphical and command-line support	No longer supported

Capture methodologies are not always appropriate for low-level debugging. In addition to Xilinx's ChipScope Pro, the offerings from Sandbyte and Synopsys fall into this category as well.

The obstacles to providing a fast and visible architecture for debugging are the limited access points into an FPGA and an unobstructed view of the design. JTAG has been a reliable and robust standard for debugging for well over a decade. However it is slow, serial, and typically applied globally across the device. With JTAG, the debug vantage point is the

equivalent to that of a long, narrow tube: significant time and effort is required to pass information along the tube and the tube must be emptied each time, regardless of how much information is relevant. A summary of the debug products covered in this section is given in Table 3.2. Domain-specific approaches, such as those implemented for the NPU processor, the Lappeenranta University project, and the work undertaken at Los Alamos were not included. The most innovative approaches, those generated from research projects quickly faded. There is little documented evidence to explain why these projects failed to achieve commercial success. However the growing concern over IP protection suggests that tools such as JBits, BoardScope, and JHDL exposed too much of the internal architectural details.

3.6 Summary

Debug solutions are limited. Vendors provide embedded-logic analyzers to extend the original development environment found in simulation to debugging, while third-party vendors are rather limited in their approaches. Applications for JTAG are the most common, however the least likely to scale to future platforms. JTAG is a universally adopted and open standard, but places the debugging context at the end of a slow serial line. Some third-party approaches also include stand-alone development platforms alleviating some of the shortcomings, however this does not permit development directly on the final target platform. The most innovative approaches have been research products, but these typically exposed more architecture-specific details.

A significant shortcoming in almost all the approaches reviewed is one of perspective. The debugging context is placed outside the FPGA, with a restricted view onto the FPGA's state. Embedded solutions, such as those requiring cores to be instantiated into the design

are based on capture methodologies. The storage requirements limit the width or depth of information viewable. With JTAG solutions, this view is serialized requiring the entire state to be shifted out regardless of how much information is needed. A new perspective, one from within the FPGA with random access, is needed.

Chapter 4

Improving Abstraction and Turnaround Time

The model verification gap and the deficiencies discussed in the previous chapters highlight some of the weaknesses in FPGA development flows. The model verification gap exists because high-level reference models are used primarily to verify against simulation models and because they can not be directly coupled to the hardware. In many cases, validation against high-level models is manual. Furthermore, there are significant gaps in the ability to analyze the implemented hardware in detail without compromising time or accuracy.

FPGA development environments lack the flexibility found in software development environments, most notably intuitive means for quickly testing modifications and validating designs. Three prominent attributes of software development are identified and targeted in this research for FPGAs: *visibility*, *controllability*, and *agility*.

Visibility is the extent to which design elements, such as signals, ports, and registers, are observable once implemented in hardware. In software debugging, special annotations such as symbol tables are compiled into the binaries and performance-enhancing optimizations

are avoided to improve the observability of the application. Once a variable is in scope, it can be printed or "watched" within a debugger where changes to the variable are trapped. No such standardized facility is natively available in FPGAs other than diagnostic LEDs, and debugging remains largely technology- and vendor-dependent as well as a learned skill. *Controllability* is the extent to which a design's state can be manipulated or altered during execution. Examples include forcing variables to values at runtime, a feature widely supported by software debuggers but not architecturally supported in FPGAs. Altering the state of the debug mechanism is also not natively supported. Software breakpoint modifications do not require re-compilation or restarting of the affected unit, while most FPGA vendor tools implement capture or trigger mechanisms as part of the monolithic design flow. Finally, *agility* is defined as the ease and efficiency at which modifications can be made to a design. It is a common occurrence during FPGA development that a trivial change requires a re-implementation of the entire design, a lengthy process taking potentially tens of hours for large designs [20]. Software build tools such as `Make` selectively rebuild only the affected units within the dependency tree of the modified unit. FPGA flows have support for incremental or module-based development methods, but these have strict requirements which are difficult to attain. Agility as applied to debugging is similar. While most vendor-specific means of debugging and design analysis are built as part of the design, agility in this context refers to modifying the debug configuration.

There are two phases of the development cycle where these weaknesses are most prominent: initial modeling and hardware validation. High-level reference models made during initial design planning lack integration with the rest of the flow. This inability to integrate reduces the model's effectiveness and hinders productivity during validation. Later on during low-level debugging, the lack of insight into executing hardware during runtime and the inability to inspect it are significant shortcomings of all FPGAs. These are addressed through two separate approaches, *High-Level Validation* and *Low-Level Debug*.

The remainder of this chapter is organized as follows:

High-Level Validation

In Section 4.1, the High-Level Validation (HLV) framework is presented. HLV seeks to couple a normal, high-level reference model written in a standard HLL to synthesized and implemented hardware without significantly modifying either while simultaneously raising the layer of abstraction for validation. In essence, HLV serves as a software-based, unit-testing framework for hardware. Aspects of mapping software and hardware elements are discussed.

Low-Level Debug

In Section 4.2, the low-level aspects of improving debugging time turnaround are discussed. Low-Level Debug (LLD) changes the perspective of FPGA debugging from an external viewpoint to an internal one, removing much of the latency and design-time commitments from debugging. LLD changes the model of analyzing FPGA state from the conventional serial JTAG analysis method to treating the FPGA as a random access memory where state can be arbitrarily analyzed. Additionally, LLD leverages the rapid implementation times of partial reconfiguration by separately partitioning off debug logic so that it can be rapidly swapped out without disrupting the rest of the design. LLD retargets the three attributes of software development—controllability, visibility, and agility—for FPGAs.

Dynamic Modular Design and Validation

In Section 4.3, the Dynamic Modular Development (DMD) project is discussed. DMD is the overall framework in which this debugging framework is presented. DMD aims to improve turnaround time for the entire range of FPGA development tasks including floorplanning and implementation time as well as the debugging aspects discussed here. PATIS, a major component of DMD, is an automatic floorplanning tool that isolates all top-level modules for efficient design turnaround. PATIS is capable of adjusting floorplan boundaries in two

dimensions while also balancing resource requirements for each module. PATIS' floorplan database is populated in the background and ensures that a suitable floorplan is always instantly available without stalling the development process as a new one is produced.

4.1 High-Level Validation

The HLL functional models that are created during the first phases of the development cycle serve a critical role throughout the process. In addition to providing a guideline from which to build the hardware, they also provide critical reference data by which to validate the design's accuracy. The model verification gap exists because the reference model is physically isolated from the subsequent design steps. HLLs are not natively compatible with HDLs and high-level synthesis has not yet fully matured or been standardized for widespread use. Any validation that occurs between the reference model and subsequent design steps, such as simulation or hardware, are mostly manual or require significant user intervention. Bridging this gap by directly linking the model and simulation is a viable solution, but of limited usefulness. Both the model and the simulator can run on the same platform and frameworks such as the deprecated Program Language Interface (PLI), now succeeded by the Verilog Procedural Interface [26], provide a convenient and structured means of interfacing HDL simulations to C-language programs. However the vagaries of simulation, such as unintended differences between the simulation model and the actual implementation, make this an unreliable strategy. Instead, a means of linking the reference model directly to the hardware would provide a robust means of directly validating hardware without the use of HLS.

At first glance, high-level reference models share little in common with hardware implementations making them unlikely candidates capable of being coupled. HLLs do not characterize

any timing constructs and are mostly algorithmic in nature. Hardware designs leverage concurrency and require skillful pipelining for the best designs. *Procedural* languages provide a convenient mechanism by which to model a hardware design's structure. Individual procedures or functions provide a convenient means of logically segmenting a functional model and mapping it to individual hardware modules. Hardware modules and software procedures have much in common and are the most logical pairing for mapping between the two. Both have well-defined boundaries that hide the functionality, state, and implementation from the outside. In software, the term *scope* defines the context wherein variables exist and to what other areas of the design these are visible. Both also have a well-defined gateway through which information passes, known as an *interface*. In software, the function's arguments defined by its signature or prototype are the primary means of passing data into and out of the procedure. In hardware, the module's ports define this interface. Modules and procedures are also a convenient means by which to reuse functionality. In both cases, repeated use of functionality warrants that it be encapsulated or *refactored* into a separate procedure or module so that it can be repeatedly and reliably be instantiated where needed. Figure 4.1 shows how refactoring frequently used code into a function improves the maintainability of the code as well as reduces the likelihood of coding errors. In addition, such refactoring may improve other metrics such as final design size.

Mapping an untimed reference model to hardware can occur only if three conditions are satisfied. First, functional entities in each must be logically mapped to one another. In the simplest case, each procedure could be mapped to a corresponding module. Second, the data inputs and outputs of each should also be mapped. The simplest case again is a direct one-to-one mapping. Control signals, which are discussed in Section 4.1.2, are excluded from this mapping. Finally, the interaction of the control signals and the latency of the hardware module must be modeled in software. All three can be accomplished without invasive modification of either the reference model or the hardware.

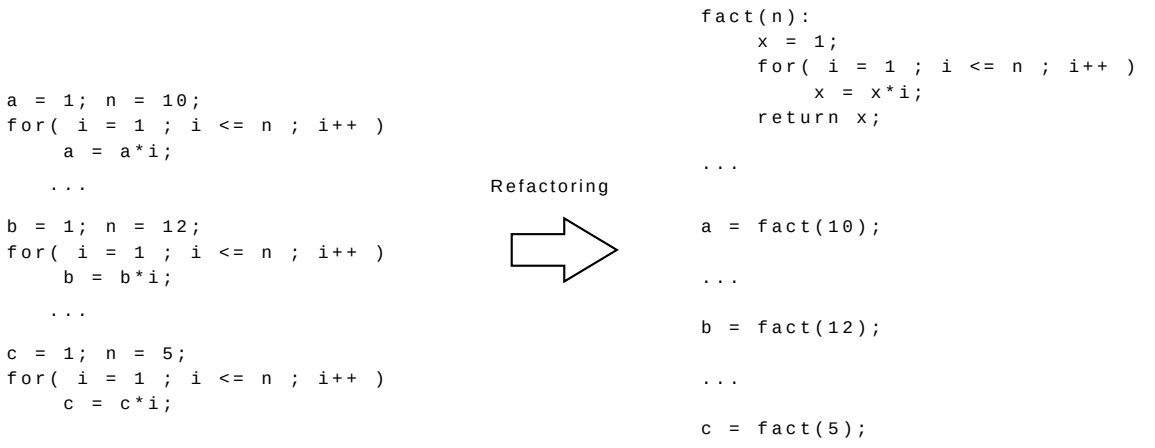

Figure 4.1: Refactoring code improves maintainability.

4.1.1 Mapping Software Procedures to Hardware Modules

HLLs are untimed, meaning that the actual execution time, measured either as real ("wall-clock") time or as a function of any parameter related to the platform on which it is running, is irrelevant. Hardware design on the other hand is characterized by its execution time as a function of the number of clock cycles required or *latency*. While the purpose of the reference model is to clearly define the desired and correct results for given inputs through an algorithm, additional metrics apply to hardware. Hardware execution is bounded by the time required for each individual stage of the operation or through some signaling mechanism to indicate that an operation has completed. Despite this mismatch, it is still possible to map a functional software representation to a hardware model.

Procedures have well-defined boundaries and a single point of entry. Likewise, hardware modules also have well-defined boundaries and it is through this similarity that a well-partitioned software model can be effectively cast in hardware. Figure 4.2 illustrates a one-to-one mapping of hardware and software elements.

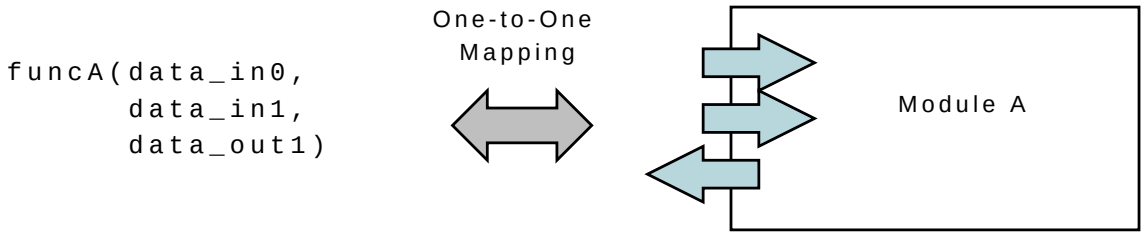

Figure 4.2: Mapping data paths in hardware and software.

4.1.2 Mapping Data and Control Signals to the Software Model

Mapping the interface of hardware modules to that of a software procedure is rarely a simple one-to-one mapping. Hardware module interfaces have two types of ports, data and control, while software procedures do not typically address these hardware constructs. Data signals are easily matched to the corresponding data variables of the software model since the model is algorithmic in nature. Control signals are a broad category of signals that can indicate actions to be taken such as an operation; the different states of the circuit such as busy, ready, or acknowledge; or meta-information about data signals. For example, pipelined designs have several stages in which distinct operations are performed. As the data moves through the pipeline, each stage must coordinate with its neighbors. A common scenario in pipelined designs is that valid data may not be available for each clock cycle, thereby introducing stages that are empty or contain invalid data. These empty stages are known as *stalls* or *bubbles*. Since hardware execution can not be stopped, the stages of the pipeline must also propagate which data is valid. Invalid data in a pipeline is a normal occurrence and caused when pipelines must wait for data to become available, such as the time in between image frames in video processing applications or the service time required for a memory access. Control signals are shifted along the pipeline with the data to indicate that the data is valid and not the result of a stall. In software models, control signals are not considered since they are not part of the algorithm. The simulation model is the first time in the development cycle that control signals are considered. However, by leveraging the

low-level detail of signal interaction visible in simulation, a software model of control signal interactions can be extracted. This model can be integrated into the reference model thereby creating a means of interacting directly with the hardware implementation. An overview of this process is illustrated in Figure 4.3.

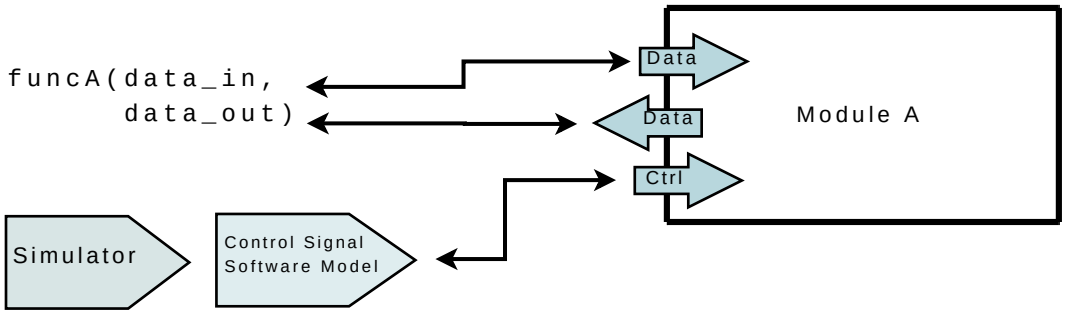

Figure 4.3: Developing a software model for hardware control signals from simulation models.

4.2 Low-Level Debug

Lack of low-level visibility, control, and agility of a design is the other deficiency of FPGA development methodologies. Once the design has been implemented, it is difficult if not impossible to see what the precise state of any register is at any given time, arbitrarily stop the design to inspect registers, or quickly change the debugging configuration. In comparison, all these tasks are easily accomplished in software development environments. The tools providing the most visibility into a design are those that are built into the design, which are at the same time are the most costly to modify.

Software debugging environments are considered the gold standard of debugging. It is at all times possible to inspect any variable and stop the application at any arbitrary location under a variety of conditions. Debugging conditions, such as breakpoints and any conditions that trigger them, can be changed instantaneously without recompilation. A model similar to this is desirable for FPGAs.

There are two primary obstacles to build a transparent and flexible FPGA debugging framework. First, FPGAs do not present a consistent or standardized internal architecture. While ordinary computers have a discrete, well-defined memory hierarchy where state is maintained, FPGAs may have a loosely structured memory hierarchy where the state of a single register can be physically distributed throughout the device in an almost random pattern. Furthermore, no two consecutive implementations of the same design are guaranteed to be implemented the same way in the device. Second, FPGAs have no standardized interface. Computers have a console consisting of a keyboard and monitor for interactive work, along with a operating system that presents an abstract interface to the hardware, all of which are absent from FPGAs. And even though FPGAs have an abundance of input and output pins, only the JTAG interface is standardized. JTAG's serial access model is not adequate for quickly inspecting large devices and has not gracefully scaled with increasing device sizes. To counter these obstacles, the three attributes of effective debugging as found in software development environments are described below in the context of FPGAs.

4.2.1 Improving Visibility

The flexible nature of an FPGA's architecture inhibits visibility into finished designs, as does the widespread adoption of ASIC development techniques which are not well-suited for FPGAs. The majority of these approaches rely on external tools, which are at a distinct disadvantage. Visibility can be improved through two methods. First, by shifting the debugging perspective from outside the FPGA to inside, direct access to the architecture is simplified. Second, visibility is improved if the FPGA can be accessed like a random access memory. Traditional access methods process the device's state by serially shifting it out in its entirety and then decoding it to reconstruct the original design's register structure. From within, the FPGA's memory structure is randomly accessible in its entirety much like

a conventional memory. This change in perspective is significantly faster since the entire structure is readily accessible. Figure 4.4 shows this comparison between the serial model and a random access model.

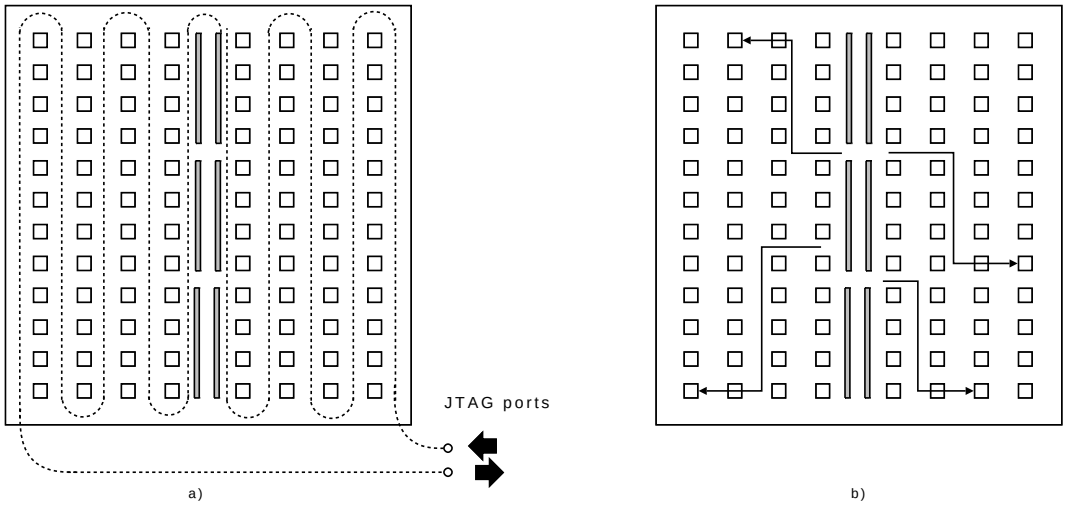

Figure 4.4: a) A serial model of accessing register state; b) a random access model in which registers are accessible as memory locations.

The random distribution of all the design's bits requires additional steps in constructing a mechanism to resolve them. In software, a *symbol table* resolves an identifier to a memory location. In debugging environments, the symbol table is expanded to include the symbol's name as it appears in the source code along with additional information such as size or data type. A similar symbol table must be built for the purposes of resolving each register's bit location so that the original register can be accurately reconstructed within the debugger.

The construction of a symbol table for an FPGA requires additional effort compared to symbol tables for software applications. Unlike a conventional computer where the details of the underlying hardware are not proprietary and standardized through interface conventions such as Hardware Abstraction Layers (HAL) and operating system Application Programming Interfaces (API), each device family is structured differently and each design turn produces a different implementation. A combined model of both the FPGA's architecture and of

the design is needed. Unlike memory locations in software applications, an FPGA register does not have a single address. Instead, individual bits are distributed across the FPGA's logic resources. Vendor tools must be manipulated in order to produce a mapping of design registers to bit locations within the FPGA's architecture. This mapping is illustrated in Figure 4.5 where the symbol table individually maps each bit of a register to a physical location within the FPGA.

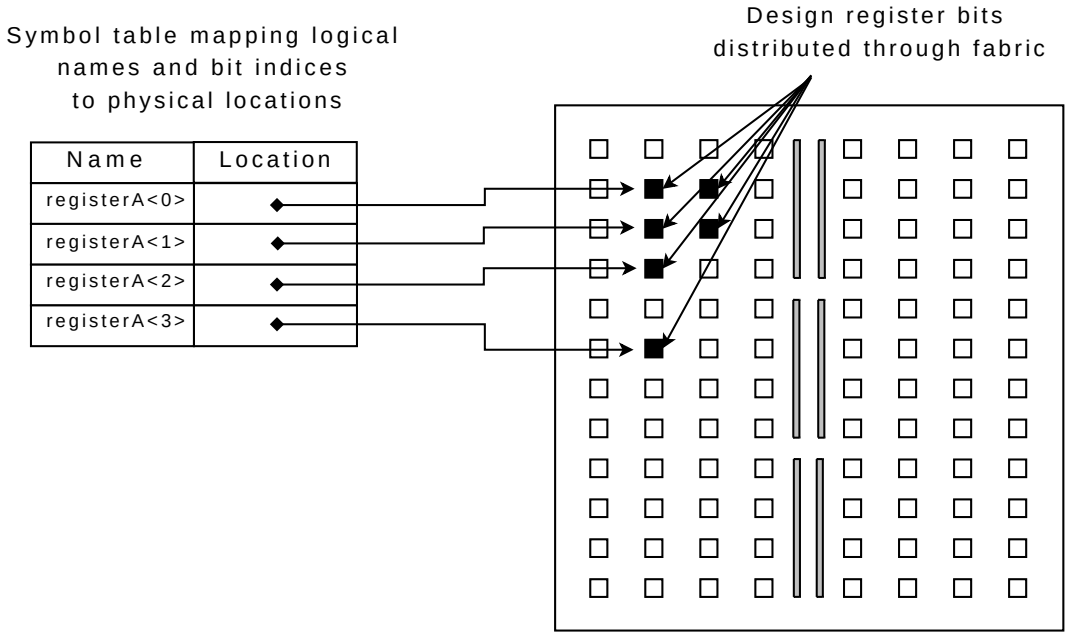

Figure 4.5: Symbol table mapping logical design elements to physical locations in an FPGA.

4.2.2 Improving Controllability

Controllability was previously defined to be the extent to which a design's state can be manipulated or altered during execution. In the context of debugging, this relates specifically to how precisely execution can be controlled. Examples of controllability include starting or stopping the execution of a design, advancing a precise number of steps, continuing execution, and defining conditions by which to suspend execution through the use of breakpoints or assertions.

In comparison to software which executes serially, hardware is a globally concurrent process: designs can be created so that many portions are simultaneously active. As such, the definition of a breakpoint solution in hardware is not defined in terms of an instruction counter as it is in software, but rather on the overall state of the design. Additionally, a hardware breakpoint mechanism must be instantaneous so that the design's state is preserved for inspection. If not, the overall state may change so significantly in a single cycle as to make any diagnosis or inspection impossible.

The most productive hardware design evaluation occurs when execution occurs at the target frequency. Some debugging frameworks allow the state to be captured at each clock cycle, however execution must be slowed considerably for this to occur. The primary concern with this approach is that timing issues can be masked since they will only occur once the design is executing at full-speed. Even a modest 100 MHz clock rate is sufficient to cause timing problems given the complex routing architectures of FPGAs. Other issues arise when certain resources require a minimum frequency for reliable operation, such as video processing components or digital clock managers. A successful control mechanism should not only allow execution to run at the design's target frequency, but be able to halt and hold the design in its current state. The maximum response time to switch from full-speed operation to a halted state must be less than a single clock cycle. Additionally, the design must be reliably held in a steady state and be able to resume execution without introducing unreliable or noisy transitions which can corrupt logic state, known as *glitches*.

Clock buffers allow clocks to be reliably controlled without introducing such glitches. Attempts to manage clock signals with ordinary logic results in *gated clocks*. Gated clocks occur when clock signals are handled as ordinary logic. Gating clocks is permissible in ASIC design, but is normally an undesirable condition in FPGAs. Unpredictably latency and loss of clock signal integrity are two risks, since the FPGA's normal signal routing architecture

Table 4.1: Truth table for a simple clock buffer.

Inputs		**Output**
In	**ENB**	**Out**
X	0	0
I	1	I

is not intended for clock signals which require special handling. Gated clocks are commonly detected by FPGA vendor tools and warnings are issued. Clock buffers can have several control inputs, clock inputs, and a clock output whose source is determined by the control lines. The most common uses of clock buffers are to reliably distribute clock signals to a large number of components (also referred to as *high fan-out*,) allow for glitch-free transition between different clock sources (for example, transitioning from a high-speed to a low-speed clock to reduce power consumption), or transition between an active clock and a constant signal which safely halts the connected components. The schematic symbol for a simple clock buffer is shown in Figure 4.6. The truth table for a clock buffer is given in Table 4.1 and shows that while enabled, the buffer passes the input clock directly to the output port, but when disabled holds the output line at a constant logic level. Clock buffers are not simply logic gates reserved for clock signals, but rather a specialized component for reliably distributing clock signals and are frequently capable of being configured to the application. For instance, clock buffers can be configured to pass logic high or low when enabled. Additionally, clock buffers have logic that coordinates the transition of the signal based on its inputs.

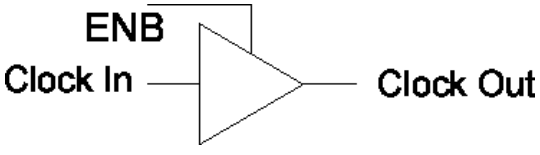

Figure 4.6: Schematic representation of a simple clock buffer.

Clock buffers are central to an effective control strategy. The most frequently used operations that define controllability center around execution. Operations such as running, stopping, and stepping are transferable to a hardware platform through the use of a clock buffer that sources the master clock of the design. With a clock buffer ensuring glitch-free transitions, execution control becomes a matter of developing a means of integrating the controlling factors, such as interactivity and breakpoint management.

Breakpoints are the final element to controllability. Software breakpoints are easy to write, typically a simple command followed by a location and optionally a condition. Their usability also stems from their conditional syntax being in the native language. Since source code location is not applicable to concurrent hardware designs, asynchronous, conditional breakpoints become the basis for a hardware-based debugger. While synchronous logic limits the complexity that can be implemented, asynchronous logic can exactly satisfy Boolean conditions of its inputs and outputs and more importantly immediately raise a signal without requiring an additional clock cycle as is the case with synchronous logic.

4.2.3 Improving Agility

The final attribute for productive FPGA debugging is agility. Agility is defined to be the ease and efficiency at which modifications can be made to a debugging configuration. In software debugging, breakpoint modification is effortless. With graphical user interfaces, the breakpoint and any associated condition is annotated in the source code or listed in a separate window. Even with command-line based debuggers such as the GNU `gdb` debugger, breakpoints can be quickly specified with a single command and a line number, file name, or function name. FPGA logic analyzers are the closest to this model of use, but require more effort to setup and much longer to modify. Modifications to embedded logic analyzers are typically treated like any design modification and may require a full design

re-implementation. Given that the general toolflow is time consuming and increases with design complexity, it is desirable to isolate any debugging logic from the rest of the design.

By physically separating debugging logic from the rest of the design, the debugging module can be re-implemented multiple times independently of the rest of the design. The debugging module's small size relative to the rest of the design would significantly reduce implementation time. Such a scenario aligns the FPGA debugging environment experience with that of a software debugging environment where relatively short times are required to alter the debugging configuration. To achieve this rapid turnaround and design isolation, partial reconfiguration can be leveraged which allows a region of the FPGA to be reserved for multiple modules that can be swapped in and out on demand. A smaller, isolated region also reduces the implementation time required. This rapid modification is ideal for interactive scenarios, such as breakpoint logic implemented in hardware. Figure 4.7 shows a separate reconfigurable region dedicated to implementing debugging logic. The distinct module allows rapid modifications of the debugging logic without re-implementing the entire design.

4.3 Dynamic Modular Design and Validation

The Dynamic Modular Design (DMD) framework envisions Xilinx's PR [67] flow not as a runtime strategy, but a design time methodology. While developers have not widely adopted PR for production designs, the research conducted for DMD has shown that rapid development turnaround times can be achieved by partitioning frequently modified modules into separate PR regions [68]. DMD's use of PR does not extend beyond the development environment since PR regions are gradually and automatically merged out of the design. The PATIS floorplanner will be discussed briefly as it relates to the overall DMD flow, however an extensive discussion can be found in [20, 37, 38, 68, 69]. There are two principal components

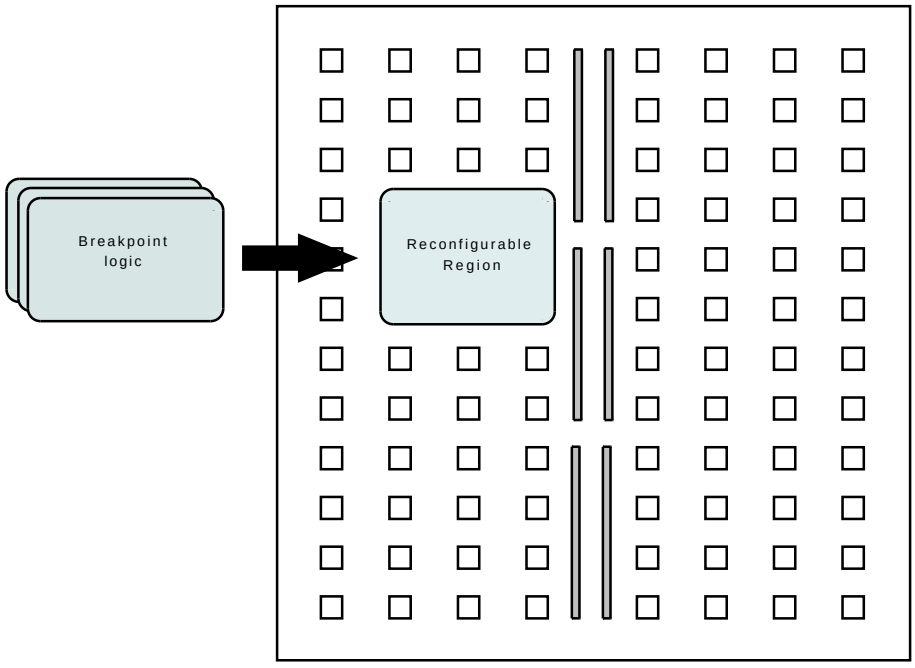

Figure 4.7: Accelerating debugging turnaround with a reconfigurable region for debugging logic.

of DMD: the PATIS floorplanner which is discussed below and the debug validation tools which are the subject of this research.

4.3.1 PATIS

DMD extends the traditional PR flow with the Partial module-producing, Automatic, Timing-aware, Incremental, Speculative (PATIS) floorplanner [20]. While the standard PR flow has traditionally been a manual process not normally associated with enhanced productivity, the PATIS tools simplify much of the implementation details of PR by automating the process. Since runtime reconfiguration capabilities are not required, PATIS uses a simplified version of the PR flow that does not have the complications found in runtime hot-swapped modules. Bus macros are automatically inserted on module boundaries and provide passive, readback-based observability of all communication between floorplanned modules [37].

The primary obstacle to fast implementation times for FPGAs is place-and-route (PAR). While vendor tools implementing parallel PAR algorithms and leveraging the lower-cost of multi-core processors have been introduced, they have yielded only a modest speedup due to the complexity of the algorithms and the large, complex data structures which saturate memory bandwidth and overrun caches. PATIS counters these obstacles through a "divide-and-conquer" strategy that creates independent floorplan variants of the design that are parallelizable on separate machines and do not require shared memory access. By separating modules that are currently under development into independent PR regions, modules can be separately implemented without reimplementing the entire design, dramatically reducing implementation times.

PATIS defines two attributes to describe maturity, *fickleness* and *viscosity*, and uses these to determine when to migrate modules from the reconfigurable regions to the static area. Fickleness is characterized by the effort required for a module to meet timing requirements, while viscosity is defined by the frequency of historical changes to the module which may provide indicators of the likelihood that it will change again.

The goal of PATIS is to recast partial reconfiguration not as runtime strategy, but as a design-time methodology for static designs. Separate reconfigurable regions for each module, as shown in Figure 4.8, enable a unique feature of PATIS: the ability to swap a previous revision of a module out for a more recent one without the need for rebuilding the entire design. This is similar to linking in software where modified files are recompiled and inserted into library archives and then linked to executables without recompiling the entire application. The time-savings can be significant.

The PATIS floorplanner creates a modestly oversized PR region for each top-level module allowing modules to grow during the course of development without disturbing the rest of the design. If an updated version of a module no longer fits within its boundaries, PATIS selects

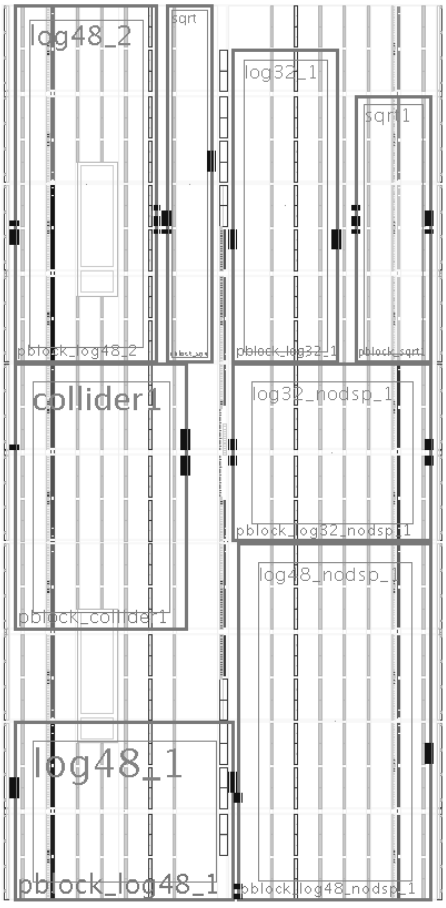

Figure 4.8: PATIS provides each top-level module its own reconfigurable region.

an appropriate floorplan from a database or re-implements the entire design. A speculative floorplanning background process runs that explores the design space, generating potential future floorplans based on estimated module completeness and past changes. Timing is analyzed across module interfaces and compared to top-level constraints. A thorough discussion of PATIS can be found in [68].

Figure 4.9 illustrates the DMD flow. Whenever a module change affects the design floorplan, PATIS tries to accommodate the changes by applying a minimum set of updates to the existing floorplan. Ripple effects are considered, and a completely new floorplan may be generated if incremental changes are inadequate. The speculative floorplanning background

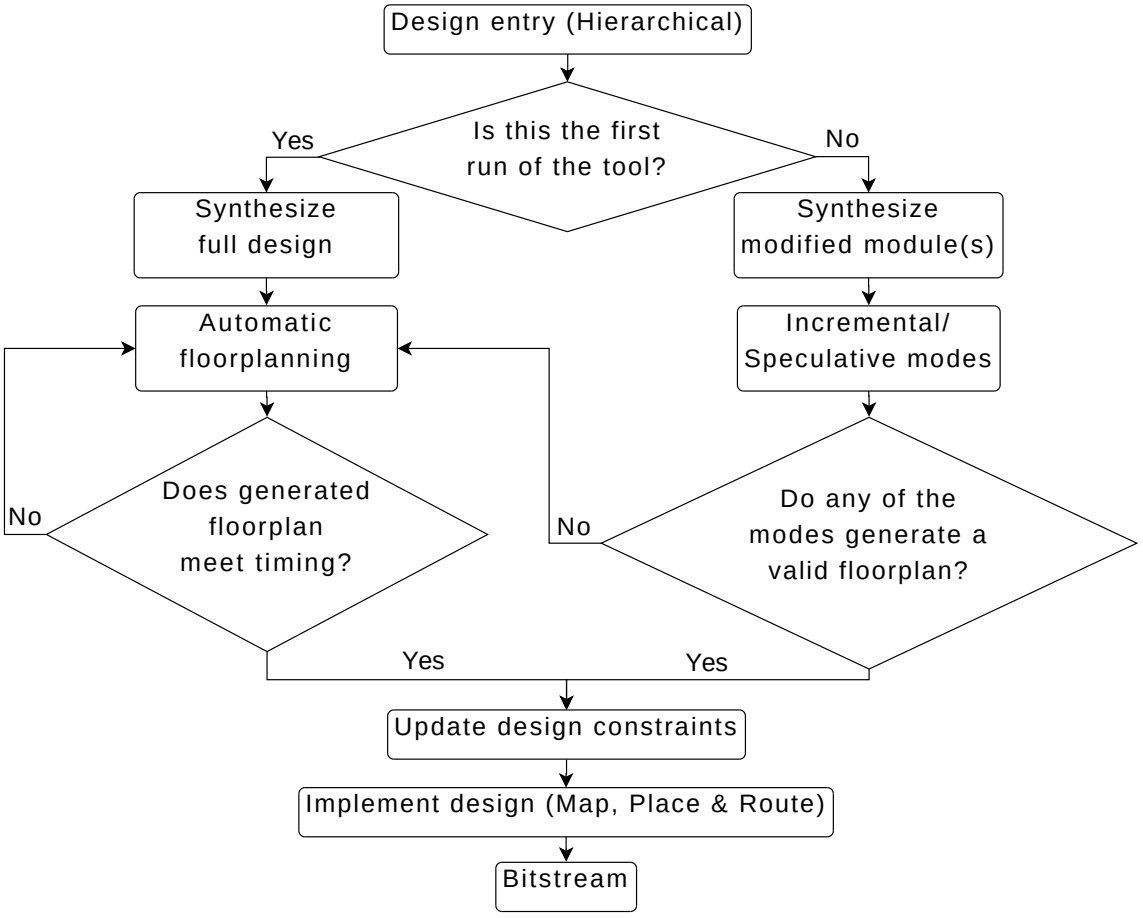

Figure 4.9: Dynamic Modular Design flow.

process populates the database with design space explorations of potential floorplans based on past changes. Floorplans include an optimized placement of bus macros. Timing is analyzed across module interfaces and compared to the top-level constraints.

Implementation acceleration comes from the reduction of a large global optimization problem to a set of smaller, independent problems. Optimization restrictions across module boundaries are generally accepted for the sake of design productivity and timing closure. Although the PATIS/PR flow adds bus macro overheads to inter-module routing, the use of registered bus macros may improve system clock frequency when critical nets span module boundaries.

4.4 Summary

This chapter explored some of the key concepts necessary for developing a validating and debugging framework. HLV seeks to tie the reference model created during the first stages of development to a functioning hardware implementation. This not only functionally validates the hardware, but additionally allows increases to testing productivity since the hardware is now under software control. HLV can assist even during later stages of development by serving as unit testing framework for individual hardware components. At the other end of the development process, LLD addresses the three attributes of low-level FPGA productivity: controllability, visibility, and agility. Methods for establishing fine-grain control over a design, repositioning the debug perspective to allow uninhibited access to the internal structures, and a means for mirroring the breakpoint model found in software were discussed. Finally, the DMD project was summarized including an overview of the PATIS floorplanner which enables the debugging framework to coexist in the rapid design turnaround environment created through partial reconfiguration.

Chapter 5

Implementation

Chapter 4 presented two approaches to improving FPGA developer productivity at different phases during the development cycle. The HLV framework seeks to bridge the model verification gap by providing a means of directly linking the initial high-level reference model to implemented hardware. The LLD framework is applied during the later stages of development by applying fine-grain control over the execution and providing detailed insight into an implemented design's state. Both these frameworks derive their use model from the debugging environments long prevalent in software. Chapter 5 presents the implementation details of these two frameworks.

5.1 High-Level Validation

HLV addresses the model verification gap wherein the reference model produced during the initial development stages becomes isolated from hardware development. HLV provides automated functional validation of synthesized hardware by binding it directly to the original, untimed reference model. The resulting framework can be applied to hardware much like

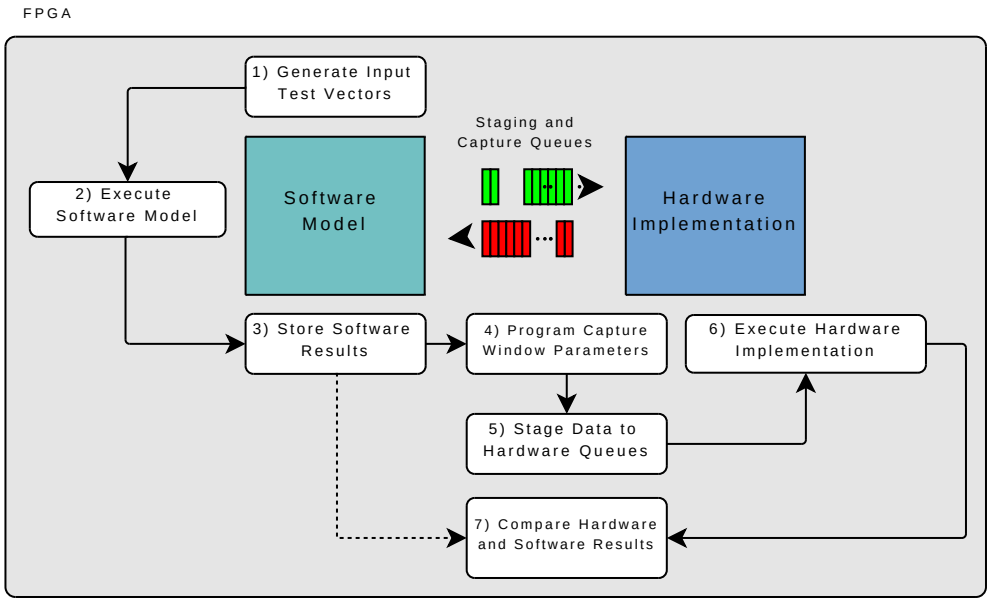

Figure 5.1: Overview of High-Level Validation.

unit testing frameworks are in software for frequent testing, such as nightly or regression testing. HLV is a hardware/software framework based on capture methodologies similar to those found in embedded logic analyzers. Since HLV is targeted towards automated tests, the resource overhead of capture techniques is considered acceptable. Figure 5.1 outlines the HLV framework which consists of an on-chip processor with a peripheral that functions as a test harness for the module. The peripheral provides input and output ports for data and control signals, queues for staging input data as well as capturing output data, and the logic for controlling the capture. The microprocessor provides a platform for the software-based testbenches, a user-interface over a serial console, as well as a platform for executing the reference model. Input data is distributed to both the software model and the hardware implementation, executed on both, and the results compared for consistency.

HLV can be used as a nightly, software-controlled, hardware unit-testing framework. Simulation data provides the model with control signal interactions and latencies to develop the software testbenches. As discussed in Section 4.1.2, control signals found in hardware

cannot be mapped to the corresponding software model. The HLV testbench software can provide software control of designs during development and debug, scanning the output for expected data produced by the software model, and reviewing output streams without re-implementing the design as with conventional hardware debugging. Software-controlled testbenching has rapid turnaround times and requires no hardware re-implementation to inspect different parts of the design, output streams, or to test different scenarios. Elaborate, software-generated unit tests can be created for each module, enabling hardware validation against the reference model. The cost for changing a testbench is no longer the time required for hardware re-implementation, but rather the time required to recompile and reintegrate the software into the hardware bitstream.

5.1.1 Reference Model Execution and Hardware Data Staging

HLV operates by first executing the software reference model on the microprocessor using prepared or randomly generated input data and the storing the results for that data set. Next, the same input data is staged in the input queues of the test harness. Control signal interaction and latencies gathered during simulation are programmed using HLV's API. The same API is then used to configure the *capture window* which defines the clock cycle range of the output data to store for comparison to the reference model. Output data that lies outside the capture window range is discarded. The use of queues for input and output data enable the design to run at its target operating frequency without having to develop a technique to control the execution of the design.

HLV uses a processor to allow users to enable programs to interact directly with the hardware to be tested. The processor provides a platform for the HLV driver software, the software testbench, and enables communication over a serial console. The Xilinx MicroBlaze processor was chosen for its small size, large collection of peripherals, and ease of customization. The

MicroBlaze is a configurable soft-processor especially designed for Xilinx FPGAs and includes a full C-language GNU gcc compiler. The MicroBlaze and its peripherals are specified with the Xilinx Embedded Development Kit (EDK), a graphical user interface used to build embedded systems on FPGAs. A wide range of peripherals, including memory controllers, UARTs, busses, and custom co-processors can be specified and configured. The processor project created in the EDK can then be instantiated into a higher-level project, such as into Xilinx's ISE, as an ordinary HDL component.

No invasive modifications to the reference model are necessary in order to integrate HLV into the DMD development cycle. The intent of HLV is to create a transparent means of linking the reference model to synthesized hardware without the need to significantly modify either. The testbench software applications to be run on the processor can be rapidly integrated into the bitstream through the use of special vendor tools which initialize internal BRAMs. Software compilation and reintegration with the bitstream can be done with a single command from within the EDK.

5.1.2 Test Harness and API

The test harness is a custom MicroBlaze peripheral that connects the processor to the hardware module to be validated. The test harness consists of input/output pairs of First In, First Out (FIFO) queues which connect the hardware component directly to the data path of the processor, enabling the DUT to operate at the target frequency. The test harness also provides logic which controls the capture process.

Through its Fast Simplex Link (FSL) architecture, the MicroBlaze provides direct access to its data path through special instructions and ports [24, 70]. FSL is a configurable length, 32-bit wide, point-to-point, unidirectional FIFO-based processor interface where data can be queued for reads or writes at high-speeds, allowing streaming data peripherals to directly

interact with the MicroBlaze's data path. A low-overhead interface such as FSL reduces the overhead and complexity between the data path and the DUT. This is preferable to interfacing through a bus-based peripheral which adds additional hardware and software overhead and thereby operates slower.

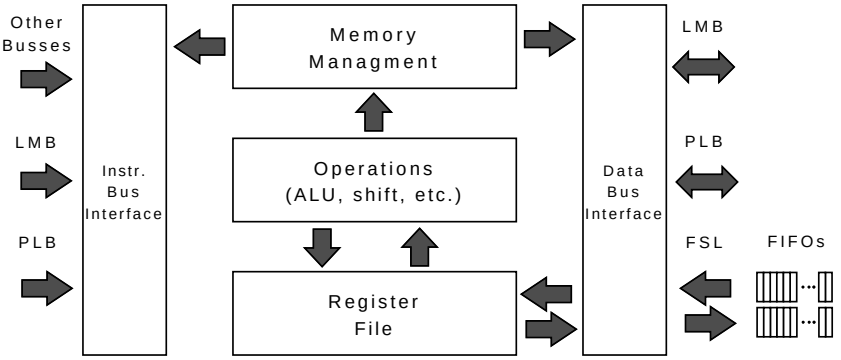

Figure 5.2: MicroBlaze architecture.

Figure 5.2 shows that unlike other busses available on the MicroBlaze, such as the PLB and LMB interfaces which connect to intermediate hardware peripherals and requires driver software, the FSL connects directly to variable length FIFOs. For this reason, FSL is the preferred choice for interfacing with custom hardware accelerators. For even higher performance, FSL can be configured to use BRAMs rather than distributed RAM implemented in ordinary configurable logic. Data can be handled deterministically, with latencies for sending or retrieving data to and from the interfaces at one to two clock cycles, depending on the configuration [24]. The MicroBlaze includes FSL instructions as part of its binary interface, further reducing overhead and latencies. Furthermore, FSL instructions are available in blocking and non-blocking variants which can halt execution when FIFOs are full or empty. These options could be used to develop complex, long-running testing scenarios.

Use of the Xilinx EDK enables peripherals to be quickly added to a MicroBlaze. Known as *co-processors*, a short series of configuration options can be set in the graphical user interface which produces a co-processor stub connected to a MicroBlaze. The stub is generated

Table 5.1: Summary of HLV API.

Function	**Description**
`write_fsl_data`	Write data to the FSL data queue
`write_fsl_ctrl`	Write control data to the FSL control queue
`write_fsl_pair`	Write data and control data to the corresponding FSL queue in unison
`hold`	Hold control and data at a constant value for the given number of clock cycles

with instructions on how to customize the HDL to produce the co-processor, as well as the framework for software drivers to access the new peripheral through memory mapped registers or streaming instructions. The test harness was created as an FSL co-processor and then customized with large-capacity FIFOs to store long streams of data and control signal information for both input and output.

The HLV API enables the FSL queues to be populated with data and control streams identical to those gathered from simulation and includes utility functions to hold individual or groups of signals at constant values or transition after pre-defined intervals. A summary of the API calls along with a brief description of their function is included in Table 5.1. The API can be used to build custom signal interactions, such as reset sequences necessary to set the hardware to a known initial state, before streaming input data from the queues. Once populated with the control and data signals, the API is used to release the signal streams to the DUT.

In addition to queuing data and control streams for the DUT, the test harness also implements its capture logic in hardware for the best performance. As noted in Chapter 3, capture methodologies are common in embedded logic analyzers which are instantiated as part of the design and internally record execution traces. A capture methodology is more appropriate in a test environment where individual components will be individually tested rather than

as part of the entire design. The resource requirements for capture, most notably on-chip memory, will not compete with the design for resources.

As part of the test configuration, the API is used to program the capture window logic which defines the range of the output stream to store in the test harness' queues. Until execution is triggered, a counter is held at zero until the control register's start bit is activated. Once set, the data stored in the input queues is released and begins streaming to the DUT. As the counter increments, the capture window is activated once the counter is between the range of the start and stop registers. Only during that window is the module's output data stored into the test harness queues and ceases being captured once the counter exceeds the stop register value. The capture window is illustrated in Figure 5.3. The test harness can be monitored through the API by a status register, while a control register allows the test harness to manipulated. Table 5.2 shows the memory map of the HLV peripheral's configuration registers.

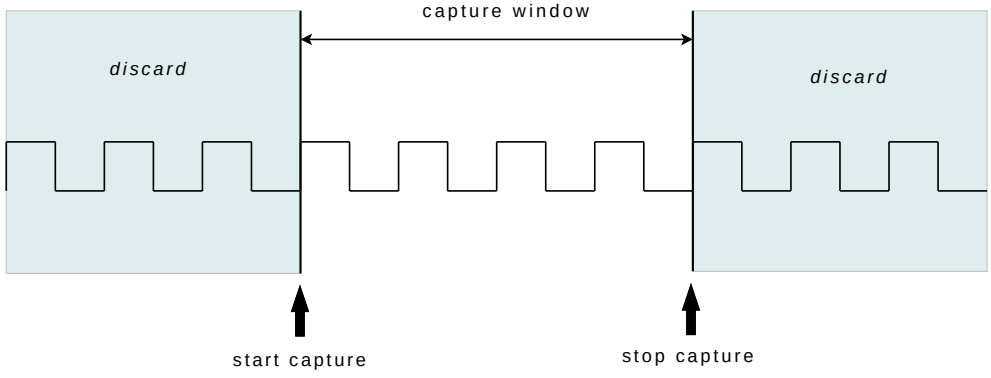

Figure 5.3: HLV capture window parameters.

HLV is unique in that execution occurs at the frequency of the design and does not require that execution be halted or occur at a slower frequency. The DUT is continuously sourced from the system clock. Since HLV can stage data that is to be streamed at full-speed, the DUT can continuously execute even without valid data at its inputs. Placing control sequences, such as reset sequences, ahead of test data in the queues allows the the DUT

Table 5.2: Register map for the HLV peripheral.

Register	**Name**	**Description**
0	Control register	Start, stop, and reset functionality for test harness and queues
1	Start capture	Capture window start boundary
2	Stop capture	Capture window stop boundary
3	Status register	Test harness and queues status register
4	Unused	
5	Unused	
6	Unused	
7	Unused	

to be placed into a known initial state and executed as if run in a full execution life-cycle. Operational data extracted from simulation, such as the end-to-end latency of any pipelines, is necessary to ensure that programming of the queues is done properly. This is particularly important for properly resetting the device before execution.

The simplified example program given in Listing 5.1 shows how a software function that has been mapped to a hardware module is executed and its data stored. The same data is then paired with control signals extracted from simulation and placed into the data and control queues. The start and stop registers defining the capture window are then set before the hardware module is set to execute. The output data is then captured and automatically compared to the software module's results.

5.2 Low-Level Debug

The LLD framework aims to introduce the interactivity and abstraction found in software debug environments to FPGAs. Similar to HLV, LLD consists of an on-chip processor controlling a custom peripheral which interfaces to the design. The processor controls the

```
// execute reference model and store results
reference_model(&result[0]);

// stage data
reset();
write_fsl_pair(0x00000012, 0x0);
feed_message1();
hold(0x2, 0x0, 12);

// configure
HLV_DRIVER_mWriteReg1((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, windowStart);
HLV_DRIVER_mWriteReg2((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, windowStart+windowLength);

// go
HLV_DRIVER_mWriteReg0((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, 1);

// retrieve and store hardware results
for (i = 0; i < windowLength; i++) {
        // read output data
        read_from_fsl(output_data[i], XPAR_FSL_HLV_FSL_DATA_OUTPUT_SLOT_ID);

        // read output control
        read_from_fsl(output_ctrl, XPAR_FSL_HLV_FSL_CTRL_OUTPUT_SLOT_ID);
}
```

Listing 5.1: Sample HLV program.

execution of the design and provides an interactive, scriptable command-line interface that interacts with a second, workstation-based application. Whereas embedded software running on an FPGA would be too slow and resource-limited to meaningfully interact with a running design, LLD uses the on-chip processor as a resident delegate to provide more visibility and insight from within the FPGA to the workstation application. The custom hardware peripheral provides rapidly interchangeable condition-based breakpoint and assertion capabilities through the use of partial reconfiguration. The breakpoint logic is capable of halting and holding the design at any arbitrary location for inspection. A state readback utility implementing the ICAP interface is capable of reading any design register in the device, treating it as a memory rather than requiring a shift chain. An overview of LLD is given in Figure 5.4.

Software models of both the FPGA device and the implemented design provide the information necessary to interact with and interrogate the FPGA directly. However, the resource and processing requirements of such models far exceed the available resources of any current

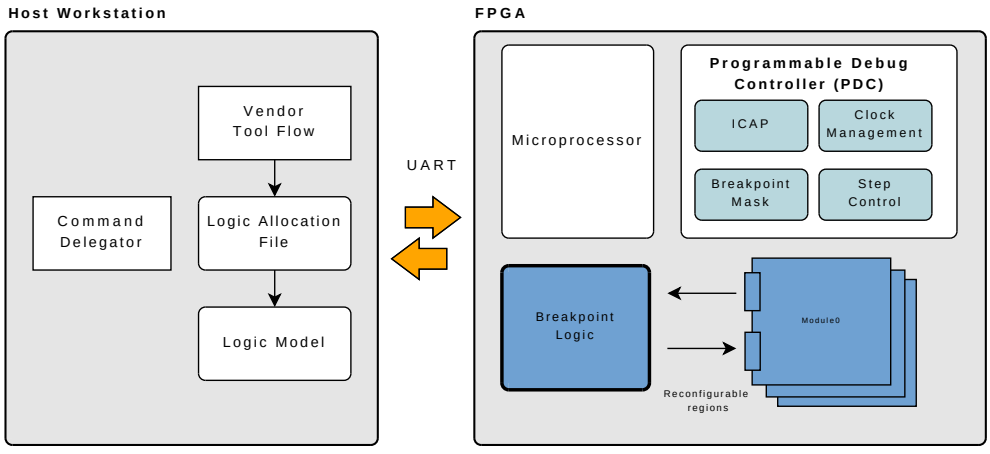

Figure 5.4: Overview of LLD.

FPGA. Therefore, a workstation application is needed to construct the model of the design and map hierarchical design names to individual bit locations in the FPGA. The application then assembles and issues a stream of commands to the on-board processor to retrieve and decode state information through the ICAP. The individual components of LLD are discussed below.

5.2.1 Programmable Debug Controller

The Programmable Debug Controller (PDC) is the interface between the processor and the design to be debugged. The PDC is a custom peripheral of the MicroBlaze that manages clock logic, status signals from the reconfigurable breakpoint module, and the ICAP to read state information. The Xilinx ISE EDK has facilities to customize and generate stubs for processor peripherals. Peripherals can be connected directly to the data path through the FSL interface or through a conventional bus such as PLB. The EDK automates the generation of interfaces, drivers, and memory-mapped registers. An API was developed to drive this peripheral using these stubs. Its register map is shown in Table 5.3 while a discussion of the subcomponents follows.

Table 5.3: Register map for Programmable Debug Controller.

Register Number	**Description**
0	Development and debug register
1	Development and debug register
2	Unused
3	Unused
4	Unused
5	Unused
6	Unused
7	Unused
8	Unused
9	Unused
10	Unused
11	Active breakpoint mask
12	Step counter target
13	Unused
14	Breakpoint enable mask
15	Control register

Clock Management

The PDC's clock management unit enables fine grain control of the design's execution. Clock buffers, which were introduced in Section 4.2.2, enable glitch-free transitions between clock sources including sources that are being held at a constant logic level. Using the PDC's software interface, the design can be run freely at its targeted design frequency or stepped an arbitrary number of clock cycles, also at the target frequency and duty cycle. At startup, the clock is disengaged and the design is halted until manually started from the command-line either through a `run` or `step` command. It then executes until either a breakpoint or assertion condition occurs, a user-issued command to `stop` is given, or if the programmable step counter expires. Figure 5.5 shows an overview of the LLD's clock control and control unit. Hardware-controlled clock stepping, rather than a software-generated clock signal, allows the design to be reliably stepped a predetermined number of cycles using the actual

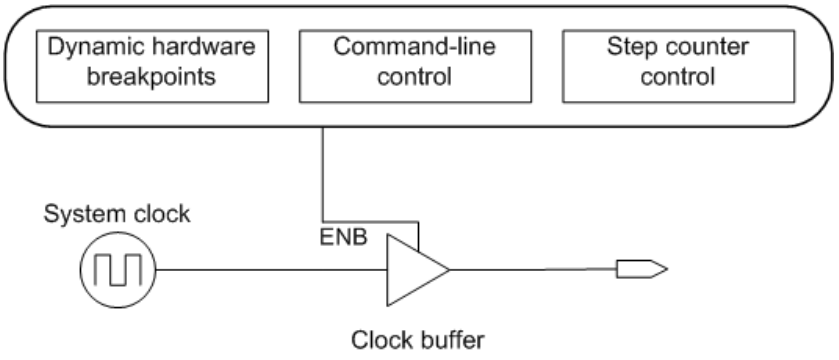

Figure 5.5: LLD clock control.

system clock which eliminates the introduction of timing and transition glitches. The clock management architecture can be extended to cross multiple clock domains by cascading clock buffers.

The PDC halts the design by enabling the clock buffer, which then holds the output clock line at a constant logic high level. Clock buffers differ from conventional buffers by taking into account rising and falling edges of their sources and deferring transition from one source to the other until the transition has completed on both sources, thereby avoiding the introduction of glitches. The clock buffers used in the PDC have a single clock source, ensuring that the reaction time is consistently one clock cycle which is sufficient to halt the design for debug purposes.

Dynamic Breakpoint and Assertion Logic

An agile and interactive breakpoint management strategy is a novel contribution of this work. Up to 32 software-addressable breakpoints sourced from top-level signals can be programmed into a dedicated, reconfigurable breakpoint region managed from the user console. Support is provided for two types of breakpoints: conventional, conditional breakpoints that suspend execution when a condition is met, and assertion-style breakpoints that suspend execution once a condition fails to be met. Assertions for simulation-based development were

introduced in Section 2.3.4. Unlike conventional assertion-based verification, the assertions found in DMD are written in conventional Verilog and respond like software assertions which interrupt execution immediately. Breakpoints can be modified and quickly re-implemented into hardware without re-implementing the entire design, achieving significant time savings especially for large or high-utilization designs. Breakpoint statements can also be selectively enabled or disabled from the command-line through a software-controlled breakpoint mask. The 32 individual bits of the breakpoint mask register each correspond to single breakpoint statement. The individual bit of the mask is logically AND'ed with the signal from each breakpoint, allowing any breakpoint statement to be temporarily disabled and then re-enabled as needed. Breakpoint logic is implemented as asynchronous logic which signals its activation immediately and causes the clock buffer to suspend design execution at the next rising clock edge. This ensures that the design is held in the precise state that caused the breakpoint to trigger. Immediate suspension of execution improves the possibility of finding the underlying cause of an assertion failure or inspecting the design in the state that caused the breakpoint to occur. A detail of the breakpoint region is shown in Figure 5.6.

Breakpoint logic is maintained in a separate, top-level, partially reconfigurable region. Implementing the debug region as a top-level module exposes the interface to the design's top-level signals and ports without necessitating changes to intermediate module interfaces. A PR region allows the breakpoint logic module to be swapped out in a matter of seconds without interrupting the execution of the design or needing to reset its state each time. Top-level signals were chosen as the most versatile means of monitoring the design since these are most likely to reflect the global state of the design.

Breakpoint tables consist of 32 asynchronous breakpoint logic conditions and reference top-level signals or ports. The dependency on top-level signals does not limit the usability of the framework since internal state is usually validated through simulation prior to implementation and the actual state of any arbitrary register can be inspected through the PDC's ICAP

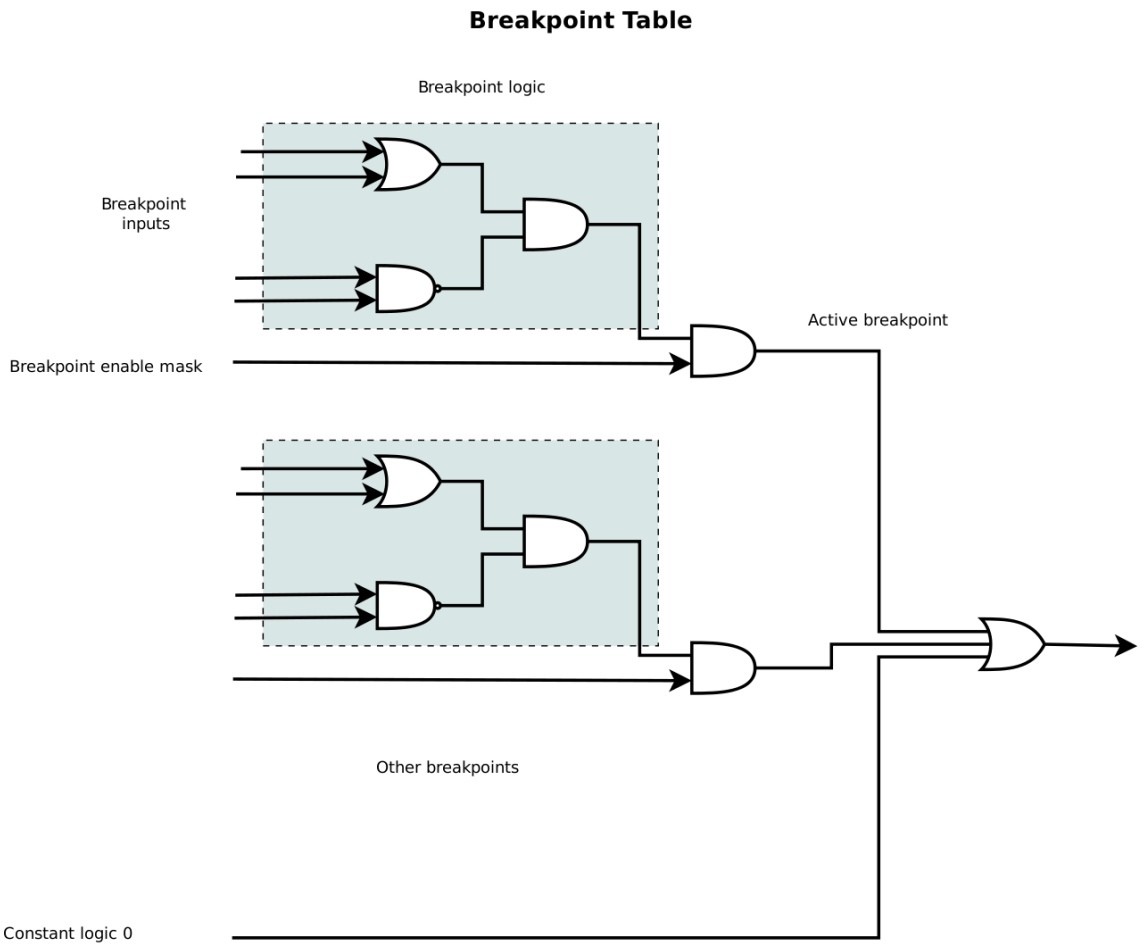

Figure 5.6: LLD breakpoint region detail.

interface once the design has been halted. An overview of the ICAP interface was given in Section 5.2.1.

Breakpoints are implemented as asynchronous logic to allow fast, single-clock cycle response times that suspend execution at the next rising clock edge. Breakpoint conditions are entered from the workstation application console as valid Verilog and can be a combination of any top-level signals or ports previously defined as the interface to the breakpoint region. If signals other than those previously selected are needed, a full re-implementation of the design is necessary as discussed in Chapter 1 to reroute signals and re-establish the

breakpoint module interface. Breakpoints unconditionally suspend execution of the design and must either be disabled through the breakpoint mask or have the activating condition changed in order for execution to resume. Software-based breakpoints were considered and would have eliminated the dedicated routing as required for this implementation. While software-based breakpoints would have created a far more agile and sophisticated breakpoint strategy, execution time would have suffered as the entire array of monitored signals and their associated conditions would need to be repolled at each clock cycle, eliminating the possibility of running the design at its target frequency.

The workstation application manages the breakpoint table that is later synthesized into the breakpoint logic module. An array of breakpoint structures, given in the Listing 5.2, is held in memory and regenerated as a Verilog module after every modification. The `breakpointText` field contains the original breakpoint statement and must have signals which correspond with the previously declared port names of the breakpoint logic module. Once entered, breakpoint conditions are written out to a Verilog module and processed with the partial bitstream tool flow.

The generated source of a sample breakpoint module is given in Listing 5.3. Empty breakpoint table entries are assigned a constant logic 0 to prevent them from interfering with the breakpoint logic even if the corresponding breakpoint mask is accidentally enabled. There are no control signals into the breakpoint module, only design signal inputs. Likewise, there are only two control outputs from the breakpoint region: an aggregate line to indicate an active breakpoint, and a mask showing the index of active breakpoints. The aggregate line is intended to signal an interrupt to the MicroBlaze when the status of any breakpoint changes to cause an interrupt routine to automatically print information about the active breakpoint. Given the small size of the breakpoint region, the resulting module is fast to implement into hardware and requires a nearly insignificant number of resources.

```
enum breakpoint_t { BREAKPOINT, ASSERTION};

class Breakpoint {
public:
    uint16_t idx;                       // Breakpoint index
    std::string breakpointText;         // Original breakpoint statement
    bool isValid;                       // Enable the deletion of breakpoints
    breakpoint_t breakpointType;        // Breakpoint type

        // Convenience function to return text of the breakpoint type
    std::string getBreakpointType() const {
        switch (breakpointType) {
        case BREAKPOINT:
            return "breakpoint";
        case ASSERTION:
            return "assertion";
        }
    }
};
```

Listing 5.2: `Breakpoint` data structure.

"Inverted" Bitstream

The increasing size and complexity of FPGAs warrants that extra precautions are taken with the implementation flows. Design rule checks (DRC) are repeatedly enforced throughout the implementation cycle ensuring that the device is not configured in a manner that would somehow physically damage the device, for instance a logic high line tied directly to ground or two different drivers on the same line. DRC is performed on the entire design, regardless of how small the change is or if even just a partial region is re-implemented. The entire design is processed for checks such as DRC, and the time required increases proportionally with the design size. For large designs on large devices, the overhead of this DRC can be significant.

To counter the lengthy runtimes that would result in regenerating a debug module, an "inverted" bitstream was devised. Inverted bitstreams refer to the replacing the logic outside the reconfigurable debug region with an empty design, while still preserving the boundaries

```
module dmd_debug_logic
(
        // design input ports
        cmd_i,
        cmd_o,
        cmd_w_i,
        text_i,
        text_o,

         // control lines
        breakpoint_active,
        breakpoint_reg
);

        input           [0 : 2]         cmd_i;
        input           [0 : 3]         cmd_o;
        input                           cmd_w_i;
        input           [0 : 31]        text_i;
        input           [0 : 31]        text_o;
        output                          breakpoint_active;
        output          [0 : 31]        breakpoint_reg;

        wire            [0 : 31]        breakpoints;

        // assign the internal breakpoint register array to the output
        assign breakpoint_reg = breakpoints;

        assign breakpoint_active = |breakpoints;

        // generated breakpoints/assertions below this line
        assign breakpoints[0] =  (text_i == 32'h62636465);  // breakpoint
        assign breakpoints[1] =  (text_o == 32'h84983E44);  // breakpoint
        assign breakpoints[2] =  (cmd_i == 3'b010);         // breakpoint
        assign breakpoints[3] =  (cmd_o == 4'h4);           // breakpoint
        assign breakpoints[4] = 1'b0;                       // empty breakpoint

        // additional breakpoints...

        assign breakpoints[31] = 1'b0;                      // empty breakpoint

endmodule
```

Listing 5.3: Generated Verilog breakpoint module for reconfigurable breakpoint region.

of the debug region. When implementing the debug region, overhead such as that with DRC is almost entirely eliminated.

Stepping Logic

Clock stepping is an another function of the PDC and utilizes a dedicated step counter and breakpoint logic as a separate control to the clock buffer. The PDC can be programmed through the API to step the design an arbitrary number of clock cycles with an accuracy of one to two clock cycles. This uncertainty is based on the operation of the clock buffer which transitions the clock on the following rising clock edge after the select line has been asserted [3]. Based on the timing of the triggering event, the actual transition may not occur until the following rising edge, resulting in a two cycle latency. The stepping function

presented here is unique in that the design's actual clock is used rather than an artificial or synthesized clock, enabling stepping to be performed at the correct frequency and duty cycle. To the best of our knowledge, this is the only debug implementation that enables debug stepping at the design's target frequency using the system clock.

A memory-mapped register in the PDC maintains the target number of clock cycles to be stepped and is set through the API. The design is then set to execute by switching the PDC's clock buffer to the system clock. Once the counter expires, a suspend signal causes the clock buffer to disconnect the clock once more and hold the clock line high, allowing inspection of the design.

While a software-controlled clock would be easier to implement and enable a variety of additional functionality, it would not guarantee glitch-free operation. Treating a clock line as ordinary logic assumes the risk that the implementation tools would route clock signals through ordinary logic lines, a condition known as *gated logic*. The logic fabric and its routing are not designed to maintain clock signal integrity. Designs which utilize ordinary logic for clocking signals do so at their own risk since the tools will no longer be able to properly assess timing information.

ICAP Interface for State Readback

The ICAP is a user-accessible configuration port, allowing similar access as with JTAG to register state. However unlike JTAG, the ICAP allows random access to arbitrary locations of the device without the need to shift out the entire device state, enables internal access directly from design logic rather than an external development tool, and allows access to configuration data as well as design register state. The ICAP is most frequently used by designs that employ partial reconfiguration, whereby a reserved region is reconfigured directly from the design as the rest of the design continues normal, uninterrupted operation. Through the

ICAP, a design can reconfigure itself as part of its operation, reconfiguring a region to perform different tasks, similar to memory overlays in software and thereby raising overall utilization of a smaller, less expensive device. Otherwise, larger devices are required to accommodate all possible logic configurations which might otherwise be left unused a majority of the time [36]. The ICAP is used to increase the visibility into a design since it has a full, unobstructed view of the entire device as a random memory.

The smallest granularity of an ICAP access is the *configuration logic frame*, a vertical stack of 1312 bits (or 41 32-bit words) that configure a *column*, while a horizontal series of columns form a *row*. Each column is a collection of various FPGA resources with a `HCLK` clock tile at the center, occupying the center 32-bit word of a configuration frame. The remaining 640 bits above and below the center word configure the resources of those regions respectively [42]. A diagram of a configuration frame is given in Figure 5.7. On generations through the Virtex-II, the configuration architecture varied within the family and different devices each had different configuration frame sizes making development and characterization difficult. Beginning with the Virtex-4 family, the configuration architecture became standardized across devices and continues to be consistent across families as described [40]. While the actual purpose of each bit within the configuration frame is proprietary, Xilinx tools have the option to produce a file that maps individual design bits to bitstream locations and configuration frame bit addresses. Such a mapping would enable the construction of a symbol table for FPGAs. The generation and parsing of such a file to produce this mapping is discussed in Section 5.2.3.

While both reading and writing to the ICAP are supported, it is not possible to directly modify a design's internal state through the ICAP. While both configuration and design state can be read through the ICAP, it is only possible to write configuration data (such as bitstreams when performing partial reconfiguration) that directly affect an entire configuration frame. Initial values for registers and memories are found in the configuration data,

however these are transferred to the state bits during device initialization when the device's Global Set/Reset (GSR) signal is asserted. As the name implies, GSR is a device-global signal which causes all the bits in the configuration data frames to be shifted into the device's logic bits which function as bits of registers. While moving state in the opposite direction is a non-destructive (read-only) operation that occurs when the ICAP's CAPTURE command is asserted, a GSR operation destructively writes all state bits of the entire device, causing the device to lose its state. The MetaWire [71] project successfully reverse-engineered the process of selectively writing configuration bits to reprogram BRAMs. This allowed a Network on Chip (NoC) to be implemented using the preexisting configuration channels and did not require the significant resources needed for an actual NoC on an FPGA. However the method was highly device- and architecture-dependent and did not lend itself well to integration within a design. An illustration of the relation of configuration frame bits to device resources and the movement of data between the two is shown in Figure 5.7 [42].

5.2.2 Unified Software Interface

The debug component of DMD utilizes two tightly-coupled software platforms to provide the access and visibility of a software debugging environment to FPGAs. One is a high-level user interface on a conventional workstation that runs both the vendor and the DMD tools allowing them to easily exchange data. The workstation application raises the abstraction of common low-level debugging tasks from an architectural-specific level to a symbolic level consistent with the original design. Additionally, traditional workstations can handle resource-intensive tasks such as building a model of the FPGA, physically mapping the design to this model, and providing an intuitive user interface similar to those found in software environments. The other software platform executes on a MicroBlaze processor on the FPGA alongside the design and provides a point-of-presence to control execution and perform state

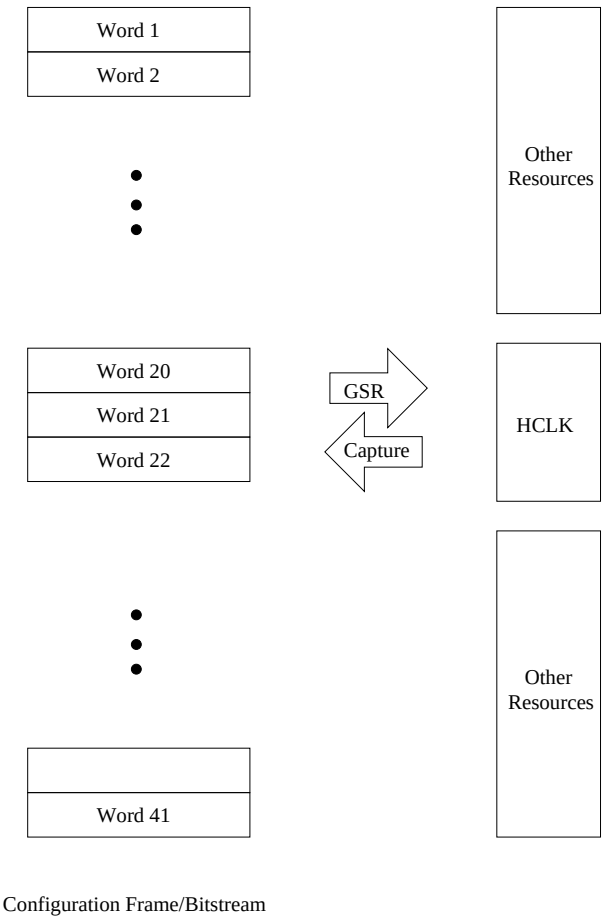

Figure 5.7: Relationship between device resources and configuration frames.

readback through the ICAP. The combined application's interface was modeled after the GNU `gdb` debugger [72] and provides an interface to FPGA design debugging familiar to software debuggers. Delegation of commands and processing is used in the combined platform to provide a single, unified interface. Commands are transparently processed either on the workstation, the FPGA-based processor, or reformatted on one in order to be processed on the other, and provide full visibility and control of a design from the single interface.

The limited resources of the onboard processor require its software to be primitive and therefore difficult, if not impossible, to manually manipulate. A compilation directive can either eliminate or include other interactive elements from the console application such as

Table 5.4: Summary of on-chip debugger commands.

Command	**Description**
`help`	Prints a help message with information on each available command
`clear`	Clears the console screen
`run`	Run the design directly connected to the system clock
`step` [cycles]	Step the design a specified number of clock cycles, one clock cycle if none are specified
`stop`	Stop the design from executing
`print` <frame><offset0>[offset1...]	Retrieve and print the value or values found at the given frame and one or more offsets.
`ping`	Respond with a message indicating that the debug software is functional.
`status`	Provide general diagnostic information about the PDC's internal registers.
`enable` <index>	Enable the breakpoint identified by the index number.
`disable` <index>	Disable the breakpoint identified by the index number.
`info mask`	Print the breakpoint mask indicating which breakpoints are enabled.
`info active`	Print the active breakpoint mask to show which breakpoints are currently active.

command-prompts, user-friendly messages and formatting, and other non-essential elements that improve the usability. These changes turn the FPGA-based processor into a debug server that the workstation application communicates with using a special protocol. Commands are encoded, replacing design names with frame and offset locations or translating intuitive command names to the local processor's more abbreviated versions. The user workstation also reformats the FPGA-based processor's responses to a more more readable format to improve usability. Table 5.4 gives a summary of the commands available on the embedded processor.

The two platforms are linked using a serial line. Communication between FPGAs and the outside world is a challenging tasks. FPGAs are targeted towards low-level, high-speed applications and can have hundreds of high-speed pins. High-level, interface connectors such as those for keyboards and monitors are available but are not standard on most development platforms. To establish a reliable, portable communication medium, a conventional serial-console was implemented. Serial links have been used reliably in computers for decades and although they have been eclipsed many times by faster and more sophisticated communication mediums, they are still universally prevalent because of their simplicity. Although mostly invisible on modern computers and even microprocessors, serial access is possible on the smallest of devices, even such commodities such as USB memory drives. At least one serial interface is found on almost all FPGA development platforms, regardless of cost and inexpensive USB-to-serial converters are available for modern workstations where the serial connector has all but vanished.

Despite relatively slow performance (the fastest transfer rates are at 100 kbps), serial links continue to thrive because of protocol's simplicity and low-cost of implementation. Serial links do not require a complex processor, protocol, memory, or software stacks, as does USB or Ethernet. Synthesizing the required cores in reconfigurable logic for serial interfaces requires few resources and IP is readily and freely available from multiple sources. And unlike both Ethernet and USB, serial implementations are easily portable across hardware vendors. Serial interfaces were implemented and a simple communication protocol was developed to leverage these benefits so that this work was portable across development boards with minimum effort.

The workstation user application was written for the Linux platform in C++ as a command-line application. Once the application is started, a command-prompt is presented. A connection to the FPGA processor is then established and validated over the serial line. Next, the

device and logic models are built in order to provide a navigable model for the mapping of design elements to the physical resources of the FPGA. The GNU Readline [73] library was used to provide the familiar command-prompt found on most Unix/Linux applications and shells, including command-completion and command history. Table 5.5 gives a summary of the workstation application commands.

5.2.3 Logic Model and Symbol Table Creation

In order to successfully retrieve and assemble the individual bits from the configuration logic frames into register values, a symbol table is required. *Logic allocation files* are an optional output file produced during bitstream file generation and map individual bits of the named design registers to both their absolute location in a serial JTAG chain and the configuration logic frame and an offset within the frame. The logic allocation file also includes additional information, such as the physical device resource corresponding to each bit. A sample line of a logic allocation file is shown in Figure 5.8. By parsing the logic allocation file, a map can be generated defining where in the FPGA fabric each bit of a design register resides.

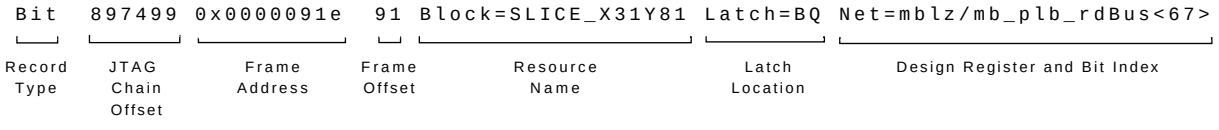

Figure 5.8: Logic allocation record format.

For a modestly-sized design, the logic allocation file can be tens of Megabytes. As there is no practical way to process such a large file on the FPGA itself, a conventional workstation is required. A parser (whose source can be found in the Appendix) was developed to read in the logic allocation file and generate a lookup table whereby hierarchical design names could be mapped to a list of configuration logic frame addresses and offsets.

Logic allocation files are space-delimited, fixed-width column ASCII text files. Two types of records are found in logic allocation files: `Info` and `Bit` records. `Info` records define special meanings for configuration bits while `Bit` records exist for each design bit and maps it a physical location in the FPGA. There are very few `Info` records in a logic allocation file, and they are unnecessary for this application. In the logic allocation file, the record type (`Bit` or `Info` identifier) is found in the first column.

The second field of a `Bit` record is the absolute location of the bit in the JTAG chain. The third and fourth fields are the frame and offset values, respectively. The frame field identifies the configuration frame in which the bit resides, while the offset defines the offset beyond the initial NULL frame where the actual bit can be found. Special care is required when determining the offset once the frame has been retrieved since a NULL frame and single word are also returned at the beginning of the frame, which must be discarded. It is assumed that this initial NULL frame represents proprietary configuration data. Therefore, space sufficient for two frames plus one additional word must be allocated in the application, and all references must be offset by that amount when accessing data read from the ICAP. Additionally, bits are inverted from their actual value and must be inverted before being interpreted as a register value.

Next, the `Block` field indicates the unique name of the FPGA resource used. Design state registers are located in Slice or Input/Output Buffer (IOB) resources. However, design registers are rarely, if ever, allocated to IOBs. The `Latch` field following the `Block` identifies which latch within the block the bit occupies. Finally, the `Net` field identifies the design's hierarchical name, its index if the register is a vector, or both the bit's index and array reference in the case of multi-dimensional vectors, such as in memories.

The construction of the symbol table only requires the use of `Bit` records and `Net` fields, easily found using a token-based parser. A `Ram` record maps out the allocation of general-purpose logic implemented as a ROM or RAM. The parser begins by discarding all comments,

```
class BitInfo {
public:
    uint32_t bitStreamReadbackLocation;
    uint32_t frameAddress;
    uint32_t frameOffset;

    int16_t signalVectorIndex;

    std::string block;
    std::string latch;
    std::string net;
};
```

Listing 5.4: `BitInfo` data structure.

delimited by a "%" and `Info` lines. `Bit` lines mapping blocks with Slice resources are read, while RAM and ROM records are discarded.

The `BitInfo` data structure, shown in Listing 5.4, represents a single bit when mapped from the logic allocation file to a location in the FPGA. Information contained in this structure includes the absolute location of the bit if shifted out of the device, the frame and offset addresses when accessed via the ICAP, as well as the index of the signal if it appears in a signal vector, such as a bus. Additional information includes the block and latch where the bit is located. Consecutive bits for vectored signals are stored sequentially in a list array in a `BitInfo` data structure; the array is then mapped to the design register name in a lookup table.

The lookup table is implemented as a map, keyed by the signal name since the full hierarchical name is globally unique in the design. This means that there is no risk of `collisions`, which occur when duplicate keys are generated. Collisions can be managed either by detecting them programmatically before inserting a new value and implementing the map's value storage container as a list or through the use of a multimap which allow duplicate key values by

design. Map complexity is logarithmic with the size of the map, denoted by $O(\log n)$. The search for keys is logarithmic as the keys are stored in a binary tree.

Once the entire logic allocation file is processed, the information is available to the console workstation application. The ample memory and disk resources of a conventional workstation make user interaction easier to enhance. The GNU Readline library is a standardized tool for creating user-friendly command-line prompts and is found in many open-source projects, including Linux shells such as `bash` and `csh`. Readline provides command-line editing capabilities, such as history and tab-completion to applications, making command-line applications more intuitive and easier to use. Using the tab-complete capabilities of Readline, the design can be explored from a command-line using the names extracted from the logic allocation file. Users can press the TAB key to list all available design names, or enter a few characters of a name and view those that match without knowing or having to memorize the design's hierarchy. The design's hierarchy can be navigated and its state inspected.

The `print` function of the user console application uses the logic model built from the logic allocation file to assemble a series of `print` commands targeted towards the embedded processor. Where possible, consecutive bits found in the same frame are combined into a single print command to accelerate processing. When entered, the register name is searched for in the register name table of the logic model, and the corresponding list of `BitInfo` structures is retrieved. The configuration frame of the first bit is read and stored, and consecutive bits are checked for the same configuration frame. Reducing the number of round-trips to the FPGA is key to reducing processing overhead, and consecutive bits in a register have a high likelihood of existing in the same or neighboring frames. As the bits are processed, the offsets are stored in a list until a non-matching frame is encountered. The set of bits existing in the same frame are sent to the embedded processor, which then retrieves

the bit values, inverting them before serially shifting them into a register and returning the result. The retrieved values are shifted into a register and then displayed once all the bits have been collected. The algorithm for retrieving register values is given in Figure 5.9.

The list of `BitInfo` structures is sorted on its first access which prevents lists that are never accessed from being unnecessarily sorted. The sort algorithm is implemented as a bubble sort with complexity $O(n \log n)$. Bubble sorts are the simplest to implement, but are not the most efficient. In a bubble sort, two adjacent elements are compared and swapped if they are out of order before advancing to the next pair of elements. The list must be iteratively traversed until all the elements are in order which is when the list can be traversed without performing any swaps. Bubble sorts are not as time efficient as other sort algorithms, but are the most space efficient. Bubble sort requires only $n + 1$ memory locations for sorting a list of length n since only one additional location is needed for temporary storage during the swap. Bubble sort is sufficient for this application since the longest list to be sorted will be of length 32.

The calling program passes the name of the target register to the procedure which first initializes variables for the resultant register value and intermediate fragments, variables for determining when the configuration frame changes, and lists for the locations of bits and offsets (lines 2–7). A list of bit locations is returned from the table which maps signal names to physical locations as seen on line 8. To prime the first round of processing, the first location in the returned list of bit locations is decomposed to a physical frame address and offset. This frame offset is the first offset to be appended to a list of frame offsets (lines 9–11). Consecutive bits have a high likelihood of being physically located close to one another, therefore a simple optimization is to group all the bits located in the same frame into a single frame access. A variable storing the previous frame is written with the current frame's address enabling the loop to distinguish when the frame address changes (line 12).

```
1:  input  : A text string identifying a SignalName
2:  output: The value of the register denoted by SignalName
3:  registerValue = 0;
4:  previousFrame = 0;
5:  currentFrame = 0;
6:  frameOffsetList = [ ];
7:  bitLocations = [ ];
8:  registerFragment = 0;
    /* Retrieve the mapping structures for the given signal                    */
9:  bitLocations = lookupMappingInfo(SignalName);
    /* The first bit must be handled outside the loop since successive frames will be
       compared                                                                */
10: bit = pop(bitLocations);
11: {frameAddress, frameOffset} = getFrameInfo(bit);
12: append (frameOffsetList, frameOffset);
13: previousFrame = currentFrame;
    /* Handle the remaining bits in order                                      */
14: for bit ∈ bitLocations do
15:     {frameAddress, frameOffset} = getFrameInfo(bit);
        /* if the frame changes, send the command to FPGA software             */
16:     if currentFrame ≠ previousFrame then
17:         registerFragment = getFrame(previousFrame, frameOffsetList);
18:         registerValue = (registerValue ≪ size(frameList)) ∨ registerFragment;
19:         clear (frameList);
20:     append (frameList, frameOffset);
21:     previousFrame = currentFrame;
22: registerFragment = getFrame (previousFrame, frameOffsetList);
23: registerValue = (registerValue ≪ size(frameList)) ∨ registerFragment;
24: return registerValue;
```

Figure 5.9: Algorithm to retrieve register values.

The remaining bits are processed in a loop which first checks if the current frame is different from the previous (line 15). If this is the case, the frame address and the list of the frame offsets is sent to the processor on the FPGA which replies with a fragment of the requested values (line 16). The returned fragment is OR'ed with the running result which must first be shifted left the width of the fragment size (line 17). If the current frame is the same as the previous frame, the offset is appended to the list of offsets and the loop is repeated (lines 19–20). Once all the bit locations have been processed, one final iteration is required to

process the remaining bits into the result register which is returned to the calling application (lines 21–23).

When printing values, the FPGA-resident processor first checks that the design is not running. A running design will cause inaccurate values to be printed. Then, a capture command is issued to the ICAP to transfer register values to the configuration frame bit locations. The selected configuration frame is read from the ICAP, first discarding the NULL frame and NULL word which precede the requested frame. The integral part of the offset divided by 32 gives the index of the target word, while the offset modulo 32 gives the correct bit of the target word. These equations are presented in Equations 5.1 and 5.2.

$$WordIndex = int(\frac{offset}{32}) + NullFrameOffset \tag{5.1}$$

$$BitIndex = offset \mod 32 \tag{5.2}$$

5.3 Summary

This chapter presented the implementation of two development and debug methodologies. Both employed an embedded processor and a custom peripheral to provide a general-purpose interface to the module or design to be tested. HLV's implementation requires considerable memory resources as it is a capture methodology, first executing the original high-level reference model, storing the results, then staging the same input data for the hardware execution. HLV's peripheral contains FIFOs for both input and output. The input data is first stored in the input peripherals, then released to the inputs of the hardware device which is continuously connected to the system clock. This implementation is unique in that the hardware implementation is compared to a high-level reference model without significantly

altering either, and the comparison is done at the target frequency. The processor serves as a platform for software-based hardware testbenches.

LLD likewise uses a microprocessor and the PDC as its custom peripheral, but for different purposes. In this case, the processor is used as a debug server on the FPGA and as a point-of-presence in what would ordinarily be an inaccessible location. A workstation application is then the client, constructing series of commands that are sent to the FPGA processor and reformatting the results. The workstation application is needed for hosting the large models of the FPGA and the design which cannot be processed on the FPGA-side processor. The two applications present a unified interface, delegating commands to the appropriate platform as needed. The PDC hosts clock control logic for error-free transitions to stop the design, controlled by a rapidly reconfigurable breakpoint module, a user-programmable step counter, or the command console. Additionally, the PDC also hosts the logic required for reading all the design's registers, although this functionality is largely controlled from both the software interfaces.

Table 5.5: Summary of workstation commands.

Command	**Description**
`run`	Begin execution of the design when stopped
`stop`	Halt execution of the design when running
`step` [steps=1]	Step the design *steps* clock cycles (default one)
`source` <filename>	Read the commands listed in the file *filename* and execute them as if they were entered at the command prompt
`connect` <port>	Connect to the serial device given in *port*
`disconnect`	Disconnect from the serial device
`continue`	Continue execution
`constrain` <condition>	Add the constraint/assertion given by *condition*
`break` <condition>	Add the breakpoint given in *condition*
`delete` <index>	Delete the breakpoint given by *index* from the breakpoint table
`info breakpoints`	Print the active breakpoint table, the status of the breakpoint masks and enables
`disable` <index>	Disable the breakpoint given by *index* by disabling its corresponding bit in the breakpoint mask
`enable` <index>	Enable the breakpoint given by *index* by disabling its corresponding bit in the breakpoint mask
`print` <index>	Print the value of the given register
`loadlogic` <file>	Load the logic-allocation file and use its contents as the logic model for the current design
`addport` <name><size>	Add a port with the given name and size to the breakpoint region for use as an input
`delport` <name>	Delete the port as a an interface port for the breakpoint region
`clear`	Clear the console screen
`ping`	Ping the remote device
`help`	Print a help message with information about all the commands
`quit`	Disconnect the remote device and exit the application

Chapter 6

Evaluation

Chapter 5 presented the implementation of both the HLV and LLD solutions that bridge the model verification gap and increase the visibility, agility, and controllability of implemented FPGA designs. In Chapter 6, both approaches are evaluated and the results compared against similar products.

To evaluate these approaches, benchmark designs were developed and validated using the HLV and LLD tools. All designs targeted a Xilinx XC5VLX110T-1 F1136 FPGA using the Linux Xilinx ISE 12.4 design suite with the PR patch running on a 2.80 GHz Intel Core i7-930 processor with 24 GB of RAM. Both HLV and LLD employ a Xilinx MicroBlaze processor with 64 kB of on-chip BRAM and built with a 115,200 Baud serial console interface for communication.

6.1 High-Level Validation

HLV links the high-level reference model to an implemented design on an FPGA. The design is tied to the processor hosting the testbench software through a custom peripheral that links it to the DUT. The benchmark design was a SHA-1 message digest, a common cryptography core in widespread use.

6.1.1 Secure Hash Algorithm

The Secure Hash Algorithm (SHA-1) is a widely used iterated cryptographic hash algorithm used to produce a 160-bit message digest from a input message block. The input message must be padded to produce a 512-bit message block. Hashes are used to uniquely fingerprint a block of data, such as a message or application, to detect if the data has been altered or corrupted and are commonly known as *checksums*. SHA-1 operates by rotating five 32-bit word segments of the data while applying bit shifts and predefined constants to the original message. Eighty rounds of these rotations or *compressions* are repetitively applied. A hash is considered unique for each message, with the probability of finding a message to produce an identical hash computationally infeasible. While vulnerabilities have been found, SHA-1 continues to be widely implemented [74]. As a useful function in both hardware and software applications, SHA-1 cores and software libraries are widely available. Many of the operations can be performed in parallel, lending itself well to a hardware-based implementation. A block diagram of the SHA-1 algorithm is shown in Figure 6.1 [75]. Each of the five input blocks is a 32-bit word which is shifted right, with the fifth word being shifted around to the first position. While some of the inputs are not modified at all, others such as the second and fifth inputs (B and E) are shifted or transformed. The function f and constants K_n and W_n are determined by the round number. An excellent discussion of the SHA-1 algorithm can be found in [74].

6.1.2 Results

Evaluation for HLV was performed on a SHA-1 core obtained from the OpenCores archives [76], a repository of open-source hardware IP. A high-level reference model of the SHA-1 algorithm was obtained from a different source. After extracting the control signal interaction

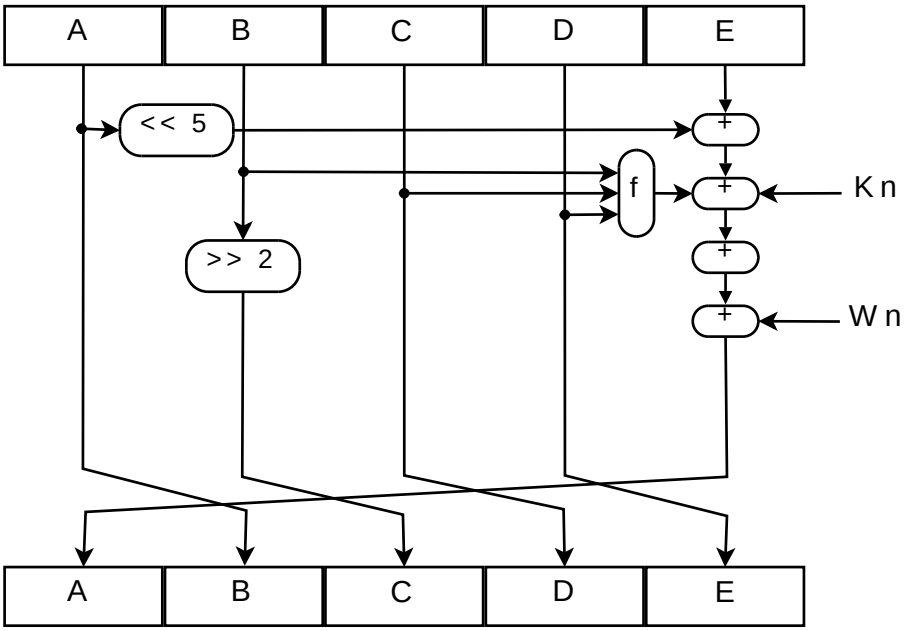

Figure 6.1: SHA-1 block diagram.

from a simulation of the hardware core, the software testbench was programmed. Programming of the testbench took approximately 20 minutes, including developing an appropriate hardware reset routine and entering the correct sequence for data and control. The simulation data provided the basis for correctly programming the framework to reproduce the correct interaction of control signals and specify the output window for results. However, once this information is incorporated into the testbench, any changes that affect latencies or final results will be detected. The captured hardware results were used to validate the hardware implementation against the simulation. It is possible to programatically expand and move the capture window's limits, enabling scanning of control or data output to detect other expected results.

The MicroBlaze processor was configured for 125 MHz operation with 64 kB of RAM for data and instruction memories, and a 115,200 Baud serial interface. The FSL input queues were configured with 256-input words while the capture queues had 32 words. The FSL queues were implemented using distributed logic to save BRAM resources for the processor or the

```
Executing software model...done.
Queuing input data for hardware model...done.
Capture window set to 175:180.
Executing hardware model...done.

Cycle    Output      Expected
175      84983E44    [84983E44]    [OK]
176      1C3BD26E    [1C3BD26E]    [OK]
177      BAAE4AA1    [BAAE4AA1]    [OK]
178      F95129E5    [F95129E5]    [OK]
179      E54670F1    [E54670F1]    [OK]
```

Listing 6.1: Transcript of an HLV test session.

design itself. The HLV testbench occupied only 14% of the Slice resources and 20% of the BRAMs. The entire application required only 13.6 kB of the total 64 kB BRAM available, allowing for larger applications. The time required to rebuild the software testbench and reintegrate with the bitstream is under 30 seconds, with compilation taking 13 seconds, and the memory integration into the bitstream taking 14 seconds.

A transcript of an HLV test session is shown in Listing 6.1 demonstrating the execution of the software reference model, the configuration of the hardware, and finally the comparison confirming the match.

First, the software model of the SHA-1 is executed on the embedded processor, and the results stored. HLV does not include any standard means of collecting the reference model results as this is left to the testbench developer depending on the requirements. Then, the data are loaded into the input queues of the DUT exactly as during simulation. Once the input data is queued, the capture window is set as a function of the number of clock cycles from the beginning of execution. The input queues are then released and a counter triggers the capture of the output once it is in the range of the capture window. The data is then compared to output of the reference model. Figure 5.1 outlines HLV's data flow and results comparison, while Figure 5.3 illustrates the capture window mechanism.

Research at the University of Florida (UF) [66, 77] investigated an approach to validating designs implemented with high-level synthesis and appears to be the only work devoted to in-circuit validation against a high-level language specification. We were unable to identify any other product—research or commercial—that compared a high-level specification directly with implemented hardware. While the UF approach relied entirely on HLS, assertions written in the native design entry language were incorporated into the application. HLS was discussed in Section 2.4. Designs implemented with HLS are typically validated in simulation and prove difficult to validate or debug once committed to hardware. This research implemented the ANSI-C assertions as asynchronous hardware logic in the design and requires a communication medium between the design and the developer platform. Assertions written into the source code are synthesized into the design as asynchronous logic that activates once the assertion becomes true.

The UF research's target objectives are much different than that of HLV. The UF project is not intended to address quick turnaround or even bridge the model verification gap as the design entry medium is already a high-level language. Rather, the UF project aims to boost developer confidence in the machine-generated RTL to catch translation errors and transparently integrate with complex designs without affecting timing closure. For instance, one concern is the overhead required to implement assertions in `for` loops. Unrolling the `for` loops would create multiple instantiations of the same assertion logic which in turn might affect timing.

6.1.3 Summary and Future Work

HLV is a unique approach to implementing software-based hardware validation. Designs were validated at target frequencies of 125 MHz without the need to control the system clock or halt the design. No other projects have been identified that bridge the model verification gap in this manner.

While not necessary for its operation, improvements could be made to HLV's software. Manually programing testbenches is laborious, requiring effort in transliterating simulation results into the HLV testbench API. Most simulators either produce an open format such as Value Change Dump (VCD) file format or have API access to the simulation engine. Automated testbench generation would greatly improve the experience and usability of HLV. Additionally, HLV lacks hardware support for reacting to design signal interactions. Instead, validation currently depends on all interactions occurring after fixed latencies. Software interaction would require additional logic to halt the application while the queues are reloaded. A more robust approach would be to enable software programmable data generators that reacted to control signals, further reducing the required capacity for the input queues, and thereby enabling more elaborate and less-scripted testbenches.

6.2 Low-Level Debug

LLD aims to raise the visibility, controllability, and agility of low-level FPGA debugging. The effectiveness of these objectives are measured through the time required to accomplish routine tasks that are normally cost- and time-prohibitive. As previously defined in Section 4.2, visibility is the ability to view arbitrary state of the design, controllability is the ability to define parameters to halt and control the design with fine-level granularity, and agility is the ease at which these changes can be made to the debugging profile. The effectiveness of LLD is evident in large or high-utilization designs where implementation times can be long, which in turn limits the number of turns-per-day. Three large benchmark designs were created to evaluate LLD's effectiveness in improving visibility, controllability, and agility.

6.2.1 Benchmark Designs

To evaluate LLD, three designs with device utilizations of more 90% of the FPGA's configurable logic resources were created. The designs were implemented to fully meet timing

(which is not the default objective of the tools), which extends the implementation times. To meet timing, PAR must iteratively place a component, attempt to route nets connected to that component, and evaluate the path based on the timing model for the device and the design's constraints. Timing analysis is complicated by the complex routing architectures and various iterative strategies such as logic duplication that consume more resources, further congest routing, but may make timing easier to achieve. The benchmark designs include the high-performance, server-class OpenSPARC, a mesh of Xilinx MicroBlaze soft-core processors interconnected by FSL links, and a network of floating-point operators generated by the FloPoCo compiler.

OpenSPARC T1 Processor

The OpenSPARC T1 processor [78] is an open-source implementation of Sun Microsystem's (now Oracle) high-performance server processor architecture based on the UltraSPARC T1 processor. The OpenSPARC is a 64-bit, 32-thread, fully pipelined processor. Unlike most soft-processors, the OpenSPARC has a virtual memory management unit capable of running a full, standard Linux distribution. Most soft-processors lack the complex logic required for virtual memory and are therefore limited to running real-time OSes which typically target small, embedded platforms. The OpenSPARC includes an embedded Xilinx MicroBlaze to handle network and other auxiliary functions. The OpenSPARC utilizes 99% of the evaluation platform's Virtex-5 LX110T's Slice resources, 17% of the DSPs, and 15% of BRAMs.

FloPoCo Floating-Point Compiler

Floating-point cores are cumbersome to implement in FPGAs due to their complexity and space requirements. Floating-point operations require multiple pipeline stages for even the

simplest operations such as addition, which takes approximately six cycles to the most complex, such as a square-root operator which can take more than 20 cycles. No generalized, standard library exists and no FPGA vendors currently implement physical floating-point cores on their devices. The FloPoCo [79] project provides a tool that creates custom, parameterizable floating-point core targeted towards a specific device, leveraging architecture-specific resources and allowing the width of the mantissa and exponent to be defined as needed. Custom operators can be tuned for the target application and device and provide control over the amount of resources consumed, rather than creating operators compliant to standards such as the IEEE 754 Floating-Point Standard, which has stringent precision requirements. The benchmark application was generated with FloPoCo 2.1.0 and occupied 98% of the Slice resources with a target frequency of 250 MHz. All operators were generated with 8-bit wide exponents and 23-bit wide mantissas.

MicroBlaze Mesh

The final benchmark application developed was a cluster of Xilinx MicroBlaze soft-core processors, such as those used throughout this research. The 10-unit FSL-interconnected cluster included one master unit with a serial UART console for user I/O and nine co-processor units. The MicroBlaze cluster consumed 92% of the target device's Slice resources and targeted an aggressive 200 MHz clock frequency.

6.2.2 Results

LLD was evaluated on the three benchmark designs by comparing the time required to implement the base design, embed Xilinx ChipScope cores (the vendor's ELA solution), and to implement the same or comparable tasks using the LLD approach. For each of the designs, real time was recorded for the implementation. Since turns-per-day are a general

metric used to measure development efficiency, real time is preferred over CPU time. CPU time is often used in conjunction with real time to determine how much time the processor was idle, presumably waiting on I/O or memory accesses. Synthesis times for subsequent builds are typically shorter since this stage is the only one in which each individual module can be independently built.

Typical debug scenarios include altering the perspective of the debug framework when different sets of signals need to be analyzed. With ELA frameworks, altering the monitored signals occurs after synthesis by inserting ELA cores into the netlist and then re-implementing the design. In contrast, LLD allows any arbitrary signal to be inspected without the need to re-implement. Furthermore, ELA frameworks embed the conditions that trigger signal capture as part of the overall design logic while LLD separates breakpoint logic from the rest of the design through the use of partial reconfiguration. While LLD does require a full re-implementation, it is limited to a small fraction of the entire design and the overhead of DRC, which takes proportionally longer with increasing design size, is eliminated by creating a separate, smaller design just for breakpoint logic. This results in a debug framework that requires a constant time to implement regardless of the overall design size or complexity.

Benchmark Designs

Base design implementation times are given in Table 6.1. For each design, timing closure was required which lengthened place-and-route times. As discussed in Section 2.2.1, PAR is responsible for routing wires through the FPGA's complex, programmable routing architecture while attempting to find routes that meet the timing constraints. All the other implementation stages were described in detail in Section 2.2.1. For the OpenSPARC, a complete build took over 3 hours, with initial synthesis accounting for approximately 55 minutes and PAR requiring nearly two hours of the implementation time. The FloPoCo design required over

Table 6.1: Benchmark design implementation times (hours:minutes:seconds).

Stage	OpenSPARC		FloPoCo		10 MicroBlaze	
	Base	ChipScope	Base	ChipScope	Base	ChipScope
Synthesis	0:56:23	-	0:05:56	-	0:02:04	-
Build	0:01:24	0:01:19	0:00:39	0:04:30	0:00:51	0:04:30
Map	0:23:57	0:23:15	1:37:00	3:17:00	0:19:00	0:18:36
PAR	1:48:43	0:30:00	8:45:00	2:56:00	8:19:00	9:36:00
Bitgen	0:05:59	0:05:07	0:02:44	0:02:58	0:03:02	0:03:14
Total	3:15:29	0:59:41	10:31:19	6:20:28	8:43:57	10:02:20

10 hours to implement, with nearly nine hours spent in place-and-route. The MicroBlaze cluster took over nine hours of runtime, with eight hours devoted to place-and-route.

Table 6.1 also shows the implementation times for inserting the Xilinx ChipScope Pro ELA tool into the benchmark designs. These times were often unpredictable, sometimes taking less time to implement than the base design itself, while at other times longer which is to be expected given the additional overhead of new resources and additional routing. Each case was run multiple times and consistently gave similar results. ChipScope insertion occurs on a synthesized netlist, therefore no time is reported for synthesis, while the rest of the flow remains unchanged. In the OpenSPARC design, the durations of all the stages were consistent with the exception of PAR which showed a significant and unexplainable speed-up. Some implementation flows incorporate previous builds as a guide for future ones, such as Xilinx's SmartGuide. This can significantly improve implementation times for design iterations where very small changes occur. However in this case, SmartGuide was not responsible for these speedups, nor is guided implementation available for all designs. Similarly, the FloPoCo design also experienced an overall decrease in implementation time, with the Map process taking twice as long as the base implementation and the PAR stage taking less. This is not uncommon with more recent versions of the tools which speculatively perform placement during mapping when the mapping can be iterated to prevent PAR from failing. It was

noted that for this design, the placement was indeed performed during the mapping, however the overall decrease in execution time remains unaccounted for as no guided implementation was used. In the MicroBlaze benchmark, the overall execution time increased. In all cases, a complex design that takes hours to implement will also take hours to insert ChipScope. The observed decrease in execution time is not guaranteed to occur in all situations and as noted earlier, the presence or removal of ChipScope can unexpectedly alter the design introducing new timing challenges. Even with the reduction of implementation times for modifications to the ChipScope-enabled design, the overall time still presents a significant obstacle to increasing the number of turns-per-day. The benchmark design results are shown in Figure 6.2.

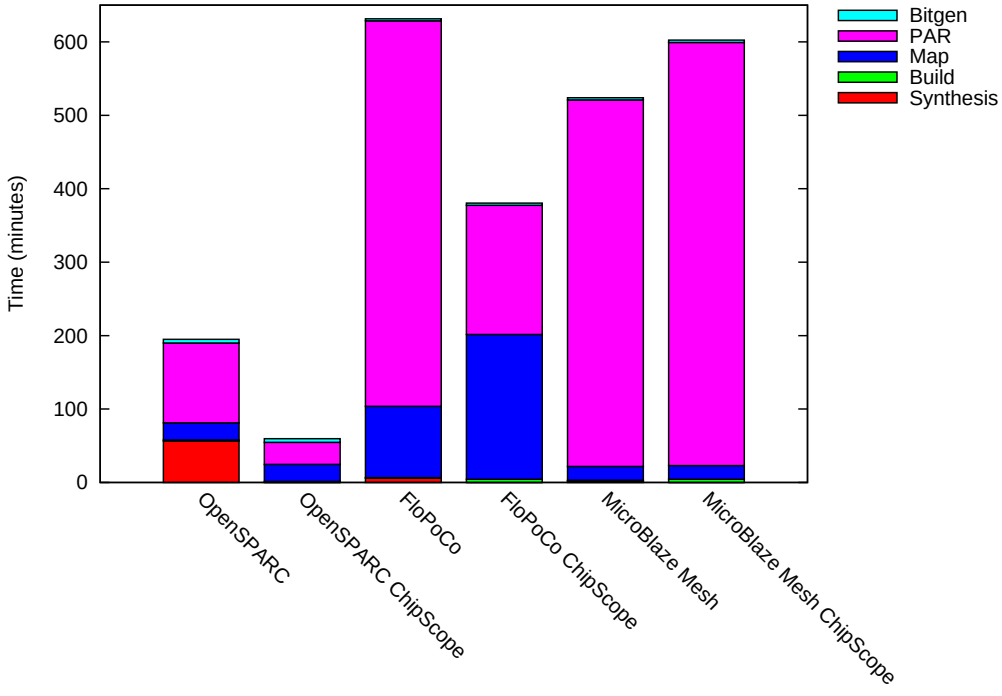

Figure 6.2: Benchmark design implementation times.

Table 6.2: LLD implementation times (minutes:seconds).

Stage	**Real time**
Synthesis	0:08
Build	0:04
Map	0:39
PAR	0:45
Bit	1:35
Total	3:11

LLD

The LLD framework is isolated from the actual design by a partially reconfigurable region that partitions the debug logic into a region that can be implemented quickly. In comparison to ChipScope, similar tasks take anywhere from a few seconds to a few minutes as opposed to several hours. This speed-up is achieved through the "inverted bitstream" scenario outlined in Section 5.2.1. This technique eliminates the processing overhead of partially reconfigurable designs in which the entire design and not just the reconfigurable region are processed. As a result, LLD implementation time is constant regardless of the number changes or the overall design size. For the benchmark designs, the total implementation time of the debugging region took slightly over three minutes until the changes were ready to be used.

Table 6.2 gives LLD implementation times for a breakpoint region comprised of 204 Slices. The time required for synthesis and build were both under 10 seconds and consist of asynchronous logic that makes up the conditional breakpoints and assertions. Map and PAR each both take around 40 seconds each, considerably less time than the corresponding phases required for re-implementation when inserting or altering a ChipScope core. The minute and a half time required for bitstream generation is mostly due to DRC. While DRC can be eliminated which will noticeably reduce implementation time, its primary purpose is to detect configurations that might damage the device, making it a worthwhile investment. Figure 6.3

shows a pie chart of data given in Table 6.2 showing the percentage of time spent at each phase of the implementation process. Unlike large designs, PAR now consumes approximately one-quarter of the execution time.

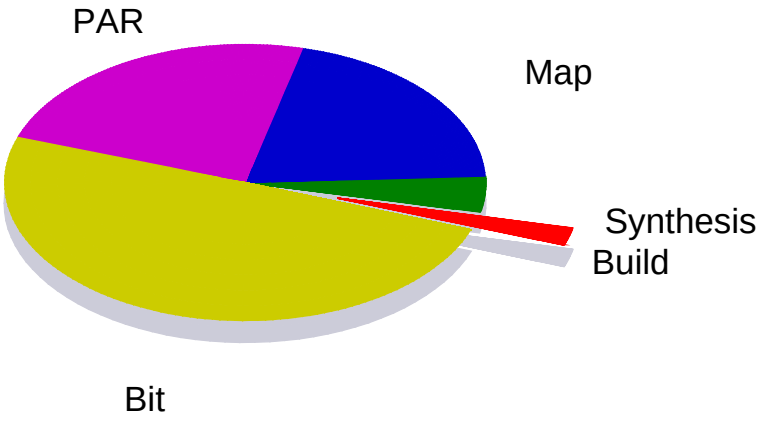

Figure 6.3: Percentage of time required for LLD implementation.

The number of resources required for a breakpoint region varies depending on the number of inputs, breakpoints, and the complexity of the logic implemented. However, this region requires only Slices which does not noticeably compete with the target design as would other resources. Even with high-utilization designs such as those implemented as benchmarks, both Map and PAR can be configured to expend extra effort and attempt to more tightly pack logic. In contrast, ELAs require a significant number of BRAM resources to record signal activity. This requirement increases proportionally with the number of signals to be monitored or the number of samples to be recorded. In one experiment, a simple set of breakpoints occupied as few as 10 Slices. Oversizing the breakpoint region can avert contention should the number or complexity of the breakpoints outsize the current region. However should this happen, the PATIS framework is capable of resizing and adjusting the floorplan as needed, likely with no additional time.

```
$ connect /dev/ttyS1
$ load-logic top.ll
$ break (text_i == 32'h62636465)
$ info breakpoints
  Breakpoint mask:    0x00000001
  Active breakpoints: 0x00000000

  Num Type  Enb Act What
  0   break y   n   text_i==32'h62636465

$ run
$ info breakpoints
  Breakpoint mask:    0x00000001
  Active breakpoints: 0x00000001

  Num Type  Enb Act What
  0   break y   y   text_i==32'h62636465

$ disable 0
$ info breakpoints
  Breakpoint mask:    0x00000000
  Active breakpoints: 0x00000001

  Num Type  Enb Act What
  0   break n   y   text_i==32'h62636465

$ print cmd_i
    cmd_i = 0x1
```

Listing 6.2: LLD debug transcript for a SHA-1 core.

Software

A transcript of a debug session is shown in Listing 6.2. The syntax and commands are similar to those found in the GNU `gdb` debugger. A debugging session is initiated by connecting to the on-chip processor through the serial line and then building the design and device model from the logic-allocation file. Reading the logic-allocation file is the single most time- and resource-intensive task and the primary reason for locating this operation to a conventional workstation. Even for small designs, logic-allocation files can be around 25 MB and for a large design such as the OpenSPARC, can exceed 255 MB.

Listing 6.2 shows how conditional breakpoints are first defined using top-level ports and signals defined in the breakpoint module. An `information` command is used to display breakpoint status. The information output begins with two status masks: the breakpoint mask shows which breakpoints are enabled, while the active breakpoint mask indicates which of the 32 breakpoints are currently active. This is followed by a detailed listing, shortened here for readability, which defines the index of the breakpoint, whether the breakpoint is a conditional breakpoint or an assertion, whether or not the breakpoint is active or enabled, and the original text of the breakpoint. As with conventional command-line debuggers, breakpoints can be individually enabled or disabled through a software-controlled mask without the need to rebuild or restart the design. The final `print` command in the listing demonstrates the ability to print the value of an arbitrary design register from the command-line.

Connections to the breakpoint logic region use FPGA routing resources out of necessity. Although this approach would be considered a shortcoming, DMD's restriction to top-level signals and the ability to arbitrarily read any design register through the processor did not present any of the expected problems such as routing issues. DMD is intended to be used as a low-level analysis tool, focusing on the signal interactions between design modules and the internal state that drives those signals. DMD's core is instantiated as part of the static region, meaning subsequent modifications of partitioned modules do not require lengthy re-routing. While a connectionless approach would have allowed any arbitrary register of the design to appear in a breakpoint condition and eliminated the need for an additional PR region, this would require a costly polling scheme of the targeted signals, reducing execution performance. LLD's ability to execute at the target design speed is unique among approaches that have full visibility of the design.

LLD breakpoints are implemented as asynchronous logic, signaling to the PDC to interrupt execution and allow software identification of active breakpoints. Each statement represents

a breakpoint as entered at the command-line. Activated rules signal to the PDC which of the 32 breakpoints triggered and suspended execution. Clock buffers respond to control signals at the next rising edge, provided that setup and hold time requirements are met. If not, transition is deferred until the following rising edge, providing at most a two-cycle latency. Consistent one-cycle response times were observed when the design was halted by breakpoints which was validated by checking register values against the programmed breakpoint. A diagnostic clock cycle counter was also implemented to determine reaction time latencies.

Often, synthesis can optimize away logic, even when instructed not to using synthesis pragmas. This can be problematic if specifically trying to observe one of these signals. These optimizations are noted in the synthesis log files. However, it is still possible to use these signals as breakpoint conditions since these are top-level ports to the debug module and this prevents them from disappearing entirely.

Tool run times, particularly Map and PAR, were highly variable. The insertion of additional complex logic is expected to extend implementation times, but was sometimes observed to reduce it in unpredictable ways. For example, in large designs the addition of a ChipScope core is expected to lengthen the implementation time. However, as shown in Table 6.1, some implementation times were reduced, but still were several hours long. Even for a design partitioned through partial reconfiguration, the tools process (but do not implement) the entire design. While only the reconfigurable region is actually re-implemented in subsequent runs, the tools perform DRC on the entire design, lengthening run times particularly on large or complex designs. For instance, this proportionally lengthens bitfile generation time even though a smaller, partial bitstream is the only portion being generated anew.

To evaluate the efficiency of this approach, several common debugging tasks were performed. In many cases ChipScope may require a post-synthesis re-implementation of the design to

accommodate a new task, dominated largely by Map and PAR runtime. However, once implemented with the LLD flow, different breakpoint scenarios were possible by only re-implementing the altered breakpoint region. With ChipScope it is not possible to arbitrarily halt the design and read a randomly selected register. The embedded logic analyzer and control cores must be reconfigured and re-implemented for a specific event. Using LLD, breakpoints were set to suspend execution so that registers could be inspected and then later the design was stepped by a small number of clock cycles to advance execution and continue inspection. LLD allows register values to be randomly read from the command-line using their full hierarchical design name in two to eight seconds depending on the width of the signal and distribution of the bits across the FPGA's configuration frames. While individual bits of a register are frequently placed as close as possible, it is usually not possible to find an entire register placed in a single configuration frame, thus resulting in fragmentation across several frames. One improvement was to group reads from the same frame into a single transaction by batching offsets from the same frame. This reduced read times by an average of 50%. While additional improvements are possible, the two to eight seconds required for an arbitrary register represent a significant improvement over having to re-implement and reroute a design to change which registers should be targeted.

The console application was similarly evaluated for its performance. The most time-consuming recurring process was the on-chip portion of reading an FPGA register. Individually, each bit requires 0.5 seconds to process a read operation, however the locality of reference of adjacent bits in the same frame prevents this from being linearly proportional. A 32-bit register takes only 7.5 seconds due to the packing of the bits. Despite utilizing a serial console—one of the slowest communications mediums still being used in modern computing systems—the entire round-trip cost is only 0.5 seconds. The most time consuming non-recurring operation is that of reading the logic-allocation file. Approximately 50% of the CPU-time, accounting for 1.6 seconds of the application's startup time, is spent parsing the logic-allocation file to

build the logic and device model. Figure 6.4, produced from the Google Perf Tools profiler suite, shows the relative time spent in a typical execution scenario. It is noteworthy that none of the blocking operations delegated to the FPGA are long enough to appear in the profiler's output as a significant source of execution time.

6.2.3 Summary and Future Work

LLD provides a software-based approach to both controlling and observing FPGA execution. An onboard processor provides an interactive user interface to the FPGA, which is in stark contrast to the post-execution, forensic-style approaches of vendor embedded logic analyzers. LLD provides an interface that is transparently distributed across a conventional workstation and the onboard processor of the FPGA to provide an all encompassing view of the device. This view is cross-referenced with design-time reports, such as the logic-allocation file, which can be leveraged to build symbol tables useful for debugging. No other debugging frameworks were identified that provide such an approach and allow the design to execute at its full speed.

Enhancements reserved for future work include extending the capabilities of the debug logic and improving the performance of the software. Currently, LLD only supports the generation of asynchronous debug logic which is a condition that can be active during a single clock cycle. Assertion-Based Verification, introduced in Section 2.3.4, allows sequences of states to be defined, equivalent to defining a state-machine that must be activated before the condition fires. A state-based approach to debugging would expand the scope of LLD to far more complex and complete scenarios.

Performance of the user application is another area reserved for future work. Although responsive, response times could be further improved. While the communication channel could be upgraded to a faster medium, such as USB or Ethernet, this would limit portability across development boards. A half-second latency in communication is tolerable, although

not optimum. The greatest latency is found in process of gathering bits from the FPGA, which if not optimized, compounds the round-trip latency over the serial line. The current implementation sequentially processes the register bits, grouping subsequent requests to the same configuration frame before querying the FPGA-based processor and proceeding to the next frame. Separate and repeated requests are made to the same configuration frame if intermediate bits reside in other frames. A more efficient approach is to globally group all the requests to the same frame. A preliminary analysis shows that this approach could reduce the time required to read a 32-bit register by nearly half.

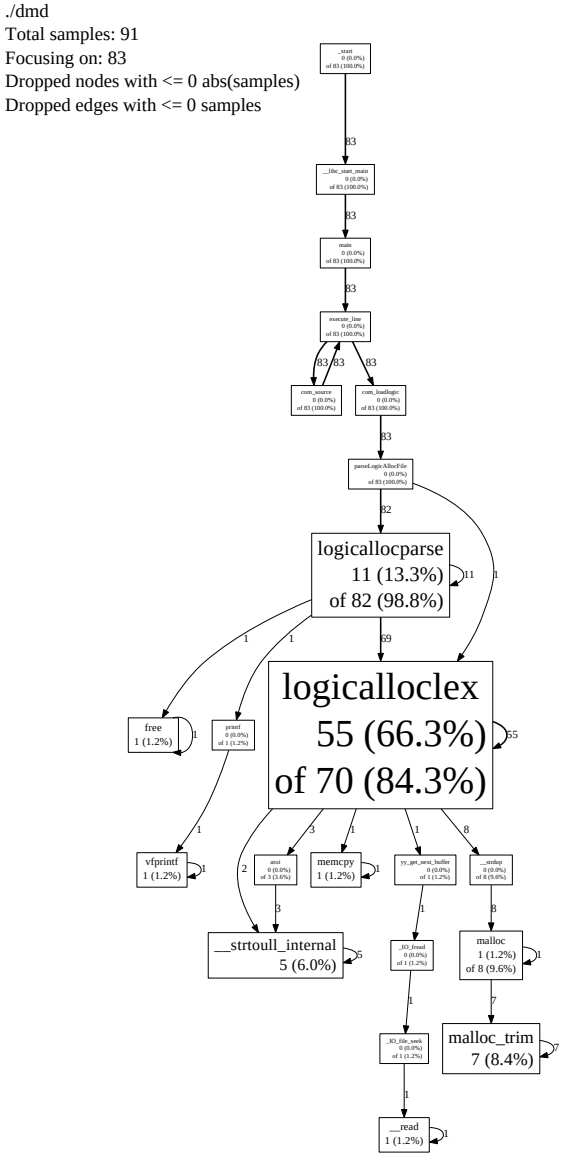

Figure 6.4: Execution call graph of console application.

Chapter 7

Conclusions and Future Work

The lack of a structured development and debug lifecycle for FPGAs causes two issues. First, initial high-level language reference models become isolated from the implemented hardware resulting in the model verification gap. This makes validating hardware against the reference model a manual task. The second issue arises once the hardware has been implemented and is inaccessible to inspection and debug. This is in stark contrast to the flexible and open development environments found in software. The thesis of this dissertation is that the same high level of design visibility and control found in software development environments could be ported to FPGAs by incorporating software control directly into the FPGA. This was achieved by first identifying the primary difficulties in the conventional FPGA design flow, and developing embedded software-based solutions to improve accessibility and raise the abstraction level of these tasks.

7.1 Review of Contributions

The contributions of this research are as follows:

- Development of a tool that addresses the model verification gap by raising the abstraction level for FPGA validation and debug.

  FPGA debug has traditionally required a detailed knowledge not only of the design being developed, but also of the target hardware. The original RTL source is often too primitive to validate against high-level functionality. The HLV framework introduced in Section 4.1 allows synthesized hardware to be directly linked to a high-level language reference model without invasively modifying either one. The reference model and hardware are concurrently executed on the FPGA with the hardware design running at the target frequency. HLV serves as a software-based unit-testing framework for FPGA designs, capable of being automated to progressively test increasing levels of the design hierarchy with software-generated test vectors with the results being immediately validated. HLV is unique in that not only does it link hardware to software reference models, but also that the validation can be performed at the hardware's target execution frequency. Debugging against high-level models is not common even in HLS flows.

  HLV is useful in that it removes the burden of manually validating reference models against the implemented hardware, even for design flows not destined for FPGAs. In most cases, hardware development and validation is repeated anew first for simulation development and then again for actual hardware which may draw from sources not used during simulation. This can open the door to unexpected and expensive design flaws. In the case of some ASIC design strategies and unlike FPGA development, none of the application HDL used in simulation can be applied. In most ASIC flows, FPGAs are used to logically and functionally validate the design before being sent to production or "taped-out". The Pentium FDIV bug [80, 81] is one such example of the consequences of the gap between reference model and implementation validation that HLV aspires to

address. HLV can also be applied to HLS flows which still have not eclipsed traditional HDL development. HLV can be used to validate HLS-generated designs since the machine-generated RTL does not lend itself well to traditional debugging or formal verification.

- Development of a tool that improves visibility, control, and agility for FPGA debug by introducing software development environment facilities.

  Unlike software development, FPGAs have few means to inspect or debug a running design such as the console messages or comprehensive debuggers that exist in software to provide detailed analysis of the design or halt execution. The internal state elements of the design have limited or no visibility, and execution occurs at millions of clock cycles per second. An error may occur and by the time it manifests itself, the state of the device may have changed so much that its diagnosis is impossible. An approach to improve visibility, control, and the agility of an FPGA development environment was introduced in Section 4.2 and implemented in Section 5.2. The developed framework brings some of the useful facilities found in software development environments to FPGAs. The system presented here allows a running, implemented design to be controlled and inspected at its target speed with the interface and agility of software development tools. While most FPGA development tools place the debugger's viewpoint outside the design, the developed framework places it within the FPGA, transforming the storage elements of the reconfigurable fabric to a conventional memory. Symbolic access to design elements is enabled by the construction of a symbol table not unlike those found in software debuggers. The agility of software breakpoints is recreated in a partially reconfigurable region that can be quickly implemented without affecting the rest of the design.

7.2 Future Work

The research presented here shows that software is a viable means to both validating and debugging FPGA designs despite the wide disparity in performance. Several enhancements and optimizations were reserved for future work, such as functional and performance-related improvements.

The HLV framework is currently programmed through manual transcription from simulation models. This is both time-consuming and error-prone and ultimately unnecessary with additional means to directly generate testbenches from simulations. HLV can also be extended to handle more complex scenarios beyond fixed-latency transactions by incorporating programmable hardware that can interact with the design without the need to buffer signal traces. Other approaches could enable complex, software-controlled interactions capable of scripting large volumes of data, but might eliminate the sustained, full-speed execution demonstrated here.

LLD demonstrated fine-grain and interactive control over an implemented design through embedded software. Given the extensive use of multiple clock domains, there remains some research to be done in implementing cascading clock control logic to reliably synchronize all clock domains. The LLD software could be further improved in the areas of performance and efficiency. Operations requiring round-trip transactions cumulatively add to the total time needed, however methods have been identified that could reduce this time by as much as 50%. This can be achieved by further optimizing the algorithms used to schedule reading and assembling design registers. Assertion-Based Verification techniques, such as those introduced in Section 2.3.4, which can define multi-state transitions for a single assertion, could be introduced to realize complex scenarios such as state-machine validation.

Bibliography

[1] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohner, "A Research Agenda for Improving Configurable Computing Design Productivity," NSF Center for High-Performance Reconfigurable Computing (CHREC), Salt Lake City, UT, Tech. Rep., June 2008. [Online]. Available: http://www.chrec.org/ftsw/BYU_VT_Report.pdf

[2] B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohner, "Design Productivity for Configurable Computing." CSREA Press, 2008, pp. 57–66.

[3] Xilinx, Inc., "UG190: Virtex-5 FPGA User Guide."

[4] ——, "XC4000E and XC4000X Series Field Programmable Gate Arrays (Product Specification)."

[5] ——, "XC5200 Series Field Programmable Gate Arrays (Product Specification)."

[6] ——, "DS003: Virtex 2.5 V Field Programmable Gate Arrays (Product Specification)."

[7] ——, "DS031: Virtex-II Platform FPGAs: Complete Data Sheet."

[8] ——, "DS083: Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet."

[9] ——, "DS100: Virtex-5 Family Overview."

[10] ——, "DS112: Virtex-4 Family Overview."

[11] ——, "DS150: Virtex-6 Family Overview."

[12] ——, "DS180: 7 Series FPGAs Overview."

[13] C. Metz, "Dennis Ritchie: The Shoulders Steve Jobs Stood On," *Wired*, October 2011. [Online]. Available: http://www.wired.com/wiredenterprise/2011/10/thedennisritchieeffect/

[14] M. Santarini, "Driver assistance revs up on xilinx fpga platforms," *Xcell Journal*, vol. 66, 2008.

[15] J. Wilcox and M. Kanellos, "Amd makes move to 1-ghz chip," *CNET News*, March 2000. [Online]. Available: http://news.cnet.com/2100-1040-237615.html

[16] C. Schalick, "Debugging FPGA designs may be harder than you expect," *EDN*, vol. 54, no. 21, p. 23, 2009.

[17] Free Software Foundation, "GNU Make." [Online]. Available: http://www.gnu.org/software/make

[18] Altera Corporation. Quartus II Incremental Compilation for Hierarchical and Team-Based Design. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii51015.pdf

[19] *Development System Reference Guide*, v10.1 ed., Xilinx, Inc., 2008. [Online]. Available: http://www.xilinx.com/itp/xilinx4/pdf/docs/dev/dev.pdf

[20] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson, "PATIS: using partial configuration to improve static FPGA design productivity," *in 17th Reconfigurable Architectures Workshop (RAW)*, 2010.

[21] *XST User Guide*, v11.3 ed., Xilinx, Inc., September 2009.

[22] Xilinx, Inc. Memory Recommendations. [Online]. Available: http://www.xilinx.com/ise/products/memory.htm

[23] N. Steiner. ChipScope scripting for batch data collection? [Online]. Available: usenet://comp.arch.fpga

[24] Xilinx, Inc., "MicroBlaze Processor Reference Guide."

[25] L. Wirbel, "FPGA survey sees sunset for gate arrays, continued dominance by Xilinx, Altera," *EE Times*, October 2008. [Online]. Available: http://www.eetimes.com/showArticle.jhtml?articleID=211200184

[26] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1 –560, 2006.

[27] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. c1 –626, 26 2009.

[28] "IEEE Standard System C Language Reference Manual," *IEEE Std 1666-2005*, pp. 1 –423, 2006.

[29] "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. C1 –1285, 2009.

[30] Cadence Design Systems, Inc. Cadence e Verification Language. [Online]. Available: http://www.cadence.com/Alliances/languages/Pages/e_page.aspx

[31] "IEEE Standard for the Functional Verification Language e," *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008)*, pp. 1 –495, 26 2011.

[32] BlueSpec, Inc. BlueSpec Language. [Online]. Available: http://www.bluespec.com/

[33] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *Design & Test of Computers, IEEE*, vol. 26, no. 4, pp. 8–17, 2009.

[34] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra, "Lessons and Experiences with High-Level Synthesis," *Design & Test of Computers, IEEE*, vol. 26, no. 4, pp. 34–45, 2009.

[35] E. Eto, *Xilinx Application Note 290: Difference-Based Partial Reconfiguration*, v2.0 ed., Xilinx, December 2007. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf

[36] S. D. Craven, "Structured approach to dynamic computing application development," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2008.

[37] G. Subbarayan, "Automatic instantiation and timing-aware placement of bus macros for partially reconfigurable FPGA designs." Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2010.

[38] A. Chandrasekharan, "Accelerating incremental floorplanning of partially reconfigurable designs to improve FPGA productivity." Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2010.

[39] "IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture," *IEEE Std 1149.7-2009*, pp. c1 –985, 10 2010.

[40] Xilinx, Inc., *UG012: Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, v4.2 ed., Xilinx, Inc., November. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug012.pdf

[41] *UG071: Virtex-4 FPGA Configuration User Guide*, v1.11 ed., Xilinx, Inc., June 2009. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug071.pdf

[42] *UG191: Virtex-5 FPGA Configuration User Guide*, v9.0 ed., Xilinx, Inc., January 2008. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[43] *UG360: Virtex-6 FPGA Configuration User Guide*, v3.4 ed., Xilinx, Inc., November 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf

[44] *UG470: 7 Series FPGAs Configuration User Guide*, v1.3 ed., Xilinx, Inc., February 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

[45] Xilinx, Inc., "Xilinx ChipScope Pro." [Online]. Available: http://www.xilinx.com/tools/cspro.htm

[46] Altera Corporation, "Design Debugging Using the SignalTap II Embedded Logic Analyzer."

[47] GateRocket, Inc., "GateRocket Product Overview," 2008.

[48] "FPGA tool vendor gaterocket folds," *EE Times*, August 2011. [Online]. Available: http://www.eetimes.com/electronics-news/4218492/FPGA-tool-vendor-GateRocket-folds

[49] Sandbyte Technologies. FPGAXpose Data Sheet. [Online]. Available: http://www.sandbyte.com/FPGAXposeDataSheet.pdf

[50] Synopsys, Inc., "Fast, Efficient RTL Debug for Programmable Logic Designs."

[51] Cadence Design Systems, Inc., "Cadence Palladium."

[52] W. Danghui, G. Deyuan, and L. Tao, "Breakpoint debugging mechanism for microprocessor design," 2003, pp. 456–458.

[53] A. Penttinen, R. Jastrzebski, R. Pollanen, and O. Pyrhonen, "Run-Time Debugging and Monitoring of FPGA Circuits Using Embedded Microprocessor," *Design and Diagnostics of Electronic Circuits and systems, IEEE*, pp. 147–148, 2006.

[54] Z. Baker and J. Monson, "In-situ FPGA debug driven by on-board microcontroller," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, April 2009, pp. 219 –222.

[55] R. Kuramoto, "Application Note 058: Xilinx In-System Programming Using an Embedded Microcontroller." [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp058.pdf

[56] 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*, 1999, pp. 12 –24.

[57] "JHDL: FPGA CAD Tools." [Online]. Available: http://www.jhdl.org

[58] C. Patterson and S. Guccione, "Jbits design abstractions," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, 29 2001-april 2 2001, pp. 251 –252.

[59] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting Bitstreams for Debugging FPGA Circuits," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington,

DC, USA: IEEE Computer Society, 2001, pp. 41–50. [Online]. Available: http://portal.acm.org/citation.cfm?id=1058426.1058867

[60] D. Levi and S. A. Guccione, "BoardScope: A debug tool for reconfigurable systems," in *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, J. Schewel, Ed. Bellingham, WA: SPIE – The International Society for Optical Engineering, November 1998, pp. 239–246.

[61] T. Price, D. Levi, and S. A. Guccione, "Debug of reconfigurable systems," *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 4212, pp. 181–187, 2000.

[62] Impulse Accelerated Technologies, "Impulse CoDeveloper C-to-FPGA Tools." [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm

[63] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, 2000, pp. 49 –56.

[64] "The Stanford SUIF Compiler Group." [Online]. Available: http://suif.stanford.edu

[65] D. Pellerin and S. Thibault, *Practical FPGA programming in C*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005.

[66] J. Curreri, G. Stitt, and A. George, "High-level synthesis techniques for in-circuit assertion-based verification," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, april 2010, pp. 1 –8.

[67] Xilinx, Inc., "Partial Reconfiguration User Guide." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

[68] A. Chandrasekharan, S. Rajagopalan, G. Subbarayan, T. Frangieh, Y. Iskander, S. Craven, and C. Patterson, "Accelerating FPGA development through the automatic parallel application of standard implementation tools," in *Field-Programmable Technology (FPT), 2010 International Conference on*, December 2010, pp. 53–60.

[69] S. Raja Gopalan, "Timing-aware automatic floorplanning of partially reconfigurable designs for accelerating FPGA productivity." Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2010.

[70] Xilinx, Inc., "DS449: LogiCORE IP Fast Simplex Link (FSL) V20 Bus."

[71] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong, "Metawire: Using FPGA configuration circuitry to emulate a network-on-chip," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept. 2008, pp. 257 –262.

[72] "GDB: The GNU Project Debugger." [Online]. Available: http://www.gnu.org/software/gdb/gdb.html

[73] "GNU Readline Library." [Online]. Available: http://www.gnu.org/software/readline

[74] D. R. Stinson, *Cryptography Theory and Practice*, 3rd ed. Chapman & Hall/CRC, 2006.

[75] "SHA-1 Collision Search Graz." [Online]. Available: http://boinc.iaik.tugraz.at/

[76] "OpenCores." [Online]. Available: http://www.opencores.org

[77] J. Curreri, G. Stitt, and A. George, "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[78] Oracle Corporation, "OpenSPARC." [Online]. Available: http://www.opensparc.net/

[79] "FloPoCo." [Online]. Available: http://flopoco.gforge.inria.fr/

[80] R. P. Colwell, *The Pentium chronicles: the people, passion, and politics behind Intel's landmark chips*. Hoboken, N.J.: John Wiley & Sons, Inc., 2006.

[81] D. Price, "Pentium FDIV flaw-lessons learned," *Micro, IEEE*, vol. 15, no. 2, pp. 86 –88, Apr 1995.

Appendix A

Open-Source and Free Software Acknowledgement

The following open-source or free software packages were instrumental in this research. Links to software homepages and/or source are included for reference.

- Ubuntu GNU/Linux. http://www.ubuntu.com.
- Flex lexer, an open-source successor to the lex lexer generator. http://flex.sourceforge.net.
- Bison parser generator, an open-source successor to the yacc parser generator. http://www.gnu.org/software/bison.
- Vim Editor. http://www.vim.org.
- Oracle VirtualBox virtualization software. http://www.virtualbox.org.
- GNU Readline Library. http://www.gnu.org/software/readline.
- Apache Xerces XML Library. http://xerces.apache.org.

Appendix B

Source Code

B.1 High-Level Validation

Listing B.1: HLV User Logic.

```
1  //----------------------------------------------------------------------------
2  // user_logic.vhd - module
3  //----------------------------------------------------------------------------
4  //
5  // ****************************************************************************
6  // ** Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.            **
7  // **                                                                        **
8  // ** Xilinx, Inc.                                                          **
9  // ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
10 // ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
11 // ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
12 // ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
13 // ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
14 // ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
15 // ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
16 // ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
17 // ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
18 // ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
19 // ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
20 // ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
21 // ** FOR A PARTICULAR PURPOSE.                                             **
22 // **                                                                        **
23 // ****************************************************************************
24 //
25 //----------------------------------------------------------------------------
26 // Filename:          user_logic.vhd
27 // Version:           1.00.a
28 // Description:       User logic module.
29 // Date:              Tue Jun 22 23:17:44 2010 (by Create and Import Peripheral Wizard)
30 // Verilog Standard:  Verilog-2001
31 //----------------------------------------------------------------------------
32 // Naming Conventions:
33 //   active low signals:                    "*_n"
34 //   clock signals:                         "clk", "clk_div#", "clk_#x"
35 //   reset signals:                         "rst", "rst_n"
36 //   generics:                              "C_*"
37 //   user defined types:                    "*_TYPE"
38 //   state machine next state:              "*_ns"
39 //   state machine current state:           "*_cs"
40 //   combinatorial signals:                 "*_com"
41 //   pipelined or register delay signals:   "*_d#"
42 //   counter signals:                       "*cnt*"
43 //   clock enable signals:                  "*_ce"
44 //   internal version of output port:       "*_i"
45 //   device pins:                           "*_pin"
46 //   ports:                                 "- Names begin with Uppercase"
47 //   processes:                             "*_PROCESS"
48 //   component instantiations:              "<ENTITY_>I_<#|FUNC>"
```

```
49  //------------------------------------------------------------------------
50
51  module user_logic
52  (
53    // -- ADD USER PORTS BELOW THIS LINE ---------------
54    // --USER ports added here
55    DMD_Fifo_Wr_En,
56    DMD_Fifo_Rd_En,
57    DMD_Fifo_Exists,
58    DMD_Fifo_Full,
59    DMD_Design_Clk,
60    // -- ADD USER PORTS ABOVE THIS LINE ---------------
61
62    // -- DO NOT EDIT BELOW THIS LINE ------------------
63    // -- Bus protocol ports, do not add to or delete
64    Bus2IP_Clk,                     // Bus to IP clock
65    Bus2IP_Reset,                   // Bus to IP reset
66    Bus2IP_Data,                    // Bus to IP data bus
67    Bus2IP_BE,                      // Bus to IP byte enables
68    Bus2IP_RdCE,                    // Bus to IP read chip enable
69    Bus2IP_WrCE,                    // Bus to IP write chip enable
70    IP2Bus_Data,                    // IP to Bus data bus
71    IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
72    IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
73    IP2Bus_Error                    // IP to Bus error response
74    // -- DO NOT EDIT ABOVE THIS LINE ------------------
75  ); // user_logic
76
77  // -- ADD USER PARAMETERS BELOW THIS LINE ------------
78  // --USER parameters added here
79  // -- ADD USER PARAMETERS ABOVE THIS LINE ------------
80
81  // -- DO NOT EDIT BELOW THIS LINE ---------------------
82  // -- Bus protocol parameters, do not add to or delete
83  parameter C_SLV_DWIDTH                   = 32;
84  parameter C_NUM_REG                      = 8;
85  // -- DO NOT EDIT ABOVE THIS LINE ---------------------
86
87  // -- ADD USER PORTS BELOW THIS LINE ------------------
88  // --USER ports added here
89  output                    DMD_Fifo_Wr_En;
90  output                    DMD_Fifo_Rd_En;
91  input                   DMD_Fifo_Exists;
92  input                     DMD_Fifo_Full;
93  input                     DMD_Design_Clk;
94  // -- ADD USER PORTS ABOVE THIS LINE ------------------
95
96  // -- DO NOT EDIT BELOW THIS LINE ---------------------
97  // -- Bus protocol ports, do not add to or delete
98  input                                     Bus2IP_Clk;
99  input                                     Bus2IP_Reset;
100 input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
101 input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
102 input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
103 input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
104 output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
105 output                                    IP2Bus_RdAck;
106 output                                    IP2Bus_WrAck;
107 output                                    IP2Bus_Error;
108 // -- DO NOT EDIT ABOVE THIS LINE ---------------------
109
110 //------------------------------------------------------------------------
111 // Implementation
112 //------------------------------------------------------------------------
113
114   // --USER nets declarations added here, as needed for user logic
115   wire                      go;
116   wire        [0 : C_SLV_DWIDTH-1]          start_capture_at;
117   wire        [0 : C_SLV_DWIDTH-1]          stop_capture_at;
118
119
120   // counter held in reset until driver is activated
121   reg    [0 : 31]           counter;
122   reg                    capture_complete;
123
124   // Nets for user logic slave model s/w accessible register example
125   reg         [0 : C_SLV_DWIDTH-1]          slv_reg0;   // control
126   reg         [0 : C_SLV_DWIDTH-1]          slv_reg1;   // start_capture_at
127   reg         [0 : C_SLV_DWIDTH-1]          slv_reg2;   // stop_capture_at
128   //reg         [0 : C_SLV_DWIDTH-1]          slv_reg3;   // status
129   reg         [0 : C_SLV_DWIDTH-1]          slv_reg4;
130   reg         [0 : C_SLV_DWIDTH-1]          slv_reg5;
131   reg         [0 : C_SLV_DWIDTH-1]          slv_reg6;
132   reg         [0 : C_SLV_DWIDTH-1]          slv_reg7;
133   wire        [0 : 7]                       slv_reg_write_sel;
134   wire        [0 : 7]                       slv_reg_read_sel;
135   reg         [0 : C_SLV_DWIDTH-1]          slv_ip2bus_data;
```

```
136     wire                                    slv_read_ack;
137     wire                                    slv_write_ack;
138     integer                                 byte_index, bit_index;
139
140     // --USER logic implementation added here
141
142     assign go = slv_reg0[31];
143     assign start_capture_at = slv_reg1;
144     assign stop_capture_at = slv_reg2;
145
146      // Write to capture fifos when the counters are in range
147     assign DMD_Fifo_Wr_En = ((counter >= start_capture_at) && (counter < stop_capture_at) && (DMD_Fifo_Full == 1'b0));
148
149     // Read from the fifos when the driver is activated and there is still data in there
150     assign DMD_Fifo_Rd_En = (go & DMD_Fifo_Exists);
151
152
153
154     always @(posedge DMD_Design_Clk)
155     begin
156     if (go == 1'b0)
157     begin
158       // our version of reset
159       counter <= 32'b0;
160       capture_complete <= 1'b0;
161     end
162     else
163     begin
164       counter <= counter + 1;
165
166       // track when the capture is complete
167       if (counter >= stop_capture_at)
168         capture_complete <= 1'b1;
169     end
170     end
171
172     // ------------------------------------------------------
173     // Example code to read/write user logic slave model s/w accessible registers
174     //
175     // Note:
176     // The example code presented here is to show you one way of reading/writing
177     // software accessible registers implemented in the user logic slave model.
178     // Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
179     // to one software accessible register by the top level template. For example,
180     // if you have four 32 bit software accessible registers in the user logic,
181     // you are basically operating on the following memory mapped registers:
182     //
183     //    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
184     //                      "1000"   C_BASEADDR + 0x0
185     //                      "0100"   C_BASEADDR + 0x4
186     //                      "0010"   C_BASEADDR + 0x8
187     //                      "0001"   C_BASEADDR + 0xC
188     //
189     // ------------------------------------------------------
190
191     assign
192       slv_reg_write_sel = Bus2IP_WrCE[0:7],
193       slv_reg_read_sel  = Bus2IP_RdCE[0:7],
194       slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2] || Bus2IP_WrCE[3] || Bus2IP_WrCE[4] || \
195     Bus2IP_WrCE[5] || Bus2IP_WrCE[6] || Bus2IP_WrCE[7],
196       slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2] || Bus2IP_RdCE[3] || Bus2IP_RdCE[4] || \
197     Bus2IP_RdCE[5] || Bus2IP_RdCE[6] || Bus2IP_RdCE[7];
198
199     // implement slave model register(s)
200     always @( posedge Bus2IP_Clk )
201       begin: SLAVE_REG_WRITE_PROC
202
203         if ( Bus2IP_Reset == 1 )
204           begin
205             slv_reg0 <= 0;
206             slv_reg1 <= 0;
207             slv_reg2 <= 0;
208           //  slv_reg3 <= 0;
209             slv_reg4 <= 0;
210             slv_reg5 <= 0;
211             slv_reg6 <= 0;
212             slv_reg7 <= 0;
213           end
214         else
215           case ( slv_reg_write_sel )
216             8'b10000000 :
217               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
218                 if ( Bus2IP_BE[byte_index] == 1 )
219                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
220                     slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
221             8'b01000000 :
222               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
```

```
223                 if ( Bus2IP_BE[byte_index] == 1 )
224                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
225                     slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
226             8'b00100000 :
227               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
228                 if ( Bus2IP_BE[byte_index] == 1 )
229                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
230                     slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
231             /* READ ONLY STATUS
232             8'b00010000 :
233               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
234                 if ( Bus2IP_BE[byte_index] == 1 )
235                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
236                     slv_reg3[bit_index] <= Bus2IP_Data[bit_index];
237             */
238             8'b00001000 :
239               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
240                 if ( Bus2IP_BE[byte_index] == 1 )
241                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
242                     slv_reg4[bit_index] <= Bus2IP_Data[bit_index];
243             8'b00000100 :
244               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
245                 if ( Bus2IP_BE[byte_index] == 1 )
246                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
247                     slv_reg5[bit_index] <= Bus2IP_Data[bit_index];
248             8'b00000010 :
249               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
250                 if ( Bus2IP_BE[byte_index] == 1 )
251                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
252                     slv_reg6[bit_index] <= Bus2IP_Data[bit_index];
253             8'b00000001 :
254               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
255                 if ( Bus2IP_BE[byte_index] == 1 )
256                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
257                     slv_reg7[bit_index] <= Bus2IP_Data[bit_index];
258             default : ;
259           endcase
260
261       end // SLAVE_REG_WRITE_PROC
262
263     // implement slave model register read mux
264     // or slv_reg3
265     always @( slv_reg_read_sel or slv_reg0 or slv_reg1 or slv_reg2 or slv_reg4 or slv_reg5 or slv_reg6 or slv_reg7 )
266       begin: SLAVE_REG_READ_PROC
267
268         case ( slv_reg_read_sel )
269           8'b10000000 : slv_ip2bus_data <= slv_reg0;
270           8'b01000000 : slv_ip2bus_data <= slv_reg1;
271           8'b00100000 : slv_ip2bus_data <= slv_reg2;
272           8'b00010000 : slv_ip2bus_data <= {4'hCAFE, 24'b0, DMD_Fifo_Wr_En, DMD_Fifo_Rd_En, go, capture_complete};
273           8'b00001000 : slv_ip2bus_data <= slv_reg4;
274           8'b00000100 : slv_ip2bus_data <= slv_reg5;
275           8'b00000010 : slv_ip2bus_data <= slv_reg6;
276           8'b00000001 : slv_ip2bus_data <= slv_reg7;
277           default : slv_ip2bus_data <= 0;
278         endcase
279
280       end // SLAVE_REG_READ_PROC
281
282     // ------------------------------------------------------------
283     // Example code to drive IP to Bus signals
284     // ------------------------------------------------------------
285
286     assign IP2Bus_Data    = slv_ip2bus_data;
287     assign IP2Bus_WrAck   = slv_write_ack;
288     assign IP2Bus_RdAck   = slv_read_ack;
289     assign IP2Bus_Error   = 0;
290
291   endmodule
```

Listing B.2: SHA Test Code.

```
1
2   #include <string.h>
3   #include <stdio.h>
4
5   #include "hlv_fsl.h"
6   #include "hlv_driver.h"
7   #include "xparameters.h"
8
9   #define printf xil_printf
10
11  int sha1_test(const char* message, u32 *result_array);
12  unsigned int clock_cycle;
13  //unsigned int clock_cycle;
14  u32 message[32];
15  u32 result[5];
16    unsigned int reset_sequence[] = {
17    0x00000000,
18    0x00000100,
19    0x00000100,
20    0x00000000,
21    0x00000000
22    };
23
24
25    const int SHIFT[] = {24,16,8,0};
26
27  const unsigned int MESSAGE_ARRAY_SIZE = 32;
28
29  char *testarray[4] = {
30    "A man cannot be too careful in the choice of his enemies.",
31    "Nothing that is worth knowing can be taught.             ",
32    "Punctuality is the thief of time.                        ",
33    "To most of us the real life is the life we do not lead.  "
34  //  "Nothing that is worth knowing can be taught.      " //,
35  //  "Punctuality is the thief of time.",
36  //  "To most of us the real life is the life we do not lead."
37    //"Anybody can be good in the country. There are no temptations."
38    };
39    /*, "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
40      "A man cannot be too careful in the choice of his enemies. - Oscar Wilde",
41      "Anybody can be good in the country. There are no temptations there. - Oscar Wilde",
42  //    "The only thing to do with good advice is to pass it on. It is never of any use to oneself. - Oscar Wilde",
43      "A true gentlemen is one who is never unintentionally rude. - Oscar Wilde"
44    };
45  */
46
47  void write_fsl_data(unsigned int data) {
48    write_into_fsl(data, XPAR_FSL_HLV_FSL_DATA_INPUT_SLOT_ID);
49  }
50
51  /*
52  void read_fsl_data(unsigned int data) {
53    read_from_fsl(data, XPAR_FSL_HLV_FSL_DATA_OUTPUT_SLOT_ID);
54  }
55    */
56  void write_fsl_ctrl(unsigned int ctrl) {
57    write_into_fsl(ctrl, XPAR_FSL_HLV_FSL_CTRL_INPUT_SLOT_ID);
58  }
59      /*
60  void read_fsl_ctrl(unsigned int &ctrl) {
61    read_from_fsl(ctrl, XPAR_FSL_HLV_FSL_CTRL_OUTPUT_SLOT_ID);
62  }
63  */
64  void write_fsl_pair(unsigned int ctrl, unsigned int data) {
65  //  xil_printf("%3d: 0x%08x\t0x%08x\r\n", clock_cycle++, ctrl, data);
66    write_into_fsl(ctrl, XPAR_FSL_HLV_FSL_CTRL_INPUT_SLOT_ID);
67    write_into_fsl(data, XPAR_FSL_HLV_FSL_DATA_INPUT_SLOT_ID);
68  }
69
70  void convertToMessageBlock(const char* messageText, u32 *message) {
71
72    const u32 messageTextLength = strlen(messageText);
73    int i = 0;
74    int j = 0;
75
76    // clear array
77    for (i = 0; i < MESSAGE_ARRAY_SIZE; i++) {
78      message[i] = 0;
79    //  printf("message[%2d] = 0x%08x\r\n", i, message[i]);
80    }
81
82    // walk the message
83    j = 0;
84    for (i = 0; i<messageTextLength; i++) {
85      j = i/4;
```

```
86      message[j] = (message[j]) | (messageText[i] << SHIFT[i%4]);
87    //  printf("message[%2d] = 0x%08x i = %d\r\n", j, message[j], i);
88    }
89
90    j = i/4;
91    // place the leading 1 bit of the pad
92    message[j] = message[j] | (1 << (SHIFT[i%4]+7));
93    //printf("message[%2d] = 0x%08x i = %d\r\n", j, message[j], i);
94
95    // place the length in bits at the end
96    message[31] = messageTextLength*8;
97
98    printf("message:");
99    for (i = 0; i < MESSAGE_ARRAY_SIZE; i++) {
100     if (!(i%4)) printf("\r\n");
101     printf("  0x%08x ", message[i]);
102   }
103
104   printf("\r\n");
105 }
106
107
108 void hold(const unsigned int control, const unsigned int data, const unsigned int count) {
109   int i;
110   for (i = 0; i < count; i++) {
111     write_fsl_pair(control, data);
112   }
113 }
114
115 void reset() {
116   const int cycles = 5;
117   int i = 0;
118
119   for (i = 0; i < cycles; i++) {
120     write_fsl_pair(reset_sequence[i], 0x0);
121   }
122 }
123
124 void feed_message1() {
125   const int cycles = 15;
126   int i = 0;
127
128   for (i = 0; i < cycles; i++) {
129     write_fsl_pair(0x2,  message[i]);
130   }
131
132 }
133
134 void feed_message2() {
135   //const int cycles = 15;
136   int i = 0;
137   write_fsl_pair(0x16,  0);
138
139   for (i = 16; i < 32; i++)
140     write_fsl_pair(0x06,  message[i]);
141 }
142
143 void send_read() {
144   int i = 0;
145   write_fsl_pair(0x11,  0);
146
147   hold(0x1, 0x0, 20);
148
149 }
150
151 int main(int argc, char **argv)
152 {
153
154   /*
155     * Disable cache and reinitialize it so that other
156     * applications can be run with no problems
157     */
158    #if XPAR_MICROBLAZE_0_USE_DCACHE
159       microblaze_disable_dcache();
160       microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
161    #endif
162
163    #if XPAR_MICROBLAZE_0_USE_ICACHE
164       microblaze_disable_icache();
165       microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
166    #endif
167
168
169   Xuint32 testIdx = 0;
170     Xuint32 counter = 0;
171   Xuint32 Reg32Value;
172   int i;
```

```
173
174     print("\r\n\n-- Entering main() --\r\n");
175
176     int g;
177      for (counter = 0; counter<4; counter++) {
178
179     // loop over 0 - 3
180     testIdx = counter; //%4;
181
182     printf("\r\n\n\n--- Test %d: '%s'\r\n", testIdx+1, testarray[testIdx]);
183     sha1_test(testarray[testIdx], &result[0]);
184
185     convertToMessageBlock(testarray[testIdx], &message[0]);
186
187     printf("\r\n Queueing data for hardware test.\r\n");
188
189     Reg32Value = HLV_DRIVER_mReadSlaveReg3((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
190     //xil_printf("   Status register: %x\n\r", Reg32Value);
191     clock_cycle = 0;
192
193     // only put in a single reset at startup
194     reset();
195     // indicate you'll be writing
196     write_fsl_pair(0x00000012, 0x0);
197
198     feed_message1();
199
200     hold(0x2, 0x0, 69);
201
202     feed_message2();
203
204     hold(0x6, 0x0, 67);
205
206     send_read();
207
208     // status reg
209     Reg32Value = HLV_DRIVER_mReadSlaveReg3((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
210     //xil_printf("   Status register set to %x\n\r", Reg32Value);
211
212     // write window start 175
213     int windowStart; //= 177;
214     int windowLength; //= 5;
215
216     if (counter == 0) {
217       windowStart = 177;
218       windowLength = 5;
219     } else {
220       windowStart = 177;
221       windowLength = 5;
222     }
223     HLV_DRIVER_mWriteSlaveReg1((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, windowStart);
224     Reg32Value = HLV_DRIVER_mReadSlaveReg1((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
225     xil_printf("   Window start register: %d\n\r", Reg32Value);
226
227     // write window stop offset from windowStart
228     HLV_DRIVER_mWriteSlaveReg2((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, windowStart+windowLength);
229     Reg32Value = HLV_DRIVER_mReadSlaveReg2((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
230     xil_printf("   Window stop register: %d\n\r", Reg32Value);
231
232     // write go
233     HLV_DRIVER_mWriteSlaveReg0((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, 1);
234     Reg32Value = HLV_DRIVER_mReadSlaveReg0((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
235     xil_printf("   Control register: %d\n\r", Reg32Value);
236
237     // status reg
238     Reg32Value = HLV_DRIVER_mReadSlaveReg3((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
239     xil_printf("   Status register: %x\n\r", Reg32Value);
240
241     // write stop
242     HLV_DRIVER_mWriteSlaveReg0((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0, 0);
243     Reg32Value = HLV_DRIVER_mReadSlaveReg0((Xuint32)XPAR_HLV_DRIVER_0_BASEADDR, 0);
244     xil_printf("   Control register set to %d\n\r", Reg32Value);
245
246     // read data from output fifos
247     const unsigned int OUTBUFFER_SIZE = 32;
248     unsigned int output_data[OUTBUFFER_SIZE];
249     unsigned int output_ctrl;
250
251     for (i = 0; i < windowLength; i++) {
252       // read output data
253       read_from_fsl(output_data[i], XPAR_FSL_HLV_FSL_DATA_OUTPUT_SLOT_ID);
254
255       // read output control
256       read_from_fsl(output_ctrl, XPAR_FSL_HLV_FSL_CTRL_OUTPUT_SLOT_ID);
257
258       xil_printf("[%2d] control = 0x%03x data = 0x%08x expected = 0x%08x [%s]\r\n",
259             i+windowStart, output_ctrl, output_data[i], result[i],
```

```
260             (output_data[i] == result[i]?"OK":"FAIL"));
261
262     }
263
264
265     // delay, delay, delay
266     u32 x;
267     for (x = 0; x<80000000; x++) ;
268
269     } // end while
270      /*
271       * Enable and initialize cache
272       */
273      #if XPAR_MICROBLAZE_0_USE_ICACHE
274         microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
275         microblaze_enable_icache();
276      #endif
277
278      #if XPAR_MICROBLAZE_0_USE_DCACHE
279         microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
280         microblaze_enable_dcache();
281      #endif
282
283      print("-- Exiting main() --\r\n");
284      return 0;
285  }
```

Listing B.3: SHA Test Code.

```
1
2  /*
3   *  sha1_hlv.c
4   *  References SHA-1 implementation found in RFC-3174.
5   */
6
7
8  #include <stdlib.h>
9  #include <xparameters.h>
10 #include <string.h>
11
12 #include "sha1.h"
13
14 #define printf xil_printf
15
16 int sha1_test(const char *message, unsigned int *result_array)
17 {
18     SHA1Context sha;
19     int i, j, err;
20     uint8_t Message_Digest[20];
21
22     /*
23      *  Perform SHA-1 tests
24      */
25
26 //        printf( "\r\nTest %d: %d, '%s'\r\n", 2, repeatcount, testarray[0]);
27
28         err = SHA1Reset(&sha);
29     /*
30         if (err)
31         {
32             fprintf(stderr, "SHA1Reset Error %d.\n", err );
33             break;    /* out of for j loop * /
34         }
35     */
36
37             err = SHA1Input(&sha,
38                   (const unsigned char *) message,
39                   strlen(message));
40           /*
41             if (err)
42             {
43                 fprintf(stderr, "SHA1Input Error %d.\n", err );
44                 break;    /* out of for i loop * /
45             }
46       */
47       /*
48
49 */
50         err = SHA1Result(&sha, Message_Digest);
51
52     /*
53         if (err)
54         {
55             fprintf(stderr,
56             "SHA1Result Error %d, could not compute message digest.\n",
57             err );
58         }
59         else  */
60             printf("\t");
61             for(i = 0; i < 20 ; ++i)
62             {
63                 printf("%02X ", Message_Digest[i]);
64         result_array[i/4] = (result_array[i/4] << 8) | Message_Digest[i];
65             }
66             printf("\r\n");
67
68     //    printf("Should match:\r\n");
69      //   printf("\t%s\r\n", resultarray[0]);
70
71     /* Test some error returns * /
72     err = SHA1Input(&sha,(const unsigned char *) testarray[1], 1);
73     printf ("\nError %d. Should be %d.\n", err, shaStateError );
74     err = SHA1Reset(0);
75     printf ("\nError %d. Should be %d.\n", err, shaNull );
76   */
77     return 0;
78 }
```

B.2 Low-Level Debug

Listing B.4: Logic Allocation Data Structure Definition Header.

```
1
2   #ifndef BITINFO_H
3   #define BITINFO_H
4
5   #include <string>
6   #include <stdint.h> // this is required for int#_t types
7   #include <list>
8
9
10  class BitInfo {
11  public:
12    uint32_t bitStreamReadbackLocation;
13    uint32_t frameAddress;
14    uint32_t frameOffset;
15
16    int16_t signalVectorIndex;
17
18    std::string block;
19    std::string latch;
20    std::string net;
21
22    BitInfo() : bitStreamReadbackLocation(0), frameAddress(0), frameOffset(0), signalVectorIndex(-1) {
23    }
24  };
25
26  typedef std::list<BitInfo> BitInfoList;
27  typedef std::map<std::string, BitInfoList > BitMap;
28
29  extern "C" int parseLogicAllocFile(const std::string &llFile, BitMap &bMap);
30
31
32  #endif
```

Listing B.5: Breakpoint Data Structure Header.

```
1
2  #ifndef _BREAKPOINT_H
3  #define _BREAKPOINT_H
4
5  #include <string>
6
7  enum breakpoint_t { BREAKPOINT, ASSERTION, CONSTRAINT };
8
9  class Breakpoint {
10 public:
11   uint16_t idx;   ///< The index of the breakpoint, matches hardware
12   std::string breakpointText; ///< Original breakpoint specification
13 //  bool isEnabled;       ///< Whether or not this breakpoint is enabled or not
14   bool isValid;       ///< To enable the deletion of breakpoints
15   breakpoint_t breakpointType; ///< What type of breakpoint this is
16
17   std::string getBreakpointType() const {
18     switch (breakpointType) {
19     case BREAKPOINT:
20       return "breakpoint";
21     case ASSERTION:
22       return "assertion";
23     case CONSTRAINT:
24       return "constraint";
25     }
26   }
27 };
28
29 #endif  // _BREAKPOINT_H
```

Listing B.6: Serial Communications Source.

```
1  // Taken from and adapted from http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html
2
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <termios.h>
7  #include <stdio.h>
8
9  // these headers are not required for C/gcc
10 #include <unistd.h>
11 #include <stdlib.h>
12 #include <string.h>
13
14 // ysi made this work by getting a working minicom session and then
15 // using stty -F /dev/ttyS0 -a and using those settings
16 // http://www.easysw.com/~mike/serial/serial.html#2_3
17
18 /* baudrate settings are defined in <asm/termbits.h>, which is
19    included by <termios.h> */
20 // #define BAUDRATE B19200
21 #define BAUDRATE B115200
22 /* change this definition for the correct port */
23 //#define MODEMDEVICE "/dev/ttyS3"
24 #define _POSIX_SOURCE 1 /* POSIX compliant source */
25
26 #define FALSE 0
27 #define TRUE 1
28
29 volatile int STOP=FALSE;
30 using namespace std;
31
32 int close_serial(int fd) {
33   close(fd);
34 }
35
36 int setup_serial(char *device)
37 {
38     int fd,c, res;
39     struct termios oldtio,newtio;
40     char buf[255];
41     /*
42        Open modem device for reading and writing and not as controlling tty
43        because we don't want to get killed if linenoise sends CTRL-C.
44      */
45     fd = open(device, O_RDWR | O_NOCTTY );
46     if (fd <0) {perror(device); exit(-1); }
47
48     tcgetattr(fd,&oldtio); /* save current serial port settings */
49     bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */
50
51     /*
52 BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
53 CRTSCTS : output hardware flow control (only used if the cable has
54 all necessary lines. See sect. 7 of Serial-HOWTO)
55 CS8     : 8n1 (8bit,no parity,1 stopbit)
56 CLOCAL  : local connection, no modem contol
57 CREAD   : enable receiving characters
58      */
59 //    newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
60     newtio.c_cflag |= BAUDRATE | CS8 | HUPCL | CREAD | CLOCAL;
61     newtio.c_cflag &= ~(CRTSCTS);
62     /*
63 IGNPAR  : ignore bytes with parity errors
64 ICRNL   : map CR to NL (otherwise a CR input on the other computer
65 will not terminate input)
66 otherwise make device raw (no other input processing)
67      */
68     newtio.c_iflag = IGNBRK;
69     newtio.c_iflag &= ~(IXON | IXOFF | IXANY | IGNPAR);  // turn off sw flow control
70
71     /*
72        Raw output.
73      */
74     newtio.c_oflag = 0;
75
76     /*
77 ICANON  : enable canonical input
78 disable all echo functionality, and don't send signals to calling program
79      */
80 //    newtio.c_lflag = ICANON;
81
82     /*
83        initialize all control characters
84        default values can be found in /usr/include/termios.h, and are given
85        in the comments, but we don't need them here
```

```
 86       */
 87      newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
 88      newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
 89      newtio.c_cc[VERASE]   = 0;     /* del */
 90      newtio.c_cc[VKILL]    = 0;     /* @ */
 91      newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
 92      newtio.c_cc[VTIME]    = 3;     /* was 0; inter-character timer unused */
 93      newtio.c_cc[VMIN]     = 0;     /* blocking read until 1 character arrives */
 94      newtio.c_cc[VSWTC]    = 0;     /* '\0' */
 95      newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
 96      newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
 97      newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
 98      newtio.c_cc[VEOL]     = 0;     /* '\0' */
 99      newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
100      newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
101      newtio.c_cc[VWERASE]  = 0;     /* Ctrl-w */
102      newtio.c_cc[VLNEXT]   = 0;     /* Ctrl-v */
103      newtio.c_cc[VEOL2]    = 0;     /* '\0' */
104
105      /*
106         now clean the modem line and activate the settings for the port
107       */
108      tcflush(fd, TCIFLUSH);
109      tcsetattr(fd,TCSANOW,&newtio);
110
111      return fd;
112
113      /*
114         terminal settings done, now handle input
115         In this example, inputting a 'z' at the beginning of a line will
116         exit the program.
117       * /
118      while (STOP==FALSE) {     // loop until we have a terminating condition * /
119          /* read blocks program execution until a line terminating character is
120             input, even if more than 255 chars are input. If the number
121             of characters read is smaller than the number of chars available,
122             subsequent reads will return the remaining chars. res will be set
123             to the actual number of characters actually read * /
124          res = read(fd,buf,255);
125          buf[res]=0;              // set end of string, so we can printf * /
126  //        printf(":%s:%d\n", buf, res);
127          printf("%s", buf);
128          if (buf[0]=='z') STOP=TRUE;
129
130          write(fd, "help\r\n", 6);
131      }
132      // restore the old port settings * /
133      tcsetattr(fd,TCSANOW,&oldtio);
134      */
135  }
```

Listing B.7: Debugger Commands Header.

```
1
2   #ifndef _CMD_DBG_H
3   #define _CMD_DBG_H
4
5   /*************************** Include Files *******************************/
6
7   #include "xparameters.h"
8   #include "stdio.h"
9   #include "xstatus.h"
10  #include "pdc.h"
11  #include "icap.h"
12
13  /************************** Constant Definitions *************************/
14  //#define LED_DELAY     1000000
15
16  #define   CTRL_STEP_STOP_CLK  (1 << 1)
17  #define   CTRL_RUN      (1 << 0)
18
19  #define   ENABLE_STEP_CLK   (1 << 31)
20  #define   ENABLE_BP_DUT0    (1 << 0)
21
22  //#define XPAR_PDC_0_BASEADDR XPAR_PDC_0_BASEADDR
23
24  void setStepCounter(const Xuint32 stepCounter);
25  Xuint32 readStepCounter();
26  void setEnableStepCounterBit(const int enable);
27  void toggleStopClock();
28  void step(const int steps);
29
30  void setRunBit(const int run);
31  void stop();
32
33  u32 readBreakpointMask();
34  void setBreakpointMask(u32 mask);
35  void enableBreakpoint(const Xuint32 signal);
36  void disableBreakpoint(const Xuint32 signal);
37
38  // toggle the debug clock once - remember that the hardware produces a pulse but you
39  // the software register must be returned regardless
40  void toggleStopClock() {
41    Xuint32 status;
42
43    status = PDC_mReadSlaveReg15(XPAR_PDC_0_BASEADDR, 0);
44
45    xil_printf("[tsc] status: %d\t", status);
46
47    // intentionally set the clock advance bit and write back in
48    status |= CTRL_STEP_STOP_CLK;
49    PDC_mWriteSlaveReg15(XPAR_PDC_0_BASEADDR, 0, status);
50
51    // read it
52    status = PDC_mReadSlaveReg15(XPAR_PDC_0_BASEADDR, 0);
53    xil_printf("[tsc] status: %d\t", status);
54
55    // toggle the bit to 0 - we know it will go to 0 because we just set it to 1
56    status ^= CTRL_STEP_STOP_CLK;
57    PDC_mWriteSlaveReg15(XPAR_PDC_0_BASEADDR, 0, status);
58    xil_printf("[tsc] status: %d\t\n\r", status);
59  }
60
61
62  // a software toggle is required even though the hardware will toggle the clock once
63  void step(const int steps) {
64    Xuint32 status;
65
66    setStepCounter(steps);
67
68    // verify that we've properly set the number of steps
69    xil_printf("steps: %d\r\n", readStepCounter());
70
71    // set the counter to run
72    //setEnableStepCounterBit(1);
73
74    //run();
75
76    /*
77    // software implementation for stepping clock
78    int i;
79    for (i = 0; i < steps; i++)
80      toggleStopClock();
81
82    // status = PDC_mReadSlaveReg0(XPAR_PDC_0_BASEADDR, 0);
83    */
84
85  }
```

```
86
87  // set/unset the run bit
88  // setting the run bit ENABLES the run bit, but the design won't run if there are breakpoints
89  // read the status registers to figure out why
90  // IMPORTANT: anytime you want to switch back to running, you must pulse the stop clock AT LEAST once in order for the BUFGMUX
91  // to toggle
92  void setRunBit(const int run) {
93    Xuint32 status;
94
95    // read the register
96    status = PDC_mReadSlaveReg15(XPAR_PDC_0_BASEADDR, 0);
97
98    if (run) {
99      // set the run bit
100     status |= CTRL_RUN;
101     PDC_mWriteSlaveReg15(XPAR_PDC_0_BASEADDR, 0, status);
102     // in order for the BUFGMUX to switch inputs, the previous clock must toggle AT LEAST once more
103     toggleStopClock();
104   } else {
105     // check if it's set, then flip
106     if (status & CTRL_RUN) {
107       status ^= CTRL_RUN;
108       PDC_mWriteSlaveReg15(XPAR_PDC_0_BASEADDR, 0, status);
109     }
110     // else do nothing, it's already not set
111   }
112
113 }
114
115
116 void stop() {
117   setRunBit(0);
118 }
119
120
121 u32 readBreakpointMask() {
122   return PDC_mReadSlaveReg14(XPAR_PDC_0_BASEADDR, 0);
123 }
124
125 u32 readActiveBreakpointMask() {
126   return PDC_mReadSlaveReg11(XPAR_PDC_0_BASEADDR, 0);
127 }
128
129 void setBreakpointMask(u32 mask) {
130   PDC_mWriteSlaveReg14(XPAR_PDC_0_BASEADDR, 0, mask);
131 }
132
133 void enableBreakpoint(const Xuint32 idx) {
134   Xuint32 status = readBreakpointMask();
135   xil_printf("idx: %d status: %d\r\n", idx, status);
136
137   status |= (1 << idx);
138   xil_printf("new status: %d\r\n", status);
139   setBreakpointMask(status);
140 }
141
142 void disableBreakpoint(const Xuint32 idx) {
143   Xuint32 status = readBreakpointMask();
144
145   if (status & (1 << idx)) {
146     // the breakpoint is set, toggle that bit
147     status ^= (1 << idx);
148     setBreakpointMask(status);
149   }
150 }
151 /*
152 void createBreakpoint0(const Xuint32 breakValue) {
153   // this next line should use the signal value to determine which register to write to
154   PDC_mWriteSlaveReg1(XPAR_PDC_0_BASEADDR, 0, breakValue);
155
156   // new breakpoints are always enabled by default
157   // enableBreakpoint(signal);
158   enableBreakpoint(0);
159 }
160 */
161
162 // write a value into the step counter
163 void setStepCounter(const Xuint32 stepCounter) {
164   PDC_mWriteSlaveReg12(XPAR_PDC_0_BASEADDR, 0, stepCounter);
165 }
166
167 Xuint32 readStepCounter() {
168   return PDC_mReadSlaveReg12(XPAR_PDC_0_BASEADDR, 0);
169 }
170
171 // enable the step counter to run
172 void setEnableStepCounterBit(const int enable) {
```

```
173   Xuint32 status;
174
175   // read the register
176   status = PDC_mReadSlaveReg14(XPAR_PDC_0_BASEADDR, 0);
177
178   if (enable) {
179     // set the run bit
180     status |= ENABLE_STEP_CLK;
181   } else {
182     // check if it's set, then flip
183     if (status & ENABLE_STEP_CLK) {
184       status ^= ENABLE_STEP_CLK;
185     }
186     // else do nothing, it's already not set
187   }
188   PDC_mWriteSlaveReg14(XPAR_PDC_0_BASEADDR, 0, status);
189 }
190
191
192 Xuint32 readBreakpointStatus() {
193   return PDC_mReadSlaveReg13(XPAR_PDC_0_BASEADDR, 0);
194 }
195 /*
196 void disableBreakPoint(const Xuint32 signal) {
197
198 }
199
200 Xuint32 readBreakpoint0() {
201   return PDC_mReadSlaveReg1(XPAR_PDC_0_BASEADDR, 0);
202 }
203
204 Xuint32 readBreakpointEnables() {
205   return PDC_mReadSlaveReg14(XPAR_PDC_0_BASEADDR, 0);
206 }
207 */
208
209 Xuint32 readDut0() {
210   Xuint32 value;
211
212   value = PDC_mReadSlaveReg0(XPAR_PDC_0_BASEADDR, 0);
213   xil_printf("DUT0: %d\r\n", value);
214   value = PDC_mReadSlaveReg1(XPAR_PDC_0_BASEADDR, 0);
215   xil_printf("DUT1: %d\r\n", value);
216
217   return value;
218 }
219
220
221 int run() {
222
223   /*
224   // check if there are any active breakpoints
225   const Xuint32 breakStatus = readBreakpointStatus();
226
227   if (breakStatus)  {
228     // there's an active breakpoint
229     // tell the user and quit
230 #ifndef DMD_SERVER
231     xil_printf("There are active breakpoints.  Breakpoint status mask: %x\r\n", breakStatus);
232     print("You must step beyond the breakpoints or disable them to continue.\r\n");
233 #endif
234     return -1;
235   }
236   */
237   setRunBit(1);
238   return 0;
239 }
240
241
242 // continue the execution
243 // Hardware: a system clock switch is issued and then hardware breakpoints take effect
244 void cont() {
245   run();
246 }
247
248
249 #endif //#define _CMD_DBG_H
```

Listing B.8: ICAP Access Routines Header.

```
#ifndef _ICAP_H
#define _ICAP_H

#include "xparameters.h"
#include "xhwicap.h"    /* HWICAP device driver */
#include "xhwicap_i.h"

#define HWICAP_DEVICE_ID    XPAR_HWICAP_0_DEVICE_ID
#define READ_FRAME_SIZE 20

#define printf xil_printf

/*
 * These are the parameters for reading a frame of data in
 * the slice SLICE_X0Y0
 */
#define HWICAP_EXAMPLE_TOP    1
#define HWICAP_EXAMPLE_BLOCK    0
#define HWICAP_EXAMPLE_HCLK   1
#define HWICAP_EXAMPLE_MAJOR    1
#define HWICAP_EXAMPLE_MINOR    20


/************************ Variable Definitions *****************************/
static XHwIcap  HwIcap; /* The instance of the HWICAP device */
static u32 isHwIcapInit;
u32  FrameData[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];

void convertFrameAddressToParts(u32 frameAddress,
    u32 *block, u32 *top, u32 *rowAddress, u32 *majorAddress, u32 *minorAddress);

u32 readFrameOffset (const u32 frame, const u32 offset);

u32 initIcap(void);

int lld_icap_test (void);

#endif // _ICAP_H
```

Listing B.9: ICAP Access Routines Source.

```
#include "icap.h"

int lld_icap_test (void) {

   print("\n\n\n -- Hola mundo.  Zapatos. --\r\n");


  int Status;
  u32 Index;
  XHwIcap_Config *CfgPtr;
  u32 block, top, rowAddress, majorAddress, minorAddress;

{
  //u32  FrameData[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];

  /*
   * Initialize the HwIcap instance.
   */
  CfgPtr = XHwIcap_LookupConfig(HWICAP_DEVICE_ID);
  if (CfgPtr == NULL) {
    print("Failure to LC.\r\n");
    return XST_FAILURE;
  }

  Status = XHwIcap_CfgInitialize(&HwIcap, CfgPtr, CfgPtr->BaseAddress);
  if (Status != XST_SUCCESS) {
    print("Failure to init.\r\n");
    return XST_FAILURE;
  }

  /*
   * Perform a self-test to ensure that the hardware was built correctly.
   */
  Status = XHwIcap_SelfTest(&HwIcap);
  if (Status != XST_SUCCESS) {
    print("Failure to self-test.\r\n");
    return XST_FAILURE;
  }

  /*
   * Read the Frame
   */
   print("Reading frame.\r\n");
  #if ((XHI_FAMILY == XHI_DEV_FAMILY_V4) || (XHI_FAMILY == XHI_DEV_FAMILY_V5))
/*  Status = XHwIcap_DeviceReadFrame(&HwIcap,
            HWICAP_EXAMPLE_TOP,
            HWICAP_EXAMPLE_BLOCK,
            HWICAP_EXAMPLE_HCLK,
            HWICAP_EXAMPLE_MAJOR,
            HWICAP_EXAMPLE_MINOR,
            (u32 *) &FrameData[0]);
            */

  convertFrameAddressToParts(0x000088a3, &block, &top, &rowAddress, &majorAddress, &minorAddress);

  Status = XHwIcap_DeviceReadFrame(&HwIcap,
           top,
           block,
           rowAddress,
           majorAddress,
           minorAddress,
           (u32 *) &FrameData[0]);

  #endif
  if (Status != XST_SUCCESS) {
    printf("Failed to Read Frame: %d \n\r", Status);
    return XST_FAILURE;
  }

  /*
   * Print Frame contents
   */
  for (Index = XHI_NUM_FRAME_WORDS + 1;
    Index <= (XHI_NUM_FRAME_WORDS << 1) ; Index++) {

    printf("Frame Word %d -> \t %x \n\r",
      (Index - XHI_NUM_FRAME_WORDS), FrameData[Index]);
  }


  print("ICAP things: \r\n");
  printf("\tDeviceCode:\t%d\r\n", HwIcap.HwIcapConfig.DeviceId);
  printf("\tBaseAddress:\t0x%x\r\n", HwIcap.HwIcapConfig.BaseAddress);
  printf("\tIsReady:\t%d\r\n", HwIcap.IsReady);
  printf("\tIsPolled:\t%d\r\n", HwIcap.IsPolled);
```

```
86    printf("\tRows:\t%d\r\n", HwIcap.Rows);
87    printf("\tCols:\t%d\r\n", HwIcap.Cols);
88    printf("\tBramCols:\t%d\r\n", HwIcap.BramCols);
89    printf("\tBytesPerFrame:\t%d\r\n", HwIcap.BytesPerFrame);
90    printf("\tWordsPerFrame:\t%d\r\n", HwIcap.WordsPerFrame);
91    printf("\tClbBlockFrames:\t%d\r\n", HwIcap.ClbBlockFrames);
92    printf("\tBramBlockFrames:\t%d\r\n", HwIcap.BramBlockFrames);
93    printf("\tHClkRows:\t%d\r\n", HwIcap.HClkRows);
94    printf("\tDSPCols:\t%d\r\n", HwIcap.DSPCols);
95
96    print("Capturing ICAP.\r\n");
97    Status = XHwIcap_CommandCapture(&HwIcap);
98
99
100   u32 Packet;
101   u32 Data;
102   u32 TotalWords;
103 //  int Status;
104   u32 WriteBuffer[READ_FRAME_SIZE];
105   u32 FrameBuffer[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];
106 //  u32 Index = 0;
107 Index = 0;
108   /*
109   XASSERT_NONVOID(InstancePtr != NULL);
110   XASSERT_NONVOID(InstancePtr->IsReady == XCOMPONENT_IS_READY);
111   XASSERT_NONVOID(FrameBuffer != NULL);
112 */
113   /*
114    * DUMMY and SYNC
115    */
116   WriteBuffer[Index++] = XHI_DUMMY_PACKET;
117   WriteBuffer[Index++] = XHI_SYNC_PACKET;
118   WriteBuffer[Index++] = XHI_NOOP_PACKET;
119   WriteBuffer[Index++] = XHI_NOOP_PACKET;
120
121   /*
122    * Reset CRC
123    */
124   Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
125   WriteBuffer[Index++] = Packet;
126   WriteBuffer[Index++] = XHI_CMD_RCRC;
127   WriteBuffer[Index++] = XHI_NOOP_PACKET;
128   WriteBuffer[Index++] = XHI_NOOP_PACKET;
129
130   /*
131    * Setup CMD register to read configuration
132    */
133   Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
134   WriteBuffer[Index++] = Packet;
135   WriteBuffer[Index++] = XHI_CMD_RCFG;
136   WriteBuffer[Index++] = XHI_NOOP_PACKET;
137   WriteBuffer[Index++] = XHI_NOOP_PACKET;
138   WriteBuffer[Index++] = XHI_NOOP_PACKET;
139
140   /*
141    * Setup FAR register.
142    */
143   Packet = XHwIcap_Type1Write(XHI_FAR) | 1;
144
145   /*
146 #if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 * /
147   Data = XHwIcap_SetupFarV4(Top, Block, HClkRow,  MajorFrame, MinorFrame);
148 #elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 * /
149   Data = XHwIcap_SetupFarV5(Top, Block, HClkRow,  MajorFrame, MinorFrame);
150 #endif
151   */
152   Data = 0x000088a3;
153   WriteBuffer[Index++] = Packet;
154   WriteBuffer[Index++] = Data;
155
156   /*
157    * Setup read data packet header.
158    * The frame will be preceeded by a dummy frame, and we need to read one
159    * extra word - see Configuration Guide Chapter 8
160    */
161   TotalWords = (HwIcap.WordsPerFrame << 1) + 1;
162
163   /*
164    * Create Type one packet
165    */
166   Packet = XHwIcap_Type1Read(XHI_FDRO) | TotalWords;
167   WriteBuffer[Index++] = Packet;
168   WriteBuffer[Index++] = XHI_NOOP_PACKET;
169   WriteBuffer[Index++] = XHI_NOOP_PACKET;
170
171   /*
172    * Write the data to the FIFO and initiate the transfer of data
```

```
173     * present in the FIFO to the ICAP device
174     */
175    Status = XHwIcap_DeviceWrite(&HwIcap, (u32 *)&WriteBuffer[0], Index);
176    if (Status != XST_SUCCESS)  {
177      print("Device write failure.\r\n");
178      return XST_FAILURE;
179    }
180
181    /*
182     * Wait till the write is done.
183     */
184    print("Waiting...\r\n");
185    while (XHwIcap_IsDeviceBusy(&HwIcap) != FALSE);
186
187
188    /*
189     * Read the frame of the data including the NULL frame.
190     */
191    Status = XHwIcap_DeviceRead(&HwIcap, FrameBuffer, TotalWords);
192    if (Status != XST_SUCCESS)  {
193      return XST_FAILURE;
194    }
195
196    /*
197     * Send DESYNC command
198     */
199    Status = XHwIcap_CommandDesync(&HwIcap);
200    if (Status != XST_SUCCESS)  {
201      return XST_FAILURE;
202    }
203
204
205    for (Index = XHI_NUM_FRAME_WORDS + 1;
206      Index <= (XHI_NUM_FRAME_WORDS << 1) ; Index++) {
207
208      printf("Frame Word %d -> \t %x \n\r",
209        (Index - XHI_NUM_FRAME_WORDS), FrameBuffer[Index]);
210    }
211
212  }
213
214
215    printf("\n\rHwIcapReadFramePolledExample Passed Successfully.\n\r\n\r");
216
217  }
218
219
220  u32 readFrameOffset (const u32 frame, const u32 offset) {
221
222  #ifdef DMD_DEBUG
223     print("\n\n\n -- Entering readFrameOffset, zapatos --\r\n");
224  #endif
225
226    int Status;
227    u32 Index;
228    XHwIcap_Config *CfgPtr;
229    u32 block, top, rowAddress, majorAddress, minorAddress;
230
231    //u32  FrameData[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];
232
233    /*
234     * Initialize the HwIcap instance.
235     */
236    CfgPtr = XHwIcap_LookupConfig(HWICAP_DEVICE_ID);
237    if (CfgPtr == NULL) {
238      print("Failure to LC.\r\n");
239      return XST_FAILURE;
240    }
241
242    Status = XHwIcap_CfgInitialize(&HwIcap, CfgPtr, CfgPtr->BaseAddress);
243    if (Status != XST_SUCCESS) {
244      print("Failure to init.\r\n");
245      return XST_FAILURE;
246    }
247
248    /*
249     * Perform a self-test to ensure that the hardware was built correctly.
250     */
251    Status = XHwIcap_SelfTest(&HwIcap);
252    if (Status != XST_SUCCESS) {
253      print("Failure to self-test.\r\n");
254      return XST_FAILURE;
255    }
256
257    /*
258     * Read the Frame
259     */
```

```
260 #ifdef DMD_DEBUG
261    print("Reading frame.\r\n");
262 #endif
263
264 //#ifndef DMD_SERVER
265   #if ((XHI_FAMILY == XHI_DEV_FAMILY_V4) || (XHI_FAMILY == XHI_DEV_FAMILY_V5))
266   Status = XHwIcap_DeviceReadFrame(&HwIcap,
267            HWICAP_EXAMPLE_TOP,
268            HWICAP_EXAMPLE_BLOCK,
269            HWICAP_EXAMPLE_HCLK,
270            HWICAP_EXAMPLE_MAJOR,
271            HWICAP_EXAMPLE_MINOR,
272            (u32 *) &FrameData[0]);
273
274
275   convertFrameAddressToParts(frame, &block, &top, &rowAddress, &majorAddress, &minorAddress);
276
277 /*  Status = XHwIcap_DeviceReadFrame(&HwIcap,
278            top,
279            block,
280            rowAddress,
281            majorAddress,
282            minorAddress,
283            (u32 *) &FrameData[0]);
284   */
285   #endif
286 //#endif
287
288   if (Status != XST_SUCCESS) {
289     printf("Failed to Read Frame: %d \n\r", Status);
290     return XST_FAILURE;
291   }
292
293   /*
294    * Print Frame contents
295   */
296   for (Index = XHI_NUM_FRAME_WORDS + 1;
297     Index <= (XHI_NUM_FRAME_WORDS << 1) ; Index++) {
298
299     printf("Frame Word %d -> \t %x \n\r",
300       (Index - XHI_NUM_FRAME_WORDS), FrameData[Index]);
301   }
302
303 #ifndef DMD_SERVER
304   print("ICAP things: \r\n");
305   printf("\tDeviceCode:\t%d\r\n", HwIcap.HwIcapConfig.DeviceId);
306   printf("\tBaseAddress:\t0x%x\r\n", HwIcap.HwIcapConfig.BaseAddress);
307   printf("\tIsReady:\t%d\r\n", HwIcap.IsReady);
308   printf("\tIsPolled:\t%d\r\n", HwIcap.IsPolled);
309   printf("\tRows:\t%d\r\n", HwIcap.Rows);
310   printf("\tCols:\t%d\r\n", HwIcap.Cols);
311   printf("\tBramCols:\t%d\r\n", HwIcap.BramCols);
312   printf("\tBytesPerFrame:\t%d\r\n", HwIcap.BytesPerFrame);
313   printf("\tWordsPerFrame:\t%d\r\n", HwIcap.WordsPerFrame);
314   printf("\tClbBlockFrames:\t%d\r\n", HwIcap.ClbBlockFrames);
315   printf("\tBramBlockFrames:\t%d\r\n", HwIcap.BramBlockFrames);
316   printf("\tHClkRows:\t%d\r\n", HwIcap.HClkRows);
317   printf("\tDSPCols:\t%d\r\n", HwIcap.DSPCols);
318
319   print("Capturing ICAP.\r\n");
320 #endif
321   Status = XHwIcap_CommandCapture(&HwIcap);
322
323
324   u32 Packet;
325   u32 Data;
326   u32 TotalWords;
327 //  int Status;
328   u32 WriteBuffer[READ_FRAME_SIZE];
329   u32 FrameBuffer[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];
330 //  u32 Index = 0;
331 Index = 0;
332   /*
333   XASSERT_NONVOID(InstancePtr != NULL);
334   XASSERT_NONVOID(InstancePtr->IsReady == XCOMPONENT_IS_READY);
335   XASSERT_NONVOID(FrameBuffer != NULL);
336 */
337   /*
338    * DUMMY and SYNC
339    */
340   WriteBuffer[Index++] = XHI_DUMMY_PACKET;
341   WriteBuffer[Index++] = XHI_SYNC_PACKET;
342   WriteBuffer[Index++] = XHI_NOOP_PACKET;
343   WriteBuffer[Index++] = XHI_NOOP_PACKET;
344
345   /*
346    * Reset CRC
```

```
347     */
348    Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
349    WriteBuffer[Index++] = Packet;
350    WriteBuffer[Index++] = XHI_CMD_RCRC;
351    WriteBuffer[Index++] = XHI_NOOP_PACKET;
352    WriteBuffer[Index++] = XHI_NOOP_PACKET;
353
354    /*
355     * Setup CMD register to read configuration
356     */
357    Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
358    WriteBuffer[Index++] = Packet;
359    WriteBuffer[Index++] = XHI_CMD_RCFG;
360    WriteBuffer[Index++] = XHI_NOOP_PACKET;
361    WriteBuffer[Index++] = XHI_NOOP_PACKET;
362    WriteBuffer[Index++] = XHI_NOOP_PACKET;
363
364    /*
365     * Setup FAR register.
366     */
367    Packet = XHwIcap_Type1Write(XHI_FAR) | 1;
368
369    /*
370  #if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 * /
371    Data = XHwIcap_SetupFarV4(Top, Block, HClkRow,  MajorFrame, MinorFrame);
372  #elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 * /
373    Data = XHwIcap_SetupFarV5(Top, Block, HClkRow,  MajorFrame, MinorFrame);
374  #endif
375    */
376    Data = frame;
377    WriteBuffer[Index++] = Packet;
378    WriteBuffer[Index++] = Data;
379
380    /*
381     * Setup read data packet header.
382     * The frame will be preceeded by a dummy frame, and we need to read one
383     * extra word - see Configuration Guide Chapter 8
384     */
385    TotalWords = (HwIcap.WordsPerFrame << 1) + 1;
386
387    /*
388     * Create Type one packet
389     */
390    Packet = XHwIcap_Type1Read(XHI_FDRO) | TotalWords;
391    WriteBuffer[Index++] = Packet;
392    WriteBuffer[Index++] = XHI_NOOP_PACKET;
393    WriteBuffer[Index++] = XHI_NOOP_PACKET;
394
395    /*
396     * Write the data to the FIFO and initiate the transfer of data
397     * present in the FIFO to the ICAP device
398     */
399    Status = XHwIcap_DeviceWrite(&HwIcap, (u32 *)&WriteBuffer[0], Index);
400    if (Status != XST_SUCCESS)  {
401      print("Device write failure.\r\n");
402      return XST_FAILURE;
403    }
404
405    /*
406     * Wait till the write is done.
407     */
408  #ifdef DMD_DEBUG
409    print("Waiting...\r\n");
410  #endif
411    while (XHwIcap_IsDeviceBusy(&HwIcap) != FALSE);
412
413
414    /*
415     * Read the frame of the data including the NULL frame.
416     */
417    Status = XHwIcap_DeviceRead(&HwIcap, FrameBuffer, TotalWords);
418    if (Status != XST_SUCCESS)  {
419      return XST_FAILURE;
420    }
421
422    /*
423     * Send DESYNC command
424     */
425    Status = XHwIcap_CommandDesync(&HwIcap);
426    if (Status != XST_SUCCESS)  {
427      return XST_FAILURE;
428    }
429
430
431    for (Index = XHI_NUM_FRAME_WORDS + 1;
432      Index <= (XHI_NUM_FRAME_WORDS << 1) ; Index++) {
433  #ifndef DMD_SERVER
```

```
434       printf("Frame Word %d -> \t 0x%08x \n\r",(Index - XHI_NUM_FRAME_WORDS), FrameBuffer[Index]);
435   #endif
436
437     }
438
439     u32 targetWordIdx = (int)(offset / 32) + (XHI_NUM_FRAME_WORDS + 1);
440     u32 targetBitIdx  = offset % 32;
441
442     u32 result = (FrameBuffer[targetWordIdx] >> targetBitIdx) & 0x1;
443   #ifndef DMD_SERVER
444     printf("FrameBuffer[%d]: 0x%08x >> %d = 0x%x",
445       targetWordIdx- (XHI_NUM_FRAME_WORDS + 1), FrameBuffer[targetWordIdx], targetBitIdx, result);
446     printf("\n\rHwIcapReadFramePolledExample Passed Successfully.\n\r\n\r");
447   #else
448     printf("%x\n\r", result);
449   #endif
450
451
452     return result;
453
454   }
455
456   u32 initIcap(void) {
457   #ifdef DMD_DEBUG
458      print("\r\n\n\n -- Entering initIcap --\r\n");
459   #endif
460
461
462     XHwIcap_Config *CfgPtr;
463     int Status;
464
465     /*
466      * Lookup HwIcap handle and initialize it.
467      */
468     CfgPtr = XHwIcap_LookupConfig(HWICAP_DEVICE_ID);
469     if (CfgPtr == NULL) {
470       print("Failure to LC.\r\n");
471       return XST_FAILURE;
472     }
473
474     Status = XHwIcap_CfgInitialize(&HwIcap, CfgPtr, CfgPtr->BaseAddress);
475     if (Status != XST_SUCCESS) {
476       print("Failure to init.\r\n");
477       return XST_FAILURE;
478     }
479
480     /*
481      * Perform a self-test to ensure that the hardware was built correctly.
482      */
483     Status = XHwIcap_SelfTest(&HwIcap);
484     if (Status != XST_SUCCESS) {
485       print("Failure to self-test.\r\n");
486       return XST_FAILURE;
487     }
488
489
490   #ifdef DMD_DEBUG
491      print("\r\n\n\n -- Leaving initIcap --\r\n");
492   #endif
493
494   }
495
496   u32 readFrame (const u32 frame, u32 *FrameBuffer) {
497   #ifdef DMD_DEBUG
498      print("\r\n\n\n -- Entering readFrame, zapatos --\r\n");
499   #endif
500
501     int Status;
502     u32 Index;
503
504   /*
505     XHwIcap_Config *CfgPtr;
506   //  u32 block, top, rowAddress, majorAddress, minorAddress;
507
508     /*
509      * Lookup HwIcap handle and initialize it.
510      * /
511     CfgPtr = XHwIcap_LookupConfig(HWICAP_DEVICE_ID);
512     if (CfgPtr == NULL) {
513       print("Failure to LC.\r\n");
514       return XST_FAILURE;
515     }
516
517     Status = XHwIcap_CfgInitialize(&HwIcap, CfgPtr, CfgPtr->BaseAddress);
518     if (Status != XST_SUCCESS) {
519       print("Failure to init.\r\n");
520       return XST_FAILURE;
```

```
521     }
522
523     /*
524      * Perform a self-test to ensure that the hardware was built correctly.
525      * /
526     Status = XHwIcap_SelfTest(&HwIcap);
527     if (Status != XST_SUCCESS) {
528       print("Failure to self-test.\r\n");
529       return XST_FAILURE;
530     }
531   */
532     // initialize it the first time throught
533     if (!isHwIcapInit)  initIcap();
534
535     /*
536      * Read the Frame
537      */
538   #ifdef DMD_DEBUG
539      print("Reading frame.\r\n");
540   #endif
541
542     // issue a capture command to capture ff
543     Status = XHwIcap_CommandCapture(&HwIcap);
544
545     u32 Packet;
546     u32 Data;
547     u32 TotalWords;
548
549     u32 WriteBuffer[READ_FRAME_SIZE];
550     Index = 0;
551     /*
552      * DUMMY and SYNC
553      */
554     WriteBuffer[Index++] = XHI_DUMMY_PACKET;
555     WriteBuffer[Index++] = XHI_SYNC_PACKET;
556     WriteBuffer[Index++] = XHI_NOOP_PACKET;
557     WriteBuffer[Index++] = XHI_NOOP_PACKET;
558
559     /*
560      * Reset CRC
561      */
562     Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
563     WriteBuffer[Index++] = Packet;
564     WriteBuffer[Index++] = XHI_CMD_RCRC;
565     WriteBuffer[Index++] = XHI_NOOP_PACKET;
566     WriteBuffer[Index++] = XHI_NOOP_PACKET;
567
568     /*
569      * Setup CMD register to read configuration
570      */
571     Packet = XHwIcap_Type1Write(XHI_CMD) | 1;
572     WriteBuffer[Index++] = Packet;
573     WriteBuffer[Index++] = XHI_CMD_RCFG;
574     WriteBuffer[Index++] = XHI_NOOP_PACKET;
575     WriteBuffer[Index++] = XHI_NOOP_PACKET;
576     WriteBuffer[Index++] = XHI_NOOP_PACKET;
577
578     /*
579      * Setup FAR register.
580      */
581     Packet = XHwIcap_Type1Write(XHI_FAR) | 1;
582
583     // this is not necessary because we have the real frame address
584     /*
585   #if XHI_FAMILY == XHI_DEV_FAMILY_V4 /* Virtex 4 * /
586     Data = XHwIcap_SetupFarV4(Top, Block, HClkRow,  MajorFrame, MinorFrame);
587   #elif XHI_FAMILY == XHI_DEV_FAMILY_V5 /* Virtex 5 * /
588     Data = XHwIcap_SetupFarV5(Top, Block, HClkRow,  MajorFrame, MinorFrame);
589   #endif
590     */
591     Data = frame;
592     WriteBuffer[Index++] = Packet;
593     WriteBuffer[Index++] = Data;
594
595     /*
596      * Setup read data packet header.
597      * The frame will be preceeded by a dummy frame, and we need to read one
598      * extra word - see Configuration Guide Chapter 8
599      */
600     TotalWords = (HwIcap.WordsPerFrame << 1) + 1;
601
602     /*
603      * Create Type one packet
604      */
605     Packet = XHwIcap_Type1Read(XHI_FDRO) | TotalWords;
606     WriteBuffer[Index++] = Packet;
607     WriteBuffer[Index++] = XHI_NOOP_PACKET;
```

```
608     WriteBuffer[Index++] = XHI_NOOP_PACKET;
609
610     /*
611      * Write the data to the FIFO and initiate the transfer of data
612      * present in the FIFO to the ICAP device
613      */
614     Status = XHwIcap_DeviceWrite(&HwIcap, (u32 *)&WriteBuffer[0], Index);
615     if (Status != XST_SUCCESS)  {
616       print("Device write failure.\r\n");
617       return XST_FAILURE;
618     }
619
620     /*
621      * Wait till the write is done.
622      */
623   #ifdef DMD_DEBUG
624     print("Waiting...\r\n");
625   #endif
626     while (XHwIcap_IsDeviceBusy(&HwIcap) != FALSE);
627
628
629     /*
630      * Read the frame of the data including the NULL frame.
631      */
632     Status = XHwIcap_DeviceRead(&HwIcap, FrameBuffer, TotalWords);
633     if (Status != XST_SUCCESS)  {
634       return XST_FAILURE;
635     }
636
637     /*
638      * Send DESYNC command
639      */
640     Status = XHwIcap_CommandDesync(&HwIcap);
641     if (Status != XST_SUCCESS)  {
642       return XST_FAILURE;
643     }
644
645   #ifndef DMD_SERVER
646     for (Index = XHI_NUM_FRAME_WORDS + 1;
647       Index <= (XHI_NUM_FRAME_WORDS << 1) ; Index++) {
648       printf("Frame Word %d -> \t 0x%08x \n\r",(Index - XHI_NUM_FRAME_WORDS), FrameBuffer[Index]);
649     }
650
651   #endif
652
653     return XST_SUCCESS;
654   }
655
656
657   // Convert the given frame address to individual components.
658   void convertFrameAddressToParts(u32 frameAddress,
659       u32 *block, u32 *top, u32 *rowAddress, u32 *majorAddress, u32 *minorAddress) {
660
661     *block     = (frameAddress >> XHI_FAR_BLOCK_SHIFT)      && XHI_FAR_BLOCK_MASK;
662     *top     = (frameAddress >> XHI_FAR_TOP_BOTTOM_SHIFT) && XHI_FAR_TOP_BOTTOM_MASK;
663     *rowAddress  = (frameAddress >> XHI_FAR_ROW_ADDR_SHIFT)   && XHI_FAR_ROW_ADDR_MASK;
664     *majorAddress= (frameAddress >> XHI_FAR_COLUMN_ADDR_SHIFT)&& XHI_FAR_COLUMN_ADDR_MASK;
665     *minorAddress= (frameAddress >> XHI_FAR_MINOR_ADDR_SHIFT) && XHI_FAR_MINOR_ADDR_MASK;
666
667     xil_printf("\r\nblock: 0x%x\r\n", *block);
668     xil_printf("top: 0x%x\r\n", *top);
669     xil_printf("rowAddress: 0x%x\r\n", *rowAddress);
670     xil_printf("majorAddress: 0x%x\r\n", *majorAddress);
671     xil_printf("minorAddress: 0x%x\r\n", *minorAddress);
672   }
```

Listing B.10: LLD Workstation Console Source.

```
/*
  Yousef S. Iskander
  Shamelessly "adapted" from
  the GNU Readline library example "fileman.c"

  GNU Copyright from original file.
Copyright (C) 1987-2009 Free Software Foundation, Inc.

    This file is part of the GNU Readline Library (Readline), a library for
    reading lines of text with interactive input and history editing.

    Readline is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Readline is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Readline.  If not, see <http://www.gnu.org/licenses/>.
 */

#include "Breakpoint.h"

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <sys/types.h>
#ifdef HAVE_SYS_FILE_H
#  include <sys/file.h>
#endif
#include <sys/stat.h>

#ifdef HAVE_UNISTD_H
#  include <unistd.h>
#endif

#include <fstream>
#include <fcntl.h>
#include <cstdio>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <errno.h>

#if defined (HAVE_STRING_H)
#  include <string.h>
#else /* !HAVE_STRING_H */
#  include <strings.h>
#endif /* !HAVE_STRING_H */

//#ifdef HAVE_STDLIB_H
#  include <stdlib.h>
//#endif

#include <readline/readline.h>
#include <readline/history.h>
#include <vector>
#include <map>
#include "BitInfo.h"
#include "serial.h"

using namespace std;

void initialize_readline();
char *command_generator PARAMS((const char *, int));
char **fileman_completion PARAMS((const char *, int, int));
char *stripwhite (char *string);
int execute_line (char *line);

bool bitInfoSignalIndexReverseSort(const BitInfo& a, const BitInfo& b);
std::string convertIntToStr(const uint32_t num);

void tokenize(const std::string& str, std::vector<string>& tokens, const std::string& delimiters = " ");

/* The names of functions that actually do the manipulation. */
int com_run PARAMS((char *));
int com_stop PARAMS((char *));
int com_step PARAMS((char *));
int com_reset PARAMS((char *));
```

```
86  int com_walk PARAMS((char *));
87  int com_source PARAMS((char *));
88  int com_connect PARAMS((char *));
89  int com_disconnect PARAMS((char *));
90  int com_continue PARAMS((char *));
91  int com_constrain PARAMS((char *));
92  int com_break PARAMS((char *));
93  int com_delete PARAMS((char *));
94  int com_info PARAMS((char *));
95  int com_disable PARAMS((char *));
96  int com_enable PARAMS((char *));
97  int com_print PARAMS((char *));
98  int com_loadlogic PARAMS((char *));
99  int com_addport PARAMS((char *));
100 int com_delport PARAMS((char *));
101 int com_display PARAMS((char *));
102 int com_undisplay PARAMS((char *));
103 int com_clear PARAMS((char *));
104 int com_ping PARAMS((char *));
105 int com_help PARAMS((char *));
106 int com_quit PARAMS((char *));
107
108 void enableBreakpoint(const uint32_t idx);
109 void disableBreakpoint(const uint32_t idx);
110 bool isBreakpointEnabled(const uint32_t idx);
111 uint32_t getBreakpointMask();
112 uint32_t getActiveBreakpointMask();
113 void breakpointCommand(const std::string &command, const uint32_t idx);
114 void makeBreakpoint(char *arg, const breakpoint_t breakpointType);
115 void createDebugLogic();
116 std::string sendRemoteCommand(const std::string &command);
117 char *signalname_generator(const char *text, int state);
118 void createPrintList(std::list<std::string> &frameList, std::string &printArgs) ;
119
120 // Globals
121 int fd;       // serial port handle, -1 is not connected
122 BitMap bitInfoMap;  // BitMap for logic allocations
123
124 /* Breakpoint storage */
125 std::vector<Breakpoint> breakpoints;
126
127 /* Port storage */
128 std::map<std::string, uint8_t> ports;
129
130 /* A structure which contains information on the commands this program
131    can understand. */
132
133 typedef struct {
134     char *name;     /* User printable name of the function. */
135     rl_icpfunc_t *func;   /* Function to call to do the job. */
136     char *doc;      /* Documentation for this function.  */
137 } COMMAND;
138
139 COMMAND commands[] = {
140     { (char*)"run", com_run, (char*)"Run the design with the primary system clock" },
141     { (char*)"stop", com_stop, (char*)"Stop execution" },
142     { (char*)"step", com_step, (char*)"Step the design a given number of clock cycles" },
143     { (char*)"reset", com_reset, (char*)"Inititate a design reset" },
144     { (char*)"walk", com_walk, (char*)"Automatically issue time single step commands" },
145     { (char*)"connect", com_connect, (char*)"Connect to the development board using the given device" },
146     { (char*)"disconnect", com_disconnect, (char*)"Disconnect from the development board." },
147     { (char*)"continue", com_continue, (char*)"Continue executing the design from the current state" },
148     { (char*)"constrain", com_constrain, (char*)"Build logic to monitor signals not to violate the given constraints" },
149     { (char*)"break", com_break, (char*)"Specify hardware breakpoints with the given conditions" },
150     { (char*)"delete", com_delete, (char*)"Delete a breakpoint" },
151     { (char*)"info", com_info, (char*)"Print information regarding breakpoints, display statements, etc." },
152     { (char*)"source", com_source, (char*)"Read commands from a file." },
153     { (char*)"disable", com_disable, (char*)"Disable the specified breakpoint" },
154     { (char*)"enable", com_enable, (char*)"Enable the specified breakpoint" },
155     { (char*)"print", com_print, (char*)"\n\t\tprint [-v | --verbose] <net> Print the current value of the register" },
156     { (char*)"display", com_display, (char*)"Specify registers to automatically print after each step" },
157     { (char*)"undisplay", com_undisplay, (char*)"Remove registers from the queue that automatically print" },
158     { (char*)"addport", com_addport, (char*)"Add a port with the given name and size" },
159     { (char*)"delport", com_delport, (char*)"Remove a port with the given name" },
160     { (char*)"load-logic", com_loadlogic, (char*)"Load the named logic allocation file and create a register model" },
161     { (char*)"clear", com_clear, (char*)"Clear the screen" },
162     { (char*)"ping", com_ping, (char*)"Ping the remote system" },
163     { (char*)"help", com_help, (char*)"Display this text" },
164     { (char*)"?", com_help, (char*)"Synonym for `help'" },
165     { (char*)"quit", com_quit, (char*)"Quit the application and exit" },
166     { (char*)"q", com_quit, (char*)"Synonym for quit" },
167     { (char *)NULL, (rl_icpfunc_t *)NULL, (char *)NULL }
168 };
169
170 /* Forward declarations. */
171 COMMAND *find_command (char *name);
172
```

```
173  /* The name of this program, as taken from argv[0]. */
174  char *progname;
175
176  /* When non-zero, this global means the user is done using this program. */
177  int done;
178
179  /* Copy a string into heap and return a pointer */
180  char *dupstr (char *s)
181  {
182      char *r;
183
184      r = (char*)malloc (strlen (s) + 1);
185      strcpy (r, s);
186      return (r);
187  }
188
189  int main (int argc, char **argv)
190  {
191      char *line, *s;
192
193      fd = -1;    // we are not yet connected
194
195      progname = argv[0];
196
197      initialize_readline (); /* Bind our completer. */
198
199      /* Loop reading and executing lines until the user quits. */
200      for ( ; done == 0; )
201      {
202          line = readline ("dmd> ");
203
204          if (!line)
205              break;
206
207          /* Remove leading and trailing whitespace from the line.
208             Then, if there is anything left, add it to the history list
209             and execute it. */
210          s = stripwhite (line);
211
212          if (*s)
213          {
214              add_history (s);
215              execute_line (s);
216          }
217
218          free (line);
219      }
220      exit (0);
221  }
222
223  /* Execute a command line. */
224  int execute_line (char *line)
225  {
226      register int i;
227      COMMAND *command;
228      char *word;
229
230      /* Isolate the command word. */
231      i = 0;
232      while (line[i] && whitespace (line[i]))
233          i++;
234      word = line + i;
235
236      while (line[i] && !whitespace (line[i]))
237          i++;
238
239      if (line[i])
240          line[i++] = '\0';
241
242      command = find_command (word);
243
244      if (!command)
245      {
246          fprintf (stderr, "%s: No such command for DMD.\n", word);
247          return (-1);
248      }
249
250      /* Get argument to command, if any. */
251      while (whitespace (line[i]))
252          i++;
253
254      word = line + i;
255
256      /* Call the function. */
257      return ((*(command->func)) (word));
258  }
259
```

```
260 /* Look up NAME as the name of a command, and return a pointer to that
261    command.  Return a NULL pointer if NAME isn't a command name. */
262 COMMAND *find_command (char *name)
263 {
264     register int i;
265
266     for (i = 0; commands[i].name; i++)
267         if (strcmp (name, commands[i].name) == 0)
268             return (&commands[i]);
269
270     return ((COMMAND *)NULL);
271 }
272
273 /* Strip whitespace from the start and end of STRING.  Return a pointer
274    into STRING. */
275 char *stripwhite (char *string)
276 {
277     register char *s, *t;
278
279     for (s = string; whitespace (*s); s++)
280         ;
281
282     if (*s == 0)
283         return (s);
284
285     t = s + strlen (s) - 1;
286     while (t > s && whitespace (*t))
287         t--;
288     *++t = '\0';
289
290     return s;
291 }
292
293 /* ************************************************************** */
294 /*                                                                */
295 /*                  Interface to Readline Completion              */
296 /*                                                                */
297 /* ************************************************************** */
298
299 /* Tell the GNU Readline library how to complete.  We want to try to complete
300    on command names if this is the first word in the line, or on filenames
301    if not. */
302 void initialize_readline ()
303 {
304     /* Allow conditional parsing of the ~/.inputrc file. */
305     rl_readline_name = "dmd";
306
307     /* Tell the completer that we want a crack first. */
308     rl_attempted_completion_function = fileman_completion;
309 }
310
311 /* Attempt to complete on the contents of TEXT.  START and END bound the
312    region of rl_line_buffer that contains the word to complete.  TEXT is
313    the word to complete.  We can use the entire contents of rl_line_buffer
314    in case we want to do some simple parsing.  Return the array of matches,
315    or NULL if there aren't any. */
316 char **fileman_completion (const char *text, int start, int end)
317 {
318     char **matches;
319
320     matches = (char **)NULL;
321
322     /* If this word is at the start of the line, then it is a command
323        to complete.  Otherwise it is the name of a file in the current
324        directory.
325        It might also be the name of a signal.
326      */
327
328     if (start == 0)
329         matches = rl_completion_matches (text, command_generator);
330     else
331         matches = rl_completion_matches (text, signalname_generator);
332
333     return (matches);
334 }
335
336 /* Generator function for command completion.  STATE lets us know whether
337    to start from scratch; without any state (i.e. STATE == 0), then we
338    start at the top of the list. */
339 char *command_generator (const char *text, int state)
340 {
341     static int list_index, len;
342     char *name;
343
344     /* If this is a new word to complete, initialize now.  This includes
345        saving the length of TEXT for efficiency, and initializing the index
346        variable to 0. */
```

```
347
348     if (!state)
349     {
350         list_index = 0;
351         len = strlen (text);
352     }
353
354     /* Return the next name which partially matches from the command list. */
355     while (name = commands[list_index].name)
356     {
357         list_index++;
358
359         if (strncmp (name, text, len) == 0)
360             return (dupstr(name));
361     }
362
363     /* If no names matched, then return NULL. */
364     return ((char *)NULL);
365 }
366
367 char *signalname_generator(const char *text, int state)
368 {
369     static BitMap::iterator bitMapIt;
370     static int len;
371     //list_index, len;
372     char *name;
373
374     /* If this is a new word to complete, initialize now.  This includes
375        saving the length of TEXT for efficiency, and initializing the index
376        variable to 0. */
377     if (!state)
378     {
379         bitMapIt = bitInfoMap.begin();
380         //        list_index = 0;
381         len = strlen (text);
382     }
383
384     /* Return the next name which partially matches from the command list. */
385     //while (name = commands[list_index].name)
386     while (bitMapIt != bitInfoMap.end())
387     {
388         //        list_index++;
389         name = const_cast<char*>(bitMapIt->first.c_str());
390         bitMapIt++;
391         if (strncmp (name, text, len) == 0)
392             return (dupstr(name));
393     }
394
395     /* If no names matched, then return NULL. */
396     return ((char *)NULL);
397 }
398 /* ********************************************************************** */
399 /*                                                                        */
400 /*                       DMD Commands                                     */
401 /*                                                                        */
402 /* ********************************************************************** */
403
404 int com_run (char *arg) {
405     printf("Starting the design. Type 'stop' to stop execution.\n");
406     std::string result = sendRemoteCommand("run");
407     cout << result << endl;
408 }
409 int com_stop (char *arg) {
410     printf("Design stopped.\n");
411     std::string result = sendRemoteCommand("stop");
412     cout << result << endl;
413 }
414
415 int com_step (char *arg) {
416
417     //const uint8_t steps = atoi(arg);
418     std::string command;
419     command.append("step ");
420     command.append(arg);
421
422     cout << sendRemoteCommand(command);
423 }
424
425 int com_reset (char *arg) {
426     printf("toggle reset line\n");
427 }
428
429 int com_walk (char *arg) {
430
431 }
432
433 int com_source(char *arg) {
```

```
434      char *s;
435      // get the file name and open a stream
436      FILE *sourceFile = fopen(arg, "r");
437
438      if (sourceFile != NULL) {
439          char *s;
440          char line [128];
441          uint32_t lineNumber = 1;
442
443          // read each line and submit to the command processor
444          while ( fgets ( line, sizeof line, sourceFile ) != NULL )
445          {
446  //            printf("Got: %s", line);
447              s = stripwhite(line);
448              if (*s) {
449                  // strip trailing \n
450                  if (s[strlen(s) - 1] == '\n') {
451                      s[strlen(s) - 1] = '\0';
452                  }
453
454                  // if there's still something left..
455                  if (strlen(s)) {
456                      // if the command fails, just exit
457                      if (execute_line(s) == -1) {
458                          printf("Execution failed on line %d: %s\n", lineNumber, s);
459                          break;
460                      }
461                  }
462              }
463              lineNumber++;
464          }
465          fclose ( sourceFile );
466
467      } else {
468          printf("There was a problem opening file: %s.\n", arg);
469          return -1;
470      }
471
472  }
473
474
475  int com_connect (char *arg) {
476      printf("Connecting on %s...", arg);
477
478      fd = setup_serial(arg);
479
480      if (fd != -1)
481        printf("done.\n");
482      else
483        printf("Problem connecting.\n");
484  }
485
486  int com_disconnect (char *arg) {
487      printf("Disconnecting from %s\n", arg);
488
489      close_serial(fd);
490  }
491
492  int com_continue (char *arg) {
493      printf("continue execution\n");
494
495      std::string result = sendRemoteCommand("continue");
496      cout << result << endl;
497  }
498
499  int com_constrain (char *arg) {
500      makeBreakpoint(arg, CONSTRAINT);
501      createDebugLogic();
502  }
503
504  int com_break (char *arg) {
505      makeBreakpoint(arg, BREAKPOINT);
506      createDebugLogic();
507  }
508
509  int com_delete(char *arg) {
510      const uint8_t breakpointNumber = atoi(arg);
511
512      // rather than erase, just reset the breakpoint
513      // slot to 1'b0 and disable in hardware
514      Breakpoint *bp = &breakpoints[breakpointNumber];
515      bp->breakpointText = "1'b0";
516
517      //breakpoints.erase(breakpoints.begin()+breakpointNumber);
518
519      // renumber the remaining breakpoints
520      /*
```

```
521      for( int i=0; i<breakpoints.size(); i++) {
522          Breakpoint *bp = &breakpoints[i];
523          bp->idx = i;
524      }*/
525
526      // disable this breakpoint
527      disableBreakpoint(breakpointNumber);
528      createDebugLogic();
529  }
530
531  /*
532  int com_delete_old(char *arg) {
533      const uint8_t breakpointNumber = atoi(arg);
534
535      breakpoints.erase(breakpoints.begin()+breakpointNumber);
536
537      // renumber the remaining breakpoints
538      for( int i=0; i<breakpoints.size(); i++) {
539          Breakpoint *bp = &breakpoints[i];
540          bp->idx = i;
541      }
542
543      createDebugLogic();
544  }
545  */
546
547  int com_loadlogic(char *arg) {
548      const std::string logicAllocFile = arg;
549
550      // parse the logic allocation model
551      parseLogicAllocFile(logicAllocFile, bitInfoMap);
552
553      cout << bitInfoMap.size() << " components available." << endl;
554  }
555
556
557  void makeBreakpoint(char *arg, const breakpoint_t breakpointType) {
558      // first scan the vector for the greatest index number (always at the end)
559      uint16_t nextIdx;
560      if (breakpoints.size())
561          nextIdx = breakpoints[breakpoints.size()-1].idx + 1;
562      else
563          nextIdx = 0;
564
565      Breakpoint bp;
566      bp.idx = nextIdx;
567      //bp.isEnabled = true;
568      bp.breakpointText = arg;
569      bp.breakpointType = breakpointType;
570
571      // add this breakpoint to the vector
572      breakpoints.push_back(bp);
573
574      // enable this breakpoint
575      enableBreakpoint(nextIdx);
576
577      printf("Breakpoint %d: %s\n", nextIdx, arg);
578  }
579
580  void enableBreakpoint(const uint32_t idx) {
581    const std::string command = "enable ";
582    breakpointCommand(command, idx);
583  }
584
585  void disableBreakpoint(const uint32_t idx) {
586    const std::string command = "disable ";
587    breakpointCommand(command, idx);
588  }
589
590  void breakpointCommand(const std::string &command, const uint32_t idx) {
591
592      if (idx >= 0 && idx < 32) {
593          std::string resultStr = sendRemoteCommand(command + convertIntToStr(idx));
594          cout << resultStr << endl;
595      } else {
596        cout << "Breakpoint index out of range [0-31]." << endl;
597      }
598  }
599
600
601  int com_info (char *arg) {
602
603      if (!strcmp(arg,"breakpoints")) {
604          if (!breakpoints.size()) {
605              printf("No breakpoints exist.\n");
606              return 0;
607          } else {
```

```
608             // populate a mask of enabled breakpoints
609             uint32_t breakpointMask = getBreakpointMask();
610             uint32_t activeBreakpointMask = getActiveBreakpointMask();
611 
612             // print a list of the breakpoints
613             printf("Num\tType\t\tEnb\tActive\tWhat\n");
614 
615             for (int i = 0; i < breakpoints.size(); i++) {
616                 const Breakpoint bp = breakpoints[i];
617 
618                 // get a breakpoint type
619                 const std::string breakpointType = bp.getBreakpointType();
620                 const std::string isEnabled = ((breakpointMask >> i) & 0x1)?"y":"n"; // bp.isEnabled?"true":"false";
621                 const std::string isActive = ((activeBreakpointMask >> i) & 0x1)?"y":"n"; // bp.isEnabled?"true":"false";
622                 printf("%d\t%s\t%s\t%s\t%s\n",
623                     bp.idx, breakpointType.c_str(), isEnabled.c_str(), isActive.c_str(), bp.breakpointText.c_str());
624             }
625         }
626     } else if (!strcmp(arg,"ports")) {
627         if (!ports.size()) {
628             printf("No ports exist.\n");
629         } else {
630             printf("Name\t\tSize\n");
631 
632             // print a list of ports
633             std::map<std::string, uint8_t>::iterator it;
634             for (it = ports.begin(); it != ports.end(); ++it) {
635                 const std::string portName = it->first;
636                 const uint8_t portSize = it->second;
637 
638                 printf("%s\t\t%d\n", portName.c_str(), portSize);
639             }
640         }
641     }
642 }
643 /*
644 int com_disable_old (char *arg) {
645     printf("disable a breakpoint\n");
646 
647     // change the argument into a int
648     const uint16_t idx = atoi(arg);
649     bool isFound = false;
650 
651     // search the breakpoint vector for the specified idx
652     for (int i = 0; i < breakpoints.size(); i++) {
653         Breakpoint *bp = &breakpoints[i];
654         if (bp->idx == idx) {
655             //bp->isEnabled = false;
656             isFound = true;
657             break;
658         }
659     }
660 
661     if (!isFound)
662         printf("No breakpoint number %d.\n", idx);
663 
664     createDebugLogic();
665 }
666 */
667 int com_disable (char *arg) {
668     const std::string command = "disable ";
669 
670     uint32_t idx;
671     int result = sscanf(arg, "%u", &idx);
672 
673     if (result) {
674       disableBreakpoint(idx);
675     } else {
676       cout << "Problem with input." << endl;
677     }
678 }
679 
680 int com_enable (char *arg) {
681     const std::string command = "enable ";
682 
683     uint32_t idx;
684     int result = sscanf(arg, "%u", &idx);
685 
686     if (result) {
687       enableBreakpoint(idx);
688     } else {
689       cout << "Problem with input." << endl;
690     }
691 }
692 
693 /*
694 int com_enable_old (char *arg) {
```

```
695         printf("enable a breakpoint\n");
696
697         // change the argument into a int
698         const uint16_t idx = atoi(arg);
699         bool isFound = false;
700
701         // search the breakpoint vector for the specified idx
702         for (int i = 0; i < breakpoints.size(); i++) {
703             Breakpoint *bp = &breakpoints[i];
704             if (bp->idx == idx) {
705                 // bp->isEnabled = true;
706                 isFound = true;
707                 break;
708             }
709         }
710
711         if (!isFound)
712             printf("No breakpoint number %d.\n", idx);
713
714         createDebugLogic();
715     }
716     */
717
718     int com_print (char *arg) {
719         const uint32_t MAX_FRAME_LIST = 16;
720         // get the target name
721         std::string target;
722         std::vector<std::string> args;
723         bool verbose = false;
724
725         tokenize(std::string(arg), args, " \t");
726
727         // if there are >=2 arguments, the first must be an option, the last the signal name
728         if (args.size() > 1) {
729             // we have options
730             const std::string option = args[0];
731
732             verbose = (option == "-v" || option == "--verbose");
733             target = args[1];
734         } else {
735             target = args[0];
736         }
737
738
739
740         // look up the target
741         BitMap::iterator it = bitInfoMap.find(target);
742         if (it == bitInfoMap.end()) {
743             cout << "Signal \"" << target << "\" was not found." << endl;
744             return 0;
745         }
746
747
748         // we have a legitimate target
749         // get and sort the list and send the frame/index pairs to MicroBlaze
750         BitInfoList bil = it->second;
751         bil.sort(bitInfoSignalIndexReverseSort);
752
753         uint32_t registerValue = 0; // result to be returned
754         uint32_t previousFrame = 0; // keep track of frame from previous round
755         uint32_t currentFrame  = 0;
756         const std::string command("print "); // command to send
757         std::list<std::string> frameList; // the argument list to the print command
758
759         // start out the iterator and grab the first BitInfo
760         BitInfoList::iterator bilIt = bil.begin();
761         BitInfo *bi = &(*bilIt);
762
763         if (verbose)
764             cout << "Processing bit " << bi->signalVectorIndex << endl;
765
766         currentFrame = bi->frameAddress;
767
768         // append the current frame offset to the arg list
769         frameList.push_back(convertIntToStr(bi->frameOffset));
770
771         // copy the current frame to be the previous
772         previousFrame = currentFrame;
773
774         // advance the iterator
775         bilIt++;
776
777         // initializer intentionally left blank, previously advanced
778         for( ; bilIt != bil.end(); ++bilIt) {
779             bi = &(*bilIt);
780
781             if (verbose)
```

```
782             cout << "Processing bit " << bi->signalVectorIndex << endl;
783
784         currentFrame = bi->frameAddress;
785
786         if ((currentFrame != previousFrame) || frameList.size() == MAX_FRAME_LIST) {
787             // it's a new frame or the list is getting too big and it's time to empty
788             // send the pending command and add to the result...
789
790             // format the print command arguments, but first add the frame as the first argument
791             std::string printArgs;
792             printArgs.append(convertIntToStr(previousFrame));
793             createPrintList(frameList, printArgs);
794
795             if (verbose) {
796                 cout << "Sending remote command: " << command << " " << printArgs << endl;
797             }
798
799             // process the result
800             std::string resultStr = sendRemoteCommand(command + printArgs);
801
802
803             // read using sscanf because it's more tolerant of whitespace (atoi will error)
804             uint32_t resultNum;
805             sscanf(resultStr.c_str(), "%x", &resultNum);
806             registerValue = (registerValue << frameList.size()) | resultNum;
807
808             if (verbose)
809                 printf("Response: %5s %#08x Current value: %#08x\n", resultStr.c_str(), resultNum, registerValue);
810
811
812             // clear the frameList...
813             frameList.clear();
814             printArgs.clear();
815
816         }
817
818         // append the frame offset as part of the current frame
819         frameList.push_back(convertIntToStr(bi->frameOffset));
820
821
822         // issue the command and get the result, shift into register
823         //        std::string result = sendRemoteCommand(command);
824         //  cout << command << " -> " << result << endl;
825
826         //    cout << "Register = " << registerValue << endl;
827
828         //  cout << bi->frameAddress << " " << bi->frameOffset << " [" << bi->signalVectorIndex << "] " << bi->latch  << endl;
829         //    cout << (*bilIt).signalVectorIndex << endl;
830
831         previousFrame = currentFrame;
832
833     }
834
835     // send the final command
836     // format the print command arguments
837     std::string printArgs;
838     printArgs.append(convertIntToStr(previousFrame));
839     createPrintList(frameList, printArgs);
840
841     if (verbose) {
842         cout << "Sending remote command: " << command << " " << printArgs << endl;
843     }
844
845     // process the result
846     std::string result = sendRemoteCommand(command + printArgs);
847
848     // cout << endl << command + printArgs << endl;
849     //  cout << "regValue (before final) = " << hex << registerValue << " frameList.size() = " << dec << frameList.size() << endl;
850     //  cout << "result (str)'" << result << "' result (atoi) " << atoi(result.c_str()) << endl;
851     //cout << "Result = '" << result << "' registerVal = " << registerValue << endl;
852
853     uint32_t resultNum;
854     sscanf(result.c_str(), "%x", &resultNum);
855     registerValue = (registerValue << frameList.size()) | resultNum;
856
857     if (verbose)
858         printf("Response: %5s %#08x Current value: %#08x\n", result.c_str(), resultNum, registerValue);
859
860     // the result must be inverted first before being used
861     // however we did this on the MBlz because short vectors were being corrupted
862     //  cout << "regValue (after final) = " << hex << registerValue << " frameList.size() = " << dec << frameList.size() << endl;
863
864     //  registerValue = ~registerValue;
865     printf("%#8x (unsigned) %u\n", registerValue, registerValue);
866     //  cout << hex << registerValue << endl;
867 }
868
```

```
869 int com_display (char *arg) {
870
871 }
872
873 int com_undisplay (char *arg) {
874
875 }
876
877 int com_addport (char *arg) {
878
879     // convert to C++ string to make this work
880     std::string argument = arg;
881     std::size_t spaceIdx = argument.find_first_of(" ");
882     std::string portName = argument.substr(0, spaceIdx);
883     uint8_t portSize = atoi(argument.substr(spaceIdx+1, argument.length()-spaceIdx).c_str());
884
885     if (portSize == 0)
886         ports[portName] = 1;
887     else
888         ports[portName] = portSize;
889
890     printf("Added port %s with size %d\n", portName.c_str(), portSize);
891     createDebugLogic();
892 }
893
894 int com_delport (char *arg) {
895     const std::string portName = arg;
896     ports.erase(portName);
897     createDebugLogic();
898 }
899
900
901 /* Print out help for ARG, or for all of the commands if ARG is
902    not present. */
903 int com_help (char *arg)
904 {
905     register int i;
906     int printed = 0;
907
908     for (i = 0; commands[i].name; i++)
909     {
910         if (!*arg || (strcmp (arg, commands[i].name) == 0))
911         {
912             printf ("%s -- %s.\n", commands[i].name, commands[i].doc);
913             printed++;
914         }
915     }
916
917     if (!printed)
918     {
919         printf ("No commands match `%s'.  Possibilties are:\n", arg);
920
921         for (i = 0; commands[i].name; i++)
922         {
923             /* Print in six columns. */
924             if (printed == 6)
925             {
926                 printed = 0;
927                 printf ("\n");
928             }
929
930             printf ("%s\t", commands[i].name);
931             printed++;
932         }
933
934         if (printed)
935             printf ("\n");
936     }
937
938     return (0);
939 }
940 /* The user wishes to quit using this program.  Just set DONE non-zero. */
941 int com_quit (char *arg)
942 {
943     done = 1;
944
945     // if the serial port is still open, close it
946     if (fd != -1) close_serial(fd);
947
948     return (0);
949 }
950
951
952 /* Return non-zero if ARG is a valid argument for CALLER, else print
953    an error message and return zero. */
954 int valid_argument (char *caller, char *arg)
955 {
```

```
956     if (!arg || !*arg)
957     {
958         fprintf (stderr, "%s: Argument required.\n", caller);
959         return (0);
960     }
961 
962     return (1);
963 }
964 
965 // to be fired after each breakpoint creation
966 // creates a module within a Verilog file that has the ports
967 // for the nets to be monitored and the debug logic
968 void createDebugLogic() {
969     std::ofstream debugFile("dmd_debug_logic.v");
970 
971     debugFile << "\n\nmodule dmd_debug_logic \n"
972         << "(\n\t// design input ports\n";
973 
974     // print a list of ports
975     std::map<std::string, uint8_t>::iterator it;
976     for (it = ports.begin(); it != ports.end(); ++it) {
977         debugFile << "\t" << it->first << ",\n";
978     }
979 
980     // static control lines
981     debugFile << "\n";
982     debugFile << "\t // control lines\n";
983     debugFile << "\tbreakpoint_active,\n\tbreakpoint_reg\n);\n\n";
984 
985     // port declarations with sizes
986     for (it = ports.begin(); it != ports.end(); ++it) {
987         const uint16_t portMax = it->second-1;
988 
989         if (portMax)
990             debugFile << "\tinput\t\t[0 : " << portMax << "]\t\t" << it->first << ";\n";
991         else
992             debugFile << "\tinput\t\t\t\t\t\t" << it->first << ";\n";
993     }
994     debugFile << "\toutput\t\t\t\t\t\tbreakpoint_active;\n"
995         << "\toutput\t\t[0 : 31]\t\tbreakpoint_reg;\n\n";
996 
997     debugFile << "\twire\t\t[0 : 31]\t\tbreakpoints;\n\n";
998 
999     debugFile << "\t// assign the internal breakpoint register array to the output\n";
1000    debugFile << "\tassign breakpoint_reg = breakpoints;\n\n";
1001 
1002    // create the breakpoint active/interrupt signal
1003    debugFile << "\tassign breakpoint_active = |breakpoints;\n\n";
1004 
1005    // cycle through the breakpoint vector and generate breakpoints and constraints
1006    //    * breakpoints are written verbatim
1007    //    * constraints are written with an inverter (trigger if violated)
1008 
1009    for (int i = 0; i < breakpoints.size(); i++) {
1010        const Breakpoint *bp = &breakpoints[i];
1011 
1012        // if the breakpoint is a constraint, preceed the expression with an inversion
1013        // if the breakpoint is disabled, write it in, it's still valid, but disabled in hardware
1014        debugFile << "\tassign breakpoints[" << bp->idx << "] = ";
1015 
1016        //if (!bp->isEnabled)
1017        //    debugFile << " 1'b0;  // disabled " << bp->getBreakpointType() << ": " << bp->breakpointText;
1018        //else
1019            debugFile << ((bp->breakpointType == CONSTRAINT) ? "~" : " ")
1020                << bp->breakpointText << ";" << "  // " << bp->getBreakpointType();
1021 
1022        debugFile << "\n";
1023    }
1024 
1025    // we must pad the rest of the breakpoints register with constant 0 to make
1026    // a good simulation where there are defined values and to make sure that
1027    // the tools don't optimize out undriven lines
1028    for (int i = breakpoints.size(); i < 32; i++) {
1029 
1030        debugFile << "\tassign breakpoints[" << i << "] = 1'b0;\n";
1031    }
1032 
1033    debugFile << "\nendmodule\n\n" << std::endl;
1034    debugFile.close();
1035 }
1036 
1037 int com_clear (char *arg) {
1038     printf("\033[H\033[J");
1039 }
1040 
1041 
1042 int com_ping (char *arg) {
```

```
1043
1044        std::string result = sendRemoteCommand("ping");
1045        if (!result.size()) {
1046            cout << "The remote system did not respond.";
1047        } else {
1048            cout << "Remote system responded: " << result;
1049        }
1050        cout << endl;
1051    }
1052
1053    std::string sendRemoteCommand(const std::string &command) {
1054
1055        int res;
1056        const int BUF_LEN = 255;
1057        char buf[BUF_LEN];
1058        std::string revisedCommand;
1059        std::string myResult = "";
1060        myResult.clear();
1061
1062        // add EOL to trigger the command
1063        revisedCommand = command;
1064        revisedCommand.append("\r\n");  // terminate the command for the interpreter
1065        //    printf("'%s'", revisedCommand.c_str());
1066
1067        write(fd, revisedCommand.c_str(), revisedCommand.length());
1068
1069        while ((res = read(fd,buf,BUF_LEN-1)) > 0) {
1070            // ordinarily doing pure C, read one less character and terminate:  buf[res] = 0;
1071            buf[res] = 0;
1072            myResult.append(buf);
1073            //      printf("%s", buf);
1074        }
1075        //    cout << "[" << myResult << "]" << endl;
1076
1077        //    buf[res]=0;             // set end of string, so we can printf * /
1078        return myResult;
1079    }
1080
1081
1082    void tokenize(const std::string& str,
1083            std::vector<string>& tokens,
1084            const std::string& delimiters)
1085    {
1086        // Skip delimiters at beginning.
1087        string::size_type lastPos = str.find_first_not_of(delimiters, 0);
1088        // Find first "non-delimiter".
1089        string::size_type pos     = str.find_first_of(delimiters, lastPos);
1090
1091        while (string::npos != pos || string::npos != lastPos)
1092        {
1093            // Found a token, add it to the vector.
1094            tokens.push_back(str.substr(lastPos, pos - lastPos));
1095            // Skip delimiters.  Note the "not_of"
1096            lastPos = str.find_first_not_of(delimiters, pos);
1097            // Find next "non-delimiter"
1098            pos = str.find_first_of(delimiters, lastPos);
1099        }
1100    }
1101
1102    std::string convertIntToStr(const uint32_t num) {
1103        std::ostringstream numStringStream;
1104        numStringStream << num;
1105        std::string numStr = numStringStream.str();
1106
1107        return numStr;
1108    }
1109
1110    void createPrintList(std::list<std::string> &frameList, std::string &printArgs) {
1111
1112        std::list<std::string>::iterator flIt;
1113        for (flIt = frameList.begin(); flIt != frameList.end(); ++flIt) {
1114            //    cout << "\t" << (*flIt) << "\n";
1115            printArgs.append(" ");
1116            printArgs.append((*flIt));
1117        }
1118        //  cout << endl;
1119    }
1120
1121    // read the breakpoint enable mask from the board
1122    uint32_t getBreakpointMask() {
1123        // send the command and parse the response
1124        std::string response = sendRemoteCommand("info mask");
1125        cout << "Breakpoint mask: " << response << endl;
1126
1127        // populate a mask of enabled breakpoints
1128        uint32_t breakpointMask;
1129        sscanf(response.c_str(), "%x", &breakpointMask);
```

```
1130
1131        return breakpointMask;
1132    }
1133
1134    uint32_t getActiveBreakpointMask() {
1135        // send the command and parse the response
1136        std::string response = sendRemoteCommand("info active");
1137        cout << "Active breakpoints: " << response << endl;
1138
1139        // populate a mask of active breakpoints
1140        uint32_t breakpointMask;
1141        sscanf(response.c_str(), "%x", &breakpointMask);
1142
1143        return breakpointMask;
1144    }
1145
1146
1147    bool isBreakpointEnabled(const uint32_t idx) {
1148      uint32_t breakpointMask = getBreakpointMask();
1149
1150      return (breakpointMask >> idx) & 0x1;
1151    }
```

Listing B.11: Main FPGA LLD Console.

```
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
#include "tokenize.h"
#include "cmd_dbg.h"
#include <xuartlite_l.h>
#include <xparameters.h>
#include <stdlib.h>
#include "icap.h"

// this is almost useless since we must use Xilinx inbyte/outbyte
// when not using an OS because the stdio getc/getchar will not read
// from stdin
//#include <stdio.h>

// User headers

// ----------------------------------------------------------------
// Local defines

// Xilinx print() function does not accept const char*, complains about discarding qualifier
#ifdef DMD_SERVER
char *prompt = "";
#else
char *prompt = "dmd$ ";
#endif

// Maximum number of argumnets to support, print is the largest of these
// I expanded print to allow 16 frame bits to be read
// Args
//  0 command name
//   1 frame address
//  2-18 frame bits
//  19  null to indicate end
#define MAX_ARGC 19

static void my_get_line(char * line, int    maxlen);

// ================================================================
static void cmd_clearscreen(int cargc, char ** cargv) {
  print("\033[H\033[J");
}


// print a bit value from the given frame address/offset pair
static void cmd_printbit(int cargc, char ** cargv) {

  // cargc: total number of valid args
  // cargv[0]: print
  // cargv[1]: frame
  // cargv[2-9]: bit indexes
  const int bitCount = cargc - 2; // num of bits is minus 'print' and frame
  Xuint32 bitIdx[16];

  int i = 1;  // skip arg 0 which is the command
  Xuint32 frame = atoi(cargv[i++]);
  //printf("frame = %d cargc = %d bitCount = %d\r\n",frame,cargc,bitCount);
  // populate the bit indices to be retrieved
  for ( ; i<cargc; i++) {
    bitIdx[i-2] = atoi(cargv[i]); // offset the index back to 0
#ifdef DMD_DEBUG
    xil_printf("%d\t", bitIdx[i-2]);
#endif
  }

  u32 FrameBuffer[XHI_NUM_WORDS_FRAME_INCL_NULL_FRAME];
  Xuint32 status = readFrame(frame, &FrameBuffer);

  // final result
  u32 result = 0;

  for (i=0; i<bitCount; i++) {
    u32 offset = bitIdx[i];

    u32 targetWordIdx = (int)(offset / 32) + (XHI_NUM_FRAME_WORDS + 1);
    u32 targetBitIdx  = offset % 32;

    // you must invert the sense of the bit
    // this appears to be the best place to do it because
    // otherwise you must remmeber the size and mask those bits
    result = (result << 1) | (~(FrameBuffer[targetWordIdx] >> targetBitIdx) & 0x1);

  #ifndef DMD_SERVER
    printf("FrameBuffer[%d]: 0x%08x >> %d = 0x%x\r\n", targetWordIdx- (XHI_NUM_FRAME_WORDS + 1),
        FrameBuffer[targetWordIdx], targetBitIdx, result);
  #endif
```

```
86    }
87
88    // NOTICE: printing in hex!
89    printf("%x\r\n", result);
90  /*
91    // there should be two arguments: address, offset
92    Xuint32 address = atoi(cargv[1]);
93    Xuint32 offset = atoi(cargv[2]);
94  #ifndef DMD_SERVER
95    xil_printf("0x%08x : %d\r\n", address, offset);
96  #endif
97    readFrameOffset(address, offset);
98  */
99  }
100
101 // undocumented utility function for use from client to
102 // print alive message - do not include in help
103 static void cmd_ping(int cargc, char ** argv) {
104   print("DMD on-chip is alive.\r\n");
105   readDut0();
106 }
107
108 static void cmd_printhelp(int cargc, char ** cargv) {
109   print(
110   "\r\nHelp:\r\n"
111   "run - Begin the hardware's execution.\r\n"
112   "stop - Stop the hardware's execution.\r\n"
113   "step [number-of-steps=1] - Step the design by the given number of steps.\r\n"
114   "print <frame_address> <frame_offset> - Print the bit located at the given address/offset pair.\r\n"
115   "continue - Continue the hardware's execution following a breakpoint or stop.\r\n"
116   "clear - Clear screen.\r\n"
117   "enable | disable <breakpoint-number> - Enable/disable a breakpoint by index.\r\n"
118   "info {mask | active} - Print information about registers, status, etc.\r\n"
119   "status - Print status registers.\r\n"
120   "help - Show this help message\r\n\n"
121   );
122 }
123
124 static void cmd_step(int cargc, char ** cargv) {
125
126   int numSteps = 1; // default to one step
127
128   // determine how this was invoked - alone or with an argument
129   // we only accept one argument, which is the step count
130   if (cargc>1) {
131     int temp = atoi(cargv[1]);
132
133     // discard any trash, like whitespace or characters which atoi returns as 0
134     if (temp == 0)
135       numSteps = 1;
136     else
137       numSteps = temp;
138   }
139
140   step(numSteps);
141
142   return;
143 }
144
145 static void cmd_status(int cargc, char **argv) {
146   xil_printf("Register status:\r\n");
147   //xil_printf("\tStatus: 0x%8x\r\n", );
148   xil_printf("\tStep counter: %d\r\n", readStepCounter());
149   xil_printf("\tControl register: 0x%8x\r\n", PDC_mReadSlaveReg15(XPAR_PDC_0_BASEADDR, 0));
150 }
151
152
153 // ----------------------------------------------------------------
154 // --  Main
155 // ----------------------------------------------------------------
156 int main(void) {
157   int  cargc;
158   char * cargv[MAX_ARGC];
159   int len;
160   char line[128];
161
162   // initialize this to null so that the print function knows when to initialize it
163   isHwIcapInit = 0;
164 #ifndef DMD_SERVER
165   print("\r\n\n\nDMD Command-line Low-Level Debugger (LLD)\r\n");
166 #endif
167   print(prompt);
168
169   for (;;) {
170     my_get_line(line, sizeof(line));
171
172     len = strlen(line);
```

```
173       if (line[len-1] == '\n')
174         line[len-1] = 0;
175
176       tokenize(line, &cargc, cargv, MAX_ARGC);
177
178       if (cargc == 0) {
179         print(prompt);
180         continue;
181       }
182
183       if (strcmp(cargv[0], "help") == 0) {
184         cmd_printhelp(cargc, cargv);
185       }
186
187       else if (strcmp(cargv[0], "clear") == 0) {
188         cmd_clearscreen(cargc, cargv);
189       }
190
191       else if (strcmp(cargv[0], "run") == 0) {
192       #ifndef DMD_SERVER
193         print("Starting hardware execution. Type 'stop' to halt.\r\n");
194       #endif
195         run();
196       }
197
198       else if (strcmp(cargv[0], "step") ==0) {
199         cmd_step(cargc, cargv);
200         // step(1);
201       }
202
203       else if (strcmp(cargv[0], "stop") == 0) {
204         stop();
205       }
206
207       else if (strcmp(cargv[0], "print") == 0) {
208         //readDut0();
209         cmd_printbit(cargc, cargv);
210       }
211
212       else if (strcmp(cargv[0], "ping") == 0) {
213         //readDut0();
214         cmd_ping(cargc, cargv);
215       }
216
217       else if (strcmp(cargv[0], "status") == 0) {
218         cmd_status(cargc, cargv);
219       }
220
221       else if (strcmp(cargv[0], "enable") == 0) {
222         u32 breakpoint = atoi(cargv[1]);
223         enableBreakpoint(breakpoint);
224       }
225
226       else if (strcmp(cargv[0], "disable") == 0) {
227         u32 breakpoint = atoi(cargv[1]);
228         disableBreakpoint(breakpoint);
229       }
230
231       else if (strcmp(cargv[0], "info") == 0) {
232         if (strcmp(cargv[1], "mask") == 0) {
233           xil_printf("0x%08x\r\n", readBreakpointMask());
234         }
235
236         if (strcmp(cargv[1], "active") == 0) {
237           xil_printf("0x%08x\r\n", readActiveBreakpointMask());
238         }
239       }
240
241       else {
242         print("Unrecognized command \"");
243         print(cargv[0]);
244         print("\".\r\n");
245       }
246
247       print(prompt);
248
249     }
250     return 0;
251   }
252
253
254   // ------------------------------------------------------------
255   static void my_get_line(
256     char * line,
257     int    maxlen)
258   {
259     char   c = 0;
```

```
260     char * p = line;
261     int n;
262
263     *p = 0;
264     for (n = 0; n < maxlen-1; n++) {
265       c = inbyte();
266
267       // handle empty lines when the user just hits ENTER
268       if (c == '\n') {
269         // Ignore it.
270         ;
271       }
272
273       else if (c == '\r') {
274   #ifndef DMD_SERVER
275         outbyte('\r');
276         outbyte('\n');
277   #endif
278         break;
279       }
280   #ifndef DMD_SERVER
281       // Check for backspace or delete key.
282       else if ((c == '\b') || (c == 0x7F)) {
283         if (p > line) {
284           outbyte('\b'); // Write backspace
285           outbyte(' ');  // Write space
286           outbyte('\b'); // Write backspace
287           p--;
288           *p = 0;
289         }
290       }
291
292       // Check for escape key or control-U.
293       else if ((c == 0x1b) || (c == 0x15)) {
294         while (p > line) {
295           outbyte('\b');
296           outbyte(' ');
297           outbyte('\b');
298           p--;
299           *p = 0;
300         }
301       }
302   #endif
303
304       else {
305   #ifndef DMD_SERVER
306         // turn off echoing when you are a server because it makes parsing responses harder
307         outbyte(c); // Echo character back to the user.
308   #endif
309         *p = c;
310         p++;
311         *p = 0;
312       }
313     }
314     *p = 0;
315   }
```

Listing B.12: Logic Allocation Lexer.

```
1
2   %{
3    #include "logicalloc.tab.h"
4
5    #include <string.h>
6   %}
7
8   %option yylineno
9   %option noyywrap
10
11  DIGIT    [0-9]
12  ID       [a-zA-Z0-9#_-]
13
14  %%
15
16    /* logic allocation report report keywords */
17  Revision    {
18            // printf("lrevision");
19            return REVISION;
20          }
21  Info        {
22            // printf("linfo");
23            return INFO;
24          }
25  Bit         {
26            // printf("lbit");
27            return BIT;
28          }
29  Block       {
30            // printf("lblock");
31            return BLOCK;
32          }
33  Latch       {
34            // printf("llatch");
35            return LATCH;
36          }
37  Net         {
38            // printf("lnet");
39            return NET;
40          }
41  COMPARE     {
42            // printf("lcompare");
43            return COMPARE;
44          }
45  Ram         {
46            // printf("lram");
47            return RAM;
48          }
49  Rom         {
50            // printf("lrom");
51            return ROM;
52          }
53  Type        {
54            // printf("ltype");
55            return TYPE;
56          }
57
58  YES         {
59            // printf("lyes");
60            return YES;
61          }
62  NO          {
63            // printf("lno");
64            return NO;
65          }
66  BIT |
67    PARBIT    {
68            // printf("lramidbitval");
69            return RAMIDBITTYPE;
70          }
71  [A-D]Q |
72    [IO]    {
73            // printf("llatchid");
74            logicalloclval.name_val = strdup(logicalloctext);
75            return LATCHID;
76          }
77  0x[0-9a-fA-F]{8} {
78            // printf("lframe");
79  //          logicalloclval.frame_val = strtol(logicalloctext,NULL,0);
80            logicalloclval.frame_val = strtol(logicalloctext,NULL,16);
81            return FRAME;
82          }
83  [0-9]+      {
84            // printf("lnumber");
85            logicalloclval.num_val = atoi(logicalloctext);
```

```
86            return NUMBER;
87          }
88
89    /*
90  [A-Z0-9]+_X[0-9]+Y[0-9]+ |
91    [A-Z][0-9]{2} { printf("lblocklocation"); return BLOCKLOCATION; }
92    */
93
94
95  [A-Za-z0-9/\._]+  {
96            // printf("ldesignname");
97            logicalloclval.name_val = strdup(logicalloctext);
98            return ID;
99            }
100   /*
101 [A-D]       {
102           // printf("lramid");
103           return RAMID;
104         }
105   */
106 ^;.*        {
107         //  printf("lcomment_line");
108         }
109 [ \t\n]+      ;
110 .         { return logicalloctext[0]; }
111
112 %%
113
114
115 /*
116 int main(int argc, char **argv) {
117
118   return yylex();
119
120 }
121 */
```

Listing B.13: Logic Allocation Parser.

```
%{

#include <stdio.h>
#include <map>
#include <list>
#include <iostream>
#include <sstream>
#include <string>


#include "BitInfo.h"

// externs, prototypes, declarations, globals
extern char* logicalloctext;
void logicallocerror(const char *msg);
int  logicalloclex ();
extern FILE *logicallocin;     // input file

BitInfo *currentBit;
std::string currentBitName;

BitMap *bitMap;

using namespace std;
%}

  /* definition */
  /* literal keyword tokens */

%debug
%error-verbose
%defines      /* output header file for flex - also a command line opt */
%locations      /* enable yylloc */

%union {
  int num_val;  /* For returning numbers, names.  */
  unsigned int frame_val;
  char *name_val;
}

%token <num_val> NUMBER
%token <name_val> ID BLOCKLOCATION LATCHID
%token <frame_val> FRAME

%type <name_val> block

%token INFO BIT
%token REVISION
%token LATCH RAM ROM NET BLOCK
%token RAMID RAMIDBITTYPE
%token TYPE

%token COMPARE YES NO
%start logic_alloc_report

%%

logic_alloc_report: revision_header info_list bit_list
  {
    printf("\n");
  }

revision_header: REVISION NUMBER
  {
  //  printf("revision header\n");

    // this is a new file, clear out the map
    bitMap->clear();
  }
  ;

info_list: info_line
  |    info_list info_line
  {

  }
  ;

info_line: INFO ID '=' NUMBER
  {
  //  printf("infoline\n");
  }
  ;

bit_list: bit_line
```

```
86    |   bit_list bit_line
87    {
88
89    }
90    ;
91
92  bit_line: bit_preamble net
93    {
94      // this is the only situation we care about, nets
95      // push the net on the vector
96      if (currentBit->latch != "I" && currentBit->latch != "O") {
97        (*bitMap)[currentBitName].push_back(*currentBit);
98      }
99
100     delete currentBit;
101     currentBit = 0;
102 //    printf("\r%s [%d]", currentBitName.c_str(), bitMap->size());
103     printf("\rLoading: %-120s ", currentBitName.c_str());
104   }
105   |     bit_preamble ram
106   {
107 //    printf("bitinfo_ramtype");
108     delete currentBit;
109     currentBit = 0;
110     currentBitName = "";
111   }
112   |   bit_preamble type
113   {
114 //    printf("bitinfo_ramtype");
115     delete currentBit;
116     currentBit = 0;
117     currentBitName = "";
118   }
119   ;
120
121 bit_preamble: BIT NUMBER FRAME NUMBER block
122   {
123   //  printf("\ncreatenewbit");
124     currentBit = new BitInfo();
125     currentBit->bitStreamReadbackLocation = $2;
126     currentBit->frameAddress = $3;
127     currentBit->frameOffset = $4;
128     currentBit->block = $5;
129   }
130   ;
131
132 block:  BLOCK '=' ID
133   {
134   //    printf(" blocklocation ");
135   //    currentBit->block = $3;
136     $$ = $3;
137   }
138   ;
139
140 net:  latch NET '=' ID
141       {
142         std::string netName = $4;
143         currentBitName = netName;
144
145         currentBit->net = $4;
146         currentBit->signalVectorIndex = 0;
147         //printf("simplenet: %s", $4);
148       }
149   | latch NET '=' ID '<' NUMBER '>'  /* 1-D array */
150       {
151         std::string netName = $4;
152         // set the current bit name for this instance
153         currentBitName = netName;
154
155         // extract the index number for the internal name
156         std::ostringstream numStream;
157         numStream << $6;
158         netName.append("<");
159         netName.append(numStream.str());
160         netName.append(">");
161
162         currentBit->net = netName;
163
164         currentBit->signalVectorIndex = $6;
165         //printf("1Dnet: %s", netName.c_str());
166       }
167   | latch NET '=' ID '[' NUMBER ']'  /* 1-D array */
168       {
169         std::string netName = $4;
170         // set the current bit name for this instance
171         currentBitName = netName;
172
```

```
173           // extract the index number for the internal name
174           std::ostringstream numStream;
175           numStream << $6;
176           netName.append("<");
177           netName.append(numStream.str());
178           netName.append(">");
179
180           currentBit->net = netName;
181
182           currentBit->signalVectorIndex = $6;
183           //printf("1Dnet: %s", netName.c_str());
184         }
185     | latch NET '=' ID '<' NUMBER '>' '<' NUMBER '>'  /* 2-D array */
186         {
187           std::string netName = $4;
188
189           // extract the array number for the internal and external name
190           std::ostringstream numStream1;
191           numStream1 << $6;
192           netName.append("[");
193           netName.append(numStream1.str());
194           netName.append("]");
195
196           // set the current bit name for this instance
197           currentBitName = netName;
198
199           // continue building the bit index for internal name
200           std::ostringstream numStream2;
201           numStream2 << $9;
202           netName.append("<");
203           netName.append(numStream2.str());
204           netName.append(">");
205
206           currentBit->net = netName;
207
208           currentBit->signalVectorIndex = $9;
209           //printf("2Dnet: %s", netName.c_str());
210         }
211     | latch NET '=' ID '[' NUMBER ']' '[' NUMBER ']'  /* 2-D array */
212         {
213           std::string netName = $4;
214
215           // extract the array number for the internal and external name
216           std::ostringstream numStream1;
217           numStream1 << $6;
218           netName.append("[");
219           netName.append(numStream1.str());
220           netName.append("]");
221
222           // set the current bit name for this instance
223           currentBitName = netName;
224
225           // continue building the bit index for internal name
226           std::ostringstream numStream2;
227           numStream2 << $9;
228           netName.append("<");
229           netName.append(numStream2.str());
230           netName.append(">");
231
232           currentBit->net = netName;
233
234           currentBit->signalVectorIndex = $9;
235           //printf("2Dnet: %s", netName.c_str());
236         }
237     ;
238
239   latch: LATCH '=' LATCHID
240     {
241       //printf(" latch = %s ");
242         currentBit->latch = $3;
243   //    cout << "latch = " << currentBit->latch;
244     }
245     ;
246
247   ram:  RAM '=' ID ':' NUMBER
248     | RAM '=' ID ':' RAMIDBITTYPE NUMBER
249     | RAM '=' ID ':' ID
250     {
251     //    printf(" ram ");
252     }
253     ;
254
255   type:   TYPE '=' ID
256     {
257     //  printf(" type ");
258     }
259     ;
```

```
260
261
262 %%
263   /* code */
264 void logicallocerror(const char *msg) {
265   printf("error: %d,%d %s, %s\n",
266     logicallocl loc.first_line, logicallocl loc.first_column, msg, logicalloctext);
267 }
268
269
270 bool bitInfoSignalIndexReverseSort(const BitInfo& a, const BitInfo& b) {
271   return (b.signalVectorIndex < a.signalVectorIndex);
272 }
273
274 bool bitInfoSortByFrameAddress(const BitInfo& a, const BitInfo& b) {
275   return (a.frameAddress < b.frameAddress);
276 }
277
278 /*
279 int old_main(void) {
280   logicallocparse();
281
282   // iterate through the collected signals
283   //std::map<std::string, std::list<BitInfo> >::iterator it;
284 /*
285   for (it = bitMap.begin(); it!= bitMap.end(); ++it) {
286     const std::string netName = it->first;
287
288     std::list<BitInfo>::iterator bIt;
289
290     for(bIt = it->second.begin(); bIt!=it->second.end(); ++bIt) {
291       BitInfo *bitInfo = &*bIt;
292       cout << netName << "\tsignalIndx: " << bitInfo->signalVectorIndex
293          << "\tframeOffset: " << bitInfo->frameOffset << endl;
294     }
295   }
296 * /
297   BitInfoList myWord = bitMap["counter"];
298   BitInfoList::iterator it;
299
300   for(it = myWord.begin(); it!=myWord.end(); ++it) {
301       BitInfo *bitInfo = &*it;
302       cout << "\tsignalIndx: " << bitInfo->signalVectorIndex
303          << "\tframeAddress: " << bitInfo->frameAddress
304          << "\tframeOffset: " << bitInfo->frameOffset << endl;
305   }
306
307   cout << "Sorting..." << endl;
308
309   myWord.sort(bitInfoSortByFrameAddress);
310
311   for(it = myWord.begin(); it!=myWord.end(); ++it) {
312       BitInfo *bitInfo = &*it;
313       cout << "\tsignalIndx: " << bitInfo->signalVectorIndex
314          << "\tframeAddress: " << bitInfo->frameAddress
315          << "\tframeOffset: " << bitInfo->frameOffset << endl;
316   }
317
318   return 0;
319 }
320 */
321
322 extern "C" int parseLogicAllocFile(const std::string &llFile, BitMap &bMap) {
323
324   bitMap = &bMap;
325
326   // reassign yyin (renamed) to the file handle from the filename you passed in
327   logicallocin = fopen(llFile.c_str(), "r");
328
329   // call the main logicalloc parse routine
330   logicallocparse();
331 }
```

Listing B.14: Programmable Debug Controller.

```
1  //----------------------------------------------------------------------------
2  // user_logic.vhd - module
3  //----------------------------------------------------------------------------
4  //
5  // ****************************************************************************
6  // ** Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.            **
7  // **                                                                        **
8  // ** Xilinx, Inc.                                                          **
9  // ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
10 // ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
11 // ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
12 // ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
13 // ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
14 // ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
15 // ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
16 // ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
17 // ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
18 // ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
19 // ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
20 // ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
21 // ** FOR A PARTICULAR PURPOSE.                                             **
22 // **                                                                        **
23 // ****************************************************************************
24 //
25 //----------------------------------------------------------------------------
26 // Filename:          user_logic.vhd
27 // Version:           1.00.a
28 // Description:       User logic module.
29 // Date:              Tue Sep 14 11:34:52 2010 (by Create and Import Peripheral Wizard)
30 // Verilog Standard:  Verilog-2001
31 //----------------------------------------------------------------------------
32 // Naming Conventions:
33 //   active low signals:                    "*_n"
34 //   clock signals:                         "clk", "clk_div#", "clk_#x"
35 //   reset signals:                         "rst", "rst_n"
36 //   generics:                              "C_*"
37 //   user defined types:                    "*_TYPE"
38 //   state machine next state:              "*_ns"
39 //   state machine current state:           "*_cs"
40 //   combinatorial signals:                 "*_com"
41 //   pipelined or register delay signals:   "*_d#"
42 //   counter signals:                       "*cnt*"
43 //   clock enable signals:                  "*_ce"
44 //   internal version of output port:       "*_i"
45 //   device pins:                           "*_pin"
46 //   ports:                                 "- Names begin with Uppercase"
47 //   processes:                             "*_PROCESS"
48 //   component instantiations:              "<ENTITY_>I_<#|FUNC>"
49 //----------------------------------------------------------------------------
50
51 module user_logic
52 (
53   // -- ADD USER PORTS BELOW THIS LINE ---------------
54   // --USER ports added here
55   DebugModeLed,
56   Dut0,
57   Dut1,
58   SysClk,
59   SysDbgClk,
60   BreakpointInterrupt,
61   BreakpointId,
62   // -- ADD USER PORTS ABOVE THIS LINE ---------------
63
64   // -- DO NOT EDIT BELOW THIS LINE ------------------
65   // -- Bus protocol ports, do not add to or delete
66   Bus2IP_Clk,                     // Bus to IP clock
67   Bus2IP_Reset,                   // Bus to IP reset
68   Bus2IP_Data,                    // Bus to IP data bus
69   Bus2IP_BE,                      // Bus to IP byte enables
70   Bus2IP_RdCE,                    // Bus to IP read chip enable
71   Bus2IP_WrCE,                    // Bus to IP write chip enable
72   IP2Bus_Data,                    // IP to Bus data bus
73   IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
74   IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
75   IP2Bus_Error,                   // IP to Bus error response
76   IP2Bus_IntrEvent                // IP to Bus interrupt event
77   // -- DO NOT EDIT ABOVE THIS LINE ------------------
78 ); // user_logic
79
80 // -- ADD USER PARAMETERS BELOW THIS LINE ------------
81 // --USER parameters added here
82 // -- ADD USER PARAMETERS ABOVE THIS LINE ------------
83
84 // -- DO NOT EDIT BELOW THIS LINE --------------------
85 // -- Bus protocol parameters, do not add to or delete
```

```
86  parameter C_SLV_DWIDTH                   = 32;
87  parameter C_NUM_REG                      = 16;
88  parameter C_NUM_INTR                     = 1;
89  // -- DO NOT EDIT ABOVE THIS LINE --------------------
90
91  // -- ADD USER PORTS BELOW THIS LINE -----------------
92  // --USER ports added here
93  output                  DebugModeLed;
94  input   [0 : 31]          Dut0;
95  input   [0 : 31]          Dut1;
96  input                   SysClk;
97  output                  SysDbgClk;
98  output                  BreakpointInterrupt;
99  input     [0 : 31]          BreakpointId;
100 // -- ADD USER PORTS ABOVE THIS LINE -----------------
101
102 // -- DO NOT EDIT BELOW THIS LINE --------------------
103 // -- Bus protocol ports, do not add to or delete
104 input                                     Bus2IP_Clk;
105 input                                     Bus2IP_Reset;
106 input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
107 input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
108 input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
109 input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
110 output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
111 output                                    IP2Bus_RdAck;
112 output                                    IP2Bus_WrAck;
113 output                                    IP2Bus_Error;
114 output     [0 : C_NUM_INTR-1]             IP2Bus_IntrEvent;
115 // -- DO NOT EDIT ABOVE THIS LINE --------------------
116
117 //----------------------------------------------------------------------------
118 // Implementation
119 //----------------------------------------------------------------------------
120
121   // --USER nets declarations added here, as needed for user logic
122   wire sys_dbg_clk;     // patis system debug clock which will pass the system clock or the debug step clock
123   wire sys_clk;       // system clock
124  // reg stop_clk;   // internal signal to abstract the clock advance register bit
125  // reg  stop_clk_d1;   // one delay cycle to find rising edge and steady state
126   reg ctrl_run;     // control register clock select line to determine which clock to source
127  // wire bp_dut0_active;    // is breakpoint for dut0 active?
128   wire clk_sel;       // clock select control line
129   reg  [0 : 31] step_counter;
130   reg step_counter_en;
131   //wire step_counter_en;
132   wire breakpoint_active;
133   wire [0 : 31] breakpoint_mask;
134
135   wire step_counter_expired;    // high when the step clock has reached goal
136   wire [0 : 31] step_counter_target;    // alias to slv_reg12
137   wire step_counter_activate;
138   wire step_counter_target_write;
139   reg  step_counter_target_write_d1;  // delay to determine rising edge
140   wire dbg_stop_clk;
141
142   // Nets for user logic slave model s/w accessible register example
143   reg        [0 : C_SLV_DWIDTH-1]           slv_reg0;   // Dut0
144   reg        [0 : C_SLV_DWIDTH-1]           slv_reg1;   // Dut1 for development purposes only, don't use
145   reg        [0 : C_SLV_DWIDTH-1]           slv_reg2;
146   reg        [0 : C_SLV_DWIDTH-1]           slv_reg3;
147   reg        [0 : C_SLV_DWIDTH-1]           slv_reg4;
148   reg        [0 : C_SLV_DWIDTH-1]           slv_reg5;
149   reg        [0 : C_SLV_DWIDTH-1]           slv_reg6;
150   reg        [0 : C_SLV_DWIDTH-1]           slv_reg7;
151   reg        [0 : C_SLV_DWIDTH-1]           slv_reg8;
152   reg        [0 : C_SLV_DWIDTH-1]           slv_reg9;
153   reg        [0 : C_SLV_DWIDTH-1]           slv_reg10;
154   reg        [0 : C_SLV_DWIDTH-1]           slv_reg11;
155   reg        [0 : C_SLV_DWIDTH-1]           slv_reg12;    // step counter goal
156   reg        [0 : C_SLV_DWIDTH-1]           slv_reg13;
157   reg        [0 : C_SLV_DWIDTH-1]           slv_reg14;  // breakpoint mask
158   reg        [0 : C_SLV_DWIDTH-1]           slv_reg15;  // control register
159   wire       [0 : 15]                       slv_reg_write_sel;
160   wire       [0 : 15]                       slv_reg_read_sel;
161   reg        [0 : C_SLV_DWIDTH-1]           slv_ip2bus_data;
162   wire                                      slv_read_ack;
163   wire                                      slv_write_ack;
164   integer                                   byte_index, bit_index;
165
166   // --USER logic implementation added here
167   // States for step counter
168   parameter IDLE  = 2'd0, SETUP = 2'd1, DELAY = 2'd2, STEPPING = 2'd3;
169   reg [1 : 0] state;
170
171   initial
172   begin
```

```
173       step_counter_target_write_d1 = 0;
174     state = IDLE;
175     ctrl_run = 0;
176     step_counter_en = 0;
177     end
178
179     // external assigns
180     assign DebugModeLed   = ~clk_sel; // bp_dut0_active; // ~clk_sel;     // clock select indicator from mux line
181     assign SysDbgClk    = sys_dbg_clk;
182     assign sys_clk    = SysClk;
183
184     assign BreakpointInterrupt = breakpoint_active; //step_counter_enabled;
185     assign breakpoint_mask = slv_reg14;
186     // mask the breakpoint_mask with the breakpoints vector and or-reduce to determine if there is an active breakpoint
187     assign breakpoint_active = |(breakpoint_mask & BreakpointId);
188
189     // control registers assigns
190     //assign ctrl_run       = slv_reg15[31];
191     //assign stop_clk     = slv_reg15[30];
192     assign clk_sel = (~breakpoint_active & (step_counter_en | ctrl_run));
193
194     // pulses unknown number of cycles when the register is written to
195     assign step_counter_target = slv_reg12;     // alias step_counter_target register
196     assign step_counter_target_write = (slv_reg_write_sel == 16'b0000000000001000);
197     assign step_counter_activate = (step_counter_target_write == 1 && step_counter_target_write_d1 == 0);
198     assign step_counter_expired = (step_counter_target-1 == step_counter);
199
200     // rising edge detection sync logic
201    // assign dbg_stop_clk = (stop_clk == 1 && stop_clk_d1 == 0);
202
203     // control logic
204     always @(posedge Bus2IP_Clk)
205     begin
206         // control register is being written to
207         if (slv_reg_write_sel == 16'b0000000000000001)
208           ctrl_run <= Bus2IP_Data[31];
209
210         // stop the running if a breakpoint hits
211         if (breakpoint_active) ctrl_run <= 1'b0;
212     end
213
214
215     // step clock
216     always @(posedge sys_clk)
217     begin
218     case (state)
219     IDLE:
220     begin
221       step_counter_en <= 1'b0;
222       step_counter <= 32'b0;
223       state <= IDLE;
224
225       if (step_counter_activate && !ctrl_run)
226         state <= SETUP;
227     end
228
229     // start the step_counter_en one cycle to chnage the clock_sel
230     SETUP:
231     begin
232       step_counter_en <= 1'b1;
233       state <= DELAY;
234     end
235
236     DELAY:
237     begin
238       state <= STEPPING;
239     end
240
241     STEPPING:
242     begin
243       step_counter_en <= 1'b1;
244       step_counter <= step_counter + 1;
245       state <= STEPPING;
246
247       if (step_counter_expired)
248       begin
249         state <= IDLE;
250         step_counter_en <= 1'b0;
251       end
252     end
253
254     endcase
255
256
257     // feed the run bit through to pulse the stop_clk
258     //stop_clk <= ctrl_run | step_counter_en;
259   //  stop_clk_d1 <= stop_clk;
```

```
260
261   step_counter_target_write_d1 <= step_counter_target_write;
262
263   end
264
265
266   // bufgmux instantiation
267     BUFGCE dbg_clk_mux (
268   .O(sys_dbg_clk),    // Clock MUX output
269   .I(sys_clk),      // Clock1 input
270   .CE(clk_sel)      // Clock select input
271   );
272
273
274   /*
275   BUFGMUX_CTRL BUFGMUX_CTRL_inst (
276   .O(sys_dbg_clk),    // Clock MUX output
277   .I0(dbg_stop_clk),    // Clock0 input
278   .I1(sys_clk),       // Clock1 input
279   .S(clk_sel)       // Clock select input
280   );
281 */
282   // ------------------------------------------------------
283   // Example code to read/write user logic slave model s/w accessible registers
284   //
285   // Note:
286   // The example code presented here is to show you one way of reading/writing
287   // software accessible registers implemented in the user logic slave model.
288   // Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
289   // to one software accessible register by the top level template. For example,
290   // if you have four 32 bit software accessible registers in the user logic,
291   // you are basically operating on the following memory mapped registers:
292   //
293   //    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
294   //                      "1000"   C_BASEADDR + 0x0
295   //                      "0100"   C_BASEADDR + 0x4
296   //                      "0010"   C_BASEADDR + 0x8
297   //                      "0001"   C_BASEADDR + 0xC
298   //
299   // ------------------------------------------------------
300
301   assign
302     slv_reg_write_sel = Bus2IP_WrCE[0:15],
303     slv_reg_read_sel  = Bus2IP_RdCE[0:15],
304     slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2] || Bus2IP_WrCE[3] || Bus2IP_WrCE[4] \
305       || Bus2IP_WrCE[5] || Bus2IP_WrCE[6] || Bus2IP_WrCE[7] || Bus2IP_WrCE[8] || Bus2IP_WrCE[9] || Bus2IP_WrCE[10] \
306       || Bus2IP_WrCE[11] || Bus2IP_WrCE[12] || Bus2IP_WrCE[13] || Bus2IP_WrCE[14] || Bus2IP_WrCE[15],
307     slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2] || Bus2IP_RdCE[3] || Bus2IP_RdCE[4] \
308       || Bus2IP_RdCE[5] || Bus2IP_RdCE[6] || Bus2IP_RdCE[7] || Bus2IP_RdCE[8] || Bus2IP_RdCE[9] || Bus2IP_RdCE[10] \
309       || Bus2IP_RdCE[11] || Bus2IP_RdCE[12] || Bus2IP_RdCE[13] || Bus2IP_RdCE[14] || Bus2IP_RdCE[15];
310
311   // implement slave model register(s)
312   always @( posedge Bus2IP_Clk )
313     begin: SLAVE_REG_WRITE_PROC
314
315       if ( Bus2IP_Reset == 1 )
316         begin
317           slv_reg0 <= 0;
318           slv_reg1 <= 0;
319           slv_reg2 <= 0;
320           slv_reg3 <= 0;
321           slv_reg4 <= 0;
322           slv_reg5 <= 0;
323           slv_reg6 <= 0;
324           slv_reg7 <= 0;
325           slv_reg8 <= 0;
326           slv_reg9 <= 0;
327           slv_reg10 <= 0;
328           slv_reg11 <= 0;
329           slv_reg12 <= 0;
330           //slv_reg13 <= 0;
331           slv_reg14 <= 0;
332           slv_reg15 <= 0;
333         end
334       else
335         case ( slv_reg_write_sel )
336 /*          16'b1000000000000000 :
337             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
338               if ( Bus2IP_BE[byte_index] == 1 )
339                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
340                   slv_reg0[bit_index] <= Bus2IP_Data[bit_index];    */
341           16'b0100000000000000 :
342             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
343               if ( Bus2IP_BE[byte_index] == 1 )
344                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
345                   slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
346           16'b0010000000000000 :
```

```
347             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
348               if ( Bus2IP_BE[byte_index] == 1 )
349                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
350                   slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
351           16'b0001000000000000 :
352             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
353               if ( Bus2IP_BE[byte_index] == 1 )
354                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
355                   slv_reg3[bit_index] <= Bus2IP_Data[bit_index];
356           16'b0000100000000000 :
357             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
358               if ( Bus2IP_BE[byte_index] == 1 )
359                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
360                   slv_reg4[bit_index] <= Bus2IP_Data[bit_index];
361           16'b0000010000000000 :
362             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
363               if ( Bus2IP_BE[byte_index] == 1 )
364                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
365                   slv_reg5[bit_index] <= Bus2IP_Data[bit_index];
366           16'b0000001000000000 :
367             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
368               if ( Bus2IP_BE[byte_index] == 1 )
369                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
370                   slv_reg6[bit_index] <= Bus2IP_Data[bit_index];
371           16'b0000000100000000 :
372             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
373               if ( Bus2IP_BE[byte_index] == 1 )
374                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
375                   slv_reg7[bit_index] <= Bus2IP_Data[bit_index];
376           16'b0000000010000000 :
377             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
378               if ( Bus2IP_BE[byte_index] == 1 )
379                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
380                   slv_reg8[bit_index] <= Bus2IP_Data[bit_index];
381           16'b0000000001000000 :
382             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
383               if ( Bus2IP_BE[byte_index] == 1 )
384                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
385                   slv_reg9[bit_index] <= Bus2IP_Data[bit_index];
386           16'b0000000000100000 :
387             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
388               if ( Bus2IP_BE[byte_index] == 1 )
389                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
390                   slv_reg10[bit_index] <= Bus2IP_Data[bit_index];
391           16'b0000000000010000 :
392             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
393               if ( Bus2IP_BE[byte_index] == 1 )
394                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
395                   slv_reg11[bit_index] <= Bus2IP_Data[bit_index];
396           16'b0000000000001000 :
397             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
398               if ( Bus2IP_BE[byte_index] == 1 )
399                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
400                   slv_reg12[bit_index] <= Bus2IP_Data[bit_index];
401    /*         16'b0000000000000100 :
402               for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
403                 if ( Bus2IP_BE[byte_index] == 1 )
404                   for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
405                     slv_reg13[bit_index] <= Bus2IP_Data[bit_index];   */
406           16'b0000000000000010 :
407             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
408               if ( Bus2IP_BE[byte_index] == 1 )
409                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
410                   slv_reg14[bit_index] <= Bus2IP_Data[bit_index];
411           16'b0000000000000001 :
412             for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1 )
413               if ( Bus2IP_BE[byte_index] == 1 )
414                 for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
415                   slv_reg15[bit_index] <= Bus2IP_Data[bit_index];
416           default : ;
417         endcase
418
419     end // SLAVE_REG_WRITE_PROC
420
421    // implement slave model register read mux
422    always @( slv_reg_read_sel or
423           // Dut0 or bp_dut0_active or
424   //      step_clk_tripped or
425         slv_reg0 or slv_reg1 or slv_reg2 or slv_reg3 or slv_reg4 or slv_reg5 or slv_reg6 or slv_reg7 or slv_reg8 or
426         slv_reg9 or slv_reg10 or slv_reg11 or slv_reg12 or slv_reg13 or slv_reg14 or slv_reg15 )
427
428     begin: SLAVE_REG_READ_PROC
429
430       case ( slv_reg_read_sel )
431         16'b1000000000000000 : slv_ip2bus_data <= Dut0; // slv_reg0;
432         16'b0100000000000000 : slv_ip2bus_data <= Dut1;
433         16'b0010000000000000 : slv_ip2bus_data <= slv_reg2;
```

```
434           16'b0001000000000000 : slv_ip2bus_data <= slv_reg3;
435           16'b0000100000000000 : slv_ip2bus_data <= slv_reg4;
436           16'b0000010000000000 : slv_ip2bus_data <= slv_reg5;
437           16'b0000001000000000 : slv_ip2bus_data <= slv_reg6;
438           16'b0000000100000000 : slv_ip2bus_data <= slv_reg7;
439           16'b0000000010000000 : slv_ip2bus_data <= slv_reg8;
440           16'b0000000001000000 : slv_ip2bus_data <= slv_reg9;
441           16'b0000000000100000 : slv_ip2bus_data <= slv_reg10;
442           16'b0000000000010000 : slv_ip2bus_data <= BreakpointId; // slv_reg11;
443           16'b0000000000001000 : slv_ip2bus_data <= slv_reg12;  // step counter target
444           16'b0000000000000100 : slv_ip2bus_data <= slv_reg13;  // step_clk_tripped, 31'b0
445           16'b0000000000000010 : slv_ip2bus_data <= slv_reg14;  // breakpoint mask
446           16'b0000000000000001 : slv_ip2bus_data <= {31'b0,ctrl_run}; // slv_ip2bus_data <= slv_reg15;  // ctrl
447           default : slv_ip2bus_data <= 0;
448         endcase
449
450       end // SLAVE_REG_READ_PROC
451
452     // ------------------------------------------------------------
453     // Example code to drive IP to Bus signals
454     // ------------------------------------------------------------
455
456     assign IP2Bus_Data    = slv_ip2bus_data;
457     assign IP2Bus_WrAck   = slv_write_ack;
458     assign IP2Bus_RdAck   = slv_read_ack;
459     assign IP2Bus_Error   = 0;
460
461   endmodule
```

Listing B.15: Example of Generated Breakpoint Logic.

```
1
2
3   module dmd_debug_logic
4   (
5     // design input ports
6     cmd_i,
7     cmd_o,
8     cmd_w_i,
9     text_i,
10    text_o,
11
12     // control lines
13    breakpoint_active,
14    breakpoint_reg
15  );
16
17    input   [0 : 2]   cmd_i;
18    input   [0 : 3]   cmd_o;
19    input           cmd_w_i;
20    input   [0 : 31]    text_i;
21    input   [0 : 31]    text_o;
22    output            breakpoint_active;
23    output    [0 : 31]    breakpoint_reg;
24
25    wire    [0 : 31]    breakpoints;
26
27    // assign the internal breakpoint register array to the output
28    assign breakpoint_reg = breakpoints;
29
30    assign breakpoint_active = |breakpoints;
31
32    assign breakpoints[0] =  (text_i == 32'h62636465);  // breakpoint
33    assign breakpoints[1] =  (text_o == 32'h84983E44);  // breakpoint
34    assign breakpoints[2] =  (cmd_i == 3'b010);  // breakpoint
35    assign breakpoints[3] =  (cmd_o == 4'h4);  // breakpoint
36    assign breakpoints[4] = 1'b0;
37    assign breakpoints[5] = 1'b0;
38    assign breakpoints[6] = 1'b0;
39    assign breakpoints[7] = 1'b0;
40    assign breakpoints[8] = 1'b0;
41    assign breakpoints[9] = 1'b0;
42    assign breakpoints[10] = 1'b0;
43    assign breakpoints[11] = 1'b0;
44    assign breakpoints[12] = 1'b0;
45    assign breakpoints[13] = 1'b0;
46    assign breakpoints[14] = 1'b0;
47    assign breakpoints[15] = 1'b0;
48    assign breakpoints[16] = 1'b0;
49    assign breakpoints[17] = 1'b0;
50    assign breakpoints[18] = 1'b0;
51    assign breakpoints[19] = 1'b0;
52    assign breakpoints[20] = 1'b0;
53    assign breakpoints[21] = 1'b0;
54    assign breakpoints[22] = 1'b0;
55    assign breakpoints[23] = 1'b0;
56    assign breakpoints[24] = 1'b0;
57    assign breakpoints[25] = 1'b0;
58    assign breakpoints[26] = 1'b0;
59    assign breakpoints[27] = 1'b0;
60    assign breakpoints[28] = 1'b0;
61    assign breakpoints[29] = 1'b0;
62    assign breakpoints[30] = 1'b0;
63    assign breakpoints[31] = 1'b0;
64
65  endmodule
```