Efficient Programming of Reconfigurable Hardware through Direct Verification

Kevin Brandon Camera

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-80
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-80.html

June 9, 2008

Efficient Programming of Reconfigurable Hardware through Direct Verification

by

Kevin Brandon Camera

B.S. (University of California, Berkeley) 1998
M.S. (University of California, Berkeley) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Robert W. Brodersen, Chair
Professor Jan M. Rabaey
Professor Paul K. Wright

Fall 2008

Efficient Programming of Reconfigurable Hardware through Direct Verification

Abstract

Efficient Programming of Reconfigurable Hardware
through Direct Verification

by

Kevin Brandon Camera
Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert W. Brodersen, Chair

Reconfigurable hardware devices, such as field-programmable gate arrays (FPGAs), have been shown to achieve greater net throughput and power, energy, and cost efficiency compared to traditional microprocessors in high-performance computing and signal processing applications. The primary drawback to the use of such devices, however, is the perceived difficulty of the overall programming process, which includes the complete design and verification of the system.

In an attempt alleviate this net system design problem, the approach of *direct verification* was conceived and implemented on the BEE2 hardware platform. Direct verification utilizes the resources of the platform to provide variables in the hardware domain, which feature read/write access to data, runtime streaming of data to off-chip storage, and fully automated dynamic assertion checking. By regulating the design clock, the verification infrastructure can also control the execution of the design under test, both via manual interaction and same-cycle breakpoints triggered by variable assertion failures. In addition, all this functionality is accessible in the original design environment via a remote network service provided by the software layer of the verification infrastructure, which allows data to be generated and analyzed at the same level of abstraction previously only present during simulation.

The resource requirements to enable direct verification on the BEE2 platform were measured both independently and as part of two real-world design examples.

The base infrastructure was found to occupy about 12% of an XCV2P70 FPGA (8% of which was purely due to the DDR2 memory controller), and the addition of a typical 16-bit variable required 75 logic slices (0.23% of the device) on average. The operating frequency of the design under test is not severely impacted unless the device utilization approaches 100%, and runtime system throughput is limited primarily by the bandwidth and latency of the attached storage medium.

Professor Robert W. Brodersen
Dissertation Committee Chair

Acknowledgments

While the number of names on the cover of this dissertation may be few, it certainly would not have been possible without the tremendously appreciated support of many others.

First and foremost, I would like to give my deepest thanks to my research advisor, Prof. Bob Brodersen, for his support, guidance, and inspiration over the years... and having been at Berkeley ever since starting as an undergraduate, it has certainly been a great number of years. In addition to the generous financial support which was always made available to me as a graduate student researcher, the way in which he has motivated me to see this work through to its completion, even at times when I became unsure whether research was in my blood, has made this final result possible. The faith that he has shown, both in accepting me as one of his students and encouraging me to continue on for a Ph.D. (during a two-hour conversation one evening at the 2001 BWRC summer retreat), has truly meant a lot and is something I will take with me perhaps even more valuable than the degree.

I would also like to thank Prof. Jan Rabaey for agreeing to participate on my dissertation committee and for his many hours of work as co-director and co-founder of BWRC. Every aspect of the environment which he and Prof. Brodersen have created here, from the facilities to the research mentality to the character of all the students and staff, has been truly inspirational and permanently shaped my vision of how research can and should be done. And as anyone who has worked with us knows well, thanks to Tom Boot for all the kindness and hard work he puts into the job of keeping BWRC on its feet on a day-to-day basis. Thanks as well to Brian Richards and Kevin Zimmerman for keeping the tools and systems we rely on for all this work running.

I am also grateful to Prof. John Wawrzynek for graciously serving as my qualifying exam committee chair and being such a valuable resource on reconfigurable computing going all the way back to my preliminary exam. Many thanks also go out to Prof. Paul Wright for generously participating on my (and countless other BWRC students') dissertation and qualifying exam committees.

I would like to thank all the students at BWRC who have come and gone during my extended stay here for making the weekdays more enjoyable and the retreats

such a pleasure. In particular, however, I would like to thank Hayden So and Chen Chang for their help in answering all my questions on BORPH and BEE so quickly and consistently, and for all the feedback they provided along the way which helped shape my own work.

Last, but certainly not least, I want to express my deepest gratitude to my family and friends, whose constant and unwavering support have kept me going during my graduate school career, which included some of the most difficult events I've had yet to face. I consider myself truly blessed to have such people in my life. So to my parents, Butch and Cheryl, and all my family and closest friends, I only hope that you can understand how appreciated you are, and I dedicate this work to you, as you have all made it possible.

In loving memory of my grandmother,
Irma Jean Riefer

Contents

Acknowledgments **i**

Contents **iv**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 BEE2 4

1.2 BORPH 6

1.3 Dissertation outline 7

2 Motivation **9**

2.1 Hardware implementation process 10

2.2 Design verification techniques 14

2.3 The direct verification approach 17

3 Platform Characteristics **19**

3.1 Design environment 19

3.1.1 BEE2 library extensions 21

3.1.2 Verification-specific library extensions 23

3.2 Implementation flow 25

3.3 System requirements 28

4 Hardware Architecture **30**

4.1 Variable network 31

4.2 Core debug controller . . . 34
4.2.1 Control operations . . . 38
4.2.2 Variable management logic . . . 42
4.2.3 Clock management logic . . . 43
4.3 External storage interface . . . 44

5 Software Interface 48
5.1 Debugger software process . . . 49
5.1.1 Hardware child process . . . 49
5.1.2 Hardware state cache . . . 52
5.2 Remote network service . . . 54
5.2.1 Hardware-specific functions . . . 55
5.2.2 General diagnostic functions . . . 58
5.3 Runtime interaction . . . 61

6 Programmability Improvements 67
6.1 Design time improvement . . . 68
6.2 Proof of concept . . . 70

7 Performance Results 75
7.1 Base hardware resources . . . 76
7.2 Timing and throughput . . . 79
7.3 Additional design examples . . . 81
7.3.1 Singular value decomposition . . . 82
7.3.2 Pocket spectrometer . . . 87

8 Conclusion 92
8.1 Summary of results . . . 92
8.2 Future opportunities . . . 94

Bibliography 97

A Hardware implementation data 99
A.1 Variable unit implementation . . . 99
A.1.1 `bdb_variable` . . . 99

A.1.2 bdb_variable_logic . . . 101
A.2 Core controller implementation . . . 105
A.2.1 bdb_core . . . 105
A.3 Memory interface implementation . . . 113
A.3.1 bdb_core_ddr_ctrl . . . 113

B Software implementation data **120**
B.1 BORPH-based bdb process . . . 120
B.1.1 main.h . . . 120
B.1.2 main.c . . . 121
B.1.3 client_handler.h . . . 126
B.1.4 client_handler.c . . . 126
B.1.5 varcmds.h . . . 140
B.1.6 varcmds.c . . . 142
B.2 Matlab verification routines . . . 171
B.2.1 bdb_connect . . . 171
B.2.2 bdb_disconnect . . . 172
B.2.3 bdb_readhist . . . 173
B.2.4 bdb_lookup_varids . . . 174
B.2.5 bdb_lookup_varnames . . . 174
B.2.6 bdb_lookup_varparams . . . 175
B.2.7 bdb_scale_values . . . 176

List of Figures

1.1 Physical architecture of a BEE2 board 5
1.2 Architecture and services provided by the BORPH kernel 7

2.1 Typical phases in the creation of a reconfigurable hardware design . . 11
2.2 Simulation time required for 200 cycles of a 100MHz system 15
2.3 Example of RTL simulation of a hardware design 16

3.1 Fixed-point complex addition/subtraction in System Generator . . . 20
3.2 Library components available for use on BEE2-targeted designs . . . 22
3.3 Variable block implementation for BEE2 using System Generator . . 23
3.4 Dialog box parameters for each variable block 24
3.5 Core debugging controller block implementation for BEE2 25

4.1 Basic architecture of debugging hardware infrastructure 30
4.2 Block diagram of a variable unit 32
4.3 Variable unit connectivity 34
4.4 State machine for user command translation 36
4.5 Interface logic for operations which modify variable state 43
4.6 Organization of clock domains to regulate design execution 44
4.7 State machine for regulating runtime and on-demand memory accesses 46

5.1 Flow diagram showing the operations performed by `bdb` 50
5.2 Dialog box showing all available end-user debugging routines 63

6.1 Example of a 32-bit unsigned adder with tunable range and saturation 69
6.2 Design of a traditional, fixed 12-bit MAC unit 71
6.3 Top-level system view of replicated 12-bit MAC example 71
6.4 Design of a paramterized, 32-bit MAC unit 73
6.5 Top-level system view of replicated 32-bit MAC example 73

7.1 System model used for base hardware infrastructure results 76
7.2 Method for inserting variables into the base hardware system 77
7.3 Post-routing device utilization of base hardware infrastructure 79
7.4 Post-routing critical path measurements for base hardware infrastructure 81
7.5 Top-level view of SVD design example 83
7.6 Contents of $U\Sigma$ block in the SVD design example 83

7.7 Post-routing device utilization of SVD design 86
7.8 Post-routing critical path measurements for SVD design 86
7.9 Top-level view of spectrometer design example 88
7.10 Structure of underlying computation in spectrometer design example 88
7.11 Post-routing device utilization of spectrometer design 90
7.12 Post-routing critical path measurements for spectrometer design . . . 90

List of Tables

1.1 Key performance metrics of a comparable FPGA and DSP 3
1.2 Comparison of leading Virtex-II Pro and Virtex-5 FPGAs 3

3.1 Parameters of the core debug controller hardware block 27
3.2 Parameters of the variable unit hardware block 27

4.1 BORPH software registers accessed by core controller 35
4.2 Allocation of bits in the `bdb_status_out` register 37
4.3 Summary of core controller commands 42

5.1 Contents of `/proc/$PID/hw/` used by `bdb` 51
5.2 Elements of `var_state` structure in hardware state cache 52
5.3 General diagnostic functions supported by `bdb` and the network service 60
5.4 Properties of each named element in the variable map structure . . . 64
5.5 Specialized verification library routines in Matlab client 65

6.1 Hardware requirements of the 8-way fixed 12-bit MAC system 70
6.2 Hardware requirements of the 8-way paramterized 32-bit MAC system 72

7.1 Synthesis resource estimates for base hardware infrastructure 77
7.2 Measurement of post-routing utilization of DDR2 memory interface . 77
7.3 Synthesis and post-routing critical path measurements 80
7.4 Synthesis resource estimates for SVD design 84
7.5 Synthesis resource estimates for spectrometer design 89

Chapter 1

Introduction

With the historical increase in the computational capability of semiconductor processing devices, fueled primarily by Moore's Law and the steady shrinking of the feature size of CMOS transistors, modern-day computing platforms are changing the paradigms traditionally associated with high-performance computing architectures, including both clusters of processor-based workstations and custom computing engines. High-perfomance computing, which was once a regime dominated purely by very large-scale "supercomputers" occupying entire buildings worth of real estate, can now be achieved by a variety of architectures, perhaps the most capable of which is a *reconfigurable hardware platform* consisting of an array of FPGAs (Field Programmable Gate Arrays) or other programmable hardware devices.

A reconfigurable hardware platform is considered to be any system which consists of programmable hardware devices which can be reconfigured as the desired application changes. The scope of reconfigurable devices has changed over the years, ranging from PLDs (Programmable Logic Devices), which behaved primarily as read-only memories (ROMs) for which the outputs could be assigned to any user-defined values, to FPGAs, for which the primitive logic cell is commonly a 4-input lookup table (LUT) which is connected to a complex routing network consisting of local interconnect and globally-distributable crossbar networks.

Beyond PLDs and FPGAs, a number of alternative reconfigurable architectures have been conceived and are continuing to be envisioned. However, regardless of the underlying physical architecture, one characteristic remains true: a reconfigurable hardware platform features devices which can have *any* arbitrary computational sys-

tem mapped onto its logic fabric, and can be repeatedly reprogrammed an indefinite number of times. Typically, the input description for such a reconfigurable system is a standard *hardware description language*, or HDL. The traditional application of HDLs, such as VHDL and Verilog, is to describe the functionality of a hardware system at a level that can be *synthesized* into primitive logical functions (such as boolean gates and flip-flops) that are understood by the target architecture. For a custom application-specific integrated circuit (ASIC), the target architecture is a standard cell library which is provided by a silicon foundry. In the case of a reconfigurable hardware device, the underlying logic cells and the interconnect architecture are well-understood by the vendor tools, and the HDL description of the system will be mapped onto the device primitives as efficiently as possible given the user-defined constraints and the capacity of the target device.

For the purpose of this work, the reconfigurable device of choice is the FPGA. While many different reconfigurable architectures have been conceived to date, the FPGA has remained one of the most constant and flexible microarchitectures available. Regardless of whether the FPGA microarchitecture is the most ideal for all applications or it has simply survived longest due to the maturity of the design tools and the general understanding and robustness of the 4-input LUT primitive cell, FPGAs have remained the most commonly used reconfigurable devices on the market thus far. For this reason, the remainder of this work will focus primarily on the use of FPGAs as the target device for reconfigurable hardware applications, although all efforts have been made to make the overall approach as device- and platform-agnostic as possible.

When this project was initially conceived, the flagship reconfigurable device offered by Xilinx was the Virtex-II Pro FPGA. While the XCV2P100 FPGA was the highest-capacity device announced at the time, the slightly smaller XCV2P70 product was more readily available (and somewhat more cost-effective) and was therefore considered the most attractive high-performance FPGA on the market. Table 1.1 shows some relative performance metrics which were measured between the XCV2P70 FPGA [18] and a leading digital signal processor (DSP) at the time, the Texas Instruments C6415T [13]. The source data for this experiment can be found in [6].

As shown in Table 1.1, not only is the leading-edge FPGA architecture superior to the comparable DSP processor in terms of raw performance, but it also has an advan-

Table 1.1: Key performance metrics of a comparable FPGA and DSP

	XCV2P70-7	**C6415T-1G**
Computation rate (Gop/s)	72	4
Power efficiency (Gop/s/W)	2.72	1.84
Price/performance (Mop/s/$)	31.0	14.81

tage in terms of the price per unit of performance. This is a particularly interesting metric: although it may not be surprising that higher-performance architectures exist in comparison to special-purpose processors, the fact that it is in fact *cheaper* to implement algorithms directly on an FPGA as opposed to compiling software onto a specialized processor makes a compelling argument for using reconfigurable hardware for advanced computing applications.

Table 1.2: Comparison of leading Virtex-II Pro and Virtex-5 FPGAs

	XCV2P100	**XC5VLX330T**
Process technology	0.13 μm	65 nm
Slices	44,096	51,840
Logic cells	99,216	331,776
Block RAM	7,992 kb	11,664 kb
Peak DCM frequency	550 MHz	450 MHz

In contrast to the leading-edge FPGA device during the initial concept of this work, at the time of writing, the state-of-the-art FPGA device offered by Xilinx is the Virtex-5 platform [19]. The Virtex-5 FPGA is based on a 65nm process technology and can provide up to 330,000 logic cell equivalents in a single device. In addition to this fundamental logic capacity, the premium Virtex-5 device offers 11.7Mb of on-chip Block RAM, 12 digital clock manager (DCM) units, 24 RocketIO GTP low-power transceivers, and 960 general-purpose I/O pins. This represents a staggering amount of reconfigurable logic at the designer's disposal. The increase in overall performance which is apparent between the Virtex-II Pro and Virtex-5 generations are summarized in Table 1.2.

Despite the more recent advances in the capacity and performance of FPGA devices, for the remainder of this document, the platform which will be the focus of the underlying implementation is the BEE2 system. BEE2 represented a significant evolution in the usability of FPGA-based computing systems, and as such served as the

inspiration for the direct verification approach presented here. The following sections will outline the characteristics of the BEE2 hardware platform itself, as well as the integrated operating system intended for BEE2, known as BORPH. Finally, after an understanding of the implementation platform has been established, the remaining chapters of the document will be outlined.

1.1 BEE2

The hardware platform used as the engine for all the experiments and results in this text is BEE2 [6], the second generation of the Berkeley Emulation Engine [5]. As mentioned above, all of the work in subsequent chapters has been designed to be as platform-agnostic as possible, requiring only that the underlying hardware be somehow reprogrammable in nature, such as inherently the case with FPGAs. A basic understanding of the BEE2 platform, however, will help to understand some of the implementation details and the decisions behind them as they are presented later.

The BEE2 platform in its most common form is a single board containing 5 Xilinx XCV2P70 FPGAs with attached memory slots and an assortment of connectors and glue logic for off-board connectivity. The platform was designed to allow any number of BEE2 boards to be interconnected in a variety of styles and at different levels of abstraction, but most typical designs involve the use of a single BEE2 board as a standalone computing engine.

Although all the FPGAs on a BEE2 board have pin-to-pin interconnections to their neighbors, allowing the entire board to effectively act as a single logic fabric, the intended usage involves the central FPGA serving as a "control FPGA" and the four outer FPGAs serving as "user FPGAs". In practice, this means that the central operating system and off-board communication facilities are implemented on the control FPGA, and the raw computation is targeted to one or more of the user FPGAs. A single application can effectively consume anywhere from one to all four user FPGAs (potentially spanning multiple boards if designed accordingly), as well as custom logic using spare resources on the control FPGA if the designer chooses to modify or re-implement the base configuration. This architecture allows a BEE2 board to operate as a single computation host and an application to be arbitrarily assigned to

any user FPGA using predefined methodologies for requesting communication from the control FPGA. The physical architecture and interconnectivity exhibited on each BEE2 board is shown in Fig. 1.1.

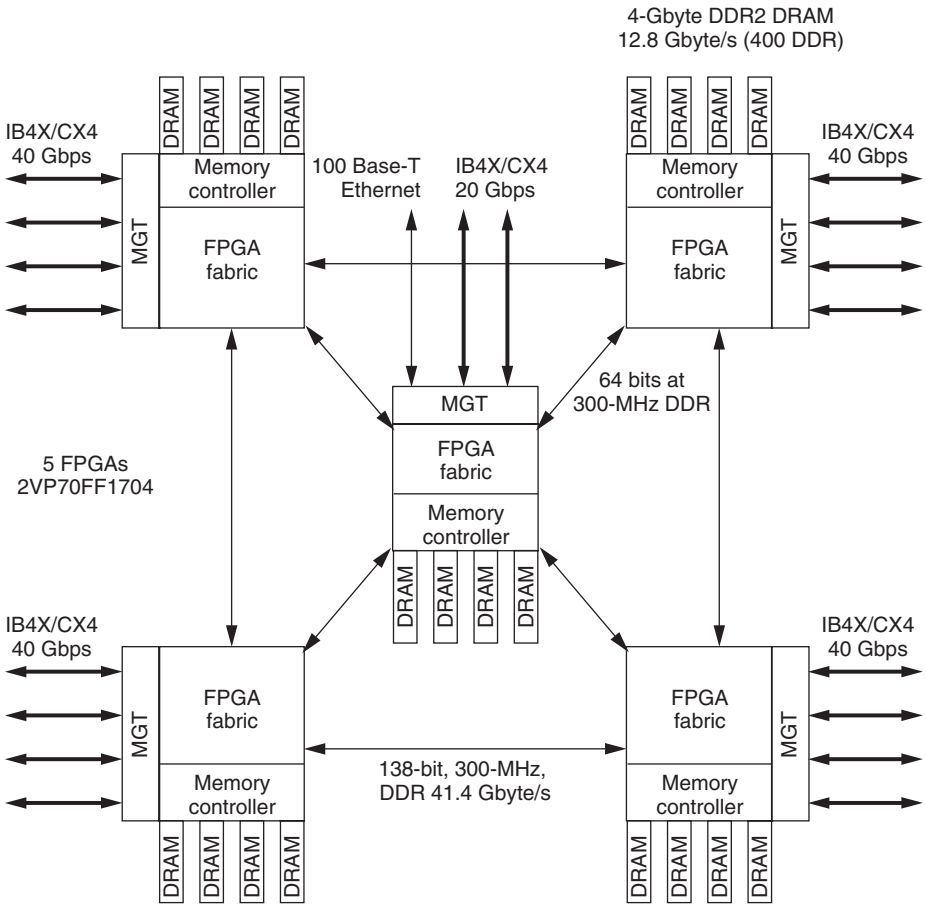

Figure 1.1: Physical architecture of a BEE2 board

All the remaining discussions in this text will assume that an application follows the originally intended architecture, where the control FPGA serves as a centralized host environment and user applications run on *only* a single user FPGA. The reason behind this restriction is due to the fact that the hardware verification infrastructure (or, more specifically, the core debug controller, presented in Sect. 4.2) must maintain control over the hardware design clock. To exert such cycle-by-cycle control over multiple FPGAs, each with internal clock management logic, presents a far more difficult synchronization challenge which is left for future work.

Attached to each of the five FPGAs (the sum of which includes both the central

control FPGA as well as all four user FPGAs) on the BEE2 board are four DDR2 DIMM slots. The pins allocated for each DIMM control interface are limited in the number of bits available (a consequence of the original BEE2 board design), effectively setting the maximum DIMM size for each module to be 1GB. Correspondingly, the total amount of memory that can be attached to each FPGA is 4GB.

While the BEE2 platform has since evolved in numerous directions in the hands of different researchers investigating different applications, the originally-intended architecture outlined here is the basis for all the work presented in this document. The complete details of the hardware components created for direct verification will be presented in Chapter 4.

1.2 BORPH

The current generation of the BEE2 platform features an integrated operating system called BORPH, the Berkeley Operating system for ReProgrammable Hardware [12], which runs on the control FPGA and manages all the resources of the BEE2 board. BORPH is an augmented version of the Linux kernel which runs on one of the integrated PowerPC cores in the control FPGA. As a fully compliant port of Linux, it inherently preserves compatibility with the existing system calls and APIs, as well as a large number of already implemented device drivers. While a full description of BORPH and its implementation is beyond the scope of this text, a basic understanding of its fundamental capabilities can be beneficial before diving into the details of the verification infrastructure.

The core purpose of BORPH, and that which is utilized in the implementation of direct verification presented in this work, is to load hardware "processes" onto an available user FPGA and establish all the external interfaces which have been declared in the hardware design itself. Some of these interfaces, such as software registers and shared on-chip memory, are directly accessible in the BORPH Linux kernel via predefined filesystem nodes called *ioreg virtual files*. Although not currently utilized by the direct verification infrastructure, BORPH can also establish and manage interconnections between hardware processes running on separate FPGAs. Fig. 1.2 shows a graphical representation of the services offered by BORPH between the software and hardware domains. The exact details of the software components relevant

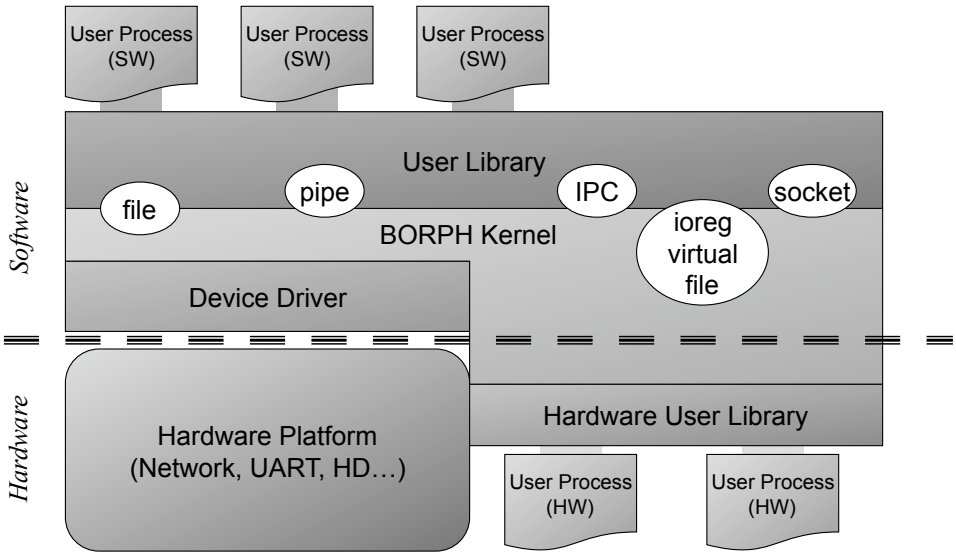

Figure 1.2: Architecture and services provided by the BORPH kernel

to direct verification are discussed in great detail in Chapter 5.

1.3 Dissertation outline

The remainder of this document will be organized as follows. Chapter 2 will discuss the current methods used for the analysis and verification of large-scale reconfigurable computing systems and highlight the key challenges which are addressed by the concept of direct verification. Chapter 3 will cover the current design environment and usage model for the BEE2 platform as well as a discussion of the platform features in general which are required for direct verification, which can be applied to virtually any platform other than BEE2. Chapter 4 will begin to present the actual implementation of the direct verification infrastructure by describing the details of the verification-specific hardware components. Chapter 5 will extend this discussion to the next level of abstraction by presenting the details of the software layer which support direct verification and connect the user to the capabilities which are available directly on the hardware. Chapter 6 will discuss how it becomes possible with a highly robust hardware verification methodology to accelerate the overall system programming time by enabling high-level functional simulation to occur at hardware speed. Chapter 7 will then summarize the actual performance results of the verifi-

cation infrastructure in terms of the additional resources required and the effect on overall system throughput. Finally, Chapter 8 will conclude this dissertation by summarizing the concepts presented throughout the previous chapters, and identifying some areas of opportunity for extending this work in the future.

Chapter 2

Motivation

Given that the net computational capacity of reconfigurable hardware platforms, such as FPGA arrays like BEE2, can significantly exceed that of traditional processor-based systems, the issue that must be addressed is what drawbacks and challenges are associated with such platforms that may limit their applicability. To choose an encompassing term, it could be said that reconfigurable hardware platforms suffer from a much higher difficulty of *programmability*, which would include the speed and efficiency in which the design description can be formulated as well as the time required to actually fine-tune and verify the ultimate functionality of an application. By including this net period of time consisting of both the creation of the hardware description and the verification of the final behavior of the design, the *programming* of a reconfigurable hardware platform through to its final configuration includes the entire process of producing a working system.

In terms of the means by which a hardware design is described, there are a large variety of programming languages and hardware generation tools available for targeting reconfigurable logic devices. Of course, the most commonly used hardware description languages (HDLs) of VHDL and Verilog can be used in virtually all systems, and are frequently used as the fundamental representation of the hardware in systems where the hardware is designed independently from form the functional models. However, as hardware capacities have become larger, and increasingly complex systems can be integrated onto a single device, numerous higher-level languages and design environments have been developed to raise the level of abstraction available for hardware design. For example, the languages SystemC [7] and HandelC [4] are based

on the standard C programming language, but with additional syntax and features useful for modeling or even driving hardware generation.

Typically, however, the ideal high-level language for describing a hardware design is dependent on the application domain. For example, SystemC may work perfectly well for systems which feature joint software and hardware behaviors and/or highly heterogeneous platform components. For signal processing and communications applications, a more powerful vector and numerical simulator, such as Matlab [10], is often used to verify the functional performance of an algorithm. More recently, hardware generation capabilities have even been built into the Matlab tool suite, and companion tools (such as Xilinx System Generator, which is used by the BEE2 design flow and is described more in Chapter 3) have also been available which add the ability to generate hardware from the underlying models. In addition, some applications, such as the Research Accelerator for Multiple Processors, or RAMP project [8], required the conception of a new language (called RDL) which mapped onto their own, custom infrastructure built directly on top of BEE2. This vast range of target applications and usage methodologies speaks to the power and flexibility of reconfigurable hardware as a computing platform, but certainly does not simplify the choice of any one "ideal" design language.

Because a variety of description languages already exist and are continuing to be developed, it was decided that an attempt to create an even more efficient, yet universally-applicable, design language would likely be fruitless. However, one common characteristic of all hardware platforms is the challenge associated with verifying the behavior of a system once it has been translated into an actual physical configuration for the hardware platform.

To better present this verification challenge, the following sections will discuss the current processes required to generate a physical hardware configuration as well as a discussion of the current tools and methods available for verifying design behavior on a hardware platform.

2.1 Hardware implementation process

Before investigating the verification techniques currently in use today, it is useful to understand the process which is involved in generating a physical configuration,

which defines the final behavior of the reconfigurable device or devices which serve as the computation engine for the hardware platform. Fig. 2.1 shows the steps which are typically involved in creating a design which will run on reconfigurable hardware.

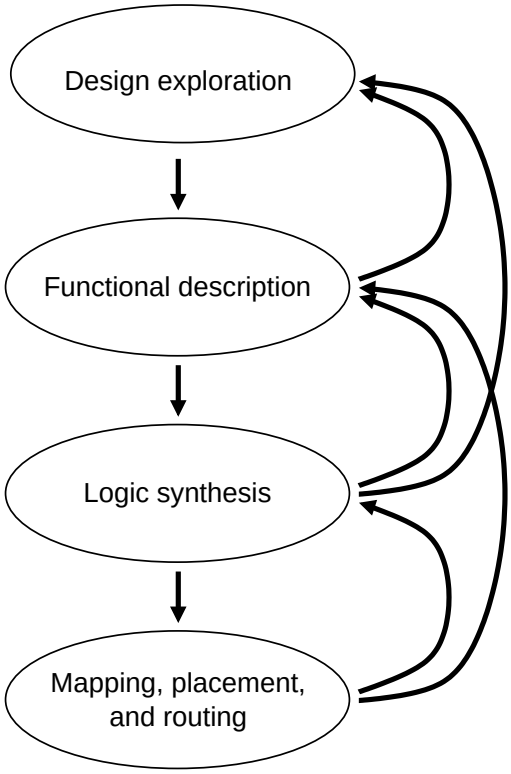

Figure 2.1: Typical phases in the creation of a reconfigurable hardware design

Like many other types of systems, the first step in the design process is to describe the desired behavior of the system. The language and environment used to create this description will vary greatly based on the application domain, as mentioned above. However, the goal of this phase still remains the same: to identify the core functionality of the system, and evaluate the ideal numerical or qualitative parameters of the design which satisfy all the constraints of the application. This *design exploration* process sometimes takes place in an environment other than that which is used to describe the physical implementation. For example, in the case of a communications system, the evaluation of high-level characteristics of the algorithm to be used may occur in Matlab by performing numerous simulations and analyzing the scope of the results. For the design of a novel networking protocol, a simulator such as `ns` may be used to evaluate the overall performance of the protocol before diving deeper into

the lower-level implementation.

Once the parameters of the system have been thoroughly investigated, the details of the *functional description* can be explored. According to the system parameters which were selected during design exploration, the functionality of the underlying computation can be derived. In the most traditional approach, this may even be performed by a separate group of designers who specialize in hardware design. With the availability of automated hardware generation built into the higher-level environment, the functional description of the system may be largely contained within the previously formulated system design. On a reconfigurable hardware platform, the functional description will be composed essentially of gate-level computational blocks, and the results will typically take the form of signal waveforms (the simulation of an RTL-level functional description is depicted in Fig. 2.3). This is a much greater level of detail than the numerical simulations which may have been performed in the previous phase, and similarly could take much more time to complete. However, only at this level are the details of the hardware implementation visible, such as clock cycles and delay estimates.

While the functional description of the hardware implementation starts to reveal the details of the physical configuration and its performance, only after the next phase of *logic synthesis* is the functional description of the hardware translated into its low-level logical functions in terms of the primitive logic cells available on the target device. Once logic synthesis has taken place, the designer can see a reasonably accurate estimate of the number of logic resources required by the system as well as an idea of its peak operating frequency. At this phase, the upward-facing arrows shown in Fig. 2.1 also start to pose a problem. If at any time these increasingly accurate representations of the final hardware implementation indicates that some system or platform constraint is no longer being met, the design process must be taken back to a previous stage (that is, if optimization techniques available at the current level of detail cannot assist in meeting the designer's goals).

Finally, once a design has been synthesized into its primitive, logic-cell-equivalent components, the physical implementation tools have the challenge of *mapping* all the logical functions in the implementation into the primitive cells of the reconfigurable device (packing multiple operations into fewer cells, when possible), *placing* all the elements in a design on the 2-dimensional array of logic cells (also in accordance to any

constraints imposed by the system or platform), and *routing* all the necessary signals between their source and destination logic cells. This is an extremely complex and time-consuming problem, and accounts for the vast majority of the overall physical implementation time. While the exact amount of time required is an unpredictable function of the overall device capacity, the amount of available freedom for placement and routing on the device, and the amount of slack available to meet user constraints, for a fully-utilized XCV2P70 device, as found on BEE2, the net time required for mapping, placement, and routing can take more than 24 hours. And once again, if at this point the designer finds that some global system constraint is no longer met, the design process must revert back to a previous stage.

Once a design has been fully placed and routed, an extremely accurate estimate of the final timing of the system is also provided. At this point, the physical implementation process is considered complete, and the actual *configuration bitstream* is created, which is used to configure the hardware device and begin computing real results. It should also be mentioned that traditionally, this entire mapping, placement, and routing procedure is completely "flat", meaning that if a single design element were to be changed, the entire hardware implementation would have to be re-generated. This is because any change, no matter how small, may affect the potential for improved mapping and placement, and will certainly affect the potential routing between elements. While most vendor tools do currently provide support for a modular implementation flow, safeguards must be taken in advance by the designer, and the areas of the device reserved by each module must be defined and floorplanned manually. This is clearly a slightly different approach to design, and is so far only useful in certain custom applications.

This section has hopefully clarified the sequence of steps involved between the conception of an application and the generation of the final configuration bitstream. There are clearly a number of complex steps involved, and therefore the minimization of iterations through this process would greatly reduce the overall time required to develop a complete system. The direct verification solution proposed in this work can help to accomplish this goal, and is introduced in Sect. 2.3. In the meantime, the following section will discuss the methods currently available for verifying a reconfigurable hardware design.

2.2 Design verification techniques

Following the sequence of phases involved in the physical implementation process, the first technique for verifying the functionality of a design is to simulate the system at the highest level of abstraction possible. The exact environment in which this takes place is, of course, dependent on the application domain. However, since the BEE2 platform featured in this work provides a design flow which offers the automated generation of hardware from a high-level system description (the exact characteristics of this design flow are described further in Chapter 3), it will be used as an example for these purposes. Because the inclusion of automated hardware generation implies that the designer does not need to perform any manual correlation of the functional hardware description to the system-level description, this model can be considered the fastest possible method for simulating the correctness of a system, since it takes place at the highest level of abstraction possible.

Fig. 2.2 shows a visual aid which demonstrates the amount of time that this coarsest-possible verification method requires. In order to simulate 200 cycles of hardware execution at 100MHz (representing 2μs of real time), the simulator requires 38.8s of CPU time. This represents *four orders of magnitude* between the two execution methods. Of course, this is most likely not a surprising result, although the exact magnitude of this difference shows the potential benefit of performing high-level system verification directly on the hardware platform. And it is important to recall that this is, in fact, the highest-level, and therefore *fastest*, method of verification possible. Plus, the classification of a system-level, numerical simulation of a design as equivalent to hardware verification is only true when the design environment in use features automated hardware generation which is assumed to be correct by construction.

The next-lower-level type of verification available is *RTL simulation*, which essentially involves the simulation of VHDL or Verilog either at the behavioral or logic gate level. While this may not incorporate the ultimate performance impacts of placement and routing, depending on the simulation models available by the hardware component library in use, it can approximate the actual delay through each hardware element. Regardless of the true delay properties of each hardware element, an RTL simulation will be cycle-accurate, in that it models the cycle-by-cycle functionality of the system, assuming the design clock frequency is chosen such that all logic opera-

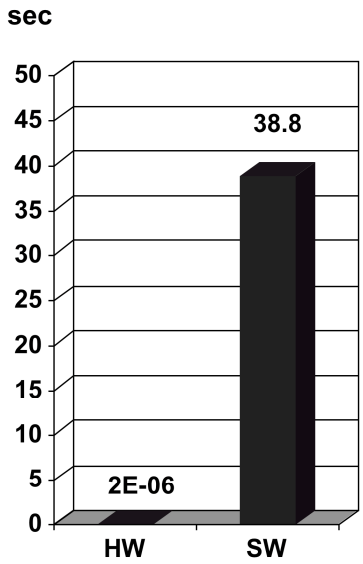

Figure 2.2: Simulation time required for 200 cycles of a 100MHz system

tions occur within the minimum period allowed. Fig. 2.3 shows the typical workspace for the RTL-level simulation of a design. As seen in the figure, RTL simulation offers a view of the actual signal waveforms which would be produced by the hardware implementation. It is currently well established that RTL simulation performs much slower than a higher level, numerical simulation. For this reason, the faster numerical simulation is preserved as the basis of comparison for direct verification.

Beyond the emulation of a hardware system within a software simulation environment, there are several methods for inspecting and verifying the correctness of data on a live reconfigurable hardware device. Vendor-supplied tools, such as Xilinx ChipScope [15], help to provide runtime access to on-chip resources. In addition, there are a variety of tools, which include Xilinx's System Generator hardware-in-the-loop feature [17], which abstract the hardware design as a simple black-box element which accepts inputs and produces outputs, and essentially serve as a software-hardware interface between the analysis environment and the running hardware. In addition, some novel approaches, such as the UNSHADES approach conceived in [14], attempt to combine the features of runtime data access and design environment integration with very minimal device overhead. And of course, in the absence of any automated tool which facilitates the verification of a running reconfigurable hardware system, the old-fashoined approach of directly capturing and monitoring signals of interest is

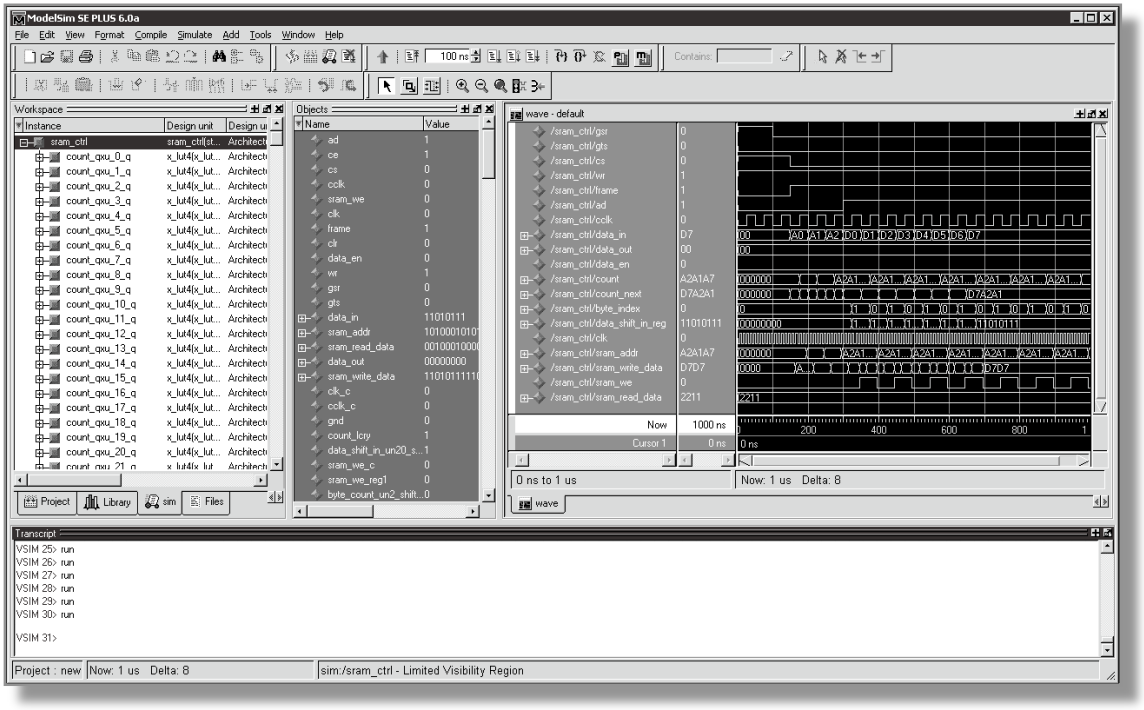

Figure 2.3: Example of RTL simulation of a hardware design

always available on any platform which provides a physical board connection for the probing of signals. Each of these approaches are discussed briefly below.

Xilinx ChipScope (as well as its analog for the Altera architecture, SignalTAP [1]), are popular tools which provide the ability to capture signals of interest on the running hardware, as well as to define conditions of interest which should cause the device to halt execution and wait for user input. In order to provide this functionality, the signals of interest defined by the user are captured in integrated on-chip RAM (a particularly finite resource). In addition, the hardware design is required to run on the lower-speed configuration clock, as the configuration subsystem on FPGAs which serves to access the device configuration and the default initial values of on-chip registers typically is controlled by a separate clock from the primary logic fabric. Because on-chip, embedded RAM is somewhat scarce on reconfigurable devices and may detract from the resources available to the design itself, as well as the fact the the configuration clock imposes an automatic reduction in performance, a superior solution was desired for use to fully verify computational systems on reconfigurable hardware.

Similarly, a large number of implementations exist for externally connecting to and receiving data from a hardware system. In this configuration, the hardware acts essentially as a simple co-processor — while data can be provided to and received from the hardware design, there is no support for the observation (and consequently, the manipulation) of internal system state. Therefore, these methods of simply communicating with the hardware device are not considered adequate for the purposes of fully verifying a reconfigurable hardware design.

Finally, alternate approaches to the verification of reconfigurable hardware platforms have been conceived, one of which is the UNSHADES system [14]. The UNSHADES tool connects to its target FPGA externally via the JTAG configuration port. This allows UNSHADES to regulate the execution of the device and capture any on-chip device data by reading back the current value of a signal. This requires virtually zero overhead in the hardware design, since purely external interfaces are utilized. However, this still requires the design to run synchronously with the JTAG configuration clock, which is much slower than the actual design clock. By integrating with the hardware generation data files themselves, UNSHADES is able to back-annotate values read from the hardware into the original design environment, which is a very powerful feature for the verification of a complete system on the hardware platform.

While all the approaches above offer a range of choices for the verification of a hardware design, none provide all the components necessary to fully assist with the verification of a complete system directly on the hardware platform. This is the goal of the direct verification approach presented in the next section.

2.3 The direct verification approach

The approach to *direct verification* which is conceived in this work attempts to improve the accessibility and mutability of data on the running hardware, which was previously only partially available by other tools. In addition, the application of direct verification can allow high-level system parameters, previously only altered during design exploration, to be characterized at runtime using the full throughput of the hardware platform.

At its root, direct verification introduces the concept of variables (one very familiar

in the software domain) to the reconfigurable hardware domain. Variables can be read or written at any time, and their cycle-by-cycle values are recorded in external storage on every cycle. In addition, the hardware design clock can be manually throttled either by the user or via dynamically-assignable variable assertions which are capable of triggering a same-cycle breakpoint in the hardware system which halts design execution until the user can observe the cause of the breakpoint and repair it. In addition, all the verification features necessary for this functionality are available in the original design environment, such that the same analysis tools which were traditionally exploited during design exploration can still be used, even though the actual execution of the computation is occurring at hardware speed.

The remaining chapters in this dissertation will present the details of the direct verification approach at every level of its implementation.

Chapter 3

Platform Characteristics

Before discussing the details of the verification methodology itself, it is important to understand the complete characteristics of the platform used in this work. This includes not only the expected features of the underlying hardware, but also the design environment used to describe the system under test and the implementation tool flow used to produce the actual hardware configuration. This verification methodology has been designed to be as platform-agnostic as possible, such that with some reworking of the hardware and software interfaces, the same approach may be taken on any alternative reconfigurable hardware system. However, some insight into the platform and environment utilized in this work is highly beneficial for understanding the background behind the hardware and software implementations in the following chapters.

The sections below are organized as follows. First, the design environment used to describe the system under test is presented. Second, the implementation tool flow and sources of automated hardware generation are described. Finally, the set of core features expected from the hardware for verification is discussed.

3.1 Design environment

The current design flow used on BEE2 platforms features a combination of Simulink [11], a graphical signal processing language in the Matlab suite of tools, and System Generator [17], a companion product offered by Xilinx which serves as a block library for use with Simulink as well as an automated driver for the hardware imple-

mentation tool flow. This graphical, signal-processing-oriented description language lends itself particularly well to communications algorithms (which was the original application domain of the BEE platform), but also supports "black box" components which can be described in the more traditional hardware description languages of VHDL and Verilog. This allows some flexibility in supporting a range of application domains which may not map ideally into a graphical, dataflow-style description at all levels.

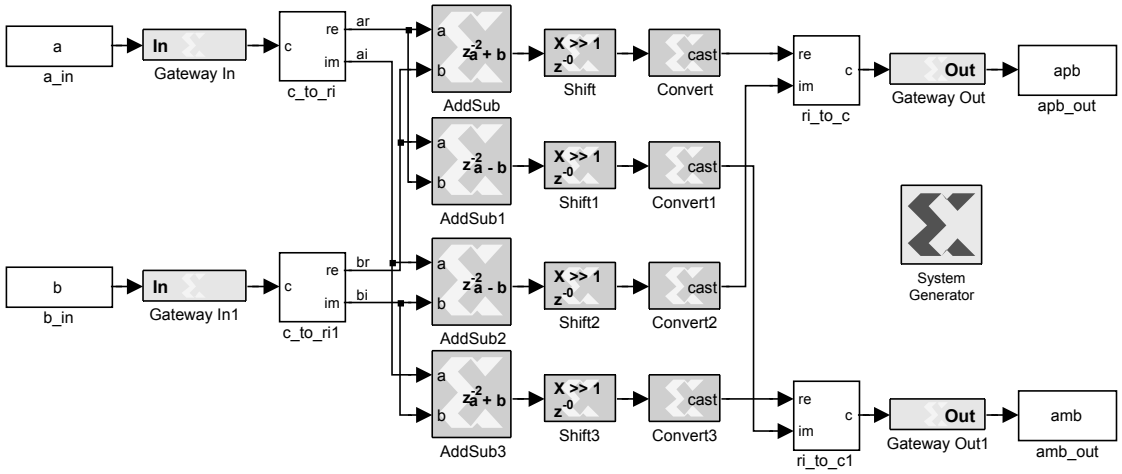

Figure 3.1: Fixed-point complex addition/subtraction in System Generator

Fig. 3.1 shows an example of how complex addition and subtraction could be implemented in this environment. The set of library components available as built-in elements in System Generator range from very primitive logical operations (such as binary boolean functions) to simple arithmetic (such as addition, subtraction, and multiplication) to highly complex signal processing routines (such as fast Fourier transforms and CORDIC division). In the figure, the System Generator primitives are identifiable by the block-X Xilinx logo. Simulink itself also supports hierarchical design via *subsystems*, and therefore the user can create any frequently-used operation as a single block in their own custom library or libraries. In Fig. 3.1, four subsystems are instantiated (with the names `c_to_ri` and `ri_to_c`), each of which performs the merging or splitting of real and imaginary components into or from a single signal bus representing a complex number.

Also visible in the example is the ability to integrate functional simulation with

the hardware design. The *gateway in* and *gateway out* blocks define the hardware boundary. Within the hardware domain, all blocks behave functionally equivalent to the underlying physical implementation, with each simulation step corresponding to one cycle of the hardware design clock. Outside of these gateway blocks, however, the user can place any arbitrary Simulink or Matlab components which can assist with input generation and/or output data analysis. Finally, all designs must include a System Generator block at the top level of the system. This is a utility block which defines several global parameters (such as the type of FPGA being targeted and the desired clock rate) and provides several pushbutton routines which will generate a hardware implementation.

While System Generator provides a starting point for developing FPGA-based processing systems, some additional infrastructure is required for a design to operate ideally on the BEE2 platform. The following subsections further describe the design elements utilized by the general BEE2 framework as well as the components specifically created for verification.

3.1.1 BEE2 library extensions

To facilitate system design on the BEE2 platform, a library of design elements is provided which includes BORPH-addressible hardware structures as well as off-chip interfaces for integrated on-board components and direct pin-to-pin I/O. This library both simplifies the accessibility of hardware resources and provides physical placeholders during the hardware assembly phase, which is discussed further along with the implementation flow details in Sect. 3.2.

Fig. 3.2 shows the library components available to BEE2 users, which can be instantiated as desired within Simulink in conjunction with System Generator. The two components specifically used for verification are the *Debug Controller* and the *variable* blocks, shown in the top-left corner of the figure and discussed further in Sect. 3.1.2. Several of the remaining blocks are designed specifically to create software-accesible memory resources from within the BORPH operating system and are named *software register*, *Shared BRAM*, and *Shared FIFO* in the figure. All the remaining blocks are provided for accessing specific hardware resources on the BEE2 board, such as general purpose I/O pins, analog-to-digital converters, and high-speed serial

interconnect.

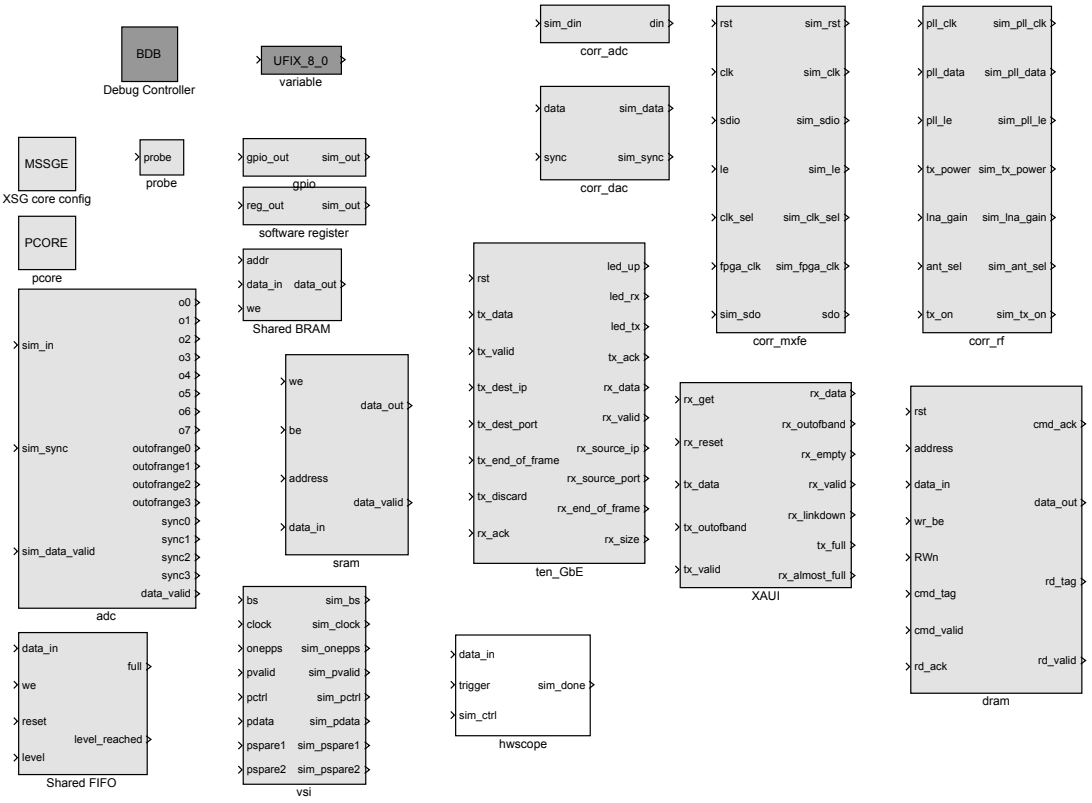

Figure 3.2: Library components available for use on BEE2-targeted designs

Similar to the importance of the System Generator utility block mentioned in Sect. 3.1 and shown in Fig. 3.1, BEE2 designs must instantiate an *XSG Core Config* block which defines important global parameters, such as the target device and desired clock source. These parameters can vary based on the user's exact platform configuration and performance requirements. This block also serves an important purpose as a placeholder component for the entire System Generator design during the hardware implementation phase, which again is covered further in Sect. 3.2. Additionally, a *pcore* block is provided to allow the user to add any custom hardware core which is already in the format expected by the implementation flow.

The BEE2 library components described here are necessary to provide convenient, configurable access to the plentiful resources of the hardware platform. For the purpose of verification, however, the only components utilized by the approach conceived

in this work are the software-accessible registers and the DDR2 memory controller. Although, on alternate platforms, these interfaces could be implemented in any possible way, and therefore the verification methodology conceived here is not dependent on any specific properties of BEE2.

3.1.2 Verification-specific library extensions

Beyond the base set of library components made available on BEE2, two additional blocks were designed for the purpose of verification, each of which correspond to the hardware structures described in great detail in Chapter 4. The first of these components, and perhaps the most critical due to its tight integration with the hardware design under test, is the *variable unit*. The second component is the core *debug controller*, which, similarly to the System Generator and *XSG Core Config* blocks mentioned above, must be instantiated exactly once at the top level of the design under test, and which defines several important global parameters relevant to verification.

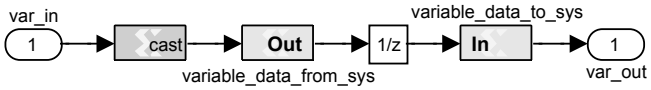

Figure 3.3: Variable block implementation for BEE2 using System Generator

Fig. 3.3 shows the model of a variable as used within System Generator on the BEE2 platform. As seen clearly by the simplicity of the figure, a designer who wishes to incorporate direct verification into their hardware design needs only to understand that the variable unit itself infers one cycle of delay into the hardware design (the exact reason for this required delay element relates to the performance of the system under test, and is discussed in Sect. 4.1). The most important aspect of each variable unit is its name within the hardware design. The name given to a variable block within Simulink becomes its unique name within the verification infrastructure. In the hardware domain, assigning a unique, identifiable name to a physical signal is a relatively advanced extension to the traditional approach of manually observing raw signals.

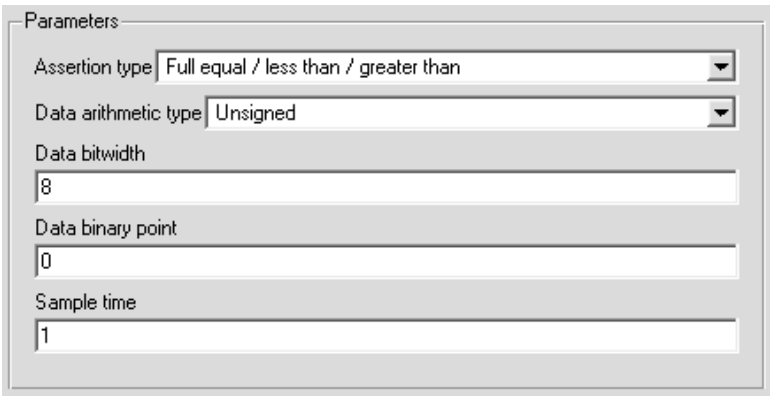

Figure 3.4: Dialog box parameters for each variable block

With respect to the parameters defined for each variable unit in the design (shown in Fig. 3.4), the first (labeled *Assertion type*), affects the style of logic inferred for runtime assertion checking (which is further described in Sect. 4.1). The next three parameters, *Data arithmetic type*, *Data bitwidth*, and *Data binary point*, are data-type and precision parameters which are specific to System Generator and affect the way in which the physical hardware signals are connected back into the design under test. Finally, the *Sample time* parameter is also specific to System Generator and affects the rate at which the register within the variable unit is actually enabled to latch its output. This is a consequence of the way in which System Generator implements multi-rate systems, which is accomplished by throttling the clock enable signal sent to each register in the hardware design.

Fig. 3.5 shows the contents of the core debug controller which manages the hardware/software and external data storage interfaces relied upon by the verification infrastructure. The most significant part of the core debug controller is the core logic block, which contains a state machine that accepts user requests from the software layer and drives the necessary signals to perform the requested operation in hardware. The exact implementation of the core debug controller is discussed in great detail in Sect. 4.2. Attached to the core debug controller is a BEE2 library component for a DDR2 memory interface. The attached DDR2 memory bank serves to store the history of all variable data samples in the design, and its physical implementation is also discussed in much greater detail in Sect. 4.3.

The library components presented in this section represent the full functionality

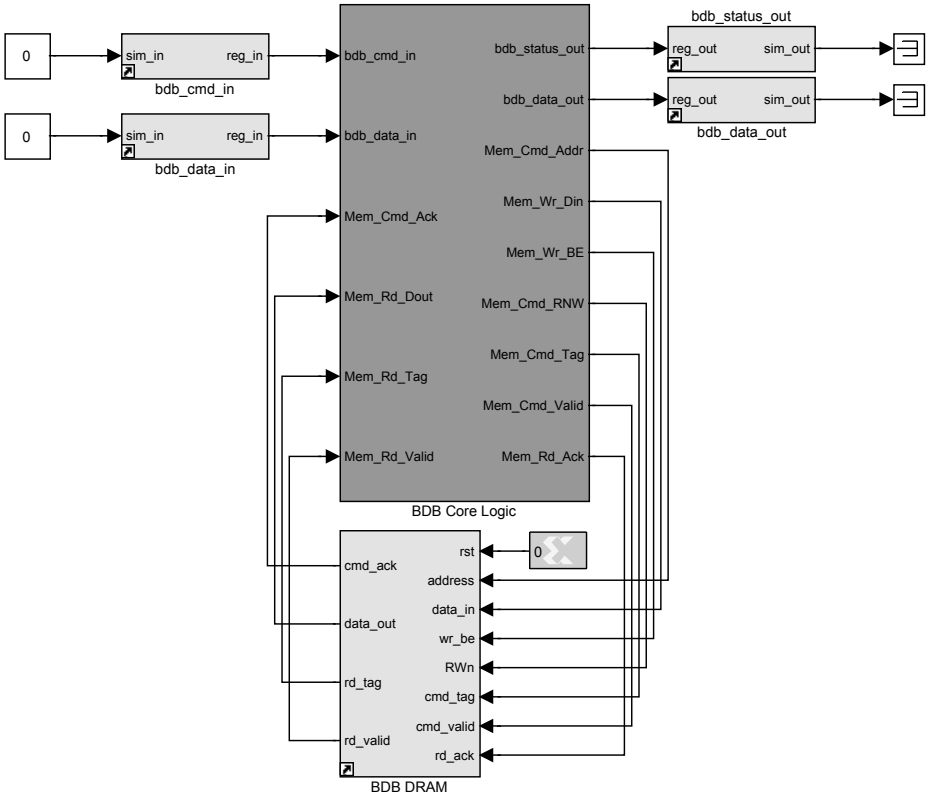

Figure 3.5: Core debugging controller block implementation for BEE2

of both the BEE2 hardware platform and the hardware verification methodology conceived in this work. While they are naturally tailored for use with BEE2 and the System Generator design framework, they could, in theory, be easily transformed into similar physical components and/or design abstractions on alternate platforms. From a design perspective, it is only advantageous that the user have some mechanism for declaring variables in their original design description, and that the implementation flow (described below) provide a means for automatically defining and interconnecting these variables to the rest of the hardware infrastructure.

3.2 Implementation flow

The library components contained within the previously described design environment provide the user with a means of declaring verification components (and consequently, hardware signals of interest) in their system. While the declaration

and parameterization of signals relevant for verification is critical to the user, in the hardware domain it is equally (if not more) unique to automatically generate the logical resources necessary to enable runtime access and analysis of in-system data. The generation of such hardware resources occurs during the physical implementation flow, which is necessary in order to abstract the underlying implementation away from the user.

From the perspective of verification, the only requirement of this approach is that there is some form of automation available between the functional description of the system (which could be any type of HDL or high-level language) and the final generation of a physical hardware configuration. In the design flow used by BEE2, this translation between custom library blocks and their hardware implementations is handled by class methods within Matlab which share a common API.

The BEE2 design flow provides a mechanism in which specially-tagged blocks within the Simulink design are recognized during compilation, and rather than have the contents of the block handled by System Generator, the generation of a hardware description is left to specific Matlab routines. This is possible because the final assembly of the hardware implementation is performed in a higher-level tool, namely, the Xilinx Embedded Design Kit (EDK) [16]. EDK was the natural choice as the final hardware representation, as it is suited for interfacing hardware to an embedded processor, which is required by the current version of the BORPH infrastructure.

While an exhaustive description of the BEE2 design infrastructure is beyond the scope of this document, a basic understanding of the means by which verification components are assimilated into the hardware design may be helpful before discussing their underlying architecture. The top level of a hardware design in EDK is defined by an MHS file (Microprocessor Hardware Specification). The main advantage to the MHS specification format is that features such as buses and processor-accessible address spaces can be easily and compactly described. Hardware cores, which can range from complete microprocessors to basic bus concatenators, are instantiated individually along with their port connectivity and parameter values.

Each specialized hardware component relevant to BEE2 and/or BORPH corresponds to a single hardware core within EDK. Each hardware core has a set of predefined parameters which are assigned for each instance of the core in the MHS file. The definition of each hardware core is part of the base hardware library which is

referenced by EDK for any system built for BEE2. The declaration and instantiation of each hardware component is performed by a specific Matlab class method for each library component (the name of which is `gen_mhs_ip`, as the method is responsible for writing the relevant section to the MHS file). In this manner, the base hardware implementation is augmented by each library instance until the net system is assembled, and the physical configuration is subsequently generated by the back-end tools.

Because of this one-to-one correspondence between the hardware cores recognized by EDK and the library components recognized by the BEE2 design flow, there are exactly two hardware cores relevant for verification: one for the core debug controller, and one for each hardware variable unit. The Matlab class definition for each of these components consists of two methods: one constructor which is required for all library components and derives parameter values from the Simulink block itself, and the `gen_mhs_ip` method which writes the necessary lines to properly instantiate the component in the MHS file. While there are additional methods available in the BEE2 API, only these two are utilized by the verification components. Table 3.1 lists the parameters which must be defined in the MHS file for the core debug controller instance, while Table 3.2 lists the parameters which are defined for each variable unit.

Table 3.1: Parameters of the core debug controller hardware block

Parameter	**Description**
`NUMVAR`	The total number of variables in the system
`SELBITS`	The number of bits required to address a variable
`W`	The number of bits allocated for each variable sample in memory

Table 3.2: Parameters of the variable unit hardware block

Parameter	**Description**
`NUMVAR`	The total number of variables in the system
`VARID`	The unique numerical ID of this variable
`BW`	The actual bit-width of the variable in the hardware design
`SW`	The number of bits allocated for each variable sample in memory
`USE_SIGNED`	Selects signed or unsigned data interpretation
`ASSERT_TYPE`	Selects type of assertion comparison logic to be generated

Underneath the core debug controller and each variable unit is a parameterized behavioral VHDL entity, ready for synthesis. For each parameter defined for the

hardware core, there exists a corresponding VHDL generic which is passed to the synthesis tool. In this manner, high-level system parameters which are initially assigned to Simulink library blocks are read by the BEE2 design flow, written to the instance declaration for each hardware core in the MHS file, and finally interpreted by the synthesis tool to drive the physical hardware configuration. This process provides all that is necessary for user-defined verification elements to be automatically generated in hardware. A complete description of the hardware architecture of the core debug controller and variable units is presented in Chapter 4.

3.3 System requirements

As mentioned frequently throughout this document, the verification methodology presented here is intended to be as general-purpose as possible, such that it could be applied, with some redesigning of the external interfaces and automation functionality, to work on any reconfigurable hardware platform. This section serves to separate out and identify the platform features which are necessary to preserve the full functionality of this approach.

First and foremost, it is fundamental that some interface be provided between the original design environment and the hardware application itself. On BEE2, the on-board operating system BORPH provides software access to on-chip hardware resources and, as an enhanced form of the standard Linux kernel, supports standard networking protocols. In the absence of BORPH on any other platform, the only expectation of the methodology conceived here is that there is some form of accessible interface between the running hardware application and the outside world. This can be as robust as BORPH with its fully functional network interfaces, or as simple as a host workstation connected via a direct wired interface such as USB, IEEE 1394, or even a traditional RS232 port. The accessibility of runtime design data back to the original design and analysis environment is critical for the overall utility of direct verification on the hardware.

Second, it is necessary for the hardware design to have access to some form of attached storage. On BEE2, this is provided by the directly-connected DDR2 DRAM banks attached to each processing FPGA. However, even by considering the current implementation of the DRAM interface (covered in detail in Sect. 4.3), it can be seen

that the runtime debug controller is really only concerned with two aspects of the storage configuration: how variable data is routed to the storage medium and when the storage medium is ready to accept a new batch of variable data. Therefore, the actual configuration of external storage is open to be of any type. Of course, like any memory system, a hierarchical storage architecture could also be leveraged to expand the net data storage capacity as large as desired. For example, by implementing either attached disk controllers or a network interface capable of communicating with remote storage services (such as NFS, or any other storage client accessible via network protocols), runtime data can pushed back to secondary or tertiary storage periodically as on-board memory is filled. The time required for pushing data off-board will naturally come with a performance penalty, but does provide flexibility as needed based on the analytical requirements of the user.

Lastly, beyond the inclusion of off-chip communication and data storage facilities, the approach conceived here only requires some form of automation in the design flow used for hardware generation (the automation features utilized in this approach are discussed in Sect. 3.2). This automation can occur at any stage of the implementation flow: with a purely HDL synthesis-based design environment, the insertion of debugging logic would most likely have to occur within the hardware description itself by modifying the user's design directly, perhaps through the use preprocessor directives which can optionally be ignored in a production build, or alternatively through verification support built into the synthesis tool. With a higher level, module-based design flow, debugging controls can be added to the library of available design units quite efficiently, as demonstrated by the verification components presented in Sect. 3.1.2 which are used in this approach.

In summary, while BEE2 is a highly scalable and flexible reprogrammable hardware platform, all the features of the verification methodology presented here are dependent on only three characteristics: off-board communication mechanisms, attached off-chip storage, and integration into the design flow for the automation of hardware generation. Any platform which can provide these features would be capable of supporting a similar approach to direct hardware verification.

Chapter 4

Hardware Architecture

The hardware components integrated into the design under test are by far the most critical ingredients of the debugging infrastructure. Is it the use of elaborations to the hardware design, rather than external software applications, that enables design verification to occur at near real-time speeds. In addition, because debugging components are integrated directly into the design under test, the applied microarchitecture can have a noticeable effect on the performance of the debuggable system.

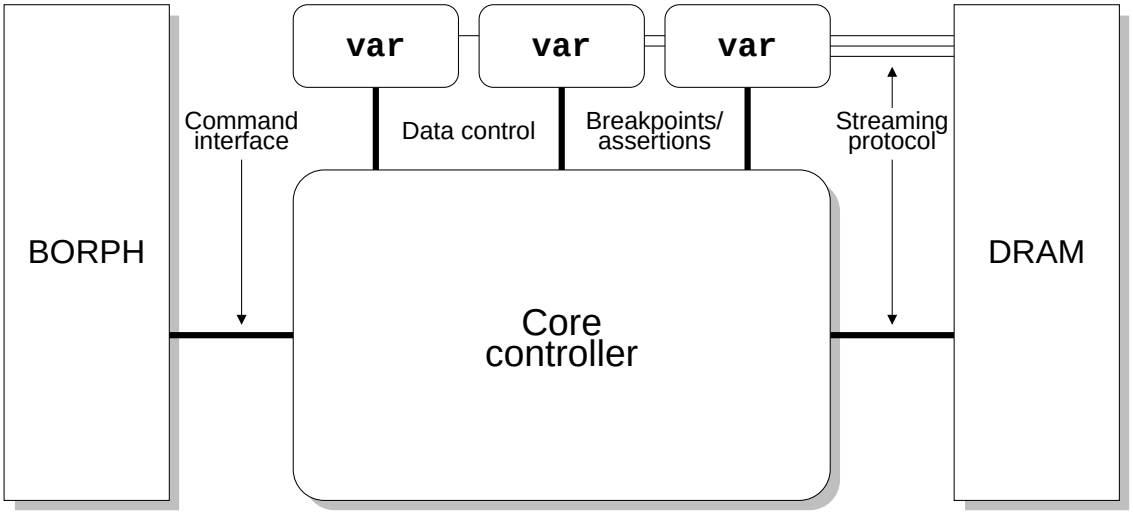

Figure 4.1: Basic architecture of debugging hardware infrastructure

There are effectively three fundamental aspects of the hardware portion of the debugger: the variable network, the core debug controller, and the external storage interface. The variable network consists of the individual variable units themselves,

each of which contains logic to control its data source and perform dynamic assertion checking. The core debug controller is responsible for temporal regulation of the hardware process, such as executing design breakpoints on assertion failures and throttling the design clock during memory access, as well as accepting and translating user commands from the software layer. Finally, the external storage interface performs the necessary tasks to stream variable data to attached memory during design execution and read back memory contents when data history is requested. Each of these features are described in full detail in the following sections.

4.1 Variable network

The variable network consists of one hardware *variable unit* for each variable declared in the user's design, as well as all the necessary connections to and from the core controller. As such, the variable network is the component of the debugger which interacts directly with the functional design itself and requires careful attention to resource efficiency and timing implications.

Each hardware variable unit consists of a register for the value itself, several parameter storage registers, an output selection mux, and assertion comparison logic. Fig. 4.2 is a block diagram which shows the organization of the variable logic. The variable unit was designed to be as efficient as possible, while providing the level of functionality needed to support the rich set of debugging features available to the designer.

There are four parameters for each variable which must be stored in local registers within each hardware unit: a *force value*, *source selection*, *threshold value*, and *condition mask*. The force value is a numeric value of the same size and precision as the variable itself which can be set by the user to override the output value of the variable. This feature is highly useful for both exploring design behavior and working around temporary bugs. The source selection is a one-bit register which holds the output mux select signal. This is necessary so that each variable may individually and persistently overridden as desired. The threshold value, also of the same size and precision as the variable itself, defines the basis for assertion comparisons. This value can be dynamically set at runtime by the user for customized assertion checking. And lastly, the condition mask is a 2- or 3-bit register, also dynamically accessible by the

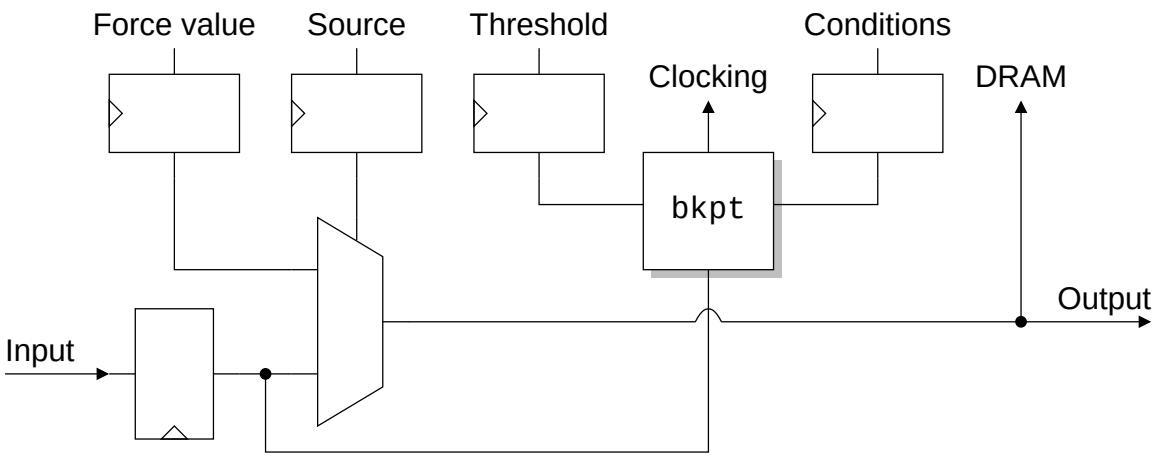

Figure 4.2: Block diagram of a variable unit

user, which defines the assertion conditions which should cause a breakpoint in the design (i.e. stop the design clock and wait for user intervention).

The input to the variable unit is directly latched to a design register, which is connected to the gated system clock. This is the only register that is "visible" to the designer, and therefore this unit cycle delay is modeled in the original design environment. The output of this register is fed to the output selection mux, which chooses whether the normal system value or the user-defined force value is passed on. By placing the mux after the design register, the computed system value is always preserved, allowing a `force` operation to be undone at any time. In order to support the manual overriding of variable values, it is unavoidable that one logic level be inserted in the design's critical path – the decision to place the mux after the design register was made purely to preserve value driven by the system. The output of the selection mux is fed back into the design with no further inline delays.

The last component of the variable unit, and the most critical in terms of the overall system timing under debugging, is the assertion comparison logic. Each variable can be defined by the user to have one of three styles of assertions: none, basic inequality, or full magnitude comparison (greater-than/less-than/equal-to). This option is given to the user for the purpose of potential resource savings. For variables for which it is known that *only* a certain, limited type of assertion checking is needed, some gates can be spared versus a full magnitude comparator. For example, the user may define a variable for a semi-constant input in their design that sets some high-

level soft parameter in the hardware. In this case, there is no reason to instantiate any comparison logic, as the input value within the design is constant. Alternatively, the user may define a variable for a signal which has a small, discrete number of possible values. In this case, a basic inequality comparator may be all that will ever be needed. It is important to note, however, that the methodology intended in this approach is to minimize iterations through the hardware implementation flow. Therefore, the user is always encouraged to opt for greater functionality at design time unless it is known with certainty that a feature will not be required.

In order to maximize the operating frequency of the hardware implementation, the comparison logic was arranged to operate on the direct output of the design register. This structure allows the comparator delay path to lie completely in parallel to the design critical path. It is also because of the comparison logic that a zero-cycle-latency variable unit results in a much more significant performance penalty. In theory, the input register could be removed and the entire variable could function as a purely combinational logic block. This, however, would insert one mux *per variable* into the design critical path, *plus* the entire delay through the comparator and clocking circuitry. While the hardware domain does already offer a very large speed advantage over simulation and can afford some slowdown, it was decided that signals of interest in synchronous logic designs typically terminate in pipeline registers already, such that the performance advantage far outweighs the potential design limitation.

In this manner, the assertion comparison logic is always active, and in the event that a condition evaluates as true and the corresponding condition mask bit is set, the variable unit will assert its *break signal*. The break signal is sent directly to the core controller, and will cause the design clock to halt within the same cycle. Management of the design clock within the core controller is covered further in Sect. 4.2.

The connections between the variable units and the core controller are summarized in Fig. 4.3. Each variable unit has its own set of four inputs and two outputs to and from the core controller, and all variables share one common value bus driven by the core controller. As only one user request can be performed at a time, it is unnecessary for each variable to have its own dedicated input value. Therefore, only independent write enable signals need to be driven to each variable unit to determine which variable parameter should be written. On the other hand, since each variable obviously has its own unique value, a separate data output must be sent to the core controller by

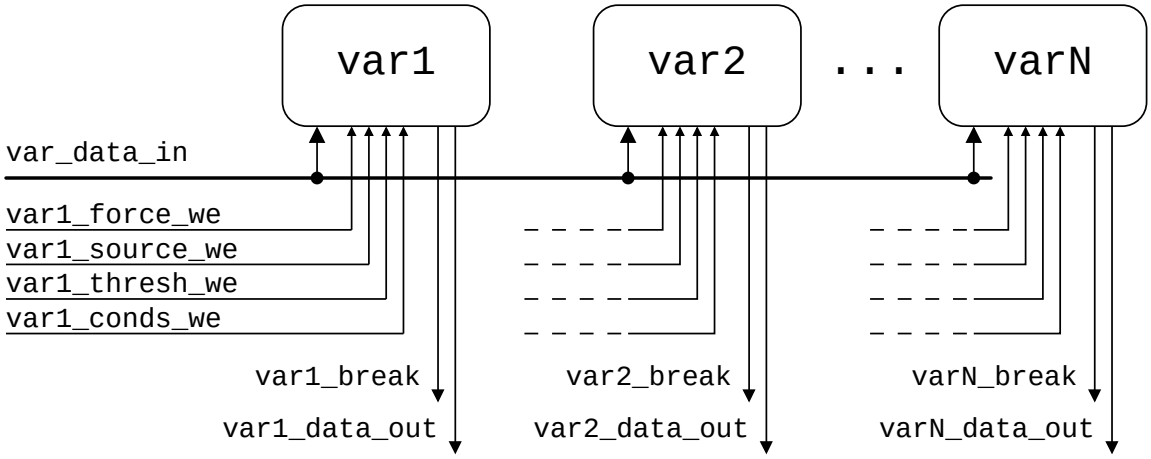

Figure 4.3: Variable unit connectivity

each variable unit. When the user issues a request to read a variable's value, the core controller can select the appropriate value to be returned from this array of signals.

4.2 Core debug controller

The core controller of the debugger is responsible for accepting and issuing user commands received from the software layer, as well as managing the design clock during runtime. It is implemented primarily as a state machine along with an assortment of glue logic, all written in behavioral VHDL and synthesized onto the FPGA fabric.

The interface between the core controller and the software layer could be implemented in a number of ways. The only requirement of the debugger is that there is some form of mutually accessible data structure which can be read and written by both the software and hardware. On the BEE2 platform utilized in this work, this software/hardware interface is provided by BORPH, the on-board operating system. BORPH provides a mechanism for declaring registers in the user's design which can either be read by software (written by hardware) or written by software (read by hardware). The direction of data transfer is always one-way. Within BORPH, these registers appear as normal UNIX files and can read or written as either ASCII strings or raw binary values (the BORPH software register interface is described further in Sect. 5.1). All the functionality performed by the core controller is encapsulated by two pairs of software registers, which are listed in Table 4.1.

Table 4.1: BORPH software registers accessed by core controller

Register name	R/W	Function
bdb_cmd_in	W	Defines the command to be performed by hardware
bdb_data_in	W	Receives arguments to commands (as needed)
bdb_status_out	R	Holds the current status of the core controller
bdb_data_out	R	Receives return value of command (as needed)

Fig. 4.4 shows the behavior of the primary state machine in the core controller. The controller effectively waits for a command to appear from the software layer via the bdb_cmd_in register and then branches to a state which begins execution of the command in hardware. While the command is being performed, the bdb_status_out register is written with a STAT_BUSY value. While most controller commands, which run at hardware speed, finish before the software layer even has a chance to read the status register, the use of this busy indicator is still necessary to detect unforeseen hardware deadlocks as well as to allow long operations to complete, such as manually running the hardware clock for a very large number of cycles (Sect. 4.2.1 describes the set of supported commands in detail). Once the command has been completed in hardware, the status register is written with a STAT_DONE value and waits for the software layer to clear the command register, which indicates that the software acknowledges completion of the command and allows the core controller to return to the idle state.

Any commands which require data to be provided from the software layer make use of a second input register, bdb_data_in. Synchronization of the command and data values coming from software is guaranteed by requiring that the data be written by the software before the command code is set. In this manner, the input data is "latched" by the hardware in the same cycle that the command code is received. Several commands, such as forcing a variable to a specified value, require two arguments to be provided (in this example, both a variable ID and a data value). To support these cases, the command register is actually interpreted in two pieces: a high half-word and a low half-word. The low half-word contains the command code itself, and the high half-word contains a *command stage counter*. This value is always zero when a command is issued, and is incremented by one for commands that require additional arguments. Once again, synchronization between the command and data is achieved

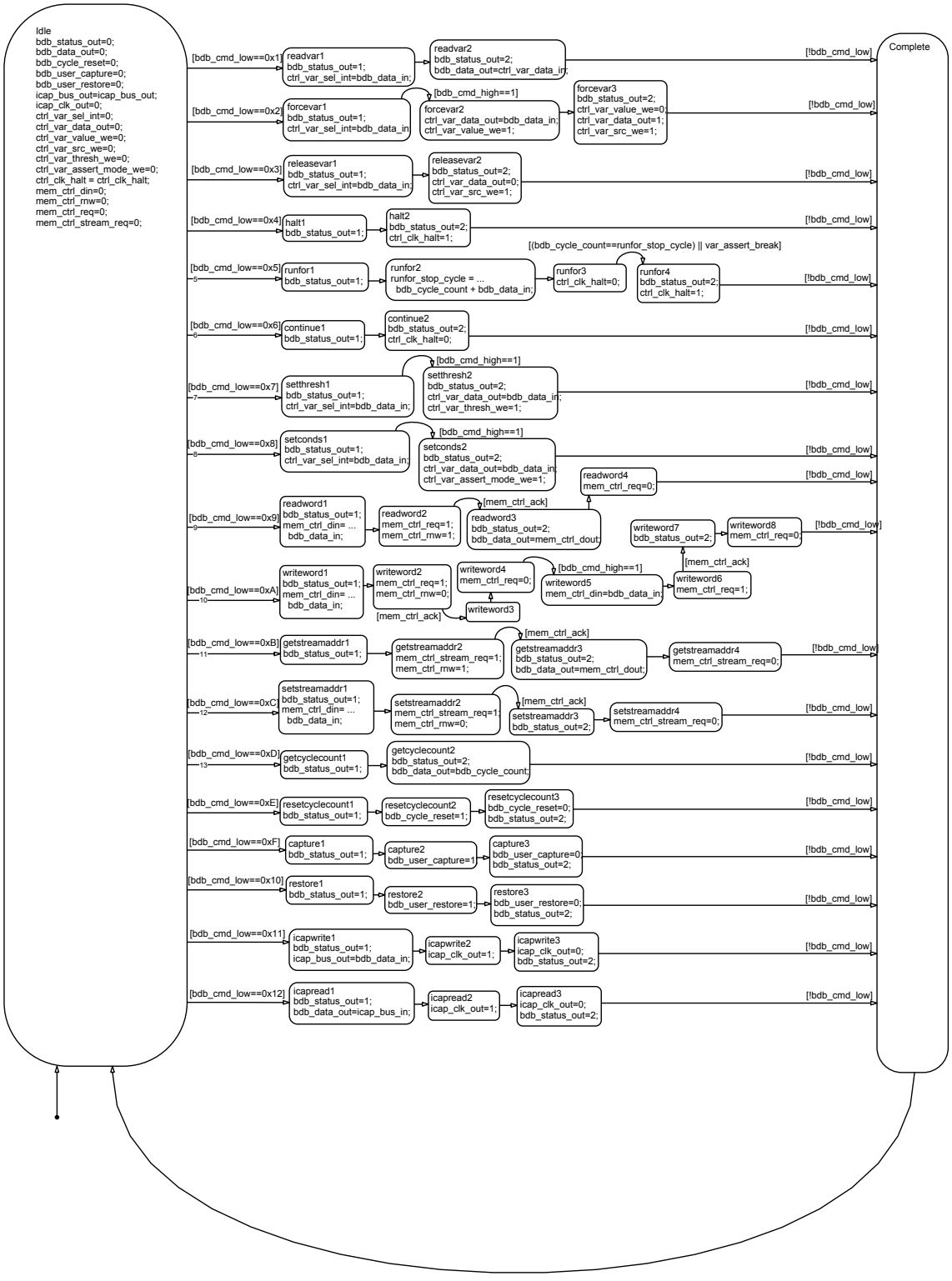

Figure 4.4: State machine for user command translation

by requiring that the data value be set before the command stage is incremented. The data is effectively latched by the hardware on the same cycle that the new command stage value is received.

Any commands which require data to be returned by the hardware make use of the `bdb_data_out` register. Of all the commands currently performed by the core controller, none return more than one word of data. Therefore, no additional status bits are needed to indicate which data element or additional argument is currently valid. As such, the presence of valid data in the data output register is indicated purely by a status of `STAT_DONE`. The value stored in the data output register is considered invalid and undefined when the controller status is set to either `STAT_IDLE` or `STAT_BUSY`.

In addition to the core controller status flags described above, the `bdb_status_out` register also contains global status bits which indicate the design clock state. These status bits are currently composed of the signals `ctrl_clk_halt`, which indicates if the design clock is currently manually halted by user request, and `var_assert_break`, which indicates if a variable breakpoint is currently active. Sect. 4.2.3 describes the clock management logic, and consequently the use of these signals, in greater detail. Table 4.2 shows the allocation of bits in the status register. The uppermost bits labeled *undefined* in the table can prove quite useful for "debugging the debugger", as signals internal to the debugging infrastructure can be exported here for observation. For this reason, it is also preferred that the software layers which read the status register simply mask off all these bits and ignore their values so that any observational outputs do not interfere with normal operation.

Table 4.2: Allocation of bits in the `bdb_status_out` register

31:10	**9**	**8**	**7:0**
Undefined	`var_assert_break`	`ctrl_clk_halt`	Core controller status

While one side of the core controller interfaces to the software layer via the previously described registers, the rest of the controller manages the hardware domain and connects to the variable network, clocking infrastructure, and external storage interface. The following subsections contain a complete description of the set of hardware commands and the physical implementation of the hardware management logic, respectively.

4.2.1 Control operations

The core controller provides a set of 18 commands to manage all aspects of hardware execution. Each of these commands are described in full detail below.

As described above, commands are received from the software layer via two pairs of shared registers. Because there is no other innate synchronization between the software and hardware layers, the point in time at which an operation is performed in hardware is not deterministic. For this reason, most commands are intended to be used when the hardware clock is stopped (such as when manually halted, or when an assertion is active). There are cases, however, when data-oriented commands are still useful even when the hardware clock is running, for example when monitoring the value of a highly static condition variable. Therefore all commands are allowed to be executed at any time, and synchronization is left as the responsibility of the user.

- `readvar` — This command reads the current value of a variable and returns it to the software layer. The variable ID to be read is provided as an argument in the data input register, and the value is returned in the data output register.

- `forcevar` — This command forces the output of a variable to a fixed value. Two arguments are expected from software: the variable ID to be forced and the specified value. In hardware, this has two effects. The force value in the variable unit is written with the value provided as an argument, and the variable output source register is written such that the output mux selects the forced value.

- `releasevar` — This command effectively undoes a force operation. Only the variable ID is expected as an argument from software. Upon execution, this command will reset the output source register of the selected variable unit such that the normally computed value is once again returned to the system.

- `setthresh` — This command defines the threshold value to be used for assertion comparisons. Two arguments are expected from software: the variable ID to be affected and the threshold value itself. In hardware, this command will write the given value into the threshold register of the specified variable unit.

- `setconds` — This command sets the assertion condition mask for a given variable. Two arguments are expected from software: the variable ID to be affected and the condition mask value itself. Valid forms of the condition mask are dependent on which style of assertion comparison logic was instantiated for the given variable (see Sect. 4.1 for more information on the assertion logic microarchitecture). Basic equal/not-equal comparisons utilize a 2-bit condition mask, and full less-than/equal/greater-than comparisons utilize a 3-bit condition mask. A *1* in a valid bit position of the condition mask enables that condition to stop the design clock when it evaluates to true. Any higher-order (i.e. non-valid) bits of the mask set by software are simply truncated and ignored by hardware.

- `halt` — This command halts the hardware clock. Internally, this is enforced by loading a high value into a register called `ctrl_clk_halt`. As such, this state where the clock is manually halted is independent of whether or not a variable breakpoint is active.

- `runfor` — This command will run the hardware clock for a specified number of cycles, provided as an argument from software. If the hardware clock is currently manually halted, it will be allowed to run for exactly the number of cycles given, and then return to the halted state. If the hardware clock is currently running, this command is effectively identical to `halt`, as the point in time at which commands are received by the hardware and take effect is not deterministic. If a variable breakpoint is currently active, this command will not have any effect on the hardware clock. This is because it is not possible to override a variable breakpoint via the manual clocking commands. In this case, the command will simply return without advancing the system clock, although the `ctrl_clk_halt` register will still be set high before returning.

- `resume` — This command returns the hardware clock to a free-running state by clearing the `ctrl_clk_halt` register. It will have no effect on system execution if the clock is already free-running, or if a variable breakpoint is active.

- `readword` — This command will read one 32-bit word from DRAM. The address to be read is received as an argument from software. The hardware does not

perform any checks with regard to the range or alignment of the address; it simply truncates the lowest two address bits to guarantee a word-aligned request and passes the remaining 29 bits to the DDR controller. The exact mechanism by which user requests to DRAM are arbitrated is covered further in Sect. 4.3.

- `writeword` — This command will write one 32-bit word to DRAM. The address and the data value to be written are expected as arguments from software. This command currently has limited use other than to verify the correctness of the DRAM controller or memory itself, as the contents of DRAM are reserved exclusively for variable data history.

- `getstreamaddr` — This command will read the current DRAM streaming address and return it to the software layer. This feature is normally used by the user interface layer when loading variable history to obtain the DRAM location which corresponds to the current cycle of execution.

- `setstreamaddr` — This command will set the current DRAM streaming address to the value provided as an argument. It is provided for advanced verification applications where the user needs to "rewind" the full system state and resume execution from a previous point in time, either for bug isolation or to verify a solution via modifying variable or system contents.

- `getcyclecount` — This command will return the current cycle number to the software layer. The number of system cycles that have been performed is stored in a 32-bit register inside the core controller. The user can use this command to query the current cycle number, for example to determine how many hardware cycles have elapsed between system events such as variable breakpoints. This cycle count is limited to 32-bits, and will wrap around to zero upon overflow, which must be accounted for by the user.

- `resetcyclecount` — This command will reset the current cycle number to zero. This feature is provided for cases when the user prefers to set a new "time zero" to use as a basis for cycle accounting.

- `capture` — This command will cause a full-chip capture of the current flip-flop contents. The implementation of this feature utilizes the *readback* capability

built into Xilinx Virtex-II FPGAs. A special block named `CAPTURE_VIRTEX2` is instantiated within the core controller. When the input to this block is sampled high, the contents of the FPGA configuration memory (the same registers used to hold the initial values for each flip-flop in the logic fabric) are loaded with the current values in the active system. This allows the user to take a "snapshot" of the entire design to be recalled at any time.

- `restore` — This command will cause a full-chip global restore of all flip-flop initial values. This feature also utilizes a special block named `STARTUP_VIRTEX2` which is instantiated in the core controller. By asserting the `GSR` input to this block, all the initial values stored in the FPGA configuration memory are reloaded into the active flip-flops. In practice, this is used as the analog to the capture operation above, allowing a user-defined snapshot to be restored at a later time.

- `icapwrite` — This command will write the argument value to the built-in ICAP interface and cycle the ICAP clock. The ICAP (*Internal Configuration Access Port*) interface is a Xilinx component which allows the issuing of FPGA configuration commands from within the logic array. This function is provided for highly advanced verification applications where the user chooses to partially reconfigure an active FPGA device.

- `icapread` — This command will cycle the ICAP clock and return the current outputs of the interface to the software layer. This function is also provided for highly advanced verification applications where the user needs the ability to directly read the current FPGA device configuration.

It should be mentioned that while the overall design goal of this verification methodology is to remain as device- and platform-independent as possible, the last four commands described in the list above use highly specific, proprietary features of Xilinx FPGAs. It is necessary to utilize these features to provide the level of abstraction needed for high-level verification in hardware. However, similar functionality could be achieved on alternate platforms either by leveraging analogous device features, or by manually adding such behavior to the debugger hardware architecture.

Table 4.3: Summary of core controller commands

Command	ID	Data in	Data out	Description
`readvar`	1	1	1	Read variable's current computed value
`forcevar`	2	2	0	Force variable to specified value
`releasevar`	3	1	0	Release variable to its computed value
`setthresh`	4	2	0	Set variable's assertion condition threshold
`setconds`	5	2	0	Set variable's assertion condition mask
`halt`	6	0	0	Halt hardware design clock
`runfor`	7	1	0	Run hardware design for N cycles
`resume`	8	0	0	Resume hardware design clock
`readword`	9	1	1	Read word from the given DRAM address
`writeword`	10	2	0	Write word to the given DRAM address
`getstreamaddr`	11	0	1	Return current DRAM stream address
`setstreamaddr`	12	1	0	Set current DRAM stream address
`getcyclecount`	13	0	1	Get current design clock cycle number
`resetcyclecount`	14	0	0	Reset design clock cycle number to zero
`capture`	15	0	0	Trigger full-chip register capture
`restore`	16	0	0	Trigger full-chip register restore
`icapwrite`	17	1	0	Write given values on ICAP input bus
`icapread`	18	0	1	Read current values on ICAP output bus

4.2.2 Variable management logic

While all the logic necessary for dealing with the system data itself is inferred as part of each variable unit (see Sect. 4.1), a small amount of interface logic is also needed in the core controller to facilitate communication with the variable network.

All the commands coming from the software layer are sent to the hardware via two 32-bit input registers (as described in Sect. 4.2). Therefore, all the commands which identify a specific variable to be accessed (namely `readvar`, `forcevar`, `releasevar`, `setthresh`, and `setconds`) reference the target variable using its numeric ID specified as a 32-bit integer value. This numeric ID is converted into a `var_select` signal, one for each variable unit in the system, using an integer to one-hot encoder. In this manner, the core controller can select the variable for which an operation is intended. For the cases where a control register inside the variable unit is being modified (`forcevar`, `releasevar`, `setthresh`, and `setconds`), the data value itself is broadcast to all variable units from the core controller, and the actual destination register to be written is selected by *AND*'ing the variable select signal with the

corresponding write enable. Fig. 4.5 shows this behavior as a block diagram.

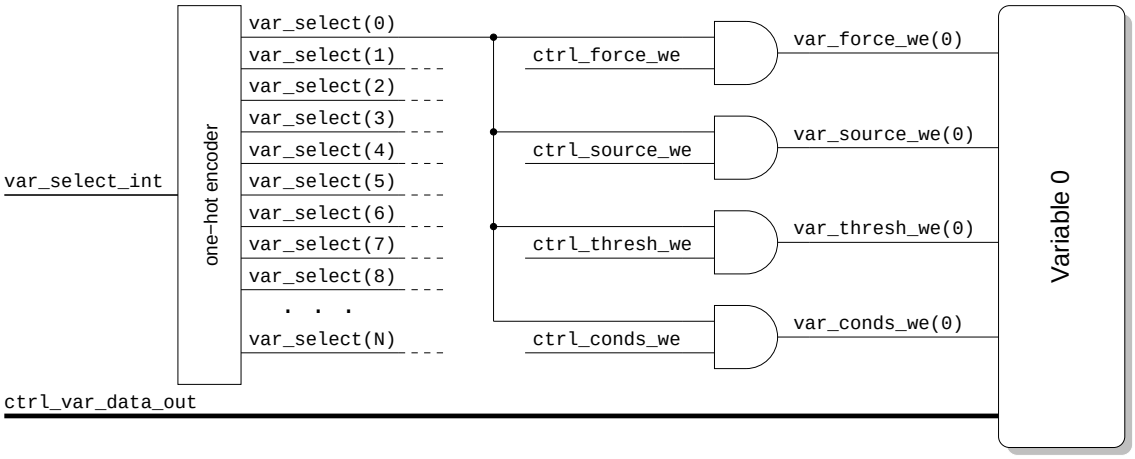

Figure 4.5: Interface logic for operations which modify variable state

Similarly, some logic is required for the core controller to select data from the proper variable during a `readvar` command. A single tri-state bus, using the same `var_select` one-hot signal above as an output selector, is used to route the desired variable into the `ctrl_var_data_in` input to the core state machine.

4.2.3 Clock management logic

The ability to have fine-grained control over the system clock, and consequently the execution of the hardware, is critical to accurately observe the vast dynamic state of the design under test. The core controller provides this functionality by instantiating and regulating the global clock buffer which feeds the design clock. Fig. 4.6 shows the resulting structure of the clock domains.

The only requirement of the debugger is that the the core logic remains on a free-running clock, and all the logic contained in the user's design lies on a controlled clock domain which can be gated with zero cycles of latency. On the platform used in this work, this functionality is achieved by connecting the debugger logic to the standard system clock which is provided directly to the FPGA device. This clock runs uninterrupted at a fixed frequency (in this case, 100MHz). A global clock buffer with an exposed clock enable input (a `BUFGCE` component on the Xilinx Virtex-II Pro FPGA) is then instantiated in the core controller. The output of this buffer is used

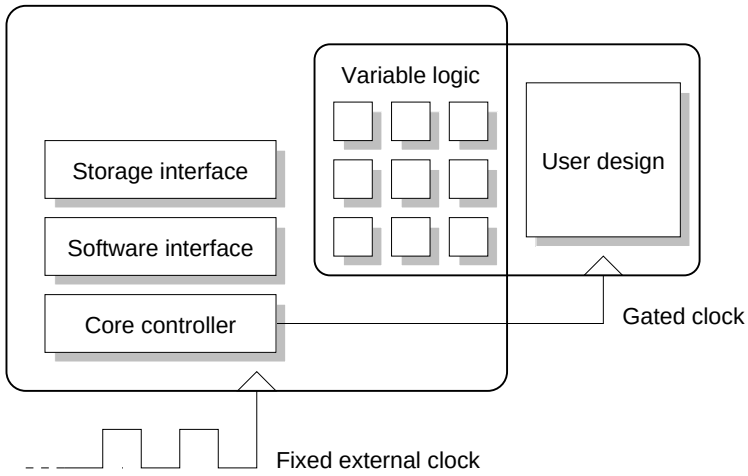

Figure 4.6: Organization of clock domains to regulate design execution

as the clock source for the user's design. The use of a gated global clock buffer is the most efficient solution available, as it does not cause the routing congestion associated with broadcasting a global clock enable signal to every register in the design. It is also highly efficient in terms of pure critical path delay, as the global clock buffer serves as the root of the clock tree and does not incur a penalty to the setup time at each register.

The clock buffer enable signal is derived by *OR*'ing three separate active high signals. The first signal, `ctrl_clk_halt`, is mentioned in Sect. 4.2.1 and is set via the `halt`, `runfor`, and `resume` manual clock control commands. The second signal, `var_assert_break`, is derived by *OR*'ing the breakpoint signals of all the variable units described in Sect. 4.1, which are driven high when an active assertion evaluates to true. The third signal, `dram_clk_wait`, is driven by the attached DRAM controller and is used to throttle the system clock so that all variable data values can be streamed to memory before the design clock advances (the behavior of the external DRAM interface is further described in Sect. 4.3).

4.3 External storage interface

One of the more powerful features of this methodology which is traditionally absent from hardware verification tools is the use of external storage to maintain a deep

history of variable data. The hardware platform utilized in this work provides four banks of attached DDR2 DRAM per FPGA on each board. When direct verification is enabled in a user's design, the first DRAM bank is reserved exclusively for storing variable data.

This design decision was made purely for simplicity and efficiency based on the architecture of the hardware platform and the availability of an existing soft DDR2 controller targeted for each individual bank. In practice, *any* type of storage could be used for this purpose. As mentioned briefly in Sect. 4.2.3, the core controller uses a single signal to determine when the system should pause the design clock while external memory is being filled. Any storage medium or architecture could be accommodated as long as the interface provides the reliable streaming of all variable data in a known sequence and a "busy" signal to the core for clock throttling. The remainder of this section describes the details of how variable data streaming is implemented on the hardware platform relevant to this work.

The role of the DRAM interface is to stream all variable values into memory during operation (while providing a signal to throttle the design clock) as well as execute user requests via `readword` and `writeword` when necessary. Arbitration between runtime streaming and user requests is handled simply by allowing any active streaming sequence to complete before performing the desired request. Care must only be taken to make sure the design clock remains in sync with the memory contents, and that all streaming operations are completed before advancing the design clock (and consequently, that the design clock is only advanced one cycle between streaming bursts). Fig. 4.7 shows the state machine that regulates these functions.

In addition to the temporal arbitration of memory accesses, some additional logic is required to properly route the necessary data to or from the DRAM interface and to drive the appropriate control signals for each operation. For user requests, this involves routing the 32-bit value provided from the software layer to the proper word location on the 256-bit wide DRAM data bus using the appropriate address bits and, in the case of a write, also setting the proper byte enable bits to prevent writing invalid data. For automated runtime streaming, a set of small counters and multiplexers are inferred to sequentially write all variable data out to memory one 256-bit row at a time. When the amount of variable data is not an integer multiple of 256 bits, the remaining bits are padded with zeros. Since the hardware infrastructure

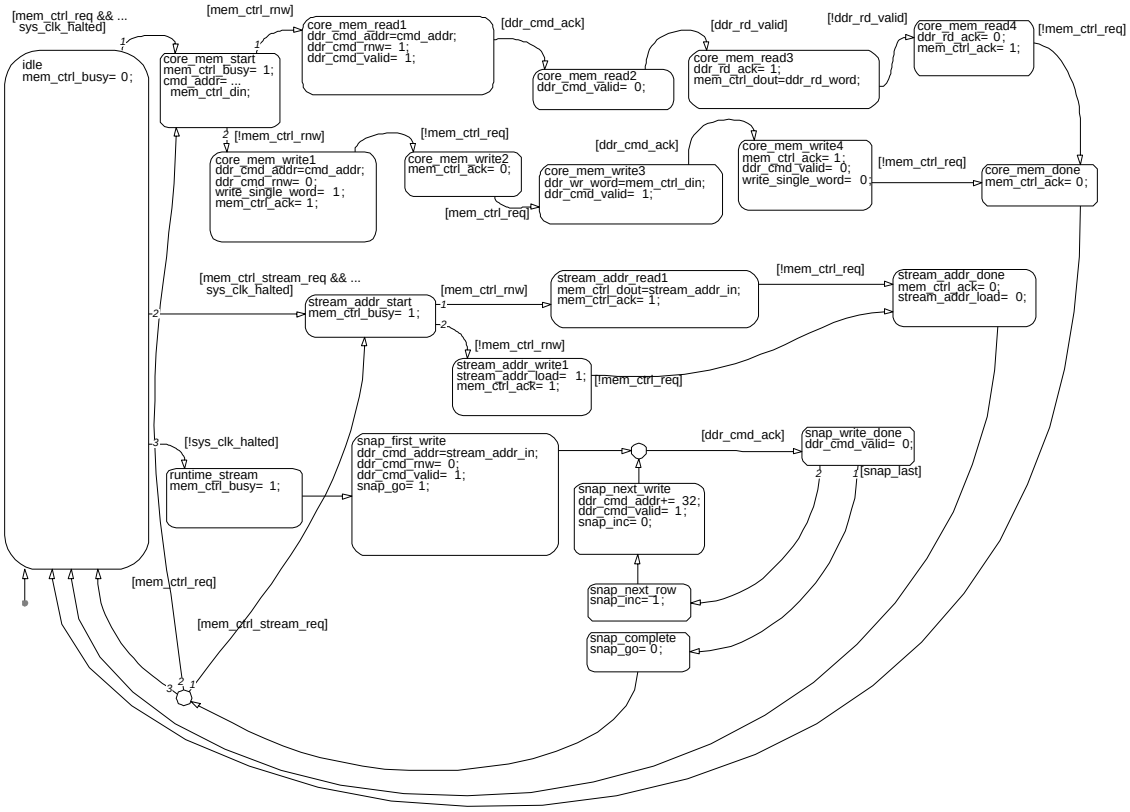

Figure 4.7: State machine for regulating runtime and on-demand memory accesses

for the debugger must be automatically generated for any shape and size of design, all this functionality is implemented in highly parameterized VHDL with liberal use of generics and generate statements. Because of this desire for pure automation of the hardware generation, the VHDL code itself is a bit complex, and can be found in Appendix A.

The organization of variable data in DRAM is kept consistent so that the software and user interface layers can use fixed memory patterns to locate the samples of specific variables at known points in time. Variables are written to DRAM in order of their numeric ID with incrementally increasing addresses. As mentioned above, in order to keep the hardware interface relatively efficient, variable data for each clock cycle is aligned to a 256-bit boundary and any bits not populated with variable data are padded with zeros. This allows the hardware simply write one 256-bit row of variable data to memory at a time, and does not require any complex shifting or reordering of values to align with the next available memory address. Of course, on

platform architectures where logic is considered exceptionally plentiful and memory is extremely scarce, the storage interface (or specifically, the address generation and data alignment logic) could be designed for maximum packing efficiency, at a cost of logic resources and perhaps even performance, in the event that the re-alignment of data to arbitrary starting addresses requires additional clock cycles.

Also for the purpose of greatly simplifying the hardware interface to memory, all variables are required to have the same storage size in memory, regardless of the actual bitwidth of the data in the design. The number of bytes allocated for each variable can be either 1, 2, or 4 and is set as a global parameter by the designer. The largest currently supported variable size is 32 bits due to the width of the software registers used to accept commands from the software layer (of course, this could be expanded, even on the BEE2 platform, by redesigning the core controller command interface to support multi-word arguments and using larger internal data registers). By providing a choice between several storage sizes, the user may select the size most appropriate for the needs of the system even though aggressive packing of data is not supported. If a design features a very small number of data values which exceed the chosen storage size, multiple variables can be used and the true value can be translated in software by the user. Quantitatively, the base address of a variable value with numeric ID i on clock cycle t can be specified as

$$A_{i,t} = 32t\lceil WN/256 \rceil + iW,$$

where N is the number of variables in the design and W is the width of stored variable data in bytes.

In summary, the hardware infrastructure which serves as the foundation of direct verification is composed of the variable network itself along with a core debug controller and some form of external storage interface for saving a history of variable data. While the exact details of the core controller and external storage interface may vary from platform to platform, the overall concept of inserting hardware variable units into the design under test and using a core debug controller to execute requests from the software layer and regulate design execution would remain the same. Generally, the architectural decisions made here gave priority to simplicity and efficiency of the hardware resources while still providing the set of features needed to support a fully robust verification strategy.

Chapter 5

Software Interface

While the integrated hardware structures described in Chapter 4 are fundamental components to the ability to directly verify applications on the hardware platform, the portal between the running hardware application and the outside world, which includes the user's original design environment, is the debugger software process.

As initially introduced in Chapter 3, the BEE2 platform used in this work features an on-board operating system called BORPH — an extension of the standard Linux kernel which runs on the integrated PowerPC core on the central control FPGA of a BEE2 board. Under BORPH, each hardware design, which in the case of FPGAs is a device configuration bitstream, is encapsulated in a standard ELF executable binary and runs as a user process on the Linux kernel. The user launches a hardware process just as he or she would on any Linux workstation. The hardware process receives a numeric process ID and appears in the process list like any other application. However, in addition to a standard Linux process, a BORPH hardware process also provides filesystem nodes for each shared hardware component which can be read or written (as allowed) by the software. It is precisely this hardware-software interface which is used by the debugger to facilitate communication between the user and the running hardware.

It is once again worth mentioning that the verification methodology conceived in this work is intended to be applicable to any reprogrammable hardware platform. The only requirement is that the platform provide some mechanism for sending shared data between the software and hardware domains. In this work, the BORPH operating system provides I/O-mapped software registers which serve this very purpose.

The following sections present a detailed description of how the software layer of the debugger is implemented on BEE2 with the BORPH operating system. The first section covers the functionality and implementation of the debugger software process itself, while the second section more specifically describes the network service provided by the debugger for accepting user commands from a remote environment.

5.1 Debugger software process

Under BORPH, the hardware design under test is launched from within a shell process, analogous to the manner in which applications are loaded by the debugger itself in the software domain. In this work, the name of this debugger process is `bdb` (which was intentionally named similarly to the popular software debugger `gdb`). The core purposes of the `bdb` process are to

- launch the hardware design under test (Sect. 5.1.1),
- cache certain aspects of the hardware state to improve efficiency and performance (Sect. 5.1.2), and
- provide a network service to listen for user commands from a remote environment (Sect. 5.2).

Fig. 5.1 is a flow chart which displays the order of tasks performed by `bdb` at launch and during operation. The remainder of this section will cover the internals of the `bdb` process, while the following section will focus specifically on the protocols used by the network service, as the network service represents an abstraction boundary independent of the chosen software architecture.

5.1.1 Hardware child process

The main `bdb` process is responsible for launching the hardware design under test as a child process. Since BORPH handles hardware applications essentially the same as software processes with the addition of some file-mapped I/O, this is performed using the same fork/exec mechanism as any other UNIX process. By default, BORPH grants full permission over the hardware-mapped files associated with a hardware

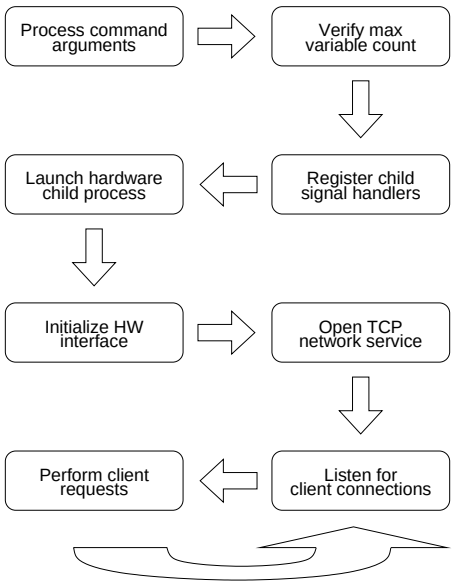

Figure 5.1: Flow diagram showing the operations performed by `bdb`

process to the user who launches it. Therefore, it is not strictly necessary for the hardware process to be launched as a child of `bdb` in order to communicate with the hardware, as an external application would still have permission to read and write from the debug software registers.

There are still two advantages, however, to implementing the hardware design under test as a child process. First, this gives `bdb` the ability to suppress the standard input and output of the hardware process and redirect any error messages to a specific file (named `borph.err` by default) in the working directory. This preserves use of the console strictly for `bdb`, but still preserves any direct output from the hardware process to a consistent location. Second, by launching the hardware design as a child process, `bdb` can register interrupt handlers for child process events such as early termination. This increases the reporting and accounting capabilities of the debugger for unexpected events, plus allows the `bdb` process to exit on its own if the hardware has already stopped running properly.

Once the child process has been launched successfully, `bdb` proceeds to initialize the file nodes which are used for communication with the hardware. All the nodes which correspond to file-mapped hardware resources are placed by BORPH in a `hw` subdirectory of the `/proc` filesystem, the typical method used by Linux to store runtime information on any active processes in the system. Table 5.1 shows the

contents of the hardware-mapped filesystem relevant to `bdb`, where `$PID` represents the numeric ID of the hardware process.

Table 5.1: Contents of `/proc/$PID/hw/` used by `bdb`

Filename	Mode	Description
`ioreg_mode`	R/W	Interpret `ioreg` data as ASCII or binary
`ioreg/bdb_cmd_in`	R/W	Debugger command input register
`ioreg/bdb_data_in`	R/W	Debugger data input register
`ioreg/bdb_status_out`	R	Debugger status output register
`ioreg/bdb_data_out`	R	Debugger data output register

First, `bdb` writes `ioreg_mode` to put BORPH into binary data access mode, as all the remaining calls operate on the register data using raw 32-bit values and not ASCII strings. Next, each of the `ioreg` software register files are opened in the appropriate access mode. BORPH allows read access for all `ioreg` files even though the direction of data transfer is always one-way. Therefore, there is no "write-only" permission on any `ioreg` file, even though the hardware input registers can only be modified by the software and not the hardware. Each of the `ioreg` files are kept open for the duration of the `bdb` process, as they will be continuously accessed on each command for the remainder of execution.

Once the `ioreg` interface is configured, `bdb` opens up a TCP network socket and begins listening for remote commands issued by the user. This is the extent of all the initialization that needs to occur before debugging can begin. By design, all hardware processes launched by BORPH begin execution as soon as the FPGA configuration bitstream has been successfully loaded. For this reason, if the user desires that the hardware design start up in an idle state and wait for user intervention before running, he or she must design such behavior into the system (for example, by declaring a variable which regulates the flow of inputs and free-running circuits in the design and setting the initial value to a disabled state).

There is one additional constraint that must be resolved by `bdb` at any time during the initialization of the hardware process — the total number of variables in the hardware design under test. This is necessary because the core controller itself does not contain any logic to perform error-checking on the variable IDs which are passed as command arguments. Rather than spend hardware resources on enforcing a constraint which should already be known by the user, this functionality was placed

within bdb. Providing the variable count could be accomplished in any number of ways, however the current implementation requires that this value be given as a command line argument when launching bdb. It was planned that future versions of bdb and BORPH be designed so that the variable count was a parameter stored in the header of the hardware process binary file itself. Until this functionality evolved, a command line argument was the simplest solution. When and how the variable count is provided to bdb on alternate platforms is completely up to the designer, so long as it is known before user requests are accepted and executed.

5.1.2 Hardware state cache

A secondary purpose of bdb is to cache certain aspects of the hardware state within memory allocated inside the software process. This functionality was added to the debugger after some practical experience was obtained as a means of improving the performance and hardware efficiency of the overall verification system.

The hardware state cache maintains an array of structures, each of which represents the current state of a variable in hardware. The array initially starts as empty, and additional space is allocated each time a new variable is accessed over the remote network service. Table 5.2 lists the elements of each variable cache entry and the initial values that are used.

Table 5.2: Elements of var_state structure in hardware state cache

Element	Type	Init	Description
valid	short unsigned	0	Entry has been written
forced	short unsigned	0	Variable is currently forced
force_val	long int	0	Value to be forced
threshold	long int	0	Assertion threshold value
cond_mask	long int	0	Assertion condition mask

When new space is allocated for the hardware state cache, each valid element is set to zero to indicate that the cache element for that variable has not yet been written by the software. The remainder of the elements of the variable state structure correspond to each of the parameter registers inside each hardware variable unit (as described in Sect. 4.1). The forced flag is set to 1 when a variable is forced to output a fixed value, which is represented by the force_val element of the state structure.

The `threshold` element holds the currently defined assertion comparison threshold set in hardware, and the `cond_mask` element equivalently holds the current condition mask.

By storing all these variable parameters in a software cache within `bdb`, it eliminates the need for any hardware resources to read back the parameter registers. Because these parameter values can only be changed via user requests, the cache will always remain consistent by updating the stored values during the execution of each command. In addition, the presence of the cache allows certain diagnostic requests, such as querying the list of all variables with nonzero assertion condition masks, to be serviced without any iterative communication with the hardware. The list of available diagnostic functions and their behavior is described further in Sect. 5.2.2.

In addition, the hardware state cache allows `bdb` to bypass communication with the hardware for requests which would return a known value. For example, if a user request is received to read the current value of a variable and the cache reports that the variable is already forced, `bdb` can return the forced value stored in the cache directly without accessing the hardware. Similarly, redundant user requests (such as trying to release a variable which is not forced, or setting the condition mask to the same value that has already been set) can be effectively ignored by simply returning a success code to the remote client without actually sending the command to hardware. The actual performance increase provided by skipping these redundant requests varies based on system load, network latency, and the architecture of the hardware/software interface itself. However, the net benefit will always be positive and justifies the implementation of such behavior in any environment.

One could also imagine that the remote client itself be responsible for caching all the hardware state on its own. Such an implementation would not only provide the benefits described above, but also spare the time required to even send redundant commands to the `bdb` service. The drawback to such an approach, however, is that it requires that the remote client and `bdb` both run persistently in tandem (i.e. the hardware process must be stopped and restarted whenever the client is restarted, and the network connection itself must not be broken). This is a direct consequence of lacking the ability to read parameters from `bdb` or the hardware itself. If for any reason such an implementation becomes more advantageous (for example, in an environment with very high-cost remote communication or extremely limited memory at the `bdb`

software layer), the most feasible solution would be to maintain the hardware state in some predefined file structure or file format in the client environment. As long as steps were taken to ensure that this file data was kept consistent across all client sessions, such an approach would enable caching to occur remotely.

5.2 Remote network service

As the portal between the user and the running hardware design, it is critical that the software component of the verification infrastructure provide a clean abstraction layer between the design and/or analysis environment and the debugger itself. On the BEE2 platform used here, this interface between the user and the debugger is provided by `bdb` as a network service. The creation of a network service is highly useful on BEE2, as the typical usage model is for the BEE2 system running BORPH to be deployed in some location not physically connected to the workstation or cluster on which design and analysis is performed.

Of course, there is no strict requirement that a network be utilized, as alternate platform architectures could be conceived where the hardware engine is directly connected to a traditional workstation (for example, via USB or PCI-E). In such cases, the software component of the debugger could potentially be integrated as a feature into the analysis environment itself. While the exact communication method between the hardware and analysis environments could vary, the design of the software layer of the debugger as a modular component with a cleanly defined service interface is still highly advantageous, as this approach allows an arbitrary type and number of higher-level environments to interface with the hardware debugging layer. The remainder of this section, however, will focus specifically on the design of the remote network service provided by `bdb` as used on the BEE2 platform.

As mentioned in Sect. 5.1, `bdb` opens up a TCP network socket to listen for client requests once the hardware child process has been initialized. Once a client has opened up a connection with the service, `bdb` enters a loop which reads a command code from the socket and then takes action based on the parameters of the specified command. Because `bdb` only manages one hardware design at a time and there is only one instance of the core controller, the network service is designed as a single-threaded loop which will only allow one concurrent client connection. Therefore, any

application which would benefit by having multiple remote clients access one hardware design must manually manage their connections by closing the socket before another client attempts to open a new connection.

All data values communicated over the network service are expressed as signed 32-bit *words* in standard network byte order. Because the actual amount of data to be transferred depends on each incoming request, the following subsections present the functionality of each command separately. All commands, however, follow the same general communication pattern of

$$\textit{Command code} \rightarrow \textit{Input arguments} \quad \cdots \quad \textit{Status word} \rightarrow \textit{Return values},$$

where the command code (sent *to* `bdb`) and status word (sent *by* `bdb`) are each one word, and the arguments and return values can be any length, including zero words.

5.2.1 Hardware-specific functions

The network service accepts a set of 18 commands with positive nonzero command codes, each of which directly corresponds to one of the commands supported by the core controller. The detailed behavior and effects of each command in hardware are presented in Sect. 4.2.1 and summarized in Table 4.3, while the format of data expected by the network service are listed below. Since the network service receives the command code and returns the status word for all commands, only the additional input arguments and return values expected are included in the description. Any commands which receive or return a varying amount of data use N to denote this amount. Note that all variable data is sent and received in the standard 32-bit signed word format; it is the responsibility of the remote client to ensure that the communicated values are not out of bounds of the physical variable size in hardware.

- `0x01 CMD_READVAR`: *1 word received, 1 word returned* — The `readvar` command expects one word with the numeric variable ID to be read, and returns one word with the current value of the variable in hardware.

- `0x02 CMD_FORCEVAR`: *2 words received, zero words returned* — The `forcevar` command expects two words, the numeric variable ID and value to be forced, and returns no data.

- `0x03 CMD_RELEASEVAR`: *1 word received, zero words returned* — The `releasevar` command expects only one word with the numeric variable ID to be released back to normal operation.

- `0x04 CMD_HALT`: *zero words received, zero words returned* — The `halt` command does not accept or return any additional data.

- `0x05 CMD_RUNFOR`: *1 word received, zero words returned* — The `runfor` command expects only one word with the number of cycles for which the hardware design clock should be run.

- `0x06 CMD_RESUME`: *zero words received, zero words returned* — The `resume` command does not accept or return any additional data.

- `0x07 CMD_SETTHRESH`: *2 words received, zero words returned* — The `setthresh` command expects two words, the numeric variable ID to be modified and the assertion threshold to be set, and returns no data.

- `0x08 CMD_SETCONDS`: *2 words received, zero words returned* — The `setconds` command expects two words, the numeric variable ID to be modified and the assertion condition mask to be set, and returns no data.

- `0x09 CMD_READMEM`: *2 words received, N words returned* — This command expects two words, the base address from which to begin reading from DRAM and the number of words to be read, and will return the exact number of words that were requested by the client. To execute this command, `bdb` actually calls the `readword` hardware command N times, filling a local memory buffer and returning all values at once to the remote client.

- `0x0A CMD_WRITEMEM`: *N + 2 words received, zero words returned* — This command initially expects two words, the starting base address and the number of values which will be written to DRAM. The exact number of words indicated by the client will then be read from the network and iteratively written to DRAM by the `writeword` hardware command. The command returns no data.

- `0x0B CMD_GETSTREAMADDR`: *zero words received, 1 word returned* — The `getstreamaddr` command expects no arguments, and returns one word with the

current DRAM stream address on the hardware.

- `0x0C CMD_SETSTREAMADDR`: *1 word received, zero words returned* — The `setstreamaddr` command expects only one word with the new DRAM stream address to be written to hardware.

- `0x0D CMD_GETCYCLECOUNT`: *zero words received, 1 word returned* — The `getcyclecount` command expects no arguments, and returns one word with the current cycle number of the hardware design clock.

- `0x0E CMD_RESETCYCLECOUNT`: *zero words received, zero words returned* — The `resetcyclecount` command does not accept or return any additional data.

- `0x0F CMD_CAPTURE`: *zero words received, zero words returned* — The `capture` command does not accept or return any additional data. However, it should be noted that `bdb` imposes a 2 second delay during this call to allow the hardware to properly settle and ensure the capture operation has completed before returning the status word to the client.

- `0x10 CMD_RESTORE`: *zero words received, zero words returned* — The `restore` command does not accept or return any additional data. However, just as with the `capture` command, `bdb` imposes a 2 second delay during this call to ensure the restore operation has fully completed. As intended, by the time `bdb` does return the status word to the client, the hardware will have resumed from the precise state at which the last `capture` operation occurred.

- `0x11 CMD_ICAPWRITE`: *N + 1 words received, zero words returned* — This command initially expects one word which indicates the number of words which will be written to the ICAP bus and then reads the exact number of words indicated by the client to be written. In hardware, the ICAP bus features a byte-wide data input, but the configuration protocol itself operates purely on 32-bit word values. To execute this command, `bdb` actually sends one `icapwrite` to the hardware to configure the bus for writing, then sends the stream of words provided by the client sequentially over the ICAP bus, and finally releases he ICAP bus back to an idle state.

- `0x12 CMD_ICAPREAD`: *1 word received, N words returned* — This command expects one word with the number of words that should be read from the ICAP bus and returns the amount of data requested by the client. Similar to the ICAP writing command, `bdb` will initially send one `icapwrite` hardware command to configure the bus for reading, then send the number of `icapread` hardware commands necessary to read the requested number of words from the ICAP bus, and finally issue one more `icapwrite` hardware command to release the bus back to an idle state.

As shown by the list above, there is a direct correspondence between the hardware-specific routines recognized by the network service and the core controller. The exceptions to the rule pertain to the DRAM and ICAP access routines, which accept arrays of data elements from the network to be iteratively sent to the hardware, and the capture and restore routines, which are analogous to their hardware equivalents but impose a short delay to ensure that the hardware and software layers have time to settle before manipulating the global device state.

5.2.2 General diagnostic functions

In addition to the hardware-specific commands understood by the network service, a set of general-purpose routines are provided which are useful for querying the overall hardware state. These diagnostic commands are all indicated by an initial command code of zero, followed by the sub-code corresponding to each individual routine. Table 5.3 summarizes the set of available diagnostic commands, and a detailed list of their behavior follows below. The prefix of "`SMD_`" before each command name stands for "subcommand", as each of the codes for these diagnostic routines is sent after the initial command code of zero in the network protocol.

- `0x1 SCMD_GETSTATUS`: *zero words received, 1 word returned* — This command will return the current value of the hardware status register, which is a single 32-bit word value.

- `0x2 SCMD_GETVARSTATE`: *1 word received, 5 words returned* — This command will return the complete state of a variable, the numeric ID of which is provided as a one-word argument to the command. The return values are 5 words, one

for each element of the variable state cache structure (described inSect. 5.1.2). These values are: whether or not the variable's cache entry is valid, if the variable is currently forced, the value which would be forced, the assertion threshold, and the assertion condition mask.

- `0x3 SCMD_GETVALIDVARS`: *zero words received, 5N + 1 words returned* — This command will return a list of al the variables which currently have valid entries in the hardware state cache. The first word returned to the client is the remaining number of words being returned, which is 5 for each matching entry in the cache. This value could be zero if no valid entries were found. What follows are 5 words for each valid entry found. The values sent for each entry are: the numeric ID of the variable, if the variable is currently forced, the value which would be forced, the assertion threshold, and the assertion condition mask.

- `0x4 SCMD_GETFORCEDVARS`: *zero words received, 2N + 1 words returned* — This command will return a list of all the variables whose outputs are currently forced to a fixed value in hardware. The first word returned to the client is the remaining number of words being returned, which could be zero if no variables are forced. The following values, 2 words for each matching entry, are the numeric ID of the forced variable and the actual value being forced.

- `0x5 SCMD_GETASSERTS`: *zero words received, 3N + 1 words returned* — This command will return a list of all the variables whose assertions are enabled (meaning, the condition mask is nonzero) in hardware. The first word returned is the remaining number of words being returned, which could be zero if no variables currently have any assertions enabled. The following 3 values are returned for each matching entry: the numeric ID of the variable, the current threshold for assertion checking, and the full condition mask.

- `0x6 SCMD_GETCOMPARES`: *zero words received, 4N + 1 words returned* — This command will return a list of all the variables whose assertions are enabled in hardware, including the current value of the variable in the system. The behavior of this command is identical to the `SCMD_ASSERTS` above with the addition of each variable's current value to each element. This command is

used extensively to determine which variable is causing an active breakpoint when the condition is detected in the hardware.

- `0x7 SCMD_GETALLVALS`: *zero words received, 6N + 1 words returned* — This command will return a list of all the variables in the system along with their current values. The first word returned is the remaining number of words being returned (this value cannot be zero, as it is not possible to enable verification of a design without declaring any variables). The following 6 values are returned for each variable in the system: whether or not the variable's cache entry is valid, if the variable is currently forced, teh value which would be forced, the assertion threshold, the assertion condition mask, and the current value of the variable in the system. The numeric variable ID is not returned, as it is assumed that the variable IDs in the system begin at zero, and all variables are accounted for incrementally in the return value array.

Table 5.3: General diagnostic functions supported by `bdb` and the network service

Command	ID	Description
SCMD_GETSTATUS	1	Return current value of status register
SCMD_GETVARSTATE	2	Return complete state of variable
SCMD_GETVALIDVARS	3	Return list of all valid variables in hardware cache
SCMD_GETFORCEDVARS	4	Return list of all currently forced variables
SCMD_GETASSERTS	5	Return list of all variables with assertions enabled
SCMD_GETCOMPARES	6	Return list of all active comparisons and their values
SCMD_GETALLVALS	7	Return list of all variables and their current value

The diagnostic routines listed above allow the user to request a snapshot of the current state of certain variables of interest in hardware. Of these routines, the `SCMD_GETVARSTATE`, `SCMD_GETVALIDVARS`, `SCMD_GETFORCEDVARS`, and `SCMD_GETASSERTS` operations are serviced purely based on the hardware state cache maintained by `bdb` and do not require any communication with the hardware. The `SCMD_GETSTATUS` operation does not require intervention from the core controller, although it does require that the current value of the `bdb_status_out` register (see Sect. 4.2) be read. The `SCMD_GETCOMPARES` and `SCMD_GETALLVALS` operations could potentially require the greatest amount of time to execute, based on the performance of the hardware/software interface. Each of these routines must perform a `readvar` hardware command for each

element of interest, which for the former case is all variables with currently active assertions, and in the latter case is all variables in the design.

5.3 Runtime interaction

All the software functionality described in the previous sections deals primarily with managing the running hardware design. The final component of the software interface that is needed for the verification infrastructure to be complete is a connection between the user and the rest of the debugger. The nature of this user interaction at runtime is highly dependent on the properties of the underlying platform and the environment used to design the system. If the computation platform most closely resembles a co-processor model, where the hardware is attached as a slave to an existing workstation, the ideal method of runtime interaction may be a command processing shell, similar to software debuggers like `gdb` in which the application under test runs on the same machine and operating system as the debugger. If the computation platform most closely resembles a standalone system capable of communicating on its own with a network (such as BEE2), the ideal method of runtime interaction most likely utilizes a remote service which accepts requests from an external analysis application, as was implemented in `bdb` and discussed in Sect. 5.2.

Regardless of which method of interaction is best suited to the hardware platform, the presence of some form of user interface during runtime is still necessary to close the verification loop. Since the BEE2 platform running BORPH behaves like a full-featured, networked Linux host, it was most practical to implement a remote service for accepting user requests in `bdb`. Consequently, since the Matlab environment (or specifically, System Generator and Simulink) is used for both design entry and functional analysis, it was also most practical to implement the user interface within Matlab. By providing functions within the design and/or analysis environment for accessing verification resources, the data generated for or retrieved from the hardware can be directly manipulated in the same way as if a pure software simulation were being performed. In this manner, direct verification on the hardware does not lose any usability compared to traditional simulation.

The ideal user interface for debugging and the best method for visualizing data during verification of a system are qualitative subjects which are far beyond the scope

of this work. However, practical experience has shown that most software tools feature either a graphical user interface (GUI) which offers point-and-click access to functions, or a library of routines or some form of API through which a user can write their own scripts to suit their needs. Each of these approaches has its own benefits based on the application under test and the current verification goal. For example, in a situation where the user is closely investigating the design and maintaining manual control over execution, a GUI is often the most practical tool for iteratively selecting individual operations to perform. On the other hand, the ability to automate operations via custom scripts is extremely powerful for algorithm performance exploration, where individual details are not as important as generating large amounts of data without intervention. For these reasons, the runtime interface in this approach to direct verification was designed as a library of user-accessible routines in Matlab, all of which are also represented in a dialog-box GUI for convenience.

Fig. 5.2 shows the GUI which can be used for runtime interaction with the hardware. Three sections of this dialog box should be quite distinguishable, as they directly correspond to the functionality of `bdb` previously described. The *Debug Actions* input pane contains a set of 10 actions, each of which correspond directly to one of the supported hardware operations presented in Sect. 4.2.1 and Sect. 5.2.1. These operations were selected as those which directly access the state of the running hardware and which a user would most frequently need to utilize. The *Diagnostic Functions* input pane also contains 10 separate operations. Eight of these operations correspond directly to the 8 diagnostic routines discussed in Sect. 5.2.2. Also included here are the two direct memory access functions, as individual memory access should not be necessary except for rare circumstances where failures in the hardware or debugger itself are suspected. Finally, the *Hardware Status* pane includes two (view-only) checkboxes which show the current state of the hardware clock. Recall from Sect. 4.2 that the status output register contains two bits, one which indicates if the design clock is manually halted, and one which indicates if a variable breakpoint is active. Because the network protocol was defined such that the hardware status is reported back to the client upon every command, these checkboxes are also updated in the GUI each time an operation is performed. These boxes give the user visual feedback as to whether the hardware clock is in a non-free-running state.

The behavior of all the client-side verification routines available in Matlab are anal-

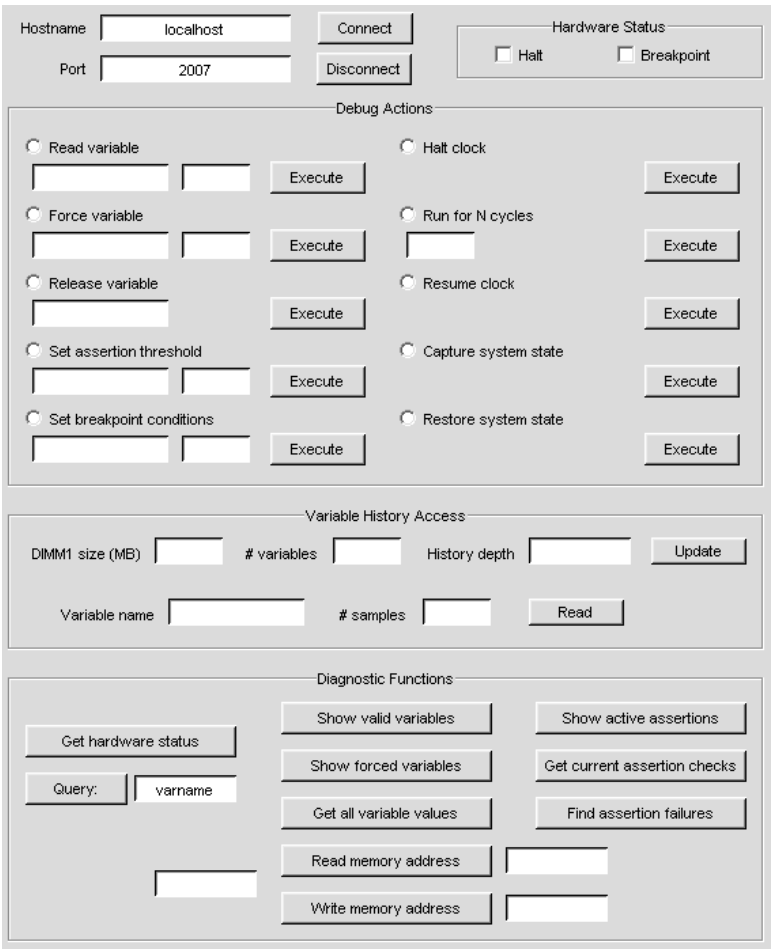

Figure 5.2: Dialog box showing all available end-user debugging routines

ogous to the behavior expected by the remote network service described in Sect. 5.2. Therefore, it is not necessary to review them here. The following features, however, are unique to the client-side runtime interface: persistent socket connection awareness, variable data precision adjustment, and automatic variable history querying.

All of the features unique to the runtime interface make use of an internal data structure which stores all the variable information for the design under test. As mentioned in Sect. 3.2, each of the verification blocks in the system are translated into a Matlab class object which contains the information necessary to drive the hardware generation tools. While this array of objects is stored only within memory as part of the Matlab workspace, a corresponding structure called the *variable map structure* is also produced which can be saved to a file and recalled during a later session. Table 5.4 shows the elements of each entry in the variable map structure. By

referencing the variable map structure internally within the client library functions, the user need not be concerned with identity or representation of variable data in hardware, and can simply refer to each variable by name via the provided routines.

Table 5.4: Properties of each named element in the variable map structure

Property name	Description
`varid`	The numerical ID of the variable in hardware
`bitwidth`	The number of bits in the hardware representation
`bin_pt`	The inferred bit position of the binary point
`storage_size`	The storage size allocated for each variable in storage
`arith_type`	Whether the value is interpreted as signed or unsigned
`assert_type`	The style of assertion logic inferred in hardware

As mentioned briefly in Sect. 5.2, `bdb` was designed to only manage one hardware design at a time. Of course, on a platform with multiple programmable devices, there is no reason why multiple instances of `bdb` cannot be launched in software and simply bind to independent devices. However, since each `bdb` instance is only responsible for one design, it is also assumed that the Matlab client is only attempting to debug a single instance of the design. Therefore, rather than leave the burden of managing the network socket to the user, all of the client library routines maintain a persistent, internal global variable which holds the numerical ID of the network socket connection to `bdb`. Besides simplifying the API for the user by eliminating one more argument to every function call, this also allows the library to automatically check for an existing socket connection before each function call, and conversely allows the socket to be closed when fatal errors are detected or when the user closes the GUI dialog box. Because of the presence of the `bdb` hardware state cache and the availability of the diagnostic routines, it is not necessary for the Matlab client to store any hardware information internally other than the network socket ID.

Also unique to the client environment is the need to convert variable values into their true numerical representation. The hardware infrastructure itself need not be aware of any properties of the variable's true numerical meaning other than the actual number of bits (which is necessary when comparing signed numbers for assertions, as higher-order bits must be ignored to obtain a correct result). In reality, each hardware variable is actually a fixed-point representation of its true numerical value. This is a consequence of the fact that System Generator by default uses fixed-point

arithmetic, the most common standard for signal processing due to its hardware efficiency compared to floating-point arithmetic. Because the hardware values are interpreted as fixed-point numbers, when exporting and importing their values from the Matlab environment (which by default is double-precision floating-point), some conversion must take place to make sure the correct values are used. This functionality is provided internally within the library routines by recalling the declared data type of the variable block from the variable map – the user does not need to be concerned with any data type conversions when communicating with the hardware.

The remaining functionality which is unique to the client-side interface has to do with the organization of variable data in memory. The exact arrangement of variable data in attached storage is described in Sect. 4.3. This is yet another detail of the verification infrastructure that does not need to be imposed on the user. Because the layout of data in attached storage is known and follows a consistent pattern which is based purely on the allocated storage size for each value and the number of variables in the system, the actual address in memory of a given variable on a given clock cycle can be automatically computed within the library routines. For this reason, *Variable History Access* also appears in its own pane on the GUI dialog in Fig. 5.2, as it is best performed via the provided routines and not by directly reading from memory. In addition to the location of variable data in memory, the variable history access functions will also properly truncate and reinterpret the received data based on the variable storage size and each variable's individual data type.

Table 5.5: Specialized verification library routines in Matlab client

Function	**Description**
`bdb_connect`	Check for and establish socket connection to `bdb`
`bdb_disconnect`	Disconnect from any existing socket connection to `bdb`
`bdb_readhist`	Read samples from variable's data history
`bdb_lookup_varids`	Lookup numerical ID for variables by name
`bdb_lookup_varnames`	Lookup name of variables by numerical ID
`bdb_lookup_varparams`	Lookup and return full structure entry for variable
`bdb_scale_values`	Convert to or from hardware and Matlab data values

Table 5.5 summarizes the functions which are available to the user in the Matlab client library. The table only lists those functions which are not directly equivalent to one of the hardware commands already presented in Sect. 5.2. The `bdb_connect`

and `bdb_disconnect` functions set the internal state of the network socket connection which is shared by all other library functions. The `bdb_readhist` function will read a requested number of samples from the history of a given variable, using the known pattern of variable data in storage and the `getstreamaddr` hardware function to determine the base address of the current cycle. The `bdb_lookup` functions all query the internal variable map and convert between numerical variable IDs and their actual names, or simply return a copy of their full parameter structure. Finally, the `bdb_scale_values` function performs the fixed-point data scaling necessary for all the routines which access raw variable data on the hardware.

In summary, the software interface of the debugger is a very important component of the overall verification infrastructure, as it is responsible for relaying data and requests from the user's design environment to and from the running hardware. The `bdb` software process manages the hardware design itself, using BORPH to transfer data to and from the hardware. In addition, `bdb` maintains a cache of the hardware state to enhance the efficiency of the underlying hardware and improve overall system performance. A network service protocol is also implemented between `bdb` and Matlab, which serves as a conduit for data between the hardware and analysis environments. Ultimately, the software interface provides an analysis experience equally rich compared to pure simulation, despite the hardware directly running the design under test.

Chapter 6

Programmability Improvements

Chapters 4 and 5 presented the details of the verification infrastructure which directly deal with the hardware design under test. These components are the foundation of the verification process and could also be considered a full-featured debugger for reconfigurable hardware platforms. While improving the accessibility of internal data and control over design execution are critical and previously challenging aspects of hardware design, one bottleneck still remains in the hardware design process which direct verification can help to address.

As discussed in Chapter 2, the physical hardware generation process is an extremely time-consuming phase of the net implementation time of a design on reconfigurable hardware platforms. While a complete understanding of the underlying challenges behind logic synthesis and placement-and-routing are beyond the scope of this document, it suffices to understand that the optimization of logical functions, the mapping of operations into the primitive logic cells of the reprogrammable device, and the two-dimensional placement and interconnection of logical elements (all of which must adhere to timing constraints) is a far more difficult optimization problem than the compilation of a software program into an instruction stream. It should be mentioned that parallel computation on distributed machines is, of course, a more challenging problem than programming on a single processor. However the burden of timing and performance in this case falls squarely on the designer, as the compilation time of each individual instruction stream is still far faster than hardware generation.

Improved solutions to the place-and-route phase have been investigated (as was also mentioned in Chapter 2), and therefore it could be conceived that aggressive

acceleration of synthesis and/or place-and-route in hardware could be achieved. It remains true, however, that avoiding iterations through the physical implementation phase can only improve the net time to implementation, and therefore the effects of direct verification on the net design time from conception to functional hardware implementation are worth exploring.

The remainder of this chapter is organized as follows. First, an overview of the practices than can improve the net time to final system implementation are presented. Finally, a proof-of-concept demonstration is constructed in which the relative cost in hardware resources of such an approach is observed at a basic level.

6.1 Design time improvement

As described above, the addition of an advanced verification methodology for re-programmable hardware platforms can clearly accelerate the process of proving the correctness of a hardware implementation. However, it is still necessary to reduce the number of *iterations* through the hardware implementation phase as much as possible. This is because traditionally every change to a hardware design, no matter how small, results in the re-implementation of the entire hardware configuration. Methods do exist for modular generation of a hardware configuration on some devices, however the use of this feature requires very careful, manual floorplanning of the hardware implementation by the user. In a dynamic, reusable, reconfigurable hardware environment, this requirement is considered too severe to be useful. Fortunately, once the concept of variables in hardware is available, the net time required between system conceptualization and a final, functional implementation can be improved on any reconfigurable platform through the use of a highly parameterized library of functional units.

The benefit of a highly parameterized library of functional units lies in the fact that minor changes to the behavior of processing elements would not require the re-implementation of the hardware configuration. For example, consider the case where the user is developing a fixed-point signal-processing system, but the exact amount of range and precision of the internal data may not be known until an exhaustive analysis of the performance of the algorithm can be determined. In this case, rather than re-generate the hardware configuration each time the user chooses to evaluate a

new precision of internal data (which traditionally would be done by simply changing the parameters of a simulation model), a variable can be forced to a different value which controls the effective precision of a functional unit's output.

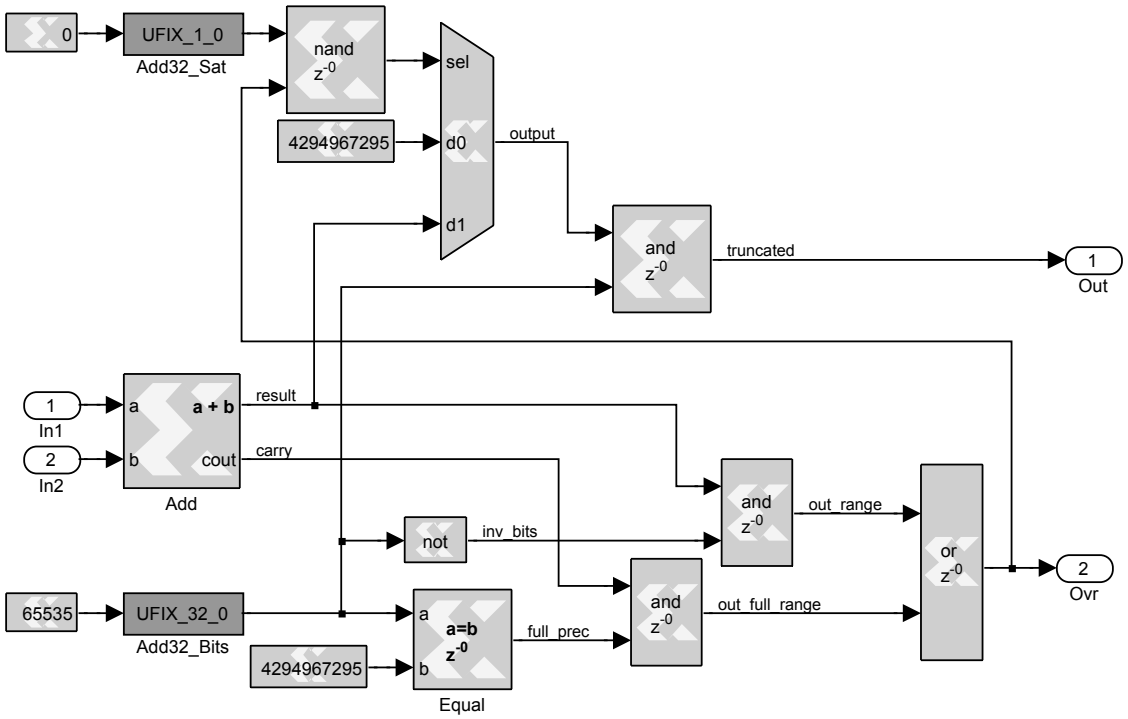

Figure 6.1: Example of a 32-bit unsigned adder with tunable range and saturation

Fig. 6.1 shows an example of a 32-bit adder which has optional bit range emulation and output saturation. The two embedded variables, `Add32_Bits` and `Add32_Sat` control the effective output bit-width and truncation/saturation behavior, respectively. By using a component such as this in place of a traditional adder, a data range from 2 to 32 bits and output saturation or wrap-around can be modeled independently simply by changing the outputs of either of the specified variables. For a fixed-point hardware implementation, the use of components such as this in lieu of a typical, fixed-size adder allow the exploration of algorithm parameters over a range of precisions. Of course, this is also done at hardware speed with the benefit of direct verification, whereas previously a significantly underpowered software simulation would have to be swept over the range of possible computational parameters. It should be noted that the example circuit shown in Fig. 6.1 is not intended to be an optimal solution for

emulating the output range or saturation behavior of an unsigned adder, but rather a simple demonstration that such functionality can be achieved through the use of addition logic beyond the basic addition component itself.

Many more complex examples could easily be imagined for any arithmetic or logical operation, and therefore a custom library of parameterizable components could be constructed based on the type of application being designed. One could consider this as being an intermediate tradeoff between custom-mapped functional units and a processor, where a processor can perform the full range of possible operations with a single, fully-functional execution unit. In this case, the library of parameterizable components is custom-tailored to the application domain at hand. In addition, because the architecture is still a synchronous, direct-mapped hardware implementation, the benefit of complete parallelism is still present, which is absent from a purely processor-based, cluster approach.

6.2 Proof of concept

In order to demonstrate the use of parameterizable functional units, a very basic example was constructed. Fig. 6.2 shows the design of a simple, 12-bit multiply-and-accumulate (MAC) unit with pure wraparound (i.e. no saturation) on overflow. In order to obtain slightly more relevant results considering the simplicity of this component, the same 12-bit MAC block was tiled 8 times, each of the outputs of which were connected to a variable. The corresponding top-level system is shown in Fig. 6.3.

Table 6.1: Hardware requirements of the 8-way fixed 12-bit MAC system

	Synthesis result	**Routed result**
Slices	6074	5385
Flip-flops	5813	5645
LUTs	7646	6902
Period	7.864 ns	9.959 ns

Table 6.1 lists the hardware resources required for this basic example. In a typical, fixed-point hardware system, the output range of a functional unit must be determined in advance and may not be changed once the hardware configuration is generated.

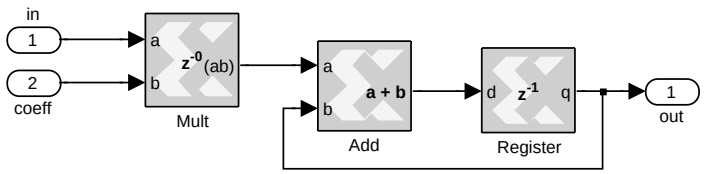

Figure 6.2: Design of a traditional, fixed 12-bit MAC unit

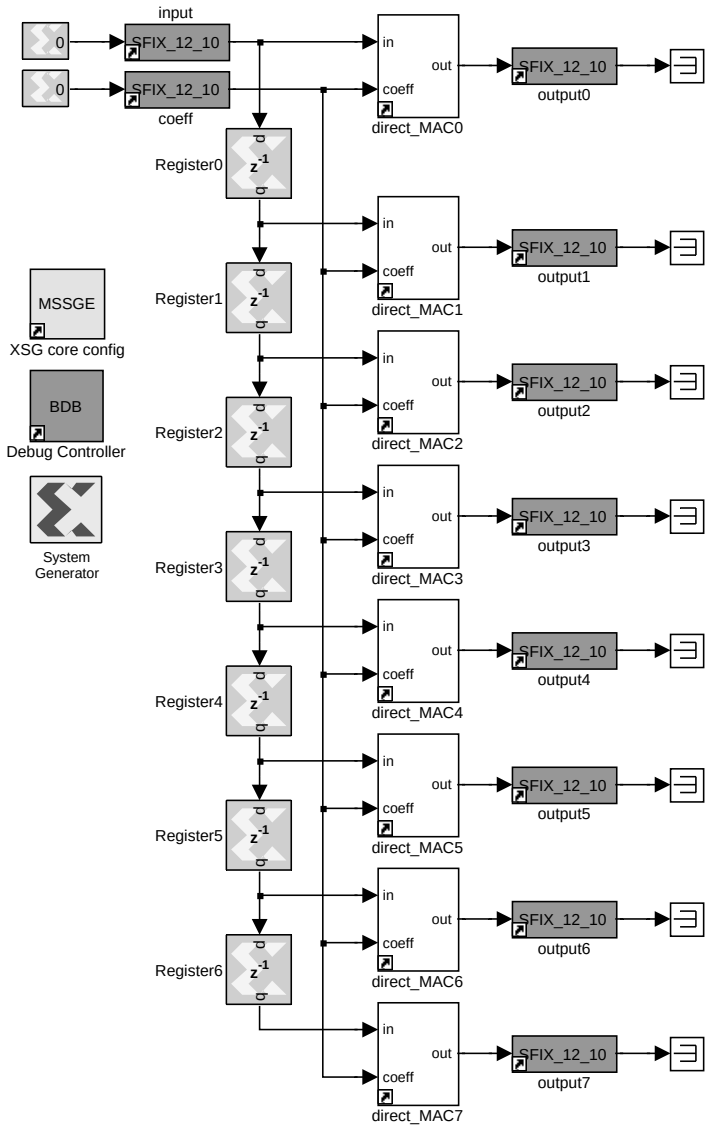

Figure 6.3: Top-level system view of replicated 12-bit MAC example

Therefore, as discussed throughout this section, if a system designer were to realize that additional bits of range were needed, or if he or she wanted to identify the effects of allowing overflow to spare the hardware cost of saturation, the system would need to be modified and completely re-implemented (potentially costing many hours of time) before being benchmarked once again.

On the other hand, Fig. 6.4 shows the design of a MAC unit for which the individual computational blocks have inputs which determine the output value range and saturation behavior of the result. Just as in the previous case, the primitive MAC component was replicated 8 times to achieve more relevant overall results, and the corresponding top-level system is shown in Fig. 6.5.

Table 6.2: Hardware requirements of the 8-way paramterized 32-bit MAC system

	Synthesis result	***% increase***	**Routed result**	***% increase***
Slices	8285	*36.4*	7397	*37.4*
Flip-flops	7215	*24.1*	6763	*19.8*
LUTs	11,162	*45.9*	10,016	*45.1*
Period	14.453 ns	*83.8*	17.253 ns	*73.2*

Table 6.2 lists the hardware resource requirements of this example featuring parameterized MAC units capable of delivering up to 32-bit values. It is clear from these results that the amount of hardware resources required by the use of fully parameterized functional units can be significant. There are several reasons for this behavior. First, the design of the parameterized MAC unit was done in a simple and straightforward, yet not necessarily efficient, manner by instantiating two 32-bit adders, one with saturation and one without, and choosing between the desired outputs. This will definitely require more resources than simply creating an adder that has both the wrapped and saturated outputs available, or which emulates saturation on overflow. Second, this example was constructed such that every 32-bit MAC unit in the system had its own pair of parameter variables. In a real system, it may well be possible to have only a limited number of parameter variables, the outputs of which could be shared by many different functional units (if not the entire design under test). Fortunately, because these parameter variables act purely as mutable constants, they do not require any assertion logic to be included.

Finally, we see a very significant increase in the critical path delay of the system.

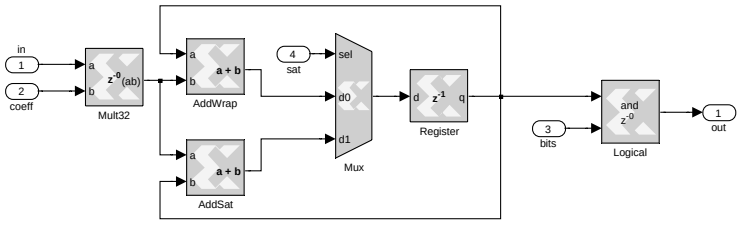

Figure 6.4: Design of a paramterized, 32-bit MAC unit

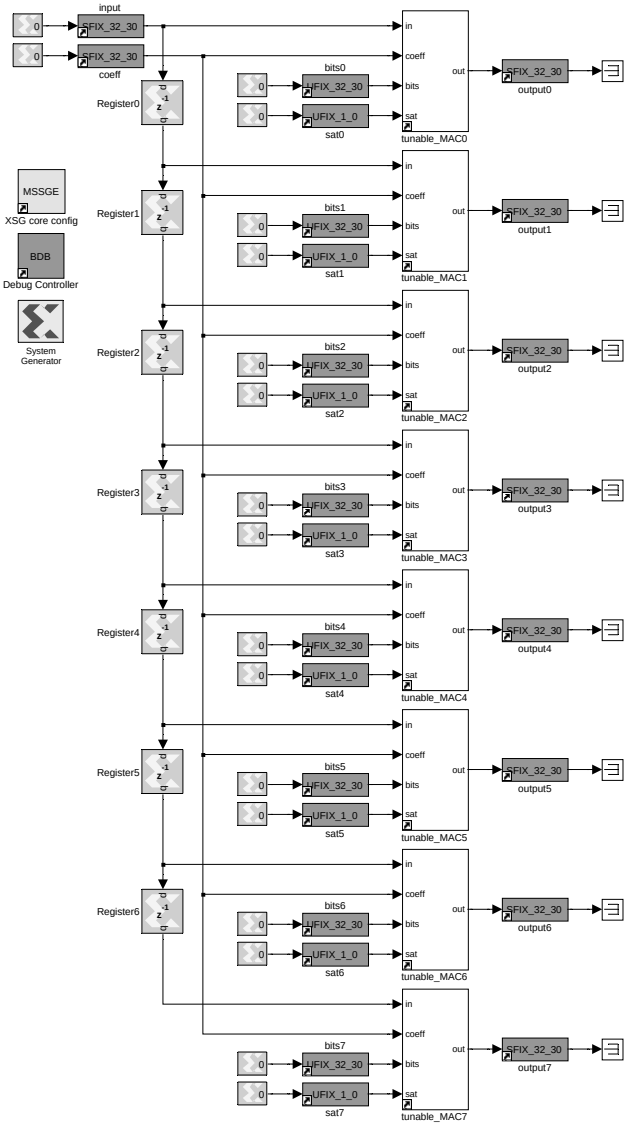

Figure 6.5: Top-level system view of replicated 32-bit MAC example

This is because the parameterized MAC unit was not designed to have any additional internal pipeline registers (and consequently, no additional latency). In a real system, this is one challenge which will remain true when using parameterized library components. The latency of each computation path in a design must always be balanced, such that the computed outputs of each subsystem in the design are properly aligned with one another. If latency is inserted into a functional unit in one path through the logic, the same amount of latency must be added to any parallel paths which depend on that value. As such, the only options available to the designer when using more complex, parameterized components is to accept this reduced operating frequency (hopefully content with the fact that the hardware is still running many orders of magnitude faster than a software simulation), or design their system to properly manage the balancing of delay elements based on the internal latency of the paramterized components. The latter could be accomplished without too much effort by defining hard parameters in the hardware description which define the number of cycles of latency which should be inserted into any shorter computation paths. In fact, many current designs are already implemented in such a way, the addition of pipeline stages and the use of register retiming are effective ways for improving the overall operating frequency of a system.

In summary, while additional hardware resources are required to allow the parameterization of the behavior of computational functional units, such an approach allows the high-level, system performance of a numerical algorithm to be evaluated at hardware speed. The vastly improved throughput of the hardware platform compared to simulation justifies the expense of some additional resources, particularly since the hardware typically remains unused during the exhaustive simulation phase. In this manner, the addition of robust variable manipulation can in fact transform the hardware platform into an even more powerful verification platform, rather than just an execution engine for the final product.

Chapter 7

Performance Results

The previous chapters have presented an approach to direct verification of a design on reconfigurable hardware platforms. Of course, the additional infrastructure required to support verification comes at a cost in terms of device logic resources and net system throughput. Since the goal of this work is to increase the usability of reconfigurable hardware and to bring simulation-like data access into the hardware domain, the performance gap between direct hardware verification and pure software simulation is extremely large (up to four orders of magnitude, as described in Chapter 2). Therefore, the performance impact of direct verification could be quite severe before one would consider the approach a net loss.

Care was taken, however, to keep the underlying verification infrastructure as lean as possible while still providing the feature set necessary to provide full access to variable data and design execution. The following sections will present the resource requirements of the verification infrastructure quantitatively and identify the core bottlenecks which would benefit most from any opportunity for improvement. First, the base infrastructure is evaluated on its own in an otherwise "empty" design. Second, the temporal performance of the system is evaluated in terms of the low-level timing of the hardware circuits and the net throughput impact on the design at runtime. Finally, two additional real-world examples are investigated which provide insight into the practical effect and potential limitations of direct verification in larger applications.

7.1 Base hardware resources

In order to evaluate the hardware resource utilization of the verification infrastructure itself, a benchmark design was created which contained only the standard BEE2 and BORPH interface logic along with a varying number of "typical" 16-bit hardware variable units with full comparison logic. The system model for these experiments is shown in Fig. 7.1, and the chain of additional variables for the 32-element example is shown in Fig. 7.2. For the sake of also measuring the resource cost of the core controller and storage interface, an even more stripped-down design was created which featured no verification support whatsoever (with regard to Fig. 7.1, the *vars* subsystem was entirely empty, and the *Debug Controller* block was removed). Due to the complex optimization that takes place during synthesis and the mapping of logic functions onto the device computation fabric, it can be difficult to precisely calculate the number of logic resources required for each verification element added to a design. However, by measuring the utilization of each of these contrived examples, it is possible to derive a reasonably accurate estimate.

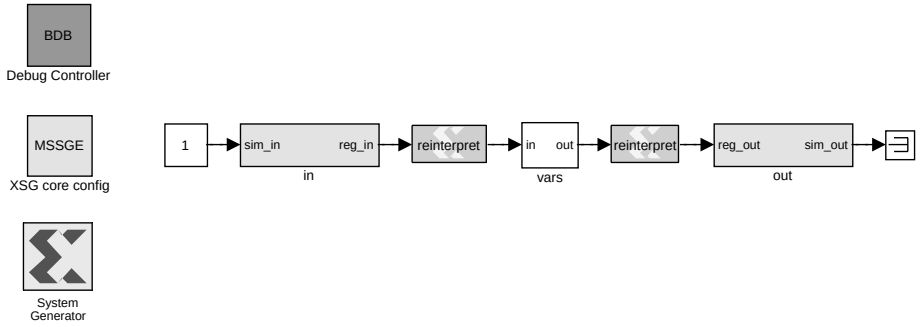

Figure 7.1: System model used for base hardware infrastructure results

Table 7.1 lists the net hardware utilization of the constructed examples described above as reported at the end of the logic synthesis phase. The post-synthesis results are most indicative of the net amount of complexity contained in the hardware description, as they represent the total number of inferred logic gates in the system, although a significant amount of optimization is still possible during the device-dependent physical implementation process.

It is clear from the results that there is a monotonic increase in the hardware

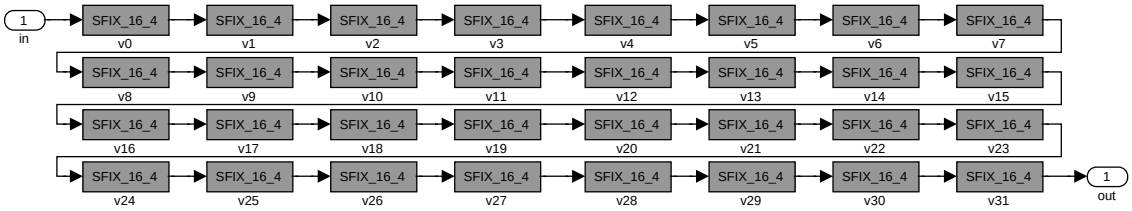

Figure 7.2: Method for inserting variables into the base hardware system

Table 7.1: Synthesis resource estimates for base hardware infrastructure

Variables	**Slices**	**FFs**	**LUTs**	***% of XCV2P70***	***Slices/variable***
0	1,530	1,535	1,876	4.62	–
2	5,562	5,615	6,883	16.81	–
4	5,658	5,720	6,986	17.10	48
8	5,900	5,933	7,281	17.83	57
16	6,653	6,356	8,447	20.11	78
32	7,696	7,186	9,777	23.26	72
64	10,079	8,852	12,987	30.46	73
128	14,656	12,188	19,239	44.29	72

utilization as the number of variables is increased. Two characteristics, however, are most informative. First is the number of additional resources which are consumed when 2 variables are added to the most basic system (labeled as containing zero variables). This increase in utilization represents the base cost of adding verification support to the system, which includes the core debug controller and the external memory interface. The results show that by adding verification support to the system (for which a minimum of 2 variables is required for the low-level hardware connections to be complete), an additional 4,032 slices are required, which corresponds to 12.19% of the XCV2P70 device used on the BEE2 platform. While this is not an overwhelming portion of the device, it is still non-trivial.

Table 7.2: Measurement of post-routing utilization of DDR2 memory interface

	Slices	**FFs**	**LUTs**	***% of XCV2P70***
With DDR2	4623	5146	4469	13.97
Without DDR2	1881	1973	1901	5.68
DDR2 total	*2742*	*3173*	*2568*	*8.29*

To further explore this last result, an incremental analysis of the base infrastructure with and without the DDR2 controller is shown in Table 7.2. This table shows that 8.29% of the device is in fact purely required by the DDR2 memory interface provided for BEE2. This result is partly due to the fact that the built-in DDR2 memory controller for BEE2 was designed as a fully asynchronous FIFO which can support any design clock frequency and still interface functionally to the fixed-frequency DDR2 memory, which must run at a 200MHz double-data-rate frequency. In addition, this result implies that by off-loading the responsibility of controlling the external storage medium, a significant number of device resources can be saved. This could be accomplished on a reconfigurable device which contained a hard-wired memory interface (which would not be mapped to the logic fabric), or also on a platform which had an independent reconfigurable device which contained the bulk of the interface logic such that the computational device was free to focus on computation.

The second metric of interest visible in Table 7.1 is the net number of device resources required for each additional variable added to the system. This value puts an upper limit on the number of variables which can be supported in a design under test before a single device can no longer hold the entire system contents. As shown in the table, the number of primitive logic elements required for each additional 16-bit variable with full comparison logic appears to reach an average maximum of about 75 slices, or 0.23% of an XCV2P70 device.

While synthesis reports provide an insight into the overall amount of logical complexity contained in a design, the final results reported after the mapping and place-and-route phases are the most accurate in terms of what the implementation tools are capable of supporting on the target device. Fig. 7.3 plots the number of slices ultimately required by the base verification infrastructure. The results track the synthesis estimates quite closely, as expected, since this contrived example contains no other design elements to assist with the packing of logic into the primitive device logic elements. In the largest test case investigated with 128 variables, the post-routing results have shown a total of 13,655 slices occupied, which corresponds to 41.27% of the device, and a net difference of 7.3% compared to synthesis estimates.

In general, the results shown here have indicated that the base verification support infrastructure requires about 4032 slices (12.2% of an XCV2P70 device), 2742 (8.29%) of which are purely associated with the DDR2 memory interface, and each additional

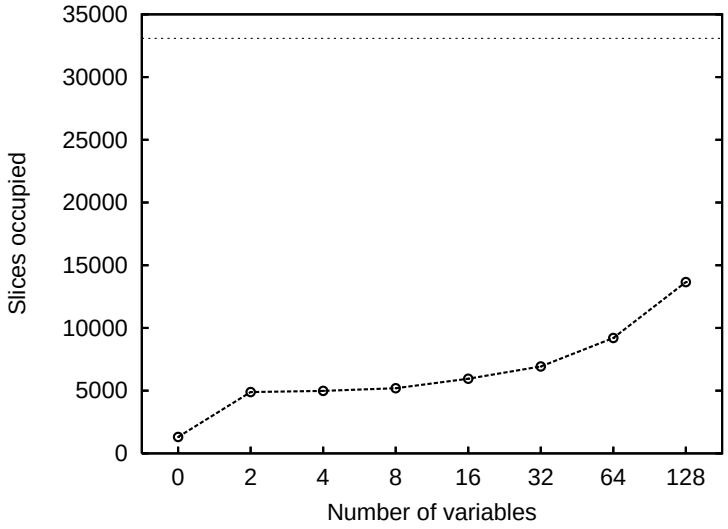

Figure 7.3: Post-routing device utilization of base hardware infrastructure

16-bit variable unit added to the design requires approximately 75 slices (0.32%) of an XCV2P70 device. For some additional data on the overhead associated with direct verification, Sect. 7.3 provides data which was yielded from larger, more practical examples.

7.2 Timing and throughput

The number of hardware resources required by the verification infrastructure is fundamentally important, as it puts an upper limit on the number of variables which can be physically accommodated in a given device. The secondary metric of interest with respect to practical system performance is the speed tradeoff that must be absorbed once verification components are inserted into the hardware design under test.

Table 7.3 lists the hardware critical path results reported after the synthesis and routing phases, respectively. In all these tests, the target frequency assigned to the tools was 100MHz, or a 10ns critical path. The synthesis results show that the verification infrastructure does not begin to stress the targeted frequency until around 100 variables are added to the system. The reason for the increase in critical path delay as variables are added is due to the logical structures for breakpoint detection

Table 7.3: Synthesis and post-routing critical path measurements

Variables	**Synthesis estimate (ns)**	**Post-routing result (ns)**
0	6.906	9.466
2	7.913	9.920
4	8.050	9.878
8	7.795	9.919
16	8.668	9.957
32	8.825	9.921
64	9.950	10.951
128	10.149	12.407

and the routing of variable data to the core controller and external storage (which are described in full detail in Sect. 4.2 and Sect. 4.3, respectively).

Once physical constraints are added to the design during the placement and routing phase, one can see from the data in Table 7.3 and the graph shown in Fig. 7.4 that even in the most basic example, the 10ns target delay is already approached due to the delay of simply sending data throughout the device. In addition, even with the lack of any actual computational elements in the design, the operating frequency of the hardware begins to be reduced at approximately 60 variables. It should be noted, however, that with such a sparse design such as this example model which has very little logic and many memory elements, there is very little opportunity for logic packing into device elements and the higher-level reduction and optimization of logical functions. Sect. 7.3, which features the inclusion of direct verification into more practical, real-world designs, shows that the implementation tools can actually achieve the target performance in far more highly constrained examples through the increased opportunity for optimization.

Finally, there is one more performance limitation of the verification infrastructure in addition to the pure impact on the operating frequency of the design, and that is the cycle-by-cycle reduction in overall system throughput. Recall that the actual design clock is regulated and throttled by the core controller (described in detail in Sect. 4.2.3). Clearly, system throughput will be stopped when the user manually halts the design clock, or if a variable assertion check fails and a breakpoint is active. However, as described in Sect. 4.3, the design clock must also be throttled to allow all the variable data in the design to be streamed to external storage before the design

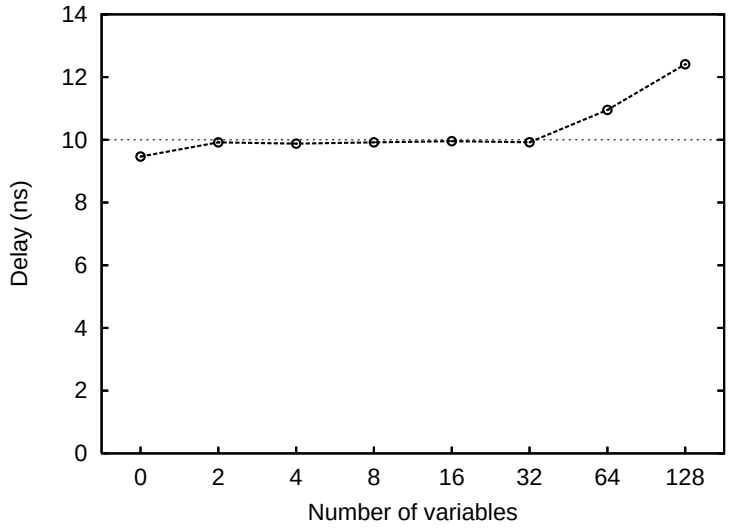

Figure 7.4: Post-routing critical path measurements for base hardware infrastructure

clock is allowed to advance by one cycle. The exact number of clock cycles required to stream all variable data off to external storage, and consequently the factor by which the actual design throughput is reduced, is purely a function of the storage architecture used on the hardware platform. For the DDR2 asynchronous memory controller provided with BEE2 used in this work, the external storage interface requires 4 clock cycles per row of variable data which needs to be written which are overlapped with the inherent latency associated with the DRAM chips themselves.

In general, it has been shown that the temporal performance of a system with direct verification support can be impacted by the critical path extension associated with variable assertion checking and data routing and also by the number of cycles required to stream all variable data to external storage. The exact quantitative effect of waiting for variable data to be streamed off-chip is a function of the storage medium and the address pattern generated, and is therefore too complex to quantify directly.

7.3 Additional design examples

The previous two sections described the resource and temporal requirements of direct verification as measured on constructed examples. However, as mentioned briefly above, the practical ability of the physical implementation tools to map logical functions into the device's primitive computational units and improve the overall

efficiency of the implementation are improved with the addition of logic into the description. Of course, this may occur to an extent until the device approached 100% utilization, at which point the lack of freedom available to assign logical and routing resources becomes severe, and the performance quickly degrades as further functionality is added.

This section provides two real-world examples of systems which have had direct verification support added to the original design. The first features an advanced singular-value decomposition calculation, and the second features a compact spectrometer design used for the determination of wide-band channel characteristics for a next-generation cognitive radio system.

7.3.1 Singular value decomposition

The first large-scale example which was augmented with verification support was an FPGA-targeted implementation of a singular value decomposition (SVD) algorithm, a highly computation-intensive algorithm which can be used for channel decoupling in multiple-antenna wireless communication applications. The original concept for this particular SVD implementation was as an FPGA emulation of a system intended for a custom ASIC design [9]. FPGA emulation of an early ASIC design is an increasingly attractive application of reconfigurable hardware, primarily for the purpose of accelerating the final verification process.

The top-level view of the SVD design is shown in Fig. 7.5. There are eight variables visible at the top level of the system, which are highlighted with dashed ovals. These variables effectively surround the *us* block, which contains the $U\Sigma$ matrix computation, which in this example represents all the heavy computation of the system. The use of variables in such a way allows the inputs and outputs of the system to be captured from (or in the case of inputs, provided to) the design under test. The remainder of the variables in each of the tests beyond the 8-variable case lie within the elements *us* block, which is a highly elaborate network of vectorized, complex arithmetic shown in Fig. 7.6.

Table 7.4 lists the synthesis estimates for the resource requirements of the original BEE2-compliant SVD design as well as each result for an increasing number of additional variables. It is clear from the original, unmodified case that with a starting

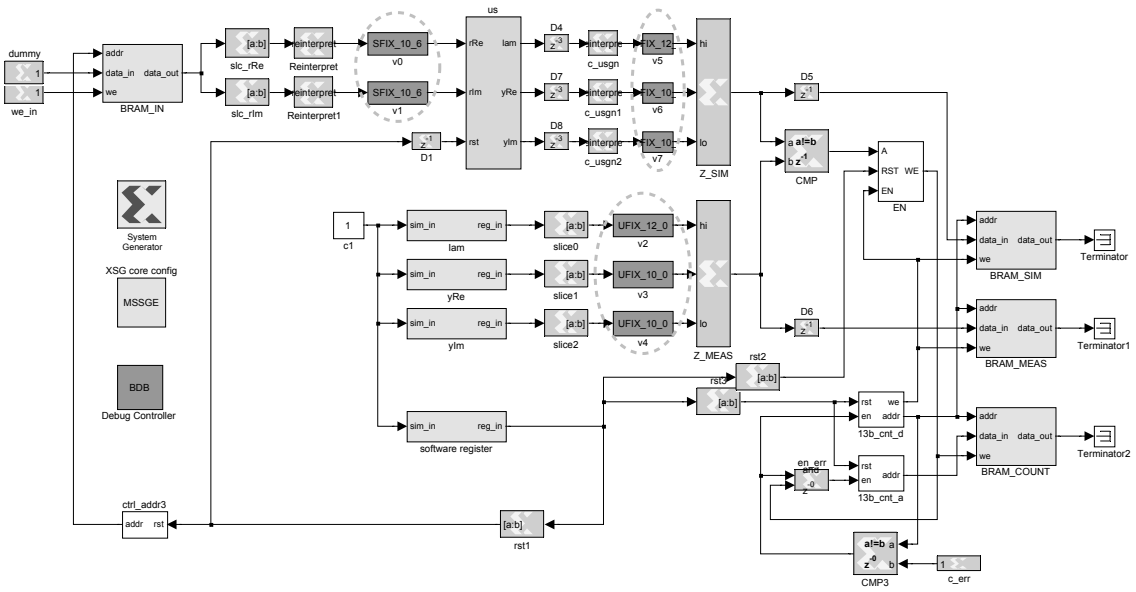

Figure 7.5: Top-level view of SVD design example

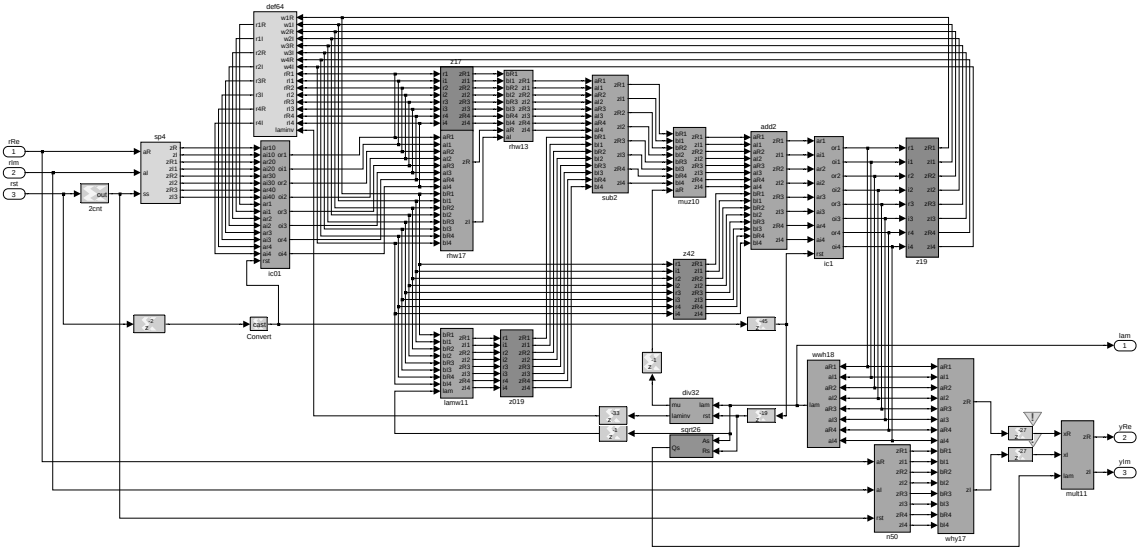

Figure 7.6: Contents of $U\Sigma$ block in the SVD design example

Table 7.4: Synthesis resource estimates for SVD design

Variables	Slices	% of device	Slices/ variable	Bits/ variable	Full compare	Basic compare
0	28,656	86.61	–	–	–	–
2	32,646	98.66	–	10	2	0
4	32,714	98.87	34	10	4	0
8	32,893	99.41	42	10.5	8	0
16	33,586	101.51	68	11.1	14	2
32	34,505	104.28	62	13.7	30	2
64	36,727	111.00	66	13.8	62	2
128	41,072	124.13	67	14.4	126	2

utilization of 87%, the target FPGA is already heavily utilized, and the addition of verification resources will significantly stress the system's performance. While there is no "magic number" for the percentage of an FGPA that should be used, it is commonly understood that design performance will degrade rapidly as utilization approaches 100%, as the freedom available for logic placement and routing becomes much more scarce. The results show that once 16 variables are added to the design, the post-synthesis estimates indicate that more than 100% of the device's resources may be required for the final implementation. This would pose a significant limitation on the availability of data to the user. At the high end, the addition of 128 variables should result in 124% of an XCV2P70 device's available logic resources.

Since this more realistic design example contains variable units which are suited to the actual data types present in the system, the variables are no longer all the same 16-bit size. Therefore, the results in Table 7.4 include three new columns which list the average number of bits per variable and the type of comparison logic present for each of the different experiments. We see that almost all the variables in each experiment contain the full greater-than/less-than/equal-to comparison logic, so there should be a negligible effect in resources due to the comparators. However, we do see that the average number of bits for the variables in the system ends up at 14.4 bits in the 128-variable case. Since this is a slightly smaller value than the 16-bit constant size used in the base infrastructure experiments, we would expect to see a slight reduction in the number of slices per variable, and in fact that is the case. The average number of slices consumed per additional variable reaches a maximum of approximately 66 slices, which is slightly less than the 75-slice value observed in

the base system.

Fortunately for the utility of direct verification, synthesis estimates alone are not an adequate indication of the requirements of the final implementation. Fig. 7.7 plots the number of slices occupied by the design after mapping, placement, and routing were complete. As you can see, the design still fits well within the 33,088 slices available on a XCV2P70 device starting at 16 additional variables. In fact, even for the 128-variable case, the design still narrowly fits within a single device. These results show that the physical-level implementation tools are still able to perform a significant amount of logic packing when it comes time to actually map logic functions into device computational elements — since the 128-variable example results in 33,086 slices being consumed after routing, this is a net savings of 7,986 slices (or 24% of the device).

Fig. 7.8 shows the post-routing minimum operating period of the design for each of the same example cases. Since the original SVD design itself resulted in an operating period of 9.999 ns (the exact value given as a constraint), there was clearly very little room for error after the addition of verification resources. The actual values reported by the post-routing timing analysis tool show that all the designs between 2 and 64 variables have minimum operating periods between 9.992 and 10.701 ns, with the 4- and 16-variable cases in fact having slightly lower periods than the base case (9.997 and 9.992 ns respectively). This demonstrates the innate variability that occurs during logic optimization and pseudo-randomness in the placement and routing algorithms. At the high end, we see a more drastic increase in the minimum period. At 14.795 ns, this is even longer than the base hardware infrastructure. The reason for this increased reduction in frequency can be easily explained: since the device is heavily over-utilized in this case (with 124% of the available slices occupied during synthesis), the physical implementation tools must perform much more aggressive area optimization rather than the usual delay optimization. For this reason, the delay is additionally impacted in order to pack the design onto one device. While the reason for this increase is important to understand, it should be recalled that the goal of this work was to dramatically increase the usability and programmability of reconfigurable hardware. As such, it is an acceptable tradeoff to degrade the operating frequency of the hardware by 48% when the hardware is already running 6 orders of magnitude faster than simulation itself.

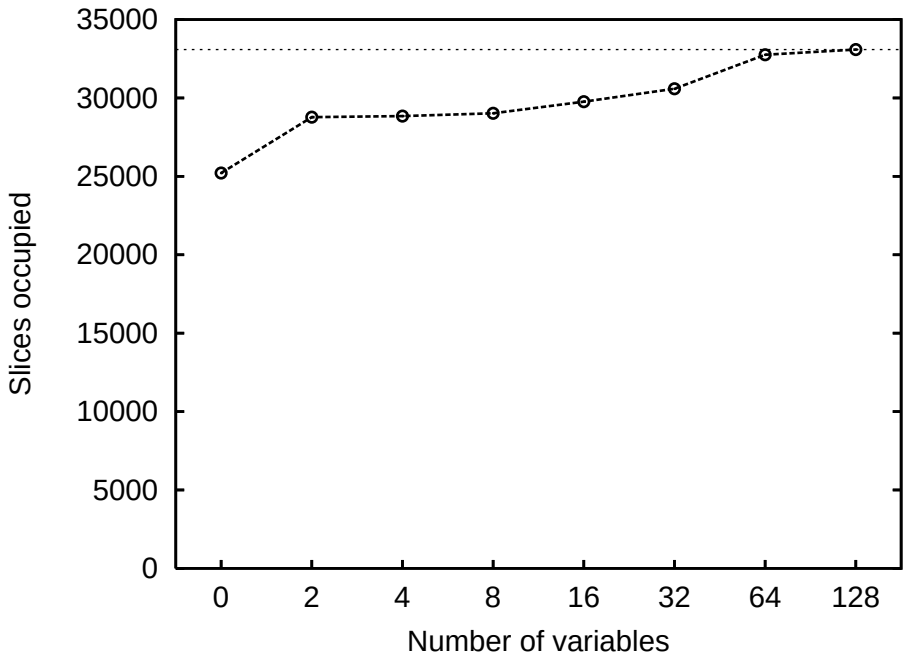

Figure 7.7: Post-routing device utilization of SVD design

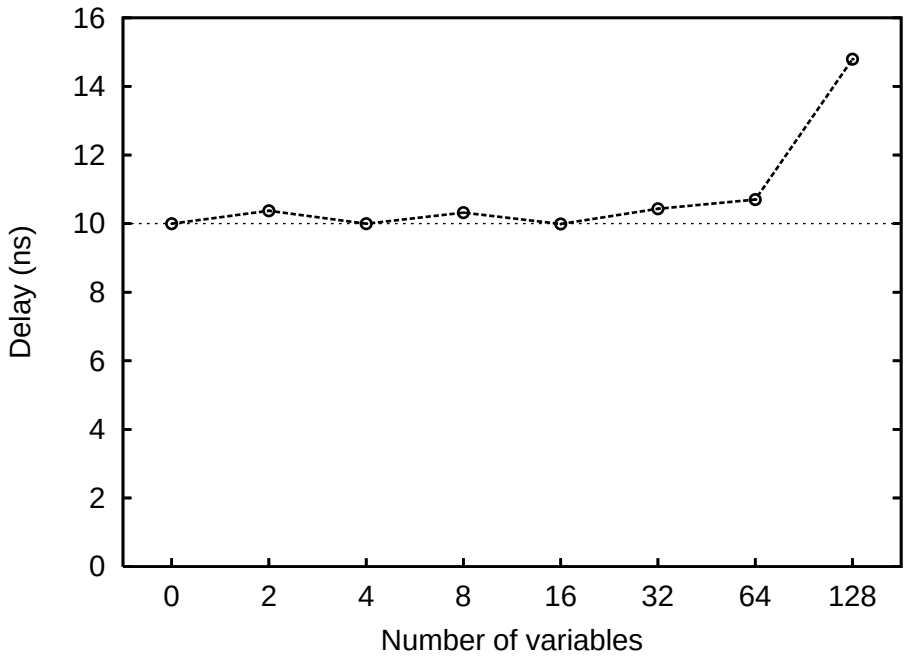

Figure 7.8: Post-routing critical path measurements for SVD design

7.3.2 Pocket spectrometer

The final real-world example investigated here was a "pocket spectrometer" design, originally conceived by the CASPER group at U. C. Berkeley, who were early adopters of the BEE2 platform, and recently adopted for the use of robust channel measurements toward the design of a cognitive radio system [2]. This spectrometer consists primarily of a polyphase filter bank followed by a fast Fourier transform (FFT) component.

The top-level view of the pocket spectrometer design is shown in Fig. 7.9. Once again, the variables which cover each of the inputs to the system are highlighted by dashed ovals. In this case, the original inputs to the system were mapped to general-purpose I/O pins on the user FPGA, which received inputs directly from an external analog front-end and analog-to-digital converter. While such an interface can still be accommodated by the system with verification support included, the user must understand that without any flow control over the input data, samples will be missed while the design under test is throttled to allow the streaming of variable data (and naturally, in the event of an assertion failure, the design clock will simply be stopped). However, even with these constraints, it may still be possible to periodically buffer and accept real-world inputs even with verification enabled, which is yet another very powerful advantage of the direct verification approach. And of course, the placement of variables on the inputs to the system would also allow input samples to be generated in the design environment (in this case, Matlab) and provided to the system through the use of the `force` mechanism.

Fig. 7.10 also shows a portion of the contents of the *pocketspec* subsystem which contains the majority of the computational logic in the system (the entire layout was quite large, and not easily legible in printed form). Here we can see the inclusion of variables between the polyphase filter bank and FFT blocks which can be useful for monitoring the high-level computed results of the system. The remainder of the variables for the larger experiments were placed beyond the visible scope of Fig. 7.10 as well as within the filter bank and FFT components.

Table 7.5 lists the synthesis estimates for the hardware resource utilization of the spectrometer design. Here we see that the original design occupies 21,735 slices, or about 66% of the device. While slightly more modest than the SVD design of

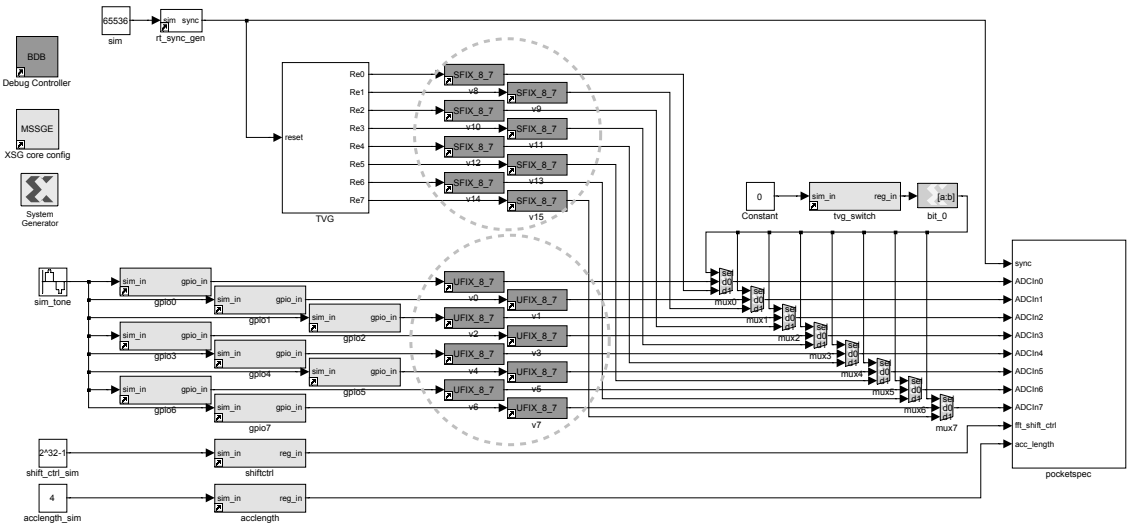

Figure 7.9: Top-level view of spectrometer design example

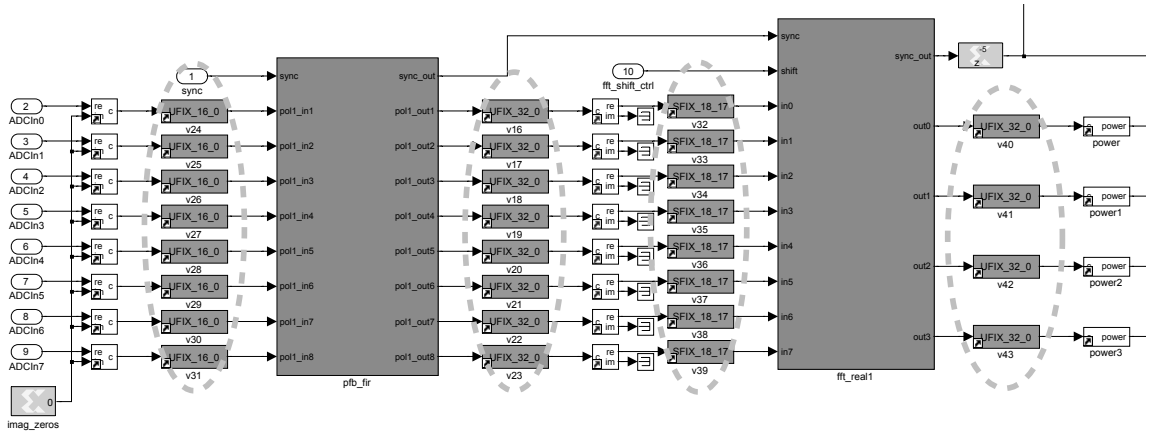

Figure 7.10: Structure of underlying computation in spectrometer design example

Table 7.5: Synthesis resource estimates for spectrometer design

Variables	**Slices**	**% of device**	**Slices/ variable**	**Bits/ variable**	**Full compares**	**Basic compares**
0	21,735	65.69	–	–	–	–
2	25,722	77.74	–	8	2	0
4	25,776	77.90	27	8	4	0
8	25,933	78.38	36	8	8	0
16	26,520	80.15	57	8	16	0
32	27,864	84.21	72	16	32	0
64	30,784	93.04	82	19.4	64	0
128	35,897	108.49	81	19.4	128	0

Sect. 7.3.1, synthesis results still indicate that the 128-variable case will require more than the number of slices available on the XCV2P70 device. Once again, the actual characteristics of the additional variables, which are configured to match the data types present in the design, are included with the resource estimates. Here, we see that all the variable units are again configured to instantiate full comparison logic, however, due to the large number of full 32-bit variable sizes, the average number of bits per variable reaches 19.4 bits — 3.4 bits more than the base infrastructure example. Correspondingly, the number of slices consumed per additional variable is slightly higher, reaching a maximum of approximately 81 slices, or 6 more slices per variable than the base infrastructure model.

Fig. 7.11 and Fig. 7.12 once again show the post-routing results for device utilization and minimum operating period, respectively. We see that the physical implementation tools again manage to pack the entire design onto the device in the 128-variable case. Interestingly, we also see that the critical path measurements almost exactly track the base infrastructure results. This helps to prove that the hardware infrastructure logic was well-designed to lie as much as possible in parallel to the design logic, and that the pure addition of verification support into a design should have a minimal direct impact on the critical path. Of course, in this case, the physical implementation tools were not nearly as stressed as in the SVD design, where an additional 24% of the target device had to be found at the expense of performance.

In summary, it has been demonstrated that the hardware infrastructure required to support verification can span a large range depending on the number of variables

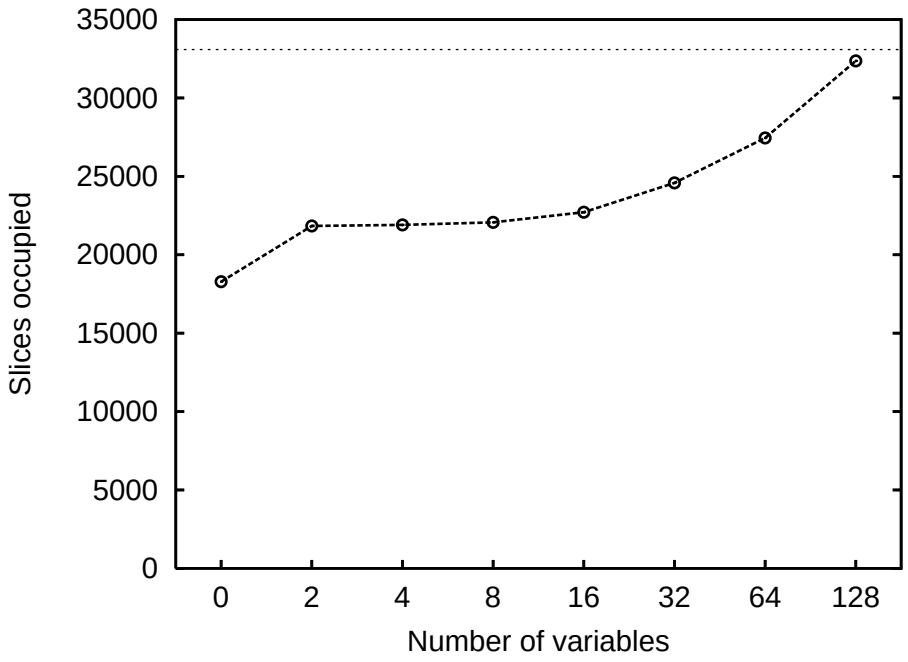

Figure 7.11: Post-routing device utilization of spectrometer design

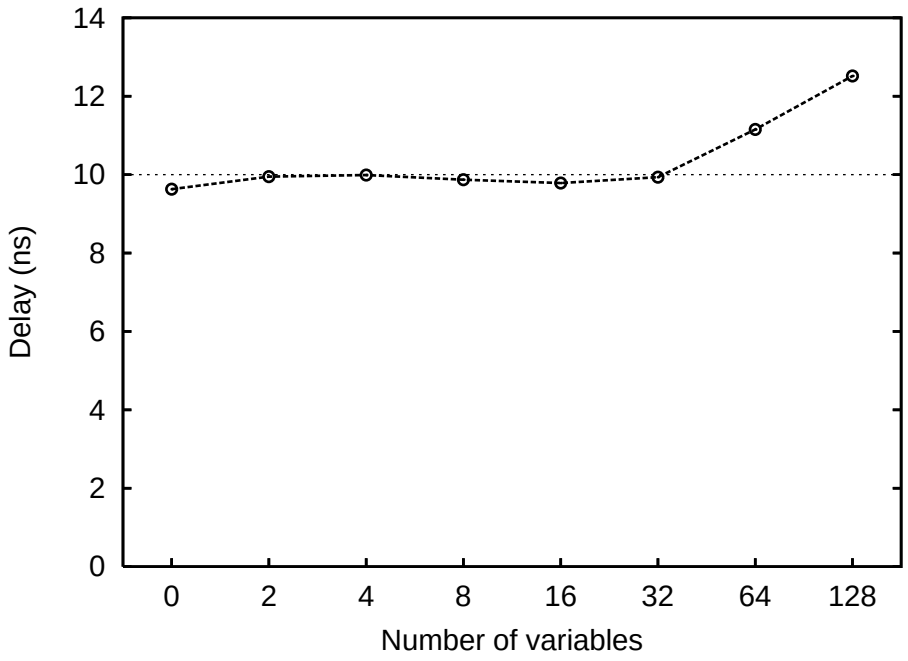

Figure 7.12: Post-routing critical path measurements for spectrometer design

added to the design. Fortunately, however, a significant amount of logic packing can be achieved by the physical implementation tools, which for these examples, reached as high as 24% of the target device. In the most basic case, the minimum number of resources required by the core infrastructure was approximately 12% of the device, although about 8% of this total was due purely to the DDR2 memory controller, which could easily be optimized into a more efficient architecture for direct streaming to external memory, or on alternate platforms could even be offloaded to an external device. In terms of the number of additional resource required for each hardware variable unit, this value scales, as expected, with the average number of bits declared for each variable. In the "nominal" case of 16 bits per variable, this value was approximately 75 slices per variable added.

Chapter 8

Conclusion

The previous chapters have presented an array of topics, beginning with the benefits of reconfigurable hardware platforms for a wide range of applications, and extending to the details of how one could implement direct verification to achieve analysis capabilities comparable to pure functional simulation, while at the same time providing virtually the entire throughput of the hardware platform. Hopefully, the arguments have been compelling that reconfigurable hardware has become a viable and often superior platform for high-performance signal processing and scientific computation; and similarly, that it is quite feasible to create a verification methodology built on and around the hardware platform which can provide an equal level of data mutability and fine-grained control over execution which were once only available in the software environment. These arguments are made at the time of writing, and will only become more viable as circuit and fabrication technology improve, and the spatial capacity of the hardware becomes even more powerful (and consequently, even more complex to model in software).

The remainder of this section serves to outline and summarize the arguments made in support of direct verification, along with its limitations and potential for further improvement.

8.1 Summary of results

By including the concept of variables in hardware, one which is intuitively apparent and familiar to software developers, a design becomes not just a powerful,

directly-mapped implementation of a given algorithm, but also a flexible and highly observable system which can be interacted with by the designer in a convenient fashion via named entities which describe their intended function. In addition, by supporting the storage of each variable's value on any given clock cycle (up to the limits of attached storage capacity), the hardware platform can be an extremely powerful computation engine, but also provides some insight into the events leading up to any given point of interest which may not have already been subject to observation.

With respect to identifying points of interest during execution, the inclusion of dynamically-definiable assertions in hardware variables is an extremely powerful mechanism for both controlling design execution and isolating the causes of failure in a design. Because assertion checking was designed to occur in parallel to the existing critical path of the design under test, this functionality also comes with a cost almost purely in terms of overall device utilization and not directly to the operating frequency of the system. Of course, higher device utilization usually does result in lower operating frequency as utilization approaches 100% and the detection of the breakpoint signal before stopping the design clock may affect the setup time of registers on the design clock net. However, these reductions in performance are considered to be within the (perhaps arbitrary) "one order of magnitude" limit, which in the case of the comparison against software simulation, is an acceptable amount of degradation. Alternate methodologies already exist to inspect an FPGA system externally with virtually no overhead, but they do not provide the level of data analysis which is made possible by direct verification and often result in far lower throughput than the minor reduction in operating frequency of the original hardware design.

In addition to supporting the variable-related features above, it has been shown that by combining a library of functional units with soft parameter inputs with direct verification, it becomes possible to truly explore the functional and numerical limits of an algorithm or model in hardware. Previously, it was often necessary to evaluate the numerical limits of an algorithm in the simulation environment long before the design was implemented in hardware. Once mutable variables are available in hardware, a designer can leverage fully parameterized functional units to perform such algorithm exploration at hardware speed, which immediately expands the range of possibilities available (or consequently, reduces the time required) for analysis.

Naturally, these features do not come without some cost in hardware resources and

the operating frequency of the design. The results which have been shown demonstrate that this overhead can cover a large range based on the number of variables added to the system. Fortunately, the physical implementation tools also offer a significant amount of optimization in the packing of logical operations into device functional units. In the examples shown above, the physical mapping process can salvage as much as 24% of the device while packing logical operations into device primitives. And even in the case where one quarter of the available resources had to be aggressively re-packed, the net impact on the operating frequency of the design was only 11.7% compared to the same number of variables in the base infrastructure. These results are yet more evidence that the amount of hardware resources which can be exploited for advanced purposes such as verification may be plentiful, and are only becoming more plentiful as silicon technology improves.

In summary, the results presented here have shown that it is possible to create a fully automated verification infrastructure in hardware which rivals the ability to access and analyze data compared to the traditional software environments. While the cost of this functionality in hardware is measurable, it is mitigated by the ability of the physical implementation tools to further optimize and pack logical functions into device primitives. And of course, it is always possible to follow the traditional methods for simplifying verification: that is, to modularly and incrementally verify components of a design individually before attempting to prove the correctness of the system as a whole. By modularly attempting to verify the components of a system, each piece of the complete design can be proven to operate correctly in a more manageable form factor before the complete behavior of the system under test is examined.

8.2 Future opportunities

The approach to direct verification presented here is quite beneficial for the testing of direct-mapped, synchronous designs targeted to reconfigurable hardware platforms. However, there are a number of systems and platforms which are not covered by direct verification which could benefit from a similar approach, if not merely an extension of the same methodology.

The first area of opportunity to extend direct verification would be multi-rate

hardware systems, which contain more than one clock domain running at different frequencies. As discussed thoroughly above, direct verification as it is conceived here deals only with single-rate systems which run on a single design clock frequency. In practice, there are many systems which require multiple clock frequencies to operate. In this case, there are multiple approaches that could be taken to extend the methodology proposed here. In the event that all the clock domains are integer multiples of one another, it is possible that the same methodology could be taken, although the core debug controller would need to drive a clock enable input directly into the clock multiplier/divider itself. This may well have an impact on the same-cycle breakpoint dependency of the hardware assertion checking logic, depending on the behavior of the clock generation circuitry. However, in such cases, it may be possible with the addition of some extra logic to detect the number of derived clock cycles which have occurred since the breakpoint took place. The exact mechanism by which this happens, and the constraints which it must follow, are left to future work.

Another area of opportunity lies in the application of direct verification to co-designed hardware/software systems, where the algorithm under test contains both a software application and an attached (and simultaneously running) hardware system. There already exists a large range of hardware/software systems which vary greatly in their underlying behavior and timing patterns. However, one fundamental feature of such algorithms is that the non-determinism of a processor with cache running a software application is often difficult to quantify in tandem with a fixed-freqency, well-characterized hardware component. While there are no features of direct verification which appear to address this software concurrency issue, it may be possible to extend the concept of variables and cycle-by-cycle storage to such architectures for the purpose of characterizing the concurrent state of software and hardware. This aspect of design has not yet been explored, and remains an open area of research in terms of complete system verification.

Other than these two specific areas of concern with respect to the application of direct verification, some additional inspection into the underlying timing issues of the hardware infrastructure would be worthwhile. As explained often in the preceeding chapters, the primary goal of this work has been to deliver the concepts of variables and high-level process control to the hardware environment. In comparison to the traditional software simulation environments, direct verification offers such an ex-

treme advantage in performance that smaller tradeoffs in operating frequency were neglected. Of course, the fact remains that by more deeply investigating the target device architecture (in this case, FPGAs, and more specifically, the Virtex microarchitecture), it may be possible to improve the critical path delay of a system under test, and therefore the overall throughput of the system. In addition, it was directly mentioned that the external architecture used in this implementation was taken from the already-available component provided with the BEE2 platform. A much more efficient controller designed purely for streaming onto attached memory may drastically reduce the number of cycles required to write variable data to memory on each design cycle. While the exact method for writing variable data to external storage is platform-dependent, the fact remains that a careful investigation into the timing performance of the external storage interface can be critical, as it will often be the source of throughput bottlenecks given current technology.

In conclusion, direct verification is a unique, new approach to providing high-level data manipulation and recovery features on reconfigurable platforms, which previously were only available in the software environments of application debugging and functional analysis. While there exist some constraints on the current capabilities of direct verification, the overall benefit of the approach conceived here provides software-like data access and process control, while simultaneously enabling the net system throughput of the underlying hardware platform.

Bibliography

[1] Altera Corporation. *Design Debugging Using the SignalTap II Embedded Logic Analyzer*, May 2008. http://www.altera.com/products/software/products/-quartus2/verification/signaltap2/sig-index.html.

[2] Robert W. Brodersen, Adam Wolisz, Danijela Cabrić, Shridhar Mubaraq Mishra, and Daniel Willkomm. CORVUS: A Cognitive Radio Approach for Usage of Virtual Unlicensed Spectrum. White paper, University of California, Berkeley, 2004. http://bwrc.eecs.berkeley.edu/Research/MCMA/CR_White_paper_final1.pdf.

[3] Kevin Camera. SF2VHD: A Stateflow to VHDL translator. Master's thesis, University of California, Berkeley, May 2001.

[4] Celoxica Limited. *Handel-C Language Reference Manual*, 2005.

[5] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. W. Brodersen, and B. Nikolić. Rapid design and analysis of communication systems using the BEE hardware emulation environment. In *Proc. IEEE Rapid System Prototyping Workshop*, June 2003.

[6] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22(2):114–125, March/April 2005.

[7] IEEE. *IEEE Standard SystemC Language Reference Manual*, March 2006.

[8] Alex Krasnov, Andrew Schulz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. RAMP blue: A message-passing manycore system in FPGAS. In *Proc. Field Programmable Logic and Applications (FPL)*, pages 54–61, August 2007.

[9] Dejan Marković, Borivoje Nokolić, and Robert W. Brodersen. Power and area minimization for multidimensional signal processing. *IEEE Journal of Solid-State Circuits*, 42(4):922–934, April 2007.

[10] The Mathworks. *Matlab 7 Getting Started Guide*, July 2007. http://www.mathworks.com/products/matlab.

[11] The Mathworks. *Simulink Product User Guide*, July 2007. http://www.mathworks.com/products/simulink.

[12] Hayden Kwok-Hay So and Robert W. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)*, 7, February 2008.

[13] Texas Instruments, Inc. *TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors*, February 2008. http://focus.ti.com/docs/-prod/folders/print/tms320c6415t.html.

[14] J. Tombs, M. A. Aguirre Echanóve, F. Muñoz, V. Baena, A. Torralba, A. Fernandez-León, and F. Tortosa. The implementation of a FPGA hardware debugger system with minimal system overhead. In *Proc. Field Programmable Logic and Applications (FPL)*, pages 1062–1066, August 2004.

[15] Xilinx, Inc. *ChipScope Pro Software and Cores User Guide*, May 2007. http://www.xilinx.com/ise/optional_prod/cspro.htm.

[16] Xilinx, Inc. *Embedded Systems Tools Guide*, January 2007. http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[17] Xilinx, Inc. *System Generator for DSP User Guide*, May 2007. http://www.xilinx.com/ise/optional_prod/system_generator.htm.

[18] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, March 2007. http://www.xilinx.com/products/silicon_solutions/fpgas/-virtex/virtex_ii_pro_fpgas.

[19] Xilinx, Inc. *Virtex-5 Family Overview*, May 2008. http://www.xilinx.com/-products/silicon_solutions/fpgas/virtex/virtex5.

Appendix A

Hardware implementation data

This chapter contains the VHDL implementation files which describe the hardware components of the verification infrastructure as designed for BEE2.

A.1 Variable unit implementation

The following VHDL code represents the behavioral description of a hardware variable unit presented in Sect. 4.1. Each variable unit is instantiated with unique generic parameters as a *pcore* in the system description (MHS file) in the Xilinx EDK environment.

A.1.1 bdb_variable

```
-- Toplevel wrapper for variable unit

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity bdb_variable is

  generic
  (
    NUMVAR : integer := 2;  -- required to determine width of bus ports
    VARID : integer := 0;  -- ordinal ID of this variable
    BW : integer := 16; -- bitwidth of variable data
    SW : integer := 32; -- storage size of variable data
    USE_SIGNED : integer := 1;  -- interpret the data value as signed
    ASSERT_TYPE : integer := 2  -- the type of assertion comparison to perform
  );

  port
  (
    -- Data to/from the system/core come straight from the variable logic and
    -- are attached directly in the MHS file
    data_to_sys : out std_logic_vector((BW-1) downto 0);
    data_from_sys : in std_logic_vector((BW-1) downto 0);
    data_to_core : out std_logic_vector((SW-1) downto 0);
```

```
    data_from_core : in std_logic_vector((SW-1) downto 0);
    assert_break_out : out std_logic;
    -- Single-bit write enables are port-mapped as a whole bus from the core,
    -- as the MHS file does not allow vector indexing
    value_we_core_bus : in std_logic_vector((NUMVAR-1) downto 0);
    src_we_core_bus : in std_logic_vector((NUMVAR-1) downto 0);
    thresh_we_core_bus : in std_logic_vector((NUMVAR-1) downto 0);
    assert_mode_we_core_bus : in std_logic_vector((NUMVAR-1) downto 0);
    -- Variables require both the source and gated system clocks
    core_clk : in  std_logic;
    bdb_clk : in  std_logic
  );

end bdb_variable;


architecture structure of bdb_variable is

  component bdb_variable_logic
    generic ( VARID : integer;
              BW : integer;
              SW : integer;
              USE_SIGNED : integer;
              ASSERT_TYPE : integer );
    port ( data_to_sys : out std_logic_vector((BW-1) downto 0);
           data_from_sys : in std_logic_vector((BW-1) downto 0);
           data_to_core : out std_logic_vector((SW-1) downto 0);
           data_from_core : in std_logic_vector((SW-1) downto 0);
           value_we_core : in std_logic;
           src_we_core : in std_logic;
           thresh_we_core : in std_logic;
           assert_mode_we_core : in std_logic;
           assert_break_out : out std_logic;
           core_clk : in std_logic;
           bdb_clk : in std_logic );
  end component;

  signal this_value_we : std_logic;
  signal this_src_we : std_logic;
  signal this_thresh_we : std_logic;
  signal this_assert_mode_we : std_logic;

begin

  -- Split off the single-bit control signals for this variable
  -- MHS syntax doesn't allow indexing, so we must do it here
  this_value_we <= value_we_core_bus(VARID);
  this_src_we <= src_we_core_bus(VARID);
  this_thresh_we <= thresh_we_core_bus(VARID);
  this_assert_mode_we <= assert_mode_we_core_bus(VARID);

  -- Instantiate the variable logic
  bdb_variable_logic_inst : bdb_variable_logic
    generic map
```

```
    (
      VARID => VARID,
      BW => BW,
      SW => SW,
      USE_SIGNED => USE_SIGNED,
      ASSERT_TYPE => ASSERT_TYPE
    )
    port map
    (
      data_to_sys => data_to_sys,
      data_from_sys => data_from_sys,
      data_to_core => data_to_core,
      data_from_core => data_from_core,
      value_we_core => this_value_we,
      src_we_core => this_src_we,
      thresh_we_core => this_thresh_we,
      assert_mode_we_core => this_assert_mode_we,
      assert_break_out => assert_break_out,
      core_clk => core_clk,
      bdb_clk => bdb_clk
    );

end structure;
```

A.1.2 bdb_variable_logic

```
-- Main logic for BDB variable unit

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity bdb_variable_logic is

  generic
  (
    -- Unique ordinal ID for this variable
    VARID : integer := 0;

    -- Bitwidth of the variable data
    BW : integer := 16;

    -- Storage size of the variable data in bits
    SW : integer := 32;

    -- Whether or not the data value is interpreted as signed
    USE_SIGNED : integer := 1;

    -- Type of assertion comparison to perform
    --  0: None (all disabled)
    --  1: Equal/Not-equal only
    --  2: Equal/Greater-than/Less-than
    ASSERT_TYPE : integer := 2
```

```
  );

  port
  (
    -- Data value being driven to the user system
    data_to_sys : out std_logic_vector((BW-1) downto 0);

    -- Current data value coming from the user system
    data_from_sys : in std_logic_vector((BW-1) downto 0);

    -- Data value being driven to the BDB core
    data_to_core : out std_logic_vector((SW-1) downto 0);

    -- Control value coming from the BDB core
    --  (destination determined by the active write enable signal)
    data_from_core : in std_logic_vector((SW-1) downto 0);

    -- Write enable for the forced data value
    value_we_core : in std_logic;

    -- Write enable for the data source selector
    src_we_core : in std_logic;

    -- Write enable for the assertion threshold value
    thresh_we_core : in std_logic;

    -- Write enable for the desired assertion mode
    assert_mode_we_core : in std_logic;

    -- Signal indicating breakpoint condition was met
    assert_break_out : out std_logic;

    -- Free-running core clock
    core_clk : in std_logic;

    -- Gated (user system) clock
    bdb_clk : in std_logic
  );

end bdb_variable_logic;


architecture behavior of bdb_variable_logic is

  signal data_sys : std_logic_vector((BW-1) downto 0);
  signal data_core : std_logic_vector((BW-1) downto 0);
  signal data_out : std_logic_vector((BW-1) downto 0);
  signal sign_bit : std_logic;
  signal src : std_logic;
  signal thresh : std_logic_vector((BW-1) downto 0);
  signal break_on_gt : std_logic;
  signal break_on_lt : std_logic;
  signal break_on_eq : std_logic;
  signal val : std_logic_vector((BW-1) downto 0);
```

```
  signal val_gt_thresh : std_logic;
  signal val_lt_thresh : std_logic;
  signal val_eq_thresh : std_logic;

begin

  -- Clocking process for user system data (uses gated clock)
  gated_clock : process(bdb_clk)
  begin
    if bdb_clk'EVENT and (bdb_clk = '1') then
      data_sys <= data_from_sys;
    end if;
  end process;

  -- Clocking process for the core control registers (uses non-gated clock)
  non_gated_clock : process(core_clk)
  begin
    if core_clk'EVENT and (core_clk = '1') then
      -- Forced value from core is stored on value_we_core
      if value_we_core = '1' then
        data_core <= data_from_core(BW-1 downto 0);
      end if;
      -- Data source selector is stored on src_we_core
      if src_we_core = '1' then
        src <= data_from_core(0);
      end if;
      -- Assertion threshold value is stored on thresh_we_core
      if thresh_we_core = '1' then
        thresh <= data_from_core(BW-1 downto 0);
      end if;
      -- Assertion mode is stored on assert_mode_we_core
      if assert_mode_we_core = '1' then
        break_on_gt <= data_from_core(2);
        break_on_lt <= data_from_core(1);
        break_on_eq <= data_from_core(0);
      end if;
    end if;
  end process;

  -- Data value returned to system is controlled by source selector
  data_out <= data_sys when src = '0' else data_core;
  data_to_sys <= data_out;

  -- Data value reported to core is sign-extended version of data to system
  sign_bit <= '0' when (USE_SIGNED = 0) else data_out(BW-1);
  sign_ext_core_data : if (SW > BW) generate
  begin
    data_to_core <= (SW-1 downto BW => sign_bit) & data_out;
  end generate;
  no_ext_core_data : if (SW = BW) generate
  begin
    data_to_core <= data_out;
  end generate;
```

```
  -- KBC: There are some serious timing issues here, based on when and how
  --      the comparison is computed (i.e. before the registers, or after)
  --      and on which cycle the breakpoint takes effect.

  -- Define the value to be used for threshold comparison
  -- NOTE: This is done in case the comparison base is parameterized later
  val <= data_sys;

  -- Case: all assertions disabled (ASSERT_TYPE=0)
  compare_none : if (ASSERT_TYPE = 0) generate
  begin
    val_eq_thresh <= '0';
    val_lt_thresh <= '0';
    val_gt_thresh <= '0';
  end generate;

  -- Case: assertion comparisons for equal/not-equal (ASSERT_TYPE=1)
  compare_eq_ne_only : if (ASSERT_TYPE = 1) generate
  begin
    -- Simply compare the data value for equality to threshold
    val_eq_thresh <= '1' when (val = thresh) else '0';
    val_lt_thresh <= not val_eq_thresh;
    val_gt_thresh <= '0';
  end generate;

  -- Case: assertion comparisons for full gt/lt/eq (ASSERT_TYPE=2)
  compare_gt_lt_eq : if (ASSERT_TYPE = 2) generate
  begin
    -- Infer comparators to derive the greater-than and equal results
    compare_gte_signed : if (USE_SIGNED /= 0) generate
      val_gt_thresh <= '1' when SIGNED(val) > SIGNED(thresh) else '0';
      val_eq_thresh <= '1' when SIGNED(val) = SIGNED(thresh) else '0';
    end generate;
    compare_gte_unsigned : if (USE_SIGNED = 0) generate
      val_gt_thresh <= '1' when UNSIGNED(val) > UNSIGNED(thresh) else '0';
      val_eq_thresh <= '1' when UNSIGNED(val) = UNSIGNED(thresh) else '0';
    end generate;
    -- Less-than is derived from comparator outputs
    val_lt_thresh <= not (val_gt_thresh or val_eq_thresh);
  end generate;

  -- Set the break signal based on which conditions are enabled
  assert_break_out <= not src and  -- forced value prevents breakpoint
                      ((val_gt_thresh and break_on_gt) or
                       (val_lt_thresh and break_on_lt) or
                       (val_eq_thresh and break_on_eq));

end behavior;
```

A.2 Core controller implementation

The following VHDL code represents the behavioral description of the core debug controller presented in Sect. 4.2. A single instance of the core controller is defined in the MHS file with the corresponding generic parameters. Note that the state machine itself (the component named `bdb_core_ctrl`) is generated automatically from a Stateflow model of its behavior by the custom tool SF2VHD [3] and is therefore not reproduced here. For a clearer understanding of its functionality, please refer to Fig. 4.4.

A.2.1 bdb_core

```
-- BDB core control unit

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

library UNISIM;
use UNISIM.vcomponents.all;


entity bdb_core is

  generic
  (
    -- Number of variables in the system
    NUMVAR : integer := 50;

    -- Number of integer bits needed to select a variable (ceil[log2[NUMVAR]])
    SELBITS : integer := 6;

    -- Bitwidth of variable data
    W : integer := 8
  );

  port
  (
    -- Merged bus with data values from all variables in design
    vars_data_in : in  std_logic_vector((NUMVAR*W)-1 downto 0);

    -- Control value sent to all variables in design
    var_data_out : out std_logic_vector(W-1 downto 0);

    -- Write enables for variable forced values
    vars_value_we_out : out std_logic_vector(NUMVAR-1 downto 0);

    -- Write enables for variable source selects
    vars_src_we_out : out std_logic_vector(NUMVAR-1 downto 0);

    -- Write enables for variable assertion thresholds
    vars_thresh_we_out : out std_logic_vector(NUMVAR-1 downto 0);
```

```
    -- Write enables for variable assertion modes
    vars_assert_mode_we_out : out std_logic_vector(NUMVAR-1 downto 0);

    -- Assertion signals from all variables
    vars_assert_break_in : in std_logic_vector(NUMVAR-1 downto 0);

    -- BDB software command interface
    bdb_cmd_in : in  std_logic_vector(31 downto 0);
    bdb_data_in : in  std_logic_vector(31 downto 0);
    bdb_status_out : out std_logic_vector(31 downto 0);
    bdb_data_out : out std_logic_vector(31 downto 0);

    -- DRAM asynchronous user logic interface
    mem_cmd_addr : out std_logic_vector(31 downto 0);
    mem_wr_din : out std_logic_vector(287 downto 0);
    mem_wr_be : out std_logic_vector(35 downto 0);
    mem_cmd_rnw : out std_logic;
    mem_cmd_tag : out std_logic_vector(31 downto 0);
    mem_cmd_valid : out std_logic;
    mem_rd_ack : out std_logic;
    mem_cmd_ack : in  std_logic;
    mem_rd_dout : in  std_logic_vector(287 downto 0);
    mem_rd_tag : in  std_logic_vector(31 downto 0);
    mem_rd_valid : in  std_logic;

    -- Active-high reset signal for controller state machine
    rst : in  std_logic;

    -- Main (non-gated) system clock output
    main_clk : out std_logic;

    -- Gated system clock output
    bdb_clk : out std_logic;

    -- Non-gated clock input directly from DCM
    dcm_clk : in  std_logic
  );

end bdb_core;


architecture behavior of bdb_core is

  component bdb_core_ctrl
    port ( bdb_cmd_low : in std_logic_vector(15 downto 0);
           bdb_cmd_high : in std_logic_vector(15 downto 0);
           bdb_data_in : in std_logic_vector(31 downto 0);
           bdb_status_out : out std_logic_vector(7 downto 0);
           bdb_cycle_reset : out std_logic;
           bdb_user_capture : out std_logic;
           bdb_user_restore : out std_logic;
           icap_bus_out : out std_logic_vector(31 downto 0);
           icap_clk_out : out std_logic;
```

```
          var_assert_break : in std_logic;
          bdb_data_out : out std_logic_vector(31 downto 0);
          ctrl_var_sel_int : out std_logic_vector(31 downto 0);
          ctrl_var_data_out : out std_logic_vector(31 downto 0);
          ctrl_var_value_we : out std_logic;
          ctrl_var_src_we : out std_logic;
          ctrl_var_thresh_we : out std_logic;
          ctrl_var_assert_mode_we : out std_logic;
          ctrl_clk_halt : out std_logic;
          ctrl_var_data_in : in std_logic_vector(31 downto 0);
          bdb_cycle_count : in std_logic_vector(31 downto 0);
          icap_bus_in : in std_logic_vector(31 downto 0);
          mem_ctrl_ack : in std_logic;
          mem_ctrl_dout : in std_logic_vector(31 downto 0);
          mem_ctrl_req : out std_logic;
          mem_ctrl_stream_req : out std_logic;
          mem_ctrl_rnw : out std_logic;
          mem_ctrl_din : out std_logic_vector(31 downto 0);
          reset : in std_logic;
          ce : in std_logic;
          clk : in std_logic );
end component;

component bdb_core_ddr_ctrl
  generic ( NUMVAR : integer;
            W : integer );
  port ( ddr_cmd_addr : out std_logic_vector(31 downto 0);
         ddr_wr_din : out std_logic_vector(255 downto 0);
         ddr_wr_be : out std_logic_vector(31 downto 0);
         ddr_cmd_rnw : out std_logic;
         ddr_cmd_tag : out std_logic_vector(31 downto 0);
         ddr_cmd_valid : out std_logic;
         ddr_rd_ack : out std_logic;
         ddr_cmd_ack : in  std_logic;
         ddr_rd_dout : in  std_logic_vector(255 downto 0);
         ddr_rd_tag : in  std_logic_vector(31 downto 0);
         ddr_rd_valid : in  std_logic;
         mem_ctrl_ack : out std_logic;
         mem_ctrl_busy : out std_logic;
         mem_ctrl_dout : out std_logic_vector(31 downto 0);
         mem_ctrl_req : in  std_logic;
         mem_ctrl_stream_req : in  std_logic;
         mem_ctrl_rnw : in  std_logic;
         mem_ctrl_din : in  std_logic_vector(31 downto 0);
         vars_data_in : in  std_logic_vector((NUMVAR*W)-1 downto 0);
         dram_clk_wait : out std_logic;
         sys_clk_halted : in  std_logic;
         rst : in  std_logic;
         bdb_clk : in  std_logic;
         clk : in  std_logic );
end component;

component ICAP_VIRTEX2
  port ( BUSY : out std_logic;
```

```
         O : out std_logic_vector(7 downto 0);
         CE : in std_logic;
         CLK : in std_logic;
         I : in std_logic_vector(7 downto 0);
         WRITE : in std_logic );
end component ICAP_VIRTEX2;

component CAPTURE_VIRTEX2
  port ( CAP : in std_logic;
         CLK : in std_logic );
end component CAPTURE_VIRTEX2;

component STARTUP_VIRTEX2
  port ( CLK : in std_logic;
         GSR : in std_logic;
         GTS : in std_logic );
end component STARTUP_VIRTEX2;

component BUFGCE
  port ( I: in std_logic;
         O: out std_logic;
         CE: in std_logic );
end component;

component BUFG
  port ( I: in std_logic;
         O: out std_logic );
end component;

-- One-hot variable selection bus
signal var_sel : std_logic_vector(NUMVAR-1 downto 0);
constant var_sel_zero : std_logic_vector(NUMVAR-1 downto 0)
                      := (others => '0');

-- Core controller ports
signal ctrl_var_data_in_32 : std_logic_vector(31 downto 0);
signal ctrl_var_data_in : std_logic_vector(W-1 downto 0);
signal ctrl_var_data_out_32 : std_logic_vector(31 downto 0);
signal ctrl_var_sel_int : std_logic_vector(31 downto 0);
signal ctrl_var_value_we : std_logic;
signal ctrl_var_src_we : std_logic;
signal ctrl_var_thresh_we : std_logic;
signal ctrl_var_assert_mode_we : std_logic;
signal ctrl_clk_halt : std_logic;
signal bdb_cycle_count : unsigned(31 downto 0);
signal bdb_cycle_reset : std_logic;
signal bdb_user_capture : std_logic;
signal bdb_user_restore : std_logic;
signal icap_bus_out : std_logic_vector(31 downto 0);
signal icap_clk_out : std_logic;
signal icap_bus_in : std_logic_vector(31 downto 0);
signal bdb_status_ctrl_out : std_logic_vector(7 downto 0);

signal bdb_user_restore_delayed : std_logic;
```

```
  signal gsr_delay_count : unsigned(25 downto 0);
  signal bdb_user_capture_delayed : std_logic;
  signal cap_delay_count : unsigned(25 downto 0);

  -- DDR command controller ports
  signal mem_ctrl_ack : std_logic;
  signal mem_ctrl_busy : std_logic;
  signal mem_ctrl_dout : std_logic_vector(31 downto 0);
  signal mem_ctrl_req : std_logic;
  signal mem_ctrl_stream_req : std_logic;
  signal mem_ctrl_rnw : std_logic;
  signal mem_ctrl_din : std_logic_vector(31 downto 0);

  -- Clock buffer logic
  signal main_clk_out : std_logic;
  signal bdb_clk_out : std_logic;
  signal bdb_clk_en : std_logic;
  signal var_assert_break : std_logic;
  signal dram_clk_wait : std_logic;
  signal sys_clk_halted : std_logic;

begin

  -- Convert integer variable select from controller into one-hot bus
  gen_var_sel : for i in 0 to NUMVAR-1 generate
    constant this_int : unsigned(31 downto 0) := TO_UNSIGNED(i,32);
  begin
    var_sel(i) <= '1' when ctrl_var_sel_int(SELBITS-1 downto 0) =
                           STD_LOGIC_VECTOR(this_int(SELBITS-1 downto 0))
                  else '0';
  end generate gen_var_sel;

  -- Generate the write enable signal outputs
  gen_var_we_src : for i in 0 to NUMVAR-1 generate
  begin
    vars_value_we_out(i) <= var_sel(i) and ctrl_var_value_we;
    vars_src_we_out(i) <= var_sel(i) and ctrl_var_src_we;
    vars_thresh_we_out(i) <= var_sel(i) and ctrl_var_thresh_we;
    vars_assert_mode_we_out(i) <= var_sel(i) and ctrl_var_assert_mode_we;
  end generate gen_var_we_src;

  -- Tri-state bus to assign selected variable value to ctrl_var_data_in
  sel_var : for i in 0 to NUMVAR-1 generate
  begin
    process(var_sel, vars_data_in)
    begin
      if var_sel(i) = '1' then
        ctrl_var_data_in <= vars_data_in(((i+1)*W)-1 downto i*W);
      else
        ctrl_var_data_in <= (others => 'Z');
      end if;
    end process;
  end generate sel_var;
```

```
-- Count gated-clock cycles (currently for the 'runfor' user command)
bdb_clk_counter : process(bdb_clk_out, bdb_cycle_reset)
begin
  if (bdb_cycle_reset = '1') then
      bdb_cycle_count <= (others => '0');
  elsif (bdb_clk_out'EVENT and (bdb_clk_out = '1')) then
      bdb_cycle_count <= bdb_cycle_count + 1;
  end if;
end process bdb_clk_counter;

-- Derive the logic for the clock control signals
var_assert_break <= '0' when (vars_assert_break_in = var_sel_zero) else '1';
sys_clk_halted <= ctrl_clk_halt or var_assert_break;
bdb_clk_en <= not (sys_clk_halted or dram_clk_wait);

-- Connect status bits to the bdb_status_out register
bdb_status_out <= (31 downto 10 => '0')  -- [31:10]
                  & ctrl_clk_halt        -- 9
                  & var_assert_break     -- 8
                  & bdb_status_ctrl_out; -- [7:0]

-- Instantiate the DDR controller
bdb_core_ddr_ctrl_inst : bdb_core_ddr_ctrl
  generic map
  (
    NUMVAR => NUMVAR,
    W => W
  )
  port map
  (
    ddr_cmd_addr => mem_cmd_addr,
    ddr_wr_din => mem_wr_din(255 downto 0),
    ddr_wr_be => mem_wr_be(31 downto 0),
    ddr_cmd_rnw => mem_cmd_rnw,
    ddr_cmd_tag => mem_cmd_tag,
    ddr_cmd_valid => mem_cmd_valid,
    ddr_rd_ack => mem_rd_ack,
    ddr_cmd_ack => mem_cmd_ack,
    ddr_rd_dout => mem_rd_dout(255 downto 0),
    ddr_rd_tag => mem_rd_tag,
    ddr_rd_valid => mem_rd_valid,
    mem_ctrl_ack => mem_ctrl_ack,
    mem_ctrl_busy => mem_ctrl_busy,
    mem_ctrl_dout => mem_ctrl_dout,
    mem_ctrl_req => mem_ctrl_req,
    mem_ctrl_stream_req => mem_ctrl_stream_req,
    mem_ctrl_rnw => mem_ctrl_rnw,
    mem_ctrl_din => mem_ctrl_din,
    vars_data_in => vars_data_in,
    dram_clk_wait => dram_clk_wait,
    sys_clk_halted => sys_clk_halted,
    rst => rst,
    bdb_clk => bdb_clk_out,
    clk => main_clk_out
```

```
  );
mem_wr_din(287 downto 256) <= (others => '0');
mem_wr_be(35 downto 32) <= (others => '1');

-- Instantiate the core controller
bdb_core_ctrl_inst : bdb_core_ctrl
  port map
  (
    bdb_cmd_low => bdb_cmd_in(15 downto 0),
    bdb_cmd_high => bdb_cmd_in(31 downto 16),
    bdb_data_in => bdb_data_in,
    bdb_status_out => bdb_status_ctrl_out,
    bdb_data_out => bdb_data_out,
    bdb_cycle_reset => bdb_cycle_reset,
    bdb_user_capture => bdb_user_capture,
    bdb_user_restore => bdb_user_restore,
    icap_bus_out => icap_bus_out,
    icap_clk_out => icap_clk_out,
    var_assert_break => var_assert_break,
    ctrl_var_sel_int => ctrl_var_sel_int,
    ctrl_var_data_out => ctrl_var_data_out_32,
    ctrl_var_value_we => ctrl_var_value_we,
    ctrl_var_src_we => ctrl_var_src_we,
    ctrl_var_thresh_we => ctrl_var_thresh_we,
    ctrl_var_assert_mode_we => ctrl_var_assert_mode_we,
    ctrl_clk_halt => ctrl_clk_halt,
    ctrl_var_data_in => ctrl_var_data_in_32,
    bdb_cycle_count => STD_LOGIC_VECTOR(bdb_cycle_count),
    icap_bus_in => icap_bus_in,
    mem_ctrl_ack => mem_ctrl_ack,
    mem_ctrl_dout => mem_ctrl_dout,
    mem_ctrl_req => mem_ctrl_req,
    mem_ctrl_stream_req => mem_ctrl_stream_req,
    mem_ctrl_rnw => mem_ctrl_rnw,
    mem_ctrl_din => mem_ctrl_din,
    ce => '1',
    clk => main_clk_out,
    reset => rst
  );
var_data_out <= ctrl_var_data_out_32(W-1 downto 0);
ctrl_var_data_in_32 <= ctrl_var_data_in when (W = 32) else
                       (31 downto W => '0') & ctrl_var_data_in;

-- Instantiate an ICAP interface for manual readback capture and restore
icap_inst : ICAP_VIRTEX2
  port map
  (
    BUSY => icap_bus_in(8),
    O(0) => icap_bus_in(7),
    O(1) => icap_bus_in(6),
    O(2) => icap_bus_in(5),
    O(3) => icap_bus_in(4),
    O(4) => icap_bus_in(3),
    O(5) => icap_bus_in(2),
```

```
    O(6) => icap_bus_in(1),
    O(7) => icap_bus_in(0),
    CE => icap_bus_out(9),
    CLK => icap_clk_out,
    I(0) => icap_bus_out(7),
    I(1) => icap_bus_out(6),
    I(2) => icap_bus_out(5),
    I(3) => icap_bus_out(4),
    I(4) => icap_bus_out(3),
    I(5) => icap_bus_out(2),
    I(6) => icap_bus_out(1),
    I(7) => icap_bus_out(0),
    WRITE => icap_bus_out(8)
  );

-- Instantiate a readback capture block for register state snapshotting
capture_inst : CAPTURE_VIRTEX2
  port map
  (
    CAP => bdb_user_capture_delayed,
    CLK => main_clk_out
  );

-- Impose a delay on the CAP signal so all bus transactions are complete
cap_delay : process(bdb_user_capture, main_clk_out)
begin
  if (main_clk_out'EVENT and (main_clk_out = '1')) then
    if ((bdb_user_capture = '1') or
        (cap_delay_count /= TO_UNSIGNED(0,26))) then
      cap_delay_count <= cap_delay_count + 1;
    elsif (bdb_user_capture_delayed = '1') then
      cap_delay_count <= TO_UNSIGNED(0,26);
    end if;
  end if;
end process cap_delay;
bdb_user_capture_delayed <= '1'
                            when cap_delay_count = TO_UNSIGNED(50000000,26)
                            else '0';

-- Instantiate a global startup block for restoring configuration values
startup_inst : STARTUP_VIRTEX2
  port map
  (
    CLK => '0',
    GSR => bdb_user_restore_delayed,
    GTS => '0'
  );

-- Impose a delay on the GSR signal so all bus transactions are complete
gsr_delay : process(bdb_user_restore, main_clk_out)
begin
  if (main_clk_out'EVENT and (main_clk_out = '1')) then
    if ((bdb_user_restore = '1') or
        (gsr_delay_count /= TO_UNSIGNED(0,26))) then
```

```
        gsr_delay_count <= gsr_delay_count + 1;
      elsif (bdb_user_restore_delayed = '1') then
        gsr_delay_count <= TO_UNSIGNED(0,26);
      end if;
    end if;
  end process gsr_delay;
  bdb_user_restore_delayed <= '1'
                              when gsr_delay_count = TO_UNSIGNED(50000000,26)
                              else '0';

  -- Currently the global clock buffers are instantiated here, as the device
  -- does not allow a BUFG to feed the input of a BUFGCE.  Therefore, we need
  -- access to the direct DCM output in order to drive the gated clock.
  -- KBC: This could perhaps be integrated into the default system.mhs file
  --      by exposing the clock buffers and enable signal as an IP block...
  bufg_inst : BUFG  -- main clock buffer
    port map
    (
      I => dcm_clk,
      O => main_clk_out
    );
  bufgce_inst : BUFGCE  -- gated clock buffer
    port map
    (
      I => dcm_clk,
      CE => bdb_clk_en,
      O => bdb_clk_out
    );
  main_clk <= main_clk_out;
  bdb_clk <= bdb_clk_out;

end behavior;
```

A.3 Memory interface implementation

The following VHDL code represents the behavioral description of the memory controller interface presented in Sect. 4.3. These components are instantiated as part of the core debug controller. Again, the second entity, corresponding to the interface command state machine and named `bdb_core_ddr_cmd_ctrl`, is automatically generated by SF2VHD and not reproduced here. Please see Fig. 4.7 for a clearer understanding of its functionality.

A.3.1 `bdb_core_ddr_ctrl`

```
-- DDR user interface controller for BDB core control unit

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use IEEE.MATH_REAL.all;
```

```
entity bdb_core_ddr_ctrl is

  generic
  (
    NUMVAR : integer := 16;  -- Number of variables in system
    W : integer := 32  -- Bitwidth of variable data
  );

  port
  (
    -- DRAM user logic interface
    ddr_cmd_addr : out std_logic_vector(31 downto 0);
    ddr_wr_din : out std_logic_vector(255 downto 0);
    ddr_wr_be : out std_logic_vector(31 downto 0);
    ddr_cmd_rnw : out std_logic;
    ddr_cmd_tag : out std_logic_vector(31 downto 0);
    ddr_cmd_valid : out std_logic;
    ddr_rd_ack : out std_logic;
    ddr_cmd_ack : in  std_logic;
    ddr_rd_dout : in  std_logic_vector(255 downto 0);
    ddr_rd_tag : in  std_logic_vector(31 downto 0);
    ddr_rd_valid : in  std_logic;
    -- BDB core logic signals
    mem_ctrl_ack : out std_logic;
    mem_ctrl_busy : out std_logic;
    mem_ctrl_dout : out std_logic_vector(31 downto 0);
    mem_ctrl_req : in  std_logic;
    mem_ctrl_stream_req : in  std_logic;
    mem_ctrl_rnw : in  std_logic;
    mem_ctrl_din : in  std_logic_vector(31 downto 0);
    vars_data_in : in  std_logic_vector((NUMVAR*W)-1 downto 0);
    dram_clk_wait : out std_logic;
    sys_clk_halted : in  std_logic;
    -- Clocking
    rst : in  std_logic;
    bdb_clk : in  std_logic;
    clk : in  std_logic
  );

end bdb_core_ddr_ctrl;


architecture behavior of bdb_core_ddr_ctrl is

  function min(x, y: integer) return integer is
  begin
    if (x < y) then
      return x;
    else
      return y;
    end if;
  end function;
```

```
  component bdb_core_ddr_cmd_ctrl
    port ( signal mem_ctrl_ack : out std_logic;
           signal mem_ctrl_busy : out std_logic;
           signal mem_ctrl_dout : out std_logic_vector(31 downto 0);
           signal ddr_cmd_valid : out std_logic;
           signal ddr_cmd_rnw : out std_logic;
           signal ddr_cmd_addr : out std_logic_vector(31 downto 0);
           signal ddr_cmd_tag : out std_logic_vector(31 downto 0);
           signal ddr_wr_word : out std_logic_vector(31 downto 0);
           signal ddr_rd_ack : out std_logic;
           signal snap_go : out std_logic;
           signal snap_inc : out std_logic;
           signal stream_addr_load : out std_logic;
           signal write_single_word : out std_logic;
           signal mem_ctrl_req : in std_logic;
           signal mem_ctrl_stream_req : in std_logic;
           signal mem_ctrl_rnw : in std_logic;
           signal mem_ctrl_din : in std_logic_vector(31 downto 0);
           signal ddr_cmd_ack : in std_logic;
           signal ddr_rd_valid : in std_logic;
           signal ddr_rd_word : in std_logic_vector(31 downto 0);
           signal snap_last : in std_logic;
           signal stream_addr_in : in std_logic_vector(31 downto 0);
           signal sys_clk_halted : in std_logic;
           signal reset : in std_logic;
           signal ce : in std_logic;
           signal clk : in std_logic );
  end component;

  constant VARS_PER_CMD : integer := 256 / W;
  constant CMDS_PER_SNAP : integer := integer(ceil(real(NUMVAR)/
                                                   real(VARS_PER_CMD)));
  constant ROW_SEL_BITS : integer := integer(ceil(log2(real(CMDS_PER_SNAP))));

  signal snap_go : std_logic;
  signal snap_inc : std_logic;
  signal snap_last : std_logic;

  signal stream_addr : unsigned(31 downto 0);
  signal stream_addr_load : std_logic;

  signal ddr_cmd_addr_out : std_logic_vector(31 downto 0);
  signal ddr_rd_word : std_logic_vector(31 downto 0);
  signal ddr_wr_word : std_logic_vector(31 downto 0);
  signal ddr_wr_din_sys : std_logic_vector(255 downto 0);
  signal write_single_word : std_logic;
  signal ddr_cmd_ack_D : std_logic;
  signal ddr_rd_valid_D : std_logic;
  signal mem_ctrl_busy_out : std_logic;

begin

  -- If all data fit in one DDR command, connect directly and pad if needed
  single_ddr_cmd : if (CMDS_PER_SNAP = 1) generate
```

```
  constant PAD_BITS : integer := 256 - NUMVAR*W;
begin
  ddr_wr_din_sys <= vars_data_in when (PAD_BITS = 0)
                    else (255 downto (256-PAD_BITS) => '0') & vars_data_in;
  snap_last <= '1';  -- Always on last row
end generate;

-- If multiple DDR commands are needed per snapshot, define multiple rows of
--  data inputs which are cycled through in sequence
multi_ddr_cmd : if (CMDS_PER_SNAP > 1) generate
  type row_array_t is array (CMDS_PER_SNAP-1 downto 0)
                   of std_logic_vector(255 downto 0);
  signal din_row_array : row_array_t;
  signal din_row_sel : unsigned(ROW_SEL_BITS-1 downto 0);
  constant din_row_max : unsigned(ROW_SEL_BITS-1 downto 0)
                       := TO_UNSIGNED(CMDS_PER_SNAP-1, ROW_SEL_BITS);
begin
  -- Use a counter to cycle through the rows for each snapshot
  multi_ddr_cmd_sel : process(clk)
  begin
    if (clk'EVENT and (clk = '1')) then
      if (snap_go = '0') then
        -- Reset counter between snapshots
        din_row_sel <= TO_UNSIGNED(0, ROW_SEL_BITS);
      elsif ((snap_go = '1') and (snap_inc = '1')) then
        -- Increment the counter during snapshot when needed
        din_row_sel <= din_row_sel + 1;
      end if;
    end if;
  end process;
  -- Set last signal high when we're on the last row
  snap_last <= '1' when (din_row_sel = din_row_max) else '0';
  -- Generate connections for each data row
  multi_ddr_cmd_rows : for i in 1 to CMDS_PER_SNAP generate
    constant VARS_LEFT : integer := NUMVAR - (i-1)*VARS_PER_CMD;
    constant VARS_HERE : integer := min(VARS_LEFT, VARS_PER_CMD);
    constant PAD_BITS : integer := 256 - VARS_HERE*W;
    constant VARBUS_LSB : integer := (NUMVAR - VARS_LEFT) * W;
    constant VARBUS_MSB : integer := VARBUS_LSB + (VARS_HERE * W) - 1;
    constant this_row_sel : unsigned(ROW_SEL_BITS-1 downto 0)
                          := TO_UNSIGNED(i-1, ROW_SEL_BITS);
  begin
    -- Map the variable data ports for this row (pad if needed)
    din_row_array(i-1) <= vars_data_in(VARBUS_MSB downto VARBUS_LSB)
                          when (PAD_BITS = 0)
                          else (255 downto (256-PAD_BITS) => '0') &
                               vars_data_in(VARBUS_MSB downto VARBUS_LSB);
    -- Infer a tri-state for row selection
    -- KBC: Is this the only feasible architecture?  Shift registers would
    --      require 1 cycle latency, and muxes are not generate-friendly...
    ddr_wr_din_sys <= din_row_array(i-1) when (din_row_sel = this_row_sel)
                      else (others => 'Z');
  end generate;
end generate;
```

```
-- Generate the runtime streaming address as a loadable counter
-- KBC: XST is inferring this as an accumulator, but it's still functional
stream_addr_counter : process(bdb_clk, stream_addr_load)
begin
  if (stream_addr_load = '1') then
    stream_addr <= UNSIGNED(mem_ctrl_din);
  elsif (bdb_clk'EVENT and (bdb_clk = '1')) then
    stream_addr <= stream_addr + (32*CMDS_PER_SNAP);
  end if;
end process;

-- Derive the clock gating signal for memory accesses
-- Currently the DDR interface controller asserts mem_ctrl_busy whenever it
--  is accessing memory, and is analagous to pausing the system clock
dram_clk_wait <= mem_ctrl_busy_out;

-- Insert registers on DDR command signals (eliminates async. logic loops)
ddr_cmd_regs : process(clk)
begin
  if (clk = '1' and clk'EVENT) then
    ddr_cmd_ack_D <= ddr_cmd_ack;
    ddr_rd_valid_D <= ddr_rd_valid;
  end if;
end process;

-- Instantiate the DDR command controller
bdb_core_ddr_cmd_ctrl_inst : bdb_core_ddr_cmd_ctrl
  port map
  (
    mem_ctrl_ack => mem_ctrl_ack,
    mem_ctrl_busy => mem_ctrl_busy_out,
    mem_ctrl_dout => mem_ctrl_dout,
    ddr_cmd_valid => ddr_cmd_valid,
    ddr_cmd_rnw => ddr_cmd_rnw,
    ddr_cmd_addr => ddr_cmd_addr_out,
    ddr_cmd_tag => ddr_cmd_tag,
    ddr_wr_word => ddr_wr_word,
    ddr_rd_ack => ddr_rd_ack,
    snap_go => snap_go,
    snap_inc => snap_inc,
    stream_addr_load => stream_addr_load,
    write_single_word => write_single_word,
    mem_ctrl_req => mem_ctrl_req,
    mem_ctrl_stream_req => mem_ctrl_stream_req,
    mem_ctrl_rnw => mem_ctrl_rnw,
    mem_ctrl_din => mem_ctrl_din,
    ddr_cmd_ack => ddr_cmd_ack_D,
    ddr_rd_valid => ddr_rd_valid_D,
    ddr_rd_word => ddr_rd_word,
    snap_last => snap_last,
    stream_addr_in => STD_LOGIC_VECTOR(stream_addr),
    sys_clk_halted => sys_clk_halted,
    reset => rst,
```

```
    ce => '1',
    clk => clk
  );
ddr_cmd_addr <= ddr_cmd_addr_out(31 downto 5) & "00000";
mem_ctrl_busy <= mem_ctrl_busy_out;

-- Use a mux to route the correct word to the core controller on reads
ddr_rd_word_mux : process(ddr_cmd_addr_out, ddr_rd_dout)
begin
  case ddr_cmd_addr_out(4 downto 2) is
    when "000" =>
      ddr_rd_word <= ddr_rd_dout(31 downto 0);
    when "001" =>
      ddr_rd_word <= ddr_rd_dout(63 downto 32);
    when "010" =>
      ddr_rd_word <= ddr_rd_dout(95 downto 64);
    when "011" =>
      ddr_rd_word <= ddr_rd_dout(127 downto 96);
    when "100" =>
      ddr_rd_word <= ddr_rd_dout(159 downto 128);
    when "101" =>
      ddr_rd_word <= ddr_rd_dout(191 downto 160);
    when "110" =>
      ddr_rd_word <= ddr_rd_dout(223 downto 192);
    when "111" =>
      ddr_rd_word <= ddr_rd_dout(255 downto 224);
    when others =>
      NULL;
  end case;
end process;

-- Generate the DDR write data inputs and byte enable signal
-- For manual controller writes, route data word and set specific BE bits
-- For runtime streaming, connect system variable data and set all BE bits
ddr_wr_logic : for i in 0 to 7 generate
  constant WORD : std_logic_vector := STD_LOGIC_VECTOR(TO_UNSIGNED(i,3));
begin
  ddr_wr_word_sel : process(write_single_word, ddr_cmd_addr_out,
                            ddr_wr_word, ddr_wr_din_sys)
  begin
    if (write_single_word = '1') then
      if (WORD = ddr_cmd_addr_out(4 downto 2)) then
        ddr_wr_din(32*i+31 downto 32*i) <= ddr_wr_word;
        ddr_wr_be(4*i+3 downto 4*i) <= "1111";
      else
        ddr_wr_din(32*i+31 downto 32*i) <= (others => '0');
        ddr_wr_be(4*i+3 downto 4*i) <= "0000";
      end if;
    else
      ddr_wr_din(32*i+31 downto 32*i) <= ddr_wr_din_sys(32*i+31 downto 32*i);
      ddr_wr_be(4*i+3 downto 4*i) <= "1111";
    end if;
  end process;
end generate;
```

```
end behavior;
```

Appendix B

Software implementation data

This chapter contains the software implementation files for the `bdb` process designed on BORPH as well as the unique verification routines available in the Matlab design environment.

B.1 BORPH-based `bdb` process

The `bdb` process (presented in Sect. 5.1) was written in standard C and designed for BORPH via the use of software-accessible registers. Each of the source and header files which compose the `bdb` process are included here.

B.1.1 `main.h`

```
#ifndef _MAIN_H_
#define _MAIN_H_

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

#include "client_handler.h"
#include "varcmds.h"

#define STDOUTFILE "borph.out"
#define STDERRFILE "borph.err"

void cleanup(int status);

#endif /* _MAIN_H_ */
```

B.1.2 main.c

```
#include "main.h"

int sock = -1; /* server socket */
int clisock = -1; /* client connection socket */
int hwpid = -1;  /* process ID for the hardware child process */
int fd_cmd = -1;  /* bdb_cmd_in file descriptor */
int fd_din = -1;  /* bdb_data_in file descriptor */
int fd_stat = -1;  /* bdb_status_out file descriptor */
int fd_dout = -1;  /* bdb_data_out file descriptor */
int numvar = 0;  /* number of variables in design under test */

extern struct var_state *var_state_cache;


/*
** Print usage information and exit.
*/
void
usage(char *cmd)
{
    printf("\nUsage: %s [-p port] [-n numvar] executable.bof\n\n", cmd);
    exit(1);
}


/*
** Signal handler for unexpected child process termination.
**
** NOTE: There are some timing issues here, such that this handler can be
**       triggered during parent termination (i.e. the child process
**       dies first and triggers this signal).  Perhaps we need our own
**       termination handlers for the parent, so that the child is manually
**       killed after the handler is de-registered.
*/
void
hwproc_exit_handler()
{
    fprintf(stderr, "\nHardware process terminated unexpectedly: "
                    "check %s or %s\n\n",
            STDOUTFILE, STDERRFILE);
    cleanup(1);
}


/*
** Process entry point.
*/
int
main(int argc, char **argv)
{
    int i, result;
```

```
short port = 2007;
struct sigaction sa, sa_old;
sigset_t sigs;
int fd_bof;
int chldin, chldout, chlderr;
struct sockaddr_in saddr;
char *bofpath;

/* Process command line arguments */
if (argc < 2) usage(argv[0]);
for (i = 1; i < argc; i++) {
    char *cmd = argv[0];
    char *arg = argv[i];
    if (arg[0] != '-') { /* end of dashed options */
        if (i+1 != argc)
            printf("WARNING: Additional command line arguments ignored\n");
        bofpath = arg;
        break;  /* exits argument loop */
    }
    switch (arg[1]) {
    case 'p': /* port number */
        if (argc <= ++i) usage(cmd);
        if (sscanf(argv[i],"%hd",&port) < 1) usage(cmd);
        break;
    case 'n':
        if (argc <= ++i) usage(cmd);
        if (sscanf(argv[i],"%d",&numvar) < 1) usage(cmd);
        break;
    default:
        usage(cmd);
    }
}

/*
** Launch and connect to hardware child process to be debugged
*/

/* Check for valid access to the BOF file */
fd_bof = open(bofpath, O_RDONLY);
if (fd_bof < 0) {
    perror("Failed to open BOF file");
    cleanup(-1);
}
close(fd_bof);

/* KBC: Eventually it may be best to add an element to the BOF file header
        which specifies the number of variables in the design.  That check
        would go here once it's implemented. */

/* Make sure number of variables has been specified before starting */
if (numvar <= 0) {
    fprintf(stderr, "Number of variables cannot automatically be "
                    "determined from the BOF file...\n"
                    "Please manually specify the variable count with the "
```

```
                    "-n switch to bdb.\n");
    cleanup(1);
}

/* Register signal handler for hardware child process termination */
result = sigemptyset(&sigs);
if (result == -1) {
    perror("Failed to initialize signal set");
    cleanup(-1);
}
sa.sa_handler = &hwproc_exit_handler;
sa.sa_mask = sigs;
sa.sa_flags = 0;
result = sigaction(SIGCHLD, &sa, &sa_old);
if (result == -1) {
    perror("Failed to register signal handler for child termination");
    cleanup(-1);
}

/* Fork the hardware process */
printf("\nLaunching BORPH executable: %s\n", bofpath);
hwpid = fork();
if (hwpid == -1) {
    perror("Failed to fork hardware process");
    cleanup(-1);
}
if (hwpid == 0) { /* child process instance */
    fflush(stdout);
    /* KBC: Currently changing stdin causes major BORPH slowdown */
    /* close(0); chldin = open("/dev/null", O_RDONLY); */
    chldin = 0;
    close(1); chldout = open(STDOUTFILE, O_WRONLY|O_CREAT|O_TRUNC, 00666);
    close(2); chlderr = open(STDERRFILE, O_WRONLY|O_CREAT|O_TRUNC, 00666);
    if ((chldin == -1) || (chldout == -1) || (chlderr == -1)) {
        perror("Failed to redirect I/O for hardware process");
        cleanup(-1);
    }
    result = execl(bofpath, bofpath, (char*)NULL);
    if (result == -1) {
        perror("Failed to launch hardware process");
        cleanup(-1);
    }
}
else { /* parent process instance */
    sleep(1); /* KBC: Is there a signal we can watch instead? */
    printf("Hardware process forked with PID %d\n", hwpid);
}

/* Initialize the interface to the hardware process via BORPH */
result = initialize_bdb();
if (result == -1) {
    fprintf(stderr, "BDB hardware interface initialization failed");
    cleanup(-1);
}
```

```
else
    printf("BDB hardware interface initialized\n");

/*
** Initialize the server (listening) socket
*/

/* Create the server socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
    perror("Failed to create socket");
    cleanup(-1);
}

/* Set up to listen on any local interface on the given port */
saddr.sin_family = AF_INET;
saddr.sin_port = htons(port);
saddr.sin_addr.s_addr = INADDR_ANY;

/* Bind to the local interface */
result = bind(sock, (struct sockaddr *)&saddr, sizeof(saddr));
if (result == -1) {
    perror("Failed to bind local interface");
    cleanup(-1);
}

/* Configure socket for listening */
result = listen(sock, 0); /* additional connections will be refused */
if (result == -1) {
    perror("Failed to set up socket for listening");
    cleanup(-1);
}

printf("\nListening for connections on port %d\n", port);

/*
** Begin the client connection loop
*/

/* Loop while listening for connections until terminated */
while (1) {
    result = accept_client_connection();
    if (result == -1) {
        fprintf(stderr,
                "Critical error while servicing client... aborting\n");
        cleanup(-1);
    }
    if (result)
        printf("Client connection terminated unepectedly\n");
    else
        printf("Client connection complete\n");
}

/* Not reached */
```

```
    return 0;
}


/*
** Exit the process with the given status value, cleaning up any loose ends as
** needed.
*/
void
cleanup(int status)
{
    int result;

    /* KBC: Do we need to unregister the signal handler here first? */
    /* Terminate the hardware process, if running */
    if (hwpid != -1) {
        result = kill(hwpid, SIGTERM);
        if (result) perror("Failed to terminate hardware process");
    }

    /* Close the server socket, if open */
    if (sock != -1) {
        result = close(sock);
        if (result) perror("Failed to close server socket");
    }

    /* Close the client socket, if open */
    if (clisock != -1) {
        result = close(clisock);
        if (result) perror("Failed to close client socket");
    }

    /* Close the bdb_cmd_in file, if open */
    if (fd_cmd != -1) {
        result = close(fd_cmd);
        if (result) perror("Failed to close bdb_cmd_in");
    }

    /* Close the bdb_data_in file, if open */
    if (fd_din != -1) {
        result = close(fd_din);
        if (result) perror("Failed to close bdb_data_in");
    }

    /* Close the bdb_status_out file, if open */
    if (fd_stat != -1) {
        result = close(fd_stat);
        if (result) perror("Failed to close bdb_status_out");
    }

    /* Close the bdb_data_out file, if open */
    if (fd_dout != -1) {
        result = close(fd_dout);
        if (result) perror("Failed to close bdb_data_out");
```

```
    }

    /* Free the variable state cache */
    if (var_state_cache != NULL) free(var_state_cache);

    /* Exit */
    exit(status);
}
```

B.1.3 client_handler.h

```
#ifndef _CLIENT_HANDLER_H_
#define _CLIENT_HANDLER_H_

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <assert.h>

#include "varcmds.h"

/* Special-purpose client/service commands */
#define SCMD_GETSTATUS 0x1
#define SCMD_GETVARSTATE 0x2
#define SCMD_GETVALIDVARS 0x3
#define SCMD_GETFORCEDVARS 0x4
#define SCMD_GETASSERTS 0x5
#define SCMD_GETCOMPARES 0x6
#define SCMD_GETALLVALS 0x7

int accept_client_connection();

#endif /* _CLIENT_HANDLER_H_ */
```

B.1.4 client_handler.c

```
#include "client_handler.h"

extern int sock, clisock;


/*
** Accept incoming client connections and call the corresponding handler
** function for each client request.
**
** Return values: 0 - Successful completion
**                1 - Session interrupted (socket read/write error, etc.)
**               -1 - Critical failure (required local operation failed)
**
*/
```

```
int
accept_client_connection()
{
    int result;
    struct sockaddr_in cliaddr;
    unsigned int addrsize;
    long int buf, hw_cmd, sub_cmd, varid, data, addr, status, count;
    long int *entries, *membuf = NULL, *icapbuf = NULL;
    unsigned long int membuflen = 0, icapbuflen = 0;

    assert(sizeof(buf) == 4);

    /* Accept incoming connection */
    addrsize = sizeof(cliaddr);
    clisock = accept(sock, (struct sockaddr *)&cliaddr, &addrsize);
    if (clisock == -1) {
        perror("Failed to accept client connection");
        return -1;
    }

    printf("\nReceived client connection from %s\n",
           inet_ntoa(cliaddr.sin_addr));

    /* Receive commands over the socket until the session ends */
    /* NOTE: All data is sent in network byte order */
    while (1) {

        /* Receive the 4-byte hardware command ID */
        errno = 0;
        result = recv(clisock, &buf, 4, 0);
        if (errno) {
            perror("Failed to receive command header");
            return 1;
        }
        else if (result != 4) {
            /* A non-error code but no data read means a closed socket */
            printf(" Socket closed on remote end\n");
            return 1;
        }
        hw_cmd = ntohl(buf);

        /* Take action as requested */
        printf("Received client command code 0x%lX\n", hw_cmd);
        switch (hw_cmd) {

        /* Zero is reserved for specialized operations which don't
           translate directly to hardware commands */
        case 0:
            /* Receive the 4-byte sub-command code */
            result = recv(clisock, &buf, 4, 0);
            if (result != 4) {
                perror("Failed to read sub-command code");
                return 1;
            }
```

```
sub_cmd = ntohl(buf);

/* Choose a sub-command to perform */
switch(sub_cmd) {

case SCMD_GETSTATUS: /* return status register contents */
    /* Read the status register */
    result = get_status(&status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to check status\n");
        return -1;
    }
    /* Send the result back to the client */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case SCMD_GETVARSTATE: /* return state of desired variable */
    /* Receive the 4-byte variable ID to query */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read variable ID value");
        return 1;
    }
    varid = ntohl(buf);
    /* Fetch the cache state of the given variable */
    result = get_var_state(varid, &count, &entries);
    if (result == -1) {
        fprintf(stderr, "Error looking up variable state\n");
        return -1;
    }
    /* Send the length of the list in words (errors are <0) */
    result = send(clisock, &count, 4, 0);
    if (result != 4) {
        perror("Failed to send variable state length");
        return 1;
    }
    /* Send the variable state contents */
    if (count > 0) {
        result = send(clisock, entries, count*4, 0);
        if (result != count*4) {
            perror("Failed to send variable state");
            return 1;
        }
    }
    break;

case SCMD_GETVALIDVARS: /* return valid variable entries */
    /* Fetch an array of all valid variable entries */
    result = get_valid_variables(&count, &entries);
    if (result == -1) {
```

```
            fprintf(stderr,
                    "Error attempting to get all valid variables\n");
            return -1;
        }
        /* Send the length of the list in words (errors are <0) */
        result = send(clisock, &count, 4, 0);
        if (result != 4) {
            perror("Failed to send valid variable count");
            return 1;
        }
        /* Send the array of valid variables */
        if (count > 0) {
            result = send(clisock, entries, count*4, 0);
            if (result != count*4) {
                perror("Failed to send valid variables");
                return 1;
            }
        }
        break;

    case SCMD_GETFORCEDVARS: /* return forced variables */
        /* Fetch an array of all forced variables */
        result = get_forced_variables(&count, &entries);
        if (result == -1) {
            fprintf(stderr,
                    "Error attempting to get all forced variables\n");
            return -1;
        }
        /* Send the length of the list in words (errors are <0) */
        result = send(clisock, &count, 4, 0);
        if (result != 4) {
            perror("Failed to send valid variable count");
            return 1;
        }
        /* Send the array of forced variables */
        if (count > 0) {
            result = send(clisock, entries, count*4, 0);
            if (result != count*4) {
                perror("Failed to send forced variables");
                return 1;
            }
        }
        break;

    case SCMD_GETASSERTS: /* return enabled assertions */
        /* Fetch an array of all enabled assertions */
        result = get_active_assertions(&count, &entries);
        if (result == -1) {
            fprintf(stderr, "Error attempting to check assertions\n");
            return -1;
        }
        /* Send the length of the list in words (errors are <0) */
        result = send(clisock, &count, 4, 0);
        if (result != 4) {
```

```
            perror("Failed to send valid variable count");
            return 1;
        }
        /* Send the array of assertions */
        if (count > 0) {
            result = send(clisock, entries, count*4, 0);
            if (result != count*4) {
                perror("Failed to send active assertions");
                return 1;
            }
        }
        break;

    case SCMD_GETCOMPARES: /* return active assertion comparisons */
        /* Fetch an array of all active comparisons */
        result = get_active_comparisons(&count, &entries);
        if (result == -1) {
            fprintf(stderr, "Error attempting to check comparisons\n");
            return -1;
        }
        /* Send the length of the list in words (errors are <0) */
        result = send(clisock, &count, 4, 0);
        if (result != 4) {
            perror("Failed to send valid variable count");
            return 1;
        }
        /* Send the array of comparisons */
        if (count > 0) {
            result = send(clisock, entries, count*4, 0);
            if (result != count*4) {
                perror("Failed to send comparison results");
                return 1;
            }
        }
        break;

    case SCMD_GETALLVALS: /* return all variable values */
        /* Receive the 4-byte max variable ID */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read variable ID value");
            return 1;
        }
        varid = ntohl(buf);
        /* Fetch an array of all variable values */
        result = get_all_values(varid, &count, &entries);
        if (result == -1) {
            fprintf(stderr,
                    "Error attempting to get all variable values\n");
            return -1;
        }
        /* Send the length of the list in words (errors are <0) */
        result = send(clisock, &count, 4, 0);
        if (result != 4) {
```

```
            perror("Failed to send valid variable count");
            return 1;
        }
        /* Send the array of variable values */
        if (count > 0) {
            result = send(clisock, entries, count*4, 0);
            if (result != count*4) {
                perror("Failed to send all valid variables");
                return 1;
            }
        }
        break;

    default: /* unknown sub-command */
        printf("Invalid sub-command: 0x%08lX\n", sub_cmd);
        close(clisock);
        clisock = -1;
        return 1;
    }
    break;

case CMD_READVAR:
    /* Receive the 4-byte variable ID to read */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read variable ID value");
        return 1;
    }
    varid = ntohl(buf);
    /* Read the variable value from the hardware */
    result = readvar(varid, &data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to read variable\n");
        return -1;
    }
    /* Respond with the hardware status and the value (if valid) */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    if (status >= 0) {
        result = send(clisock, &data, 4, 0);
        if (result != 4) {
            perror("Failed to send variable value");
            return 1;
        }
    }
    break;

case CMD_FORCEVAR:
    /* Receive the 4-byte variable ID to force */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
```

```
            perror("Failed to read variable ID value");
            return 1;
        }
        varid = ntohl(buf);
        /* Receive the 4-byte data value to write */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read variable value");
            return 1;
        }
        data = ntohl(buf);
        /* Force the variable value to the hardware */
        result = forcevar(varid, data, &status);
        if (result == -1) {
            fprintf(stderr, "Error attempting to write variable\n");
            return -1;
        }
        /* Respond with the hardware status */
        result = send(clisock, &status, 4, 0);
        if (result != 4) {
            perror("Failed to send response header");
            return 1;
        }
        break;

    case CMD_RELEASEVAR:
        /* Receive the 4-byte variable ID to release */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read variable ID value");
            return 1;
        }
        varid = ntohl(buf);
        /* Release the variable on the hardware */
        result = releasevar(varid, &status);
        if (result == -1) {
            fprintf(stderr, "Error attempting to release variable\n");
            return -1;
        }
        /* Respond with the hardware status */
        result = send(clisock, &status, 4, 0);
        if (result != 4) {
            perror("Failed to send response header");
            return 1;
        }
        break;

    case CMD_HALT:
        /* Send the halt command to hardware */
        result = halt(&status);
        if (result == -1) {
            fprintf(stderr, "Error attempting to halt clock\n");
            return -1;
        }
```

```
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header\n");
        return 1;
    }
    break;

case CMD_RUNFOR:
    /* Receive the 4-byte cycle count */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read cycle count");
        return 1;
    }
    data = ntohl(buf);
    /* Send the runfor command to hardware */
    result = runfor(data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to run design clock");
        return -1;
    }/* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_RESUME:
    /* Send the continue command to hardware */
    result = resume(&status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to resume clock\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header\n");
        return 1;
    }
    break;

case CMD_SETTHRESH:
    /* Receive the 4-byte variable ID to modify */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read variable ID value");
        return 1;
    }
    varid = ntohl(buf);
    /* Receive the 4-byte threshold value to write */
    result = recv(clisock, &buf, 4, 0);
```

```
    if (result != 4) {
        perror("Failed to read threshold value");
        return 1;
    }
    data = ntohl(buf);
    /* Set the threshold value on the hardware */
    result = setthresh(varid, data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to set threshold\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_SETCONDS:
    /* Receive the 4-byte variable ID to modify */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read variable ID value");
        return 1;
    }
    varid = ntohl(buf);
    /* Receive the 4-byte condition mask to write */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read condition mask value");
        return 1;
    }
    data = ntohl(buf);
    /* Set the condition mask on the hardware */
    result = setconds(varid, data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to set condition mask\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_READMEM:
    /* Receive the 4-byte base address */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read memory base address");
        return 1;
```

```
        }
        addr = ntohl(buf);
        /* Receive the 4-byte word count */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read word count");
            return 1;
        }
        count = ntohl(buf);
        /* Check the size of the local memory data buffer */
        if (count > membuflen) {
            long int *newbuf = (long int *) malloc(count*4);
            if (newbuf == NULL) {
                perror("Failed to allocate buffer memory");
                cleanup(-1);
            }
            if (membuf != NULL) free(membuf);
            membuflen = count;
            membuf = newbuf;
        }
        /* Read from the hardware memory */
        result = readmem(addr, membuf, count, &status);
        if (result == -1) {
            fprintf(stderr, "Error attempting to read memory\n");
            return -1;
        }
        /* Respond with the hardware status and values (if valid) */
        result = send(clisock, &status, 4, 0);
        if (result != 4) {
            perror("Failed to send response header");
            return 1;
        }
        if (status >= 0) {
            result = send(clisock, membuf, 4*count, 0);
            if (result != 4*count) {
                perror("Failed to send memory contents");
                return 1;
            }
        }
        break;

    case CMD_WRITEMEM:
        /* Receive the 4-byte base address */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read memory base address");
            return 1;
        }
        addr = ntohl(buf);
        /* Receive the 4-byte word count */
        result = recv(clisock, &buf, 4, 0);
        if (result != 4) {
            perror("Failed to read word count");
            return 1;
```

```
    }
    count = ntohl(buf);
    /* Check the size of the local memory data buffer */
    if (count > membuflen) {
        long int *newbuf = (long int *) malloc(count*4);
        if (newbuf == NULL) {
            perror("Failed to allocate buffer memory");
            cleanup(-1);
        }
        if (membuf != NULL) free(membuf);
        membuflen = count;
        membuf = newbuf;
    }
    /* Receive the memory values to be written */
    result = recv(clisock, membuf, 4*count, MSG_WAITALL);
    if (result != 4*count) {
        perror("Failed to read memory values");
        return 1;
    }
    /* Write to the hardware memory */
    result = writemem(addr, membuf, count, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to write memory\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_GETSTREAMADDR:
    /* Fetch the current stream address from the hardware */
    result = getstreamaddr(&data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to fetch stream address\n");
        return -1;
    }
    /* Respond with the hardware status and pointer value (if valid) */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    if (status >= 0) {
        result = send(clisock, &data, 4, 0);
        if (result != 4) {
            perror("Failed to send stream pointer value");
            return 1;
        }
    }
    break;
```

```
case CMD_SETSTREAMADDR:
    /* Receive the 4-byte stream address to be set */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to receive stream pointer value");
        return 1;
    }
    data = ntohl(buf);
    /* Set the current stream address on the hardware */
    result = setstreamaddr(data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to set stream address\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_GETCYCLECOUNT:
    /* Fetch the current stream address from the hardware */
    result = getcyclecount(&data, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to fetch cycle count\n");
        return -1;
    }
    /* Respond with the hardware status and count value (if valid) */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    if (status >= 0) {
        result = send(clisock, &data, 4, 0);
        if (result != 4) {
            perror("Failed to send stream pointer value");
            return 1;
        }
    }
    break;

case CMD_RESETCYCLECOUNT:
    /* Send the resetcyclecount command to hardware */
    result = resetcyclecount(&status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to reset cycle count\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
```

```
    if (result != 4) {
        perror("Failed to send response header\n");
        return 1;
    }
    break;

case CMD_CAPTURE:
    /* Send the capture command to hardware */
    result = capture(&status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to trigger capture\n");
        return -1;
    }
        /* Respond with the hardware status */
        result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header\n");
        return 1;
    }
    break;

case CMD_RESTORE:
    /* Send the restore command to hardware */
    result = restore(&status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to trigger restore\n");
        return -1;
    }
        /* Respond with the hardware status */
        result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header\n");
        return 1;
    }
    break;

case CMD_ICAPWRITE:
    /* Receive the 4-byte word count */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read word count");
        return 1;
    }
    count = ntohl(buf);
    /* Check the size of the local ICAP data buffer */
    if (count > icapbuflen) {
        long int *newbuf = (long int *) malloc(count*4);
        if (newbuf == NULL) {
            perror("Failed to allocate buffer memory");
            cleanup(-1);
        }
        if (icapbuf != NULL) free(icapbuf);
        icapbuflen = count;
        icapbuf = newbuf;
```

```
    }
    /* Receive the memory values to be written */
    result = recv(clisock, icapbuf, 4*count, MSG_WAITALL);
    printf("Received %d bytes (%ld intended)\n",result,4*count);
    if (result != 4*count) {
        perror("Failed to receive ICAP data values");
        return 1;
    }
    /* Write the data sequentially on ICAP bus */
    result = icapwritewords(icapbuf, count, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to drive ICAP bus\n");
        return -1;
    }
    /* Respond with the hardware status */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    break;

case CMD_ICAPREAD:
    /* Receive the 4-byte word count */
    result = recv(clisock, &buf, 4, 0);
    if (result != 4) {
        perror("Failed to read word count");
        return 1;
    }
    count = ntohl(buf);
    /* Check the size of the local ICAP data buffer */
    if (count > icapbuflen) {
        long int *newbuf = (long int *) malloc(count*4);
        if (newbuf == NULL) {
            perror("Failed to allocate buffer memory");
            cleanup(-1);
        }
        if (icapbuf != NULL) free(icapbuf);
        icapbuflen = count;
        icapbuf = newbuf;
    }
    /* Read the data sequentially on ICAP bus */
    result = icapreadwords(icapbuf, count, &status);
    if (result == -1) {
        fprintf(stderr, "Error attempting to read ICAP bus\n");
        return -1;
    }
    /* Respond with the hardware status and values (if valid) */
    result = send(clisock, &status, 4, 0);
    if (result != 4) {
        perror("Failed to send response header");
        return 1;
    }
    if (status >= 0) {
```

```
                result = send(clisock, icapbuf, 4*count, 0);
                if (result != 4*count) {
                    perror("Failed to send ICAP data");
                    return 1;
                }
            }
            break;

        case -1: /* special case for client to indicate it is finished */
            printf(" Client requested to end session\n");
            close(clisock);
            clisock = -1;
            return 0;

        default: /* unknown command ID */
            printf("Invalid hardware command: 0x%08lX\n", hw_cmd);
            close(clisock);
            clisock = -1;
            return 1;
        }

    }
}
```

B.1.5 varcmds.h

```
#ifndef _VARCMDS_H_
#define _VARCMDS_H_

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <assert.h>
#include <errno.h>
#include <sys/time.h>

#include "main.h"

/* BORPH I/O modes */
#define IOM_ASCII 0
#define IOM_BINARY 1

/* BDB hardware command codes */
#define CMD_READVAR 0x1
#define CMD_FORCEVAR 0x2
#define CMD_RELEASEVAR 0x3
#define CMD_HALT 0x4
```

```
#define CMD_RUNFOR 0x5
#define CMD_RESUME 0x6
#define CMD_SETTHRESH 0x7
#define CMD_SETCONDS 0x8
#define CMD_READWORD 0x9
#define CMD_READMEM CMD_READWORD
#define CMD_WRITEWORD 0xA
#define CMD_WRITEMEM CMD_WRITEWORD
#define CMD_GETSTREAMADDR 0xB
#define CMD_SETSTREAMADDR 0xC
#define CMD_GETCYCLECOUNT 0xD
#define CMD_RESETCYCLECOUNT 0xE
#define CMD_CAPTURE 0xF
#define CMD_RESTORE 0x10
#define CMD_ICAPWRITE 0x11
#define CMD_ICAPREAD 0x12

/* BDB hardware status codes */
#define STAT_CTRL_MASK 0xFF
#define STAT_IDLE 0x0
#define STAT_BUSY 0x1
#define STAT_DONE 0x2
#define STAT_BREAK 0x100
#define STAT_HALT 0x200

/* BDB service error codes */
#define ERR_VARIDRANGE -1

/* This global determines how many additional cache slots are added when a
   new variable ID is requested and a resize operation is necessary */
#define CACHE_RESIZE_INC 128

/* This global determines how many extra entries are added to each static
   result buffer for all the "get_*" sub-commands during resizing */
#define RESULTBUF_RESIZE_INC 16

struct var_state {
    short unsigned valid;
    short unsigned forced;
    long int force_val;
    long int threshold;
    long int cond_mask;
};

int initialize_bdb();

int get_status(long int *status);
int get_var_state(long int varid, long int *count, long int **results);
int get_valid_variables(long int *count, long int **results);
int get_forced_variables(long int *count, long int **results);
int get_active_assertions(long int *count, long int **results);
int get_active_comparisons(long int *count, long int **results);
int get_all_values(long int varid, long int *count, long int **results);
```

```
int readvar(long int varid, long int *val, long int *status);
int forcevar(long int varid, long int val, long int *status);
int releasevar(long int varid, long int *status);
int halt(long int *status);
int runfor(long int val, long int *status);
int resume(long int *status);
int setthresh(long int varid, long int val, long int *status);
int setconds(long int varid, long int val, long int *status);
int readmem(long int addr, long int *vals, long int len, long int *status);
int writemem(long int addr, long int *vals, long int len, long int *status);
int getstreamaddr(long int *val, long int *status);
int setstreamaddr(long int val, long int *status);
int getcyclecount(long int *val, long int *status);
int resetcyclecount(long int *status);
int capture(long int *status);
int restore(long int *status);
int icapwritewords(long int *vals, long int len, long int *status);
int icapreadwords(long int *vals, long int len, long int *status);

#endif /* _VARCMDS_H_ */
```

B.1.6 varcmds.c

```
#include "varcmds.h"

#define DEBUG_OUTPUT 0

extern int fd_cmd, fd_din, fd_stat, fd_dout;
extern int hwpid;
extern int numvar;

/* The following data structure holds the cached state of all modified
   variables in hardware.  It is addressed by variable ID and expands
   dynamically based on the highest ID that has been cached. */
struct var_state *var_state_cache = NULL;

/* The highest accessed variable ID is needed to know the valid cache size */
long int max_varid = -1;

/* The following globals keep track of overall system quantities */
long int num_valid_vars = 0;
long int num_forced_vars = 0;
long int num_active_assertions = 0;

/* Time structures for simple performance monitoring */
struct timeval t1, t2;
#define tdiff(x,y) y.tv_usec-x.tv_usec+1000000*(y.tv_sec-x.tv_sec)


/*
** Set the BORPH I/O mode for all ioregs
*/
```

```
static
int
set_iomode(int mode)
{
    int fd, result;
    char modestr[2], path[MAXPATHLEN];

    assert((mode >= 0) && (mode < 10));

    sprintf(path, "/proc/%d/hw/ioreg_mode", hwpid);
    sprintf(modestr, "%d", mode);

    fd = open(path, O_WRONLY);
    if (fd == -1) { perror("Failed to open ioreg_mode"); return -1; }
    result = write(fd, modestr, 1);
    if (result != 1) {
        perror("Failed to write to ioreg_mode");
        close(fd);
        return -1;
    }
    close(fd);
    return 0;
}


/*
** Initialize the BORPH registers used for the BDB command interface
*/
int
initialize_bdb()
{
    int result;
    char path[MAXPATHLEN];

    /* Make sure we're using the correct word size */
    assert(sizeof(long int) == 4);

    /* Set BORPH to use binary I/O mode */
    result = set_iomode(IOM_BINARY);

    /* Open the connection to bdb_cmd_in */
    sprintf(path, "/proc/%d/hw/ioreg/Debug_Controller_bdb_cmd_in", hwpid);
    fd_cmd = open(path, O_RDWR|O_NONBLOCK);
    if (fd_cmd == -1) {
        perror("Failed to open file for bdb_cmd_in");
        return -1;
    }

    /* Open the connection to bdb_data_in */
    sprintf(path, "/proc/%d/hw/ioreg/Debug_Controller_bdb_data_in", hwpid);
    fd_din = open(path, O_RDWR|O_NONBLOCK);
    if (fd_din == -1) {
        perror("Failed to open file for bdb_data_in");
        return -1;
```

```
    }

    /* Open the connection to bdb_status_out */
    sprintf(path, "/proc/%d/hw/ioreg/Debug_Controller_bdb_status_out", hwpid);
    fd_stat = open(path, O_RDONLY);
    if (fd_stat == -1) {
        perror("Failed to open file for bdb_status_out");
        return -1;
    }

    /* Open the connection to bdb_data_out */
    sprintf(path, "/proc/%d/hw/ioreg/Debug_Controller_bdb_data_out", hwpid);
    fd_dout = open(path, O_RDONLY);
    if (fd_dout == -1) {
        perror("Failed to open file for bdb_data_out");
        return -1;
    }

    return 0;
}


/*
** Read the 32-bit software register specified by its open file descriptor and
** write the result into the specified location.  If name is NULL, no infor-
** mational message is printed.
**
** Return value is 0 for success or -1 for an error;
*/
static
int
read_reg(int fd, long int *data, char *name)
{
    int result;

    result = lseek(fd, 0, SEEK_SET);
    if (result != 0) {
        if (name != NULL)
            printf("Failed to seek file for %s: %s\n", name, strerror(errno));
        return -1;
    }
    errno = 0;
    result = read(fd, data, 4);
    if (result != 4) {
        if (name != NULL)
            printf("Failed to read from %s (read returned %d): %s\n",
                   name, result, strerror(errno));
        return -1;
    }
    if ((name != NULL) && DEBUG_OUTPUT)
        printf("  => Read %s: 0x%08lX\n", name, *data);
    return 0;
}
```

```
/*
** Write the specified value into the 32-bit software register specified by its
** open file descriptor.  If name is NULL, no informational message is printed.
**
** Return value is 0 for success or -1 for an error;
*/
static
int
write_reg(int fd, long int data, char *name)
{
    int result;

    result = lseek(fd, 0, SEEK_SET);
    if (result != 0) {
        if (name != NULL)
            printf("Failed to seek file for %s: %s\n", name, strerror(errno));
        return -1;
    }
    errno = 0;
    result = write(fd, &data, 4);
    if (result != 4) {
        if (name != NULL)
            printf("Failed to write to %s (write returned %d): %s\n",
                   name, result, strerror(errno));
        return -1;
    }
    if ((name != NULL) && DEBUG_OUTPUT)
        printf("  => Wrote %s: 0x%08lX\n", name, data);
    return 0;
}


/*
** Wait for the hardware task to complete by checking the status regsiter.
**
** Return value: 0 - Hardware task completed (status ended up STAT_DONE)
**              -1 - critical error (failed to access register)
*/
static
int
wait_for_hardware()
{
    int result;
    long int status;

    /* Poll the status register as long as the hardware is busy */
    do {
        result = read_reg(fd_stat, &status, "bdb_status_out");
        if (result) return result;
    } while ((status & STAT_CTRL_MASK) == STAT_BUSY);

    return 0;
}
```

```
/*
** Clear the completed hardware command and make sure the controller has reset.
**
** Return value is the contents of bdb_status_out on success or -1 on error
*/
static
long int
end_command()
{
    int result;
    long int status;

    /* Clear the command */
    result = write_reg(fd_cmd, 0, "bdb_cmd_in");
    if (result) return result;

    /* Double-check that the controller has reset */
    result = read_reg(fd_stat, &status, "bdb_status_out");
    if (result) return result;
    if (status & STAT_CTRL_MASK) {
        printf("Hardware error: controller status stuck at %ld\n", status);
        return -1;
    }

    return status;
}


/*
** Check for the presence of the specified variable in the state cache.  The
** cache will also be resized if it currently is not large enough to hold the
** desired variable.
**
** Return values: 1 - variable is present in the state cache
**                0 - variable is not present in the state cache
**
** The function will automatically call cleanup() if calloc() fails.
**
** KBC: There is no checking here on the value of varid (or anywhere else in
**      caller functions).  That could lead to a very nasty memory explosion
**      if a malicious variable ID is passed, as the size of the cache is
**      adjusted to the highest variable ID requested.  This may be reason
**      enough to provide some way of passing the variable count to BDB.
**
** KBC: This function will also create "valid" cache entries for non-existent
**      variables, which will confuse the client tools when unknown variable
**      IDs are returned from the diagnostic sub-commands.
*/
static
int
check_var_state_cache(long int varid)
{
```

```
    /* Keep track of the number of allocated cache slots across calls */
    static unsigned cache_size = 0;

    unsigned new_size;
    struct var_state *new_cache;

    /* Update the maximum valid ID */
    if (varid > max_varid) {
        if (DEBUG_OUTPUT)
            printf("  => New maximum variable ID is %ld\n", varid);
        max_varid = varid;
    }

    /* If the cache is large enough, look up the variable and return */
    if (cache_size >= varid+1) {
        if (DEBUG_OUTPUT)
            printf("  => Cache large enough for variable %ld (valid: %hd)\n",
                   varid, var_state_cache[varid].valid);
        return var_state_cache[varid].valid;
    }

    /* Otherwise, allocate the new cache space */
    new_size = varid + 1 + CACHE_RESIZE_INC;
    new_cache = calloc(new_size, sizeof(struct var_state));
    if (new_cache == NULL) {
        perror("Failed to allocate memory for variable cache");
        cleanup(-1); /* don't bother continuing */
    }

    /* Copy the old cache values */
    memcpy(new_cache, var_state_cache, cache_size*sizeof(struct var_state));

    /* Free the old cache space and set the new cache values */
    if (DEBUG_OUTPUT) printf("  => Cache expanded from %d to %d entries\n",
                             cache_size, new_size);
    free(var_state_cache);
    var_state_cache = new_cache;
    cache_size = new_size;

    return 0;  /* Variable inherently not found if we just created space */
}


/*********************************
 * Begin diagnostic sub-commands *
 *********************************/


/*
** Read the status register and store it in the given location
*/
int
get_status(long int *status)
{
```

```
    int result;

    /* Read the status register into the provided address */
    result = read_reg(fd_stat, status, "bdb_status_out");
    printf(" Read contents of status register: 0x%08lX\n", *status);
    return result;
}


/*
** Look up the current state of the given variable in the cache and return it
** as an array pointer.  This function does not need to access the hardware.
** Errors are indicated with a return value of 1 and a (negative) code in count.
**
** The returned entry format is:
**   {long int valid, long int forced, long int force_val,
**    long int threshold, long int cond_mask}
*/
int
get_var_state(long int varid, long int *count, long int **results)
{
    static long int mem[5]; /* must be static to return its address */
    struct var_state *v;

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *count = ERR_VARIDRANGE;
        *results = NULL;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* Touch the cache to make sure an entry exists for this variable ID */
    check_var_state_cache(varid);

    /* Fill in the values for this variable */
    v = &var_state_cache[varid];
    mem[0] = (long int) v->valid;
    mem[1] = (long int) v->forced;
    mem[2] = v->force_val;
    mem[3] = v->threshold;
    mem[4] = v->cond_mask;
    if (DEBUG_OUTPUT)
        printf("  => Variable %ld state: v %ld, f %ld, fv %ld, t %ld, m %ld\n",
               varid, mem[0], mem[1], mem[2], mem[3], mem[4]);

    /* Set the results in the given pointers */
    *results = mem;
    *count = 5;

    printf(" Fetched cache state of variable %ld\n", varid);
    return 0;
}
```

```
/*
** Look up all valid variable entries in the cache.  Memory for the array is
** managed inside this function and should not be freed elsewhere.  This
** function operates purely within the cache and does not access the hardware.
**
** The returned array format per entry is:
**   {long int varid, long int forced, long int force_val,
**    long int threshold, long int cond_mask}
*/
#define VALID_ENTRY_SIZE 5
int
get_valid_variables(long int *count, long int **results)
{
    static long int *mem = NULL;
    static long int memlen = 0;
    struct var_state *v;
    long int *newmem, newlen;
    long int i, j;

    /* Check our buffer space and resize if needed */
    if (num_valid_vars > memlen) {
        newlen = num_valid_vars + RESULTBUF_RESIZE_INC;
        newmem = malloc(newlen * VALID_ENTRY_SIZE * 4);
        if (newmem == NULL) {
            perror("Failed to allocate memory for valid variables");
            cleanup(-1);
        }
        memcpy(newmem, mem, memlen * VALID_ENTRY_SIZE * 4);
        if (mem != NULL) free(mem);
        if (DEBUG_OUTPUT)
            printf("  => Valid variable buffer resized from %ld to %ld\n",
                   memlen, newlen);
        mem = newmem;
        memlen = newlen;
    }

    /* Scan the cache for all valid variables and write them to the buffer */
    for (i = 0, j = 0; i <= max_varid; i++) {
        v = &var_state_cache[i];
        if (!v->valid) continue;
        mem[VALID_ENTRY_SIZE*j    ] = i;
        mem[VALID_ENTRY_SIZE*j + 1] = (long int)v->forced;
        mem[VALID_ENTRY_SIZE*j + 2] = v->force_val;
        mem[VALID_ENTRY_SIZE*j + 3] = v->threshold;
        mem[VALID_ENTRY_SIZE*j + 4] = v->cond_mask;
        j++;
        if (DEBUG_OUTPUT)
            printf("  => Variable %ld found and is valid (%ld of %ld)\n",
                   i, j, num_valid_vars);
    }
    assert(j == num_valid_vars);

    /* Set the results in the given pointers */
```

```
    *results = mem;
    *count = j * VALID_ENTRY_SIZE;

    printf(" Fetched %ld valid variables\n", j);
    return 0;
}


/*
** Look up all forced variable entries in the cache.  Memory for the array is
** managed inside this function and should not be freed elsewhere.  This
** function operates purely within the cache and does not access the hardware.
**
** The returned array format per entry is:
**   {long int varid, long int force_val}
*/
#define FORCED_ENTRY_SIZE 2
int
get_forced_variables(long int *count, long int **results)
{
    static long int *mem = NULL;
    static long int memlen = 0;
    struct var_state *v;
    long int *newmem, newlen;
    long int i, j;

    /* Check our buffer space and resize if needed */
    if (num_forced_vars > memlen) {
        newlen = num_forced_vars + RESULTBUF_RESIZE_INC;
        newmem = malloc(newlen * FORCED_ENTRY_SIZE * 4);
        if (newmem == NULL) {
            perror("Failed to allocate memory for forced variables");
            cleanup(-1);
        }
        memcpy(newmem, mem, memlen * FORCED_ENTRY_SIZE * 4);
        if (mem != NULL) free(mem);
        if (DEBUG_OUTPUT)
            printf("  => Forced variable buffer resized from %ld to %ld\n",
                   memlen, newlen);
        mem = newmem;
        memlen = newlen;
    }

    /* Scan the cache for all forced variables and write them to the buffer */
    for (i = 0, j = 0; i <= max_varid; i++) {
        v = &var_state_cache[i];
        if (!v->valid) continue;
        if (v->forced) { /* a forced variable */
            mem[FORCED_ENTRY_SIZE*j    ] = i;
            mem[FORCED_ENTRY_SIZE*j + 1] = v->force_val;
            j++;
            if (DEBUG_OUTPUT)
                printf("  => Variable %ld found and is forced (%ld of %ld)\n",
                       i, j, num_forced_vars);
```

```
        }
    }
    assert(j == num_forced_vars);

    /* Set the results in the given pointers */
    *results = mem;
    *count = j * FORCED_ENTRY_SIZE;

    printf(" Fetched %ld forced variables\n", j);
    return 0;
}


/*
** Look up all active assertions and create an array of entries.  Memory for
** the array is managed inside this function and should not be freed elsewhere.
** This function operates purely within the cache and does not access the
** hardware.
**
** The returned array format per entry is:
**   {long int varid, long int threshold, long int cond_mask}
*/
#define ASSERT_ENTRY_SIZE 3
int
get_active_assertions(long int *count, long int **results)
{
    static long int *mem = NULL;
    static long int memlen = 0;
    struct var_state *v;
    long int *newmem, newlen;
    long int i, j;

    /* Check our buffer space and resize if needed */
    if (num_active_assertions > memlen) {
        newlen = num_active_assertions + RESULTBUF_RESIZE_INC;
        newmem = malloc(newlen * ASSERT_ENTRY_SIZE * 4);
        if (newmem == NULL) {
            perror("Failed to allocate memory for active assertions");
            cleanup(-1);
        }
        memcpy(newmem, mem, memlen * ASSERT_ENTRY_SIZE * 4);
        if (mem != NULL) free(mem);
        if (DEBUG_OUTPUT)
            printf("  => Active assertion buffer resized from %ld to %ld\n",
                   memlen, newlen);
        mem = newmem;
        memlen = newlen;
    }

    /* Scan the cache for all active assertions and write them to the buffer */
    for (i = 0, j = 0; i <= max_varid; i++) {
        v = &var_state_cache[i];
        if (!v->valid) continue;
        if (v->cond_mask) { /* an active assertion */
```

```
            mem[ASSERT_ENTRY_SIZE*j    ] = i;
            mem[ASSERT_ENTRY_SIZE*j + 1] = v->threshold;
            mem[ASSERT_ENTRY_SIZE*j + 2] = v->cond_mask;
            j++;
            if (DEBUG_OUTPUT)
                printf("  => Variable %ld found and asserting (%ld of %ld)\n",
                       i, j, num_active_assertions);
        }
    }
    assert(j == num_active_assertions);

    /* Set the results in the given pointers */
    *results = mem;
    *count = j * ASSERT_ENTRY_SIZE;

    printf(" Fetched %ld active assertions\n", j);
    return 0;
}


/*
** Look up all active assertions and read the current variable value from the
** hardware.  This effectively gives a snapshot of all current assertion/
** threshold comparisons.  Memory for the array is managed inside this function
** and should not be freed elsewhere.  Each call requires one hardware
** variable read per active assertion.
**
** The returned array format per entry is:
**   {long int varid, long int threshold, long int cond_mask, long int value}
*/
#define COMPARE_ENTRY_SIZE 4
int
get_active_comparisons(long int *count, long int **results)
{
    static long int *mem = NULL;
    static long int memlen = 0;
    struct var_state *v;
    long int *newmem, newlen, data, status;
    long int i, j, result;

    /* Check our buffer space and resize if needed */
    if (num_active_assertions > memlen) {
        newlen = num_active_assertions + RESULTBUF_RESIZE_INC;
        newmem = malloc(newlen * COMPARE_ENTRY_SIZE * 4);
        if (newmem == NULL) {
            perror("Failed to allocate memory for active assertions");
            cleanup(-1);
        }
        memcpy(newmem, mem, memlen * COMPARE_ENTRY_SIZE * 4);
        if (mem != NULL) free(mem);
        if (DEBUG_OUTPUT)
            printf("  => Active comparison buffer resized from %ld to %ld\n",
                   memlen, newlen);
        mem = newmem;
```

```
        memlen = newlen;
    }

    /* Scan the cache for all active assertions, read the variable's value
       from the hardware, and write all the results to the buffer */
    for (i = 0, j = 0; i <= max_varid; i++) {
        v = &var_state_cache[i];
        if (!v->valid) continue;
        if (v->cond_mask) { /* an active assertion */
            printf(" |");
            result = readvar(i, &data, &status);
            if (result) return result;
            mem[COMPARE_ENTRY_SIZE*j    ] = i;
            mem[COMPARE_ENTRY_SIZE*j + 1] = v->threshold;
            mem[COMPARE_ENTRY_SIZE*j + 2] = v->cond_mask;
            mem[COMPARE_ENTRY_SIZE*j + 3] = data;
            j++;
            if (DEBUG_OUTPUT)
                printf("  => Variable %ld found and asserting (%ld of %ld)\n",
                       i, j, num_active_assertions);
        }
    }
    assert(j == num_active_assertions);

    /* Set the results in the given pointers */
    *results = mem;
    *count = j * COMPARE_ENTRY_SIZE;

    printf(" Fetched %ld active comparisons\n", j);
    return 0;
}


/*
** Look up all the current variable values from the cache and/or hardware.
** Memory for the array is managed inside this function and should not be
** freed elsewhere.  Each call requires one hardware read per variable up to
** the maximum variable ID provided.  Soft errors are indicated with a return
** value of 1 and a (negative) code in count.
**
** The returned array format per entry is:
**   {long int valid, long int forced, long int force_val,
**    long int threshold, long int cond_mask, long int value}
*/
#define ALLVAL_ENTRY_SIZE 6
int
get_all_values(long int varid, long int *count, long int **results)
{
    static long int *mem = NULL;
    static long int memlen = 0;
    struct var_state *v;
    long int *newmem, newlen, data, status;
    long int i, result;
```

```
    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *count = ERR_VARIDRANGE;
        *results = NULL;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* Check our buffer space and resize if needed */
    if (varid + 1 > memlen) {
        newlen = varid + 1 + RESULTBUF_RESIZE_INC;
        newmem = malloc(newlen * ALLVAL_ENTRY_SIZE * 4);
        if (newmem == NULL) {
            perror("Failed to allocate memory for all-value buffer");
            cleanup(-1);
        }
        memcpy(newmem, mem, memlen * ALLVAL_ENTRY_SIZE * 4);
        if (mem != NULL) free(mem);
        if (DEBUG_OUTPUT)
            printf("  => All-value buffer resized from %ld to %ld\n",
                   memlen, newlen);
        mem = newmem;
        memlen = newlen;
    }

    /* Check the cache entry and read the value of all variables up to the
       given maximum */
    for (i = 0; i <= varid; i++) {
        printf(" |");
        result = readvar(i, &data, &status); /* touches cache */
        if (result) return result;
        v = &var_state_cache[i];
        mem[ALLVAL_ENTRY_SIZE*i    ] = (long int) v->valid;
        mem[ALLVAL_ENTRY_SIZE*i + 1] = (long int) v->forced;
        mem[ALLVAL_ENTRY_SIZE*i + 2] = v->force_val;
        mem[ALLVAL_ENTRY_SIZE*i + 3] = v->threshold;
        mem[ALLVAL_ENTRY_SIZE*i + 4] = v->cond_mask;
        mem[ALLVAL_ENTRY_SIZE*i + 5] = data;
    }

    /* Set the results in the given pointers */
    *results = mem;
    *count = i * ALLVAL_ENTRY_SIZE;

    printf(" Fetched %ld variable values\n", i);
    return 0;
}


/**************************
 * Begin hardware commands *
 **************************/
```

```
/*
** Read the given variable's value and store it in the given pointer location
*/
int
readvar(long int varid, long int *val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *status = ERR_VARIDRANGE;
        *val = 0;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* First check if the variable is in the cache and forced */
    if (check_var_state_cache(varid) && var_state_cache[varid].forced) {
        *val = var_state_cache[varid].force_val;
        printf(" Read variable %ld from cache: 0x%08lX (%ld)\n",
               varid, *val, *val);
        /* Read the status register since the service response expects it */
        result = read_reg(fd_stat, status, "bdb_status_out");
        if (result) return result;
        return 0;
    }

    /* Set the variable ID */
    result = write_reg(fd_din, varid, "bdb_data_in");
    if (result) return result;

    /* Send the readvar command */
    cmd = CMD_READVAR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* Read the output data */
    result = read_reg(fd_dout, val, "bdb_data_out");
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Read variable %ld from hardware: 0x%08lX (%ld)\n",
           tdiff(t1,t2), varid, *val, *val);
```

```
    return 0;
}


/*
** Force the given variable's value as specified
*/
int
forcevar(long int varid, long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *status = ERR_VARIDRANGE;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* First check if the variable is in the cache and identically forced */
    if (check_var_state_cache(varid) && var_state_cache[varid].forced &&
        (var_state_cache[varid].force_val == val)) {
        printf(" Variable %ld already forced to 0x%08lX (%ld)\n",
               varid, val, val);
        /* Read the status register since the service response expects it */
        result = read_reg(fd_stat, status, "bdb_status_out");
        if (result) return result;
        return 0;
    }

    /* Set the variable ID */
    result = write_reg(fd_din, varid, "bdb_data_in");
    if (result) return result;

    /* Send the forcevar command */
    cmd = CMD_FORCEVAR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Send the data value to be written */
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the second phase command (value latched upon command change) */
    cmd = CMD_FORCEVAR | (1 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;
```

```
    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    /* Update the variable state cache */
    if (!var_state_cache[varid].valid) num_valid_vars++;
    var_state_cache[varid].valid = 1;
    if (!var_state_cache[varid].forced) num_forced_vars++;
    var_state_cache[varid].forced = 1;
    var_state_cache[varid].force_val = val;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Forced variable %ld to 0x%08lX (%ld)\n",
           tdiff(t1,t2), varid, val, val);
    return 0;
}


/*
** Release the given variable (undoes a force operation)
*/
int
releasevar(long int varid, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *status = ERR_VARIDRANGE;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* First check if the variable is in the cache and not forced */
    if (!check_var_state_cache(varid) || !var_state_cache[varid].forced) {
        printf(" Variable %ld not forced\n", varid);
        /* Read the status register since the service response expects it */
        result = read_reg(fd_stat, status, "bdb_status_out");
        if (result) return result;
        return 0;
    }

    /* Set the variable ID */
    result = write_reg(fd_din, varid, "bdb_data_in");
    if (result) return result;

    /* Send the releasevar command */
    cmd = CMD_RELEASEVAR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;
```

```
    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    /* Update the variable state cache */
    if (!var_state_cache[varid].valid) num_valid_vars++;
    var_state_cache[varid].valid = 1;
    if (var_state_cache[varid].forced) num_forced_vars--;
    var_state_cache[varid].forced = 0;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Released variable %ld\n",
           tdiff(t1,t2), varid);
    return 0;
}



/*
** Halt the design clock
*/
int
halt(long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the halt command */
    cmd = CMD_HALT | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Design clock halted\n", tdiff(t1,t2));
    return 0;
}


/*
```

```
** Run the design clock for the given number of cycles (remains in halt state)
*/
int
runfor(long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Set the cycle count */
    if (val <= 0) {  /* KBC: Should this just be an unsigned value? */
        printf("Invalid cycle count: %ld\n", val);
        return 1;
    }
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the runfor command */
    cmd = CMD_RUNFOR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Design clock run for %ld cycles\n",
           tdiff(t1,t2), val);
    return 0;
}


/*
** Resume the system clock to normal operation
*/
int
resume(long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the continue command */
    cmd = CMD_RESUME | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;
```

```
    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Design clock resumed\n", tdiff(t1,t2));
    return 0;
}


/*
** Set the assertion check threshold for a variable to the given value
*/
int
setthresh(long int varid, long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *status = ERR_VARIDRANGE;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* First check if the variable cached with the same threshold */
    if (check_var_state_cache(varid) &&
        (var_state_cache[varid].threshold == val)) {
        printf(" Threshold for variable %ld "
               "already set to 0x%08lx (%ld)\n",
               varid, val, val);
        /* Read the status register since the service response expects it */
        result = read_reg(fd_stat, status, "bdb_status_out");
        if (result) return result;
        return 0;
    }

    /* Set the variable ID */
    result = write_reg(fd_din, varid, "bdb_data_in");
    if (result) return result;

    /* Send the setthresh command */
    cmd = CMD_SETTHRESH | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Send the threshold value to be written */
```

```
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the second phase command (value latched upon command change) */
    cmd = CMD_SETTHRESH | (1 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    /* Update the variable state cache */
    if (!var_state_cache[varid].valid) num_valid_vars++;
    var_state_cache[varid].valid = 1;
    var_state_cache[varid].threshold = val;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Threshold for variable %ld set to 0x%08lX (%ld)\n",
           tdiff(t1,t2), varid, val, val);
    return 0;
}


/*
** Set the assertion condition mask for a variable to the given value
*/
int
setconds(long int varid, long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Check the range of the requested variable ID */
    if ((varid >= numvar) || (varid < 0)) {
        *status = ERR_VARIDRANGE;
        printf(" Error: Variable ID %ld out of range\n", varid);
        return 1;
    }

    /* First check if the variable cached with the same condition mask */
    if (check_var_state_cache(varid) &&
        (var_state_cache[varid].cond_mask == val)) {
        printf(" Condition mask for variable %ld "
               "already set to 0x%08lx (%ld)\n",
               varid, val, val);
        /* Read the status register since the service response expects it */
        result = read_reg(fd_stat, status, "bdb_status_out");
```

```
        if (result) return result;
        return 0;
    }

    /* Set the variable ID */
    result = write_reg(fd_din, varid, "bdb_data_in");
    if (result) return result;

    /* Send the setconds command */
    cmd = CMD_SETCONDS | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Send the condition mask to be written */
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the second phase command (value latched upon command change) */
    cmd = CMD_SETCONDS | (1 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    /* Update the variable state cache */
    if (!var_state_cache[varid].valid) num_valid_vars++;
    var_state_cache[varid].valid = 1;
    if (val && !var_state_cache[varid].cond_mask) num_active_assertions++;
    if (!val && var_state_cache[varid].cond_mask) num_active_assertions--;
    var_state_cache[varid].cond_mask = val;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Condition mask for variable %ld set to 0x%08lX (%ld)\n",
           tdiff(t1,t2), varid, val, val);
    return 0;
}


/*
** Read the requested word-aligned memory address and set its 32-bit value in
** the given pointer location
*/
static
int
readword(long int addr, long int *val, long int *status)
{
    long int cmd;
    int result;
```

```
    gettimeofday(&t1, NULL);

    /* Set the address */
    result = write_reg(fd_din, addr, "bdb_data_in");
    if (result) return result;

    /* Send the readword command */
    cmd = CMD_READWORD | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* Read the output data */
    result = read_reg(fd_dout, val, "bdb_data_out");
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Read from memory address 0x%08lX: 0x%08lX (%ld)\n",
           tdiff(t1,t2), addr, *val, *val);
    return 0;
}


/*
** Wrapper function which calls readword() sequentially to read a block of
** hardware memory into the specified buffer location
*/
int
readmem(long int addr, long int *vals, long int len, long int *status)
{
    int i, result;

    for (i = 0; i < len; i++) {
        printf(" |");
        result = readword(addr+(i*4), &vals[i], status);
        if (result) return result;
    }

    printf(" Read %ld words from memory\n", len);
    return 0;
}


/*
** Write the given 32-bit data value to the requested word-aligned memory
** address
```

```
*/
static
int
writeword(long int addr, long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Set the address */
    result = write_reg(fd_din, addr, "bdb_data_in");
    if (result) return result;

    /* Send the writeword command */
    cmd = CMD_WRITEWORD | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Send the data value to be written */
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the second phase command (value latched upon command change) */
    cmd = CMD_WRITEWORD | (1 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Wrote address 0x%08lX with value 0x%08lX (%ld)\n",
           tdiff(t1,t2), addr, val, val);
    return 0;
}


/*
** Wrapper function which calls writeword() sequentially to write a block of
** hardware memory using values from the specified buffer location
*/
int
writemem(long int addr, long int *vals, long int len, long int *status)
{
    int i, result;

    for (i = 0; i < len; i++) {
        printf(" |");
```

```
        result = writeword(addr+(i*4), vals[i], status);
        if (result) return result;
    }

    printf(" Wrote %ld words to memory\n", len);
    return 0;
}


/*
** Look up the current stream address on the hardware and return it the
** provided pointer location
*/
int
getstreamaddr(long int *val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the getstreamaddr command */
    cmd = CMD_GETSTREAMADDR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* Read the output data */
    result = read_reg(fd_dout, val, "bdb_data_out");
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Fetched stream pointer address: 0x%08lX\n",
           tdiff(t1,t2), *val);
    return 0;
}


/*
** Set the current stream address on the hardware to the provided value
*/
int
setstreamaddr(long int val, long int *status)
{
    long int cmd;
    int result;
```

```
    gettimeofday(&t1, NULL);

    /* Set the pointer address to write */
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the setstreamaddr command */
    cmd = CMD_SETSTREAMADDR | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Set stream pointer to 0x%08lX\n",
           tdiff(t1,t2), val);
    return 0;
}


/*
** Look up the current cycle count on the hardware and return it the provided
** pointer location
*/
int
getcyclecount(long int *val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the getcyclecount command */
    cmd = CMD_GETCYCLECOUNT | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* Read the output data */
    result = read_reg(fd_dout, val, "bdb_data_out");
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;
```

```
    gettimeofday(&t2, NULL);
    printf(" [%ldus] Fetched cycle count: 0x%08lX\n",
           tdiff(t1,t2), *val);
    return 0;
}


/*
** Reset the cycle count to zero
*/
int
resetcyclecount(long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the resetcyclecount command */
    cmd = CMD_RESETCYCLECOUNT | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Cycle count reset to zero\n", tdiff(t1,t2));
    return 0;
}


/*
** Trigger a readback capture
*/
int
capture(long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the capture command */
    cmd = CMD_CAPTURE | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;
```

```
    /* Wait to be certain the delayed capture (CAP) is complete */
    sleep(2);

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Readback capture triggered\n", tdiff(t1,t2));
    return 0;
}


/*
 ** Trigger a global restore
 */
int
restore(long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the restore command */
    cmd = CMD_RESTORE | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Wait to be certain the delayed restore (GSR) is complete */
    sleep(2);

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Global restore triggered\n", tdiff(t1,t2));
    return 0;
}


/*
** Set data on the ICAP bus and pulse CCLK
*/
static
```

```
int
icapwrite(long int val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Set the ICAP data to write */
    result = write_reg(fd_din, val, "bdb_data_in");
    if (result) return result;

    /* Send the icapwrite command */
    cmd = CMD_ICAPWRITE | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Drove 0x%04lX on ICAP bus\n",
           tdiff(t1,t2), val);
    return 0;
}


int
icapwritewords(long int *vals, long int len, long int *status)
{
    int i, j, result;

    /* Ensure the ICAP bus starts idle */
    printf(" ||");
    result = icapwrite(0x300, status);
    if (result) return result;

    /* Write the sequence of bytes sent in the request */
    for (i = 0; i < len; i++) {
        for (j = 3; j >= 0; j--) {
            short int val = (vals[i] >> (j*8)) & 0x0FF;
            printf(" |");
            result = icapwrite(val, status);
            if (result) return result;
        }
    }

    /* Release the ICAP bus */
    printf(" ||");
```

```
    result = icapwrite(0x300, status);
    if (result) return result;

    printf(" Wrote %ld words on ICAP bus\n", len);
    return 0;
}


/*
** Read the data on the ICAP bus and pulse CCLK
*/
static
int
icapread(long int *val, long int *status)
{
    long int cmd;
    int result;

    gettimeofday(&t1, NULL);

    /* Send the icapread command */
    cmd = CMD_ICAPREAD | (0 << 16);
    result = write_reg(fd_cmd, cmd, "bdb_cmd_in");
    if (result) return result;

    /* Make sure the hardware task has completed */
    result = wait_for_hardware();
    if (result) return result;

    /* Read the output data */
    result = read_reg(fd_dout, val, "bdb_data_out");
    if (result) return result;

    /* End the command */
    *status = end_command();
    if (*status == -1) return -1;

    gettimeofday(&t2, NULL);
    printf(" [%ldus] Read ICAP bus: 0x%04lX\n",
           tdiff(t1,t2), *val);
    return 0;
}


int
icapreadwords(long int *vals, long int len, long int *status)
{
    int i, j, result;
    long int byte;

    /* Prepare ICAP bus for reading */
    printf(" ||");
    result = icapwrite(0x100, status);
    if (result) return result;
```

```
    /* Read the initial cycle */
    printf(" ||");
    result = icapread(&byte, status);
    if (result) return result;

    /* Read the number of requested words */
    for (i = 0; i < len; i++) {
        long int val = 0;
        for (j = 3; j >= 0; j--) {
            printf(" |");
            result = icapread(&byte, status);
            if (result) return result;
            val = val | (byte << (j*8));
        }
        vals[i] = val;
    }

    /* Read another byte to observe ICAP status (for debug only) */
    printf(" ||");
    result = icapread(&byte, status);
    if (result) return result;

    /* Release the ICAP bus */
    printf(" ||");
    result = icapwrite(0x300, status);
    if (result) return result;

    printf(" Read %ld words on ICAP bus\n", len);
    return 0;
}
```

B.2 Matlab verification routines

The client-side remote interface presented in Sect. 5.3 is implemented as both a GUI and library of functions within the Matlab environment. Because of the direct correspondence between many Matlab library routines and the hardware-specific commands which are already fully documented in Appendix A and previously in this chapter, these redundant routines are not reproduced here. However, several of the utility functions which are unique to the translation of fixed-point data to and from the hardware, as well as those which demonstrate the preservation of connection state within the client, are documented below. The selection of functions to be included here also correspond to the list chosen in Table 5.5.

B.2.1 bdb_connect

```
function bdb_connect(server, port)
% BDB_CONNECT
%
```

```
global bdb_connection

% Check arguments
if nargin < 2
    error('Two input arguments required');
elseif nargin > 2
    error('Too many input arguments');
end

% Make sure there's not already a connection
if ~isempty(bdb_connection)
    fprintf(['\nConnection to BDB already exists\n' ...
                'Disconnect before reconnecting to service\n\n']);
    return
end

% Open a socket to the given BDB service
fprintf('\n');
socket = opensocket(server, port);
fprintf('\n');

% Store this connection
bdb_connection = socket;
```

B.2.2 bdb_disconnect

```
function bdb_disconnect
% BDB_DISCONNECT

global bdb_connection

% Make sure there's a connection to close
if isempty(bdb_connection)
    fprintf('\nNo existing BDB connection\n\n');
    return
end

% Send a command ID of -1 to gracefully end the session
try
  sendbytes(bdb_connection, num2bytes(-1, 32));
catch
  % No need to do anything if we're closing up
end

% Close the socket
try
  closesocket(bdb_connection);
  fprintf('\nBDB connection closed\n\n');
catch
  % No need to do anything if we closing up
end

% Clear the connection
```

```
bdb_connection = [];
```

B.2.3 bdb_readhist

```
function [hw_values, status] = bdb_readhist(varid, count, vsize)
% BDB_READHIST

% Check arguments
if nargin ~= 3
    error('Three input arguments required');
elseif ~isnumeric(varid)
    error('Variable ID must be a numeric value');
elseif length(varid) ~= 1
    error('Variable ID must be a single scalar value');
elseif ~isnumeric(count)
    error('Sample count must be a numeric value');
elseif length(count) ~=  1
    error('Sample count must be a single scalar value');
elseif count <= 0
    error('Sample count must be a positive integer');
elseif ~isnumeric(vsize)
    error('Variable storage size must be a numeric value');
elseif length(vsize) ~=  1
    error('Variable storage size must be a single scalar value');
elseif isempty(find(vsize == [8 16 32], 1))
    error('Variable storage size must be either 8, 16, or 32');
end

% KBC: We don't explicitly check if the hardware is halted here...  If not,
%      the stream pointer fetched will be arbitrary, and there's really no
%      guarantee that the hardware isn't overwriting data during the read
%      operations, either.  This makes it the user's responsibility.

% Get the current memory pointer
base_addr = bdb_getstreamaddr;

% Check the number of variables in the design
varparams = bdb_lookup_varparams;
varnames = fieldnames(varparams);
numvars = length(varnames);

% Look up the DIMM capacity
% KBC: Should this be replaced by a wrapper function to better abstract
%      away the actual system parameter data structure in case of changes
%      (not to mention isolation and consistent error information)?
global bdb_system_params
if isempty(bdb_system_params)
    error('BDB system parameter structure not found');
end
totalmem = bdb_system_params.totalmem * 1024^2;

% Calculate the variable layout in memory in terms of byte addresses
rows_per_samp = ceil((numvars * vsize) / 256);
```

```
samp_offset = varid * (vsize / 8);
samp_interval = rows_per_samp * (256 / 8);

% Perform the memory read(s) from the hardware
% KBC: For performance reasons on large reads, this would be much better
%      done by adding an interval/sweep parameter to the bdb_memread
%      command so that only one transfer is made over the network...
hw_values = zeros(1, count);
for n = 0 : 1 : count-1
    % Calculate the address to read, accounting for DRAM wrap-around
    addr = mod(base_addr + samp_offset + n*samp_interval, totalmem);
    % Send the hardware memory read command
    [hw_values(n+1), status] = bdb_readmem(addr, 1, vsize);
end
```

B.2.4 bdb_lookup_varids

```
function varids = bdb_lookup_varids(varnames)
% BDB_LOOKUP_VARIDS  Look up the ID(s) of the variable(s) in the map structure
%
%   Return value is an array of variable IDs corresponding to the input
%   variable names

% Check argument
if ischar(varnames)
    varnames = {varnames};
elseif ~iscell(varnames)
    error('Argument must be a string or cell array of strings');
end

% Make sure we have a variable map to check
global bdb_variable_map
if isempty(bdb_variable_map)
    error('BDB variable mapping structure not found');
end

% Look up the variable name(s)
varids = zeros(1, length(varnames));
for i = 1:length(varnames)
    varname = varnames{i};
    if ~ischar(varname)
        error('All elements of argument cell array must be strings');
    elseif ~isfield(bdb_variable_map, varname)
        error('Variable "%s" not found in mapping structure', varname);
    end
    varids(i) = bdb_variable_map.(varname).varid;
end
```

B.2.5 bdb_lookup_varnames

```
function varnames = bdb_lookup_varnames(varids)
% BDB_LOOKUP_VARNAMES  Look up the name for the given ID(s) in the map structure
%
```

```
%   Return value is a cell array of variable names corresponding to the
%   input variable IDs

% Check argument
if ~isnumeric(varids)
    error('Argument must be a numeric scalar or array');
end

% Make sure we have a variable map to check
global bdb_variable_map
if isempty(bdb_variable_map)
    error('BDB variable mapping structure not found');
end

% Get the list of variable names (should be in ascending ID order)
all_names = fieldnames(bdb_variable_map);

% Look up the given variable ID(s)
varnames = cell(1,length(varids));
for i = 1:length(varids)
    varid = varids(i);
    if varid > length(all_names) - 1
        error('Variable ID "%d" not found in variable map', varid);
    end
    varnames{i} = all_names{varid + 1};
    % Sanity check
    if bdb_variable_map.(varnames{i}).varid ~= varid
        error('Internal error: Variable map not in ascending order');
    end
end
```

B.2.6 bdb_lookup_varparams

```
function params = bdb_lookup_varparams(varnames)
% BDB_LOOKUP_VARPARAMS  Look up a variable's full map structure entry
%
%   Return value is an array of variable parameter structures taken from
%   the global variable map.  If no input arguments are given, returns the
%   mapping structure itself in bdb_variable_map.(varname) format.

% Check arguments
if nargin == 0
    return_all = 1;
elseif nargin > 1
    error('Too many input arguments');
else
    return_all = 0;
    if ischar(varnames)
        varnames = {varnames};
    elseif ~iscell(varnames)
        error('Argument must be a string or cell array of strings');
    end
end
```

```
% Make sure we have a variable map to check
global bdb_variable_map
if isempty(bdb_variable_map)
    error('BDB variable mapping structure not found');
end

% See if the whole map was requested
if return_all
    params = bdb_variable_map;
    return
end

% Look up the variable entries
for i = 1:length(varnames)
    varname = varnames{i};
    if ~ischar(varname)
        error('All elements of argument cell array must be strings');
    elseif ~isfield(bdb_variable_map, varname)
        error('Variable "%s" not found in mapping structure', varname);
    end
    params(i) = bdb_variable_map.(varname);
end
```

B.2.7 bdb_scale_values

```
function scaled_vals = bdb_scale_values(values, var, mode)
% BDB_SCALE_VALUES  Scales a variable value between integer and fixed-point form
%
%   Usage:  scaled_vals = bdb_scale_values(values, var, mode)
%
%   values is the source value or vector of values to be scaled.
%   VAR can be either a variable name string, a cell array of variable names,
%    or a numeric scalar or vector of variable IDs.  In the case of a cell
%    array or vector, the length must be equal to the length of the value
%    vector.  Each variable identifier will be matched to the corresponding
%    value element.  In the case of a single identifier, all elements in
%    the value input will be treated as the specified variable.  This
%    argument is required so that the function can look up the number
%    representation of the desired variable.
%   MODE can be either 0 to scale integer hardware values into the desired
%    fixed-point form, or 1 to scale a Matlab double into a raw integer form
%    to be sent to the hardware.
%
%   SCALED_VAL is the vector of resulting values in fixed-point (mode=0) or
%    integer (mode=1) format.  In both cases, the Matlab data type is
%    double.

% Check arguments
if nargin ~= 3
    error('Three input arguments required');
elseif ~isnumeric(values)
    error('Value argument must be numeric scalar or array');
```

```
elseif ~ischar(var) && ~iscell(var) && ~isnumeric(var)
    error('Variable identifier must be string, cell, or numeric type');
elseif ischar(var)
    var = {var};
elseif iscell(var) && (length(var) ~= 1) && (length(var) ~= length(values))
    error('Variable name cell array must be singular or same length as values');
elseif isnumeric(var) && (length(var) ~= 1) && (length(var) ~= length(values))
    error('Variable ID array must be scalar or same length as values');
elseif ~isnumeric(mode) || (length(mode) ~= 1)
    error('Scaling mode must be a numeric scalar');
elseif (mode < 0) || (mode > 1)
    error('Invalid scaling mode: %d', mode);
end

% Look up the variable map entry for the given variable(s)
if isnumeric(var), varname = bdb_lookup_varnames(var);
else varname = var; end
varmap_entries = bdb_lookup_varparams(varname);

% Get the signed-ness, size and binary point position of the variable(s)
storage_size = [varmap_entries.storage_size];
arith_type   = [varmap_entries.arith_type];
bitwidth     = [varmap_entries.bitwidth];
bin_pt       = [varmap_entries.bin_pt];

% If reading values from hardware (mode=0), we trust the data range is correct
% by design (allowing or preventing overflow is the user's responsibility)
% KBC: This is still an open issue...
% KBC: Also, should we at least look for a warning for anything strange?
if ~mode
    % Sign-extend the data for any sub-32-bit storage types
    % KBC: This is done in software, as the hardware core shouldn't need to
    %      dynamically keep track of the signed-ness of the selected variable
    %      during reading
    values = values - 2.^storage_size .* (values >= 2.^(storage_size-1));
    % Account for binary point shift
    scaled_vals = values .* (2 .^ -bin_pt);
    % Correct unsigned values (received data is interpreted as signed)
    bias = (2 .^ (bitwidth-bin_pt)) .* (~arith_type & (scaled_vals < 0));
    scaled_vals = scaled_vals + bias;
    return
end

% If sending values to hardware, bias values to integer form and check bounds
int_vals = values .* (2 .^ bin_pt);

% KBC: Currently the behavior is to saturate any values with magnitudes
%      out of range and truncate and fraction underflow.  This could be
%      parameterized or changed by default if necessary.

% Check lower and upper bounds and saturate if needed
% NOTE: Bitwidth alone suffices here, as storage_size >= bitwidth by design
min_vals = -2.^(bitwidth-1) .* arith_type;  % unsigned min_val is simply zero
max_vals = ((2.^(bitwidth-1)-1) .* arith_type) + ...
```

```
            ((2.^bitwidth-1) .* ~arith_type);
clipped_vals = max(int_vals, min_vals);
clipped_vals = min(clipped_vals, max_vals);
if ~isempty(find(clipped_vals ~= int_vals, 1))
    warning('BDB:scalingDataLoss', ...
            'Out-of-range data values saturated during scaling');
end

% Check for underflow in the fractional component
% KBC: This is purely fix() right now, but could be parameterized for each
%      type of rounding, if necessary.
scaled_vals = fix(clipped_vals);
if ~isempty(find(scaled_vals ~= clipped_vals, 1))
    warning('BDB:scalingDataLoss', ...
            'Decimal underflow truncated during scaling');
end
```